**SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA**

**FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES**

Pavol Mederly

# SEMI-AUTOMATED CONSTRUCTION OF MESSAGING-BASED ENTERPRISE APPLICATION INTEGRATION SOLUTIONS

**Dissertation**

FIIT-10890-54689

**Supervisor: Prof. Pavol Návrat**

**August 2011**

Slovak University of Technology in Bratislava

Faculty of Informatics and Information Technologies

Pavol Mederly

# SEMI-AUTOMATED CONSTRUCTION OF MESSAGING-BASED ENTERPRISE APPLICATION INTEGRATION SOLUTIONS

Dissertation

FIIT-10890-54689

Supervisor: **Prof. Pavol Návrat**

Programme: **Software Systems**

Field of study: **9.2.5. Software Engineering**

Organization: **Institute of Informatics and Software Engineering**

August 2011

Author: Pavol Mederly

Fakulta informatiky a informačných technológií

Slovenská technická univerzita v Bratislave

Supervisor: Prof. Pavol Návrat (Slovenská technická univerzita v Bratislave)

Reviewers: Prof. Karol Matiaško (Žilinská univerzita v Žiline)

Dr. Zoltán Balogh (Slovenská akadémia vied)

ACM Subject Classification:

D.2.2 Design Tools and Techniques: Computer-aided software engineering;

I.2.2 Automatic Programming: Program transformation;

I.2.8 Problem Solving, Control Methods, and Search

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

## Anotácia

Integrácia informačných systémov, čiže úsilie s cieľom zaistiť, aby nezávisle od seba vyvinuté systémy dokázali spolupracovať, je už desaťročia významnou témou informatiky v prostredí podnikov, či iných organizácií. Vývoj integračných riešení je ešte stále často spojený s vysokými nákladmi a chybovosťou, a to i napriek veľkému úsiliu o zlepšenie tohto stavu, vynakladanému v podnikovej aj akademickej sfére.

Cieľom predkladanej dizertačnej práce je zefektívnenie procesu tvorby a údržby integračných riešení založených na posielaní správ. V porovnaní s existujúcimi prístupmi využívajúcimi myšlienku modelom riadeného vývoja, ktoré automatizujú generovanie kódu pre integračné riešenia, táto práca ide ďalej: usilujeme sa automatizovať nielen generovanie kódu, ale aj samotný návrh riešenia. Využívame pritom prostriedky umelej inteligencie, konkrétne plánovanie a spĺňanie ohraničení. V rámci práce prezentujeme sadu metód, ktoré pre daný abstraktný návrh integračného riešenia a požiadavky na jeho vlastnosti (ako je napríklad priepustnosť, dostupnosť, monitorovateľnosť, minimalizácia využitia komunikačných prostriedkov a podobne) vytvorí vhodný detailný návrh integračného riešenia a prípadne aj vykonateľný kód.

## Annotation

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 this dissertation 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 dissertation 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.

## Acknowledgement

First and foremost, I would like to thank my supervisor Prof. Pavol Návrat for his guidance throughout my work on this dissertation.

I am also much indebted to Prof. Mária Bieliková, who once invited me to the doctoral study at our faculty, for her encouragement throughout my study. Many thanks go to my colleagues at the Institute of Informatics and Software Engineering, for the friendly atmosphere and unselfish help. Especially I would like to thank Dr. Marián Lekavý for our successful cooperation in the area of planning and for his insightful comments to the draft of this dissertation.

I am very grateful also for the comments of reviewers of this work, Prof. Karol Matiaško and Dr. Zoltán Balogh, which helped me to improve its quality.

I would like to thank my family, especially my wife Anna for her love, support, patience, and for standing with me in all times, and also to our children, Janko, Marienka and Beátka, for bringing much joy to our lives. I am grateful to my parents and parents-in-law for their support throughout my whole study.

Most of all, I would like to thank our God for giving me the life, earthly and eternal, and for all the talents I got from Him. Only to Him be all praise and glory.

# Contents

# List of figures

# List of tables

## List of abbreviations

| | |
|---|---|
| AIS | Academic information system |
| API | application programming interface |
| BPEL | Business Process Execution Language |
| BPMN | Business Process Model and Notation (originally Business Process Modeling Notation) |
| CORBA | Common Object Request Broker Architecture |
| CSP | constraint satisfaction problem |
| DCOM | Distributed Component Object Model |
| DL/P | Data element-level, planning-based method |
| DSL | domain-specific language |
| EAI | enterprise application integration |
| ECA | Event Condition Action |
| EIP | Enterprise integration pattern |
| ESB | enterprise service bus |
| ERP | enterprise resource planning |
| HTN | hierarchical task network |
| HTTP | Hypertext Transfer Protocol |
| ISO | International Organization for Standardization |
| JDBC | Java Database Connectivity |
| JMS | Java Message Service |
| ML/CP | Message-level, constraint programming-based method |
| ML/P | Message-level, planning-based method |
| MOM | message-oriented middleware |
| MQ | message queuing |
| ODBC | Open Database Connectivity |
| OWL-S | an ontology for describing semantic web services |
| PDDL | Planning Domain Definition Language |
| QoS | quality of service |
| RPC | remote procedure call |
| SAT | Boolean satisfiability problem |
| SOAP | a protocol for a communication with web services (originally Simple Object Access Protocol) |
| STRIPS | Stanford Research Institute Problem Solver |
| U/CP | Universal constraint programming-based method |
| UML | Unified Modeling Language |
| WSDL | Web Services Description Language |
| XML | Extensible Markup Language |
| XSLT | Extensible Stylesheet Language Transformations |

# Introduction

In this dissertation we are investigating the possibilities of semi-automated construction of messaging-based enterprise application integration solutions.

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 (Hohpe and Woolf, 2004). A software system that enables such cooperation is often called an *integration solution*.

A related area, inter-enterprise application integration (sometimes known as business-to-business application integration, or B2Bi), deals with making applications in various enterprises to cooperate. Although in basic principles similar to EAI, the situation in B2Bi is more complicated as there come issues of trust, legislative, standards, and so on. For this reason the technologies used in EAI and B2Bi are slightly different.

These two flavors of application integration, together with their 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 (Papazoglou, Traverso, Dustdar, Leymann, and Krämer, 2006).

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*. Let us consider more closely the first three of them.

*Requirements specification* deals with stating requirements that the integration solution must implement or satisfy. Two significant categories of such requirements are *functional* and *non-functional* ones. The former specify the required functionality to be provided by the solution. Its key part is the *business logic* to be implemented, i.e. algorithms and rules such as "when a purchase order arrives, the system has to invoke 'check customer credit' and 'check inventory' services" or "an order coming from a customer of class 'standard' can be processed only if the customer has credit rating of at least 60 and all the ordered products are in the inventory". The latter (non-functional) requirements deal with the performance, reliability, scalability, security, monitoring, logging and auditing features of the integration solution. Generally, requirements are statements imposed by the integration solution *customer*.

An activity of *software* (or *system*) *design* deals with creating a solution blueprint for a software system that would meet the above functional and non-functional requirements. The designer has to choose an appropriate *architecture* first, and then he or she creates a *detailed system design* – the key part being a thorough decomposition of the system functionality into a set of components that together meet all the requirements. In the domain of messaging-based integration solutions the most important part of the design often lies in choosing components to be used and connecting them together appropriately. The design is a strongly creative activity where the developer engages his or her previous experience, well-known approaches (often described in the form of standard architectures and patterns), intuition, and rational thinking.

The *implementation* is concerned with the actual creation of an integration solution in chosen development environment.

Design and implementation activities are usually carried out by the *developer* or developers.

The border line between the above kinds of software engineering activities is sometimes a bit blurred. For example, an exact specification of the control flow of integration solution (obviously a part of business logic to be implemented) can be missing in the requirements document – it could be created during the design process, or it could be even automated, as seen in automatic service composition approaches (Papazoglou et al., 2006). On the other hand, the detailed design can overlap with the implementation: many modern integration platforms allow the developer to work at quite high level of abstraction – the integration solution is being constructed by choosing, configuring, and connecting pre-existing solution components (adapters and integration services), with the need to write actual code being strongly reduced.

As shown in Chapter 1, there are several approaches that assist the developer with the implementation activities in the domain of messaging-based integration solutions, namely (Scheibler and Leymann, 2009) and (Sleiman, Sultán, and Frantz, 2009). 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 chapters 4-6, we have succeeded in using traditional artificial intelligence techniques, namely planning and constraint programming, in order to reach this goal.

The structure of this dissertation is as follows: In *Chapter 1* we describe enterprise application integration – its expected benefits, major issues encountered, main approaches used to achieve it, as well as the most significant results of related research. *Chapter 2* is devoted to formulation of our research problem. *Chapter 3* contains a description of methods and tools we use. Then *chapters 4 to 6* contain our main result – methods for automating selected aspects of integration solutions design and their evaluation. The final chapter contains a conclusion and outlines possibilities for consequential research.

# 1 Enterprise application integration

In the first part of this chapter we describe enterprise application integration, its expected benefits, major issues encountered, and main approaches in this area. This provides us with a general motivation and wider context for our research problem. Describing main approaches in the area of integration also allows us to specify a domain of our work more precisely.

In the second part we describe the current state of research related to our topic.

## *1.1 The need for integration*

It seems that the integration is not "natural": information systems in an organization are usually disintegrated unless it undertakes an explicit effort to integrate them, and keep them integrated. (In literature such disintegrated systems are usually called "silo" or "stovepipe" systems, or "islands of automation".)

Reasons for this situation include:

1. When building or modernizing an enterprise information system, or a part of it, there is a strong need to balance two conflicting forces or needs: (1) needs of a particular project (or department or business unit carrying it out), focusing on its business case: achieving particular business functionality while minimizing costs, with (2) needs of the enterprise as a whole: achieving compatibility, maintainability, flexibility, functionality, efficiency and low total cost of ownership of the whole information system.

   As creating and deploying enterprise systems is hard enough by itself and integrating them with the rest of enterprise's IT landscape is much harder, very often project teams concentrate on achieving their immediate goals and have no time or other resources to take wider aspects into account.

   For more information on this issue please see (Trowbridge, Roxburgh, Hohpe, Manolescu, and Nadhan, 2004), (Cook, 1996), and (Britton, 2000).

2. The enterprise itself is typically rather fragmented – it is divided into business units and departments, frequently with their own IT departments or teams, often not communicating among themselves properly. The result is that the IT systems created by them are not communicating either. (In this context Hohpe and Woolf (2004) cite Conway's law: "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.")

   Of course, the degree of fragmentation depends heavily on the sector the organization is in – e.g. higher education institutions are more decentralized than most commercial ones – as well as individual organization. As a note, this fragmentation of organizations is often the source of the suboptimal functioning not only in the IT area: once very popular business process reengineering efforts try to overcome exactly this problem, see e.g. (Hammer and Champy, 1993).

3. Packaged (or “commercial-off-the-shelf”) products that represent significant part of enterprise IT systems are developed and maintained by independent vendors so they are not compatible “out of the box”.

4. Mergers and acquisitions contribute to the problem significantly.

When an organization has a number of systems that do not communicate among themselves, it usually encounters the following issues:

1. *Business processes that are supported by more than one system cannot be executed seamlessly* – relevant data stored in one of the systems have to be manually re-entered into the other ones. This typically results in delays in process execution, inefficient utilization of human resources, and induction of errors (e.g. when repeatedly entering the same data into several systems).

2. *It is not possible to provide meaningful and consistent information based on data from more than one system* – as the data (e.g. data about a customer) are often independently entered into more systems, it is difficult to (1) relate relevant pieces of the data together, and (2) decide what system contains the correct data in case they overlap and differ.

3. *It is very difficult to share data and services with customers and business partners* – it is impossible to provide consistent information to partners when you cannot obtain it at all, as described above. Moreover, customers and business partners usually cannot (or are not allowed to) use traditional user interfaces designed for internal employees, so relevant internal systems have to be integrated with system or systems providing the access for external users (e.g. portals).

Enterprise application integration is then a process of connecting disparate systems so that they can work together to produce a unified set of functionality. EAI allows an organization to overcome the above-mentioned limitations and provides further benefits, e.g. it allows it to add originally unforeseen functionality to the information system in a flexible and efficient way.

For more information and deeper discussion on issues related to silo systems as well as on benefits of integration, please see e.g. (Linthicum, 2003), (Britton, 2000), (Pan and Viña, 2004), (Chappell, 2004), (Hohpe and Woolf, 2004), and (Cummins, 2002).

## *1.2 Approaches to integration*

In this section we present a categorization of main approaches to integration, as described in the literature – namely in (Trowbridge et al., 2004), (Linthicum, 2003), and (Hohpe and Woolf, 2004) – and then we proceed with the characterization of messaging-based integration solutions that are the main focus of this dissertation.

As the categories presented in the above mentioned publications are rather similar and overlapping, we base our presentation on the one put forth by Trowbridge et al. that we feel is the most comprehensive one; along with certain adaptations that we consider necessary.

The integration options – or patterns as Trowbridge et al. describe them – can be looked at from various points of view:

1. Looking at the overall functionality provided by the integration solution.
2. Looking at how individual systems are connected to the integration solution.
3. Looking at the structure of the integration solution.

### 1.2.1 Classification by the overall functionality of the solution

When looking at the *overall functionality* provided by the integration solution there are the following three basic options:

1. *Portal integration* is an approach that provides end users with a comprehensive view across various systems in a visually consistent format. In its more advanced versions it is also possible to make updates to the presented data. The integration solution itself does not perform any actions upon individual systems, except for those mandated directly by the user through the portal.

2. *Entity aggregation* is an approach that provides systems with a consolidated view on data stored in various systems – again, with the possibility to make updates in more advanced versions of the pattern. This approach is often called “enterprise information integration” (Bernstein and Haas, 2008). Similarly to the case of portal integration, an integration solution of this kind does not perform any actions upon individual systems by itself – only when instructed to do so by client systems utilizing the consolidated view it provides.

3. *Process integration* is an approach that allows managing, or orchestrating, interactions between multiple systems as prescribed by business processes definitions. An integration solution in this case invokes specific functions of individual systems as required by a process definition. Such a processing is triggered either by a defined action within some of the systems or by an external event. This form of integration is perhaps the most closely related to the general meaning of the term “enterprise application integration”.

### 1.2.2 Classification by the nature of connections to systems

When looking at *connections between the integration solution and individual systems* there are the following options:

1. *Data integration* – an integration solution interacts with a system's data layer, typically with the database, either via database interface like JDBC (Java Database Connectivity, a standardized API allowing a Java client to access a database), ODBC (Open Database Connectivity, another standardized API used to access a database), or database vendor-specific API, or through database tools performing transfer and/or sharing of the data with the integration solution.[1]

2. *Functional integration* – an integration solution interacts with a system's business logic. This is the preferred way of connecting to a system from the point of view of ease of development and maintenance, reliability, and security. Technically it can be implemented in many ways, ranging from a very simple ones – using traditional "import/export" functionality provided by the system – to sophisticated ones, using synchronous or asynchronous distributed middleware like RPC (Remote Procedure Call), CORBA (Common Object Request Broker Architecture), DCOM (Distributed Component Object Model), .NET Remoting, web services, or messaging.

3. *Presentation integration* (also called "screen scraping") – an integration solution interacts with a system's presentation layer. This is sometimes the only possibility when the system being integrated is a monolithic, closed one.

In most cases there are special components of an integration solution whose role is to provide connectivity to systems being integrated. These components are usually called *adapters*.

### 1.2.3 Classification by the structure of the integration solution

An integration solution typically consists of many components:

1. adapters that provide a connectivity to systems being integrated,

2. other components that provide specific functionality of the integration solution – they contain e.g. data mapping rules or the logic of business process (in case of process integration approach).

[1] In addition to the above mentioned cases, there is also a very specific kind of EAI mentioned e.g. by Trowbridge et al. (2004) and by Hohpe and Woolf (2004). It is a situation where applications are integrated by storing their data in a shared database. Actually, this is not very common in practice as – in the general case of applications coming from independent sources – it would require considerable modification of applications involved.

These components have to communicate to form a working system. Trowbridge et al. (2004) distinguish the following major integration solution communication structures (or topologies, as they call them):

1. *Point-to-Point Connections* – communication between components is carried out by the sender sending a message to a particular recipient, knowing its address and access protocol.
2. *Broker-based* – the broker is used to decouple communicating components; it can be
    a. a direct broker that helps establish a connection between components, with the subsequent communication flowing directly between them,
    b. an indirect broker that acts as a mediator between components, processing each message flowing between them.

Current major integration technologies are diverse in this respect, for example web services cover cases (1) and (2a), while message brokers correspond to case (2b).

### 1.2.4 Messaging-based integration solutions

In this dissertation we deal with *integration solutions based on messaging* as described in (Hohpe and Woolf, 2004). These solutions can be characterized in the following way:

1. They consist of a set of components that communicate by *sending and receiving messages*, primarily – although not necessarily – in a reliable and asynchronous way.
2. As a primary – again, not exclusive – mean of communication they use *messaging middleware*, also called message-oriented middleware (MOM) or message queuing (MQ).

How are messaging-based integration solutions related to the integration approaches classification described above?

First of all, when looking at the third criterion – i.e. *integration solution structure* – we deal with solutions based on an indirect broker or brokers: this functionality is provided by the messaging middleware that transports messages between individual integration solution components.

From the point of view of the first criterion – i.e. *overall functionality of the integration solution* – we deal with all three integration approaches: although messaging-based integration solutions are a good fit for process integration, they can be used to implement portal integration and entity aggregation as well.

Finally, from the point of view of the second criterion – i.e. *the nature of connections to participating systems* – here again we deal with the full spectrum of approaches:

participating systems are connected to the integration solution using adapters (working at the level of database, business logic, or user interface) and these adapters communicate with the other solution components using messaging. In some cases it is possible for the integration solution to communicate with messaging-aware systems directly.

Generally, messaging is a very good technology for creating integration solutions, because it allows creating *loosely coupled* solutions. First of all, it allows systems to be decoupled *at run-time*: When using traditional synchronous approaches to communication (e.g. remote procedure call, synchronous web service invocations, and so on) it is required that both parties are simultaneously available for the communication to succeed. Unfortunately, this is quite an unrealistic expectation in real-life systems – we need to tolerate some downtime due to unexpected hardware or software failures, as well as for ordinary hardware and software maintenance. For integration solutions connecting many systems (perhaps tens of them) the probability of at least one of them to be unavailable is then quite high. Asynchronous messaging using message broker eliminates this problem, as the only system that has to be always available is the broker itself. And this can be implemented using up-to-date technologies like broker clustering.

Second form of loose coupling allowed by messaging-based integration solution is visible *at design time*. Many traditional integration technologies require communication parties to have the same interface. This has a consequence that when one of them changes – typically because of a system upgrade – the other has to be changed as well. Messaging-based systems allow easy insertion of *mediation components* between integrated systems that can cope with mismatches between interfaces of individual systems.

### 1.2.5 Architectures for messaging-based integration solutions: Pipes and Filters and Process Manager

There are two main implementation paradigms, or architectures, for messaging-based integration solutions:

1. *Pipes and Filters*. This approach is based on processing messages by various components (filters): transformation services, routing services, splitters, aggregators, resequencers, application adapters, and so on. These filters are organized in a predefined structure (graph), and connected by pipes. Pipes are typically implemented in memory or via messaging middleware; however, other channel implementations, e.g. web service calls, can be used as well.

   A simple example of the use of this pattern is shown in Figure 1. Here we want to process purchase orders coming e.g. from company customers or business partners. Each order has to be validated and then customer's credit as well as the inventory has to be checked – using an appropriate service, depending on the ordered product’s type. For simplicity we assume that one order contains products of one type only.

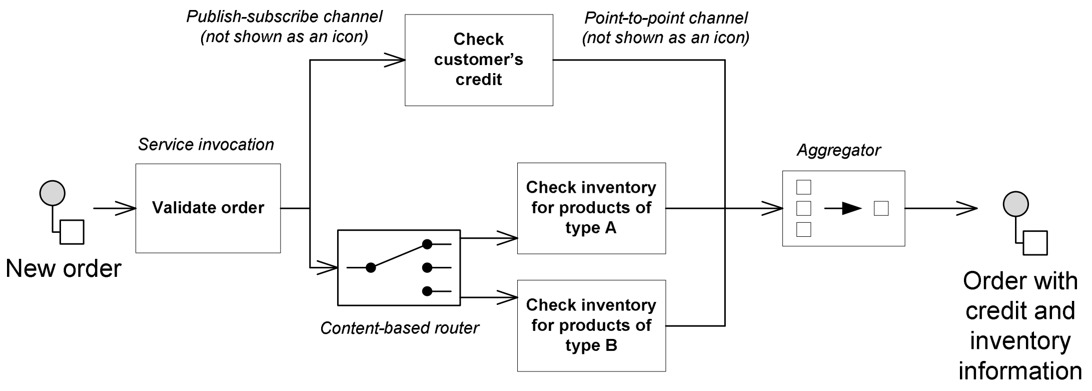

Publish-subscribe channel (not shown as an icon)
Check customer’s credit
Point-to-point channel (not shown as an icon)
Service invocation
Validate order
New order
Aggregator
Check inventory for products of type A
Check inventory for products of type B
Content-based router
Order with credit and inventory information

**Figure 1.** An example of the Pipes and Filters architectural pattern.

(Filters are denoted as boxes, pipes as lines between filters.)

An important feature of Pipes and Filters approach is that messages are self-contained and, moreover, filters do not exchange any information besides those included in messages. Everything that a filter A wants to pass to filter B must be stored in the message or messages flowing within the solution.

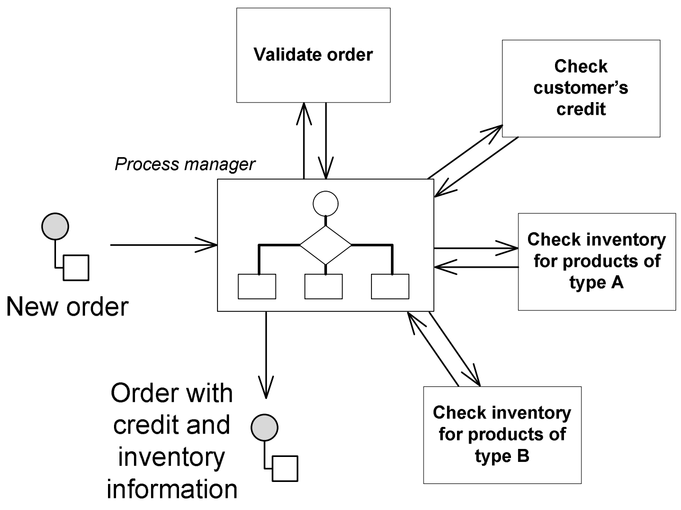

Validate order
Check customer’s credit
Process manager
New order
Check inventory for products of type A
Order with credit and inventory information
Check inventory for products of type B

**Figure 2.** An example of the Process Manager architectural pattern – adapted from (Hohpe and Woolf, 2004)

2. *Process Manager*. Here we have one central component, namely the process manager, orchestrating the whole integration process. The process manager receives incoming message, and invokes the first service (in this case, Validate order) by sending it a message. The service processes a message and sends a reply back to the manager. It then decides which service to invoke next (in this case both Check customer’s credit and one of inventory checking services), sends the message(s), gets the reply or replies and the whole process continues. An example is shown in Figure 2.

Process Manager maintains the overall state of process instance execution, i.e. the state of processing of an individual message. Because this state can contain all data that have been received in the original (trigger) message as well as all data returned by individual services, it is no longer necessary to keep all data in messages that are sent to services: each service can be provided with only those data it really needs to process.

This approach is frequently used for a web service composition.

Both approaches have advantages and disadvantages in terms of development effort, portability, maintainability, operational reliability, efficiency, and ease of administration, and – if using commercial infrastructure – also licensing and support costs. For example, the Process Manager-based solutions are generally easier to develop and maintain, yet less efficient and more costly. This approach is often used to implement long-running processes that span days or weeks and often require human interaction. On the other hand, Pipes and Filters-based solutions are usually more efficient but require more development effort. They are frequently a good fit for implementing technology-oriented, shorter-running processes. For a comparison between these two kinds of processes, please see also the description of Macro-microflow pattern in (Hentrich and Zdun, 2006). For a more detailed comparison of Process Manager and Pipes and Filters architectural styles, see (Mederly and Návrat, 2011).

When creating an integration solution, either Process Manager or Pipes and Filters-based one, a developer usually starts with a specification of *control dependencies* among services invocations executed within that solution. These dependencies can be easily modeled using BPMN (Business Process Model and Notation) or UML activity diagram. For an example please see Figure 3a. Implementing such a specification in languages used in current Process Manager implementations, like BPEL (Business Process Execution Language), is usually not a big problem. For Pipes and Filters architectures this can be quite straightforward in some cases (see for example Figure 3, inspired by a similar comparison presented in (Hohpe and Woolf, 2004)), but rather intricate in others, namely when complex non-functional constraints come into play. As an example of such a correspondence, see e.g. figures 9 and 10 in Chapter 4.

And exactly this question – how to design and implement a Pipes and Filters-based integration solution for a given set of control and data dependencies and non-functional requirements – is the main topic of this dissertation, as will be discussed in Chapter 2.

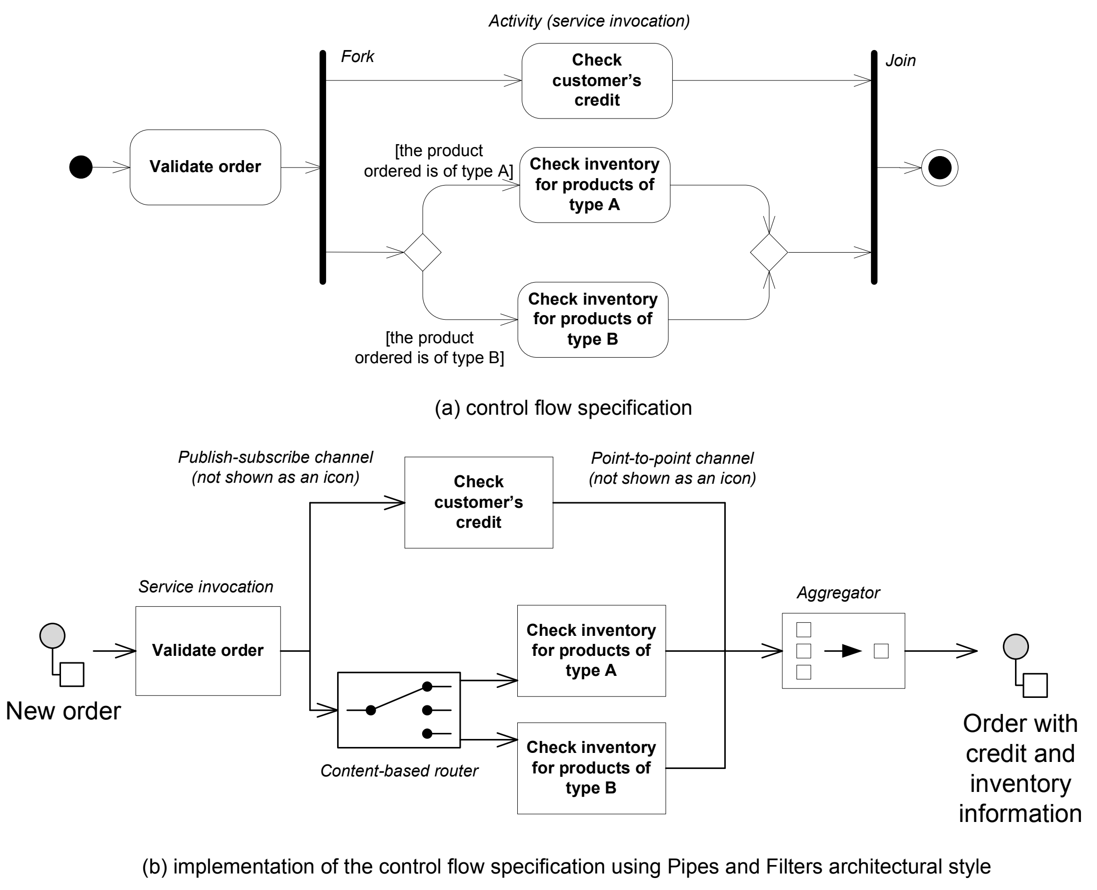

Activity (service invocation)
Fork
Check customer's credit
Join
Validate order
[the product ordered is of type A]
Check inventory for products of type A
Check inventory for products of type B
[the product ordered is of type B]
(a) control flow specification
Publish-subscribe channel (not shown as an icon)
Check customer's credit
Point-to-point channel (not shown as an icon)
Service invocation
Aggregator
Validate order
New order
Check inventory for products of type A
Check inventory for products of type B
Content-based router
Order with credit and inventory information
(b) implementation of the control flow specification using Pipes and Filters architectural style

**Figure 3.** A relation of control flow specification to its Pipes and Filters implementation.

## *1.3 State of the art in the industry*

When creating an integration solution the developers traditionally had to write a significant amount of low-level (technology-oriented) code in order to access individual systems, to implement required data transformations, message routing, splitting and aggregating, and so on.

This code had to be created and then kept up-to-date as systems and integration requirements evolved. The result was that creating and maintaining the integration solution was a tedious, error-prone, and therefore expensive, undertaking.

As this fact was recognized soon, there have been many attempts to improve this situation. Among most relevant ones have been so called EAI (enterprise application integration) tools starting to appear in the second half of 1990's followed by products called ESBs (enterprise service buses), since ca 2003. Both categories of products provided rich infrastructure, consisting of communication mechanisms (e.g. message-oriented middleware), pre-created adapters (e.g. for database systems, mail systems, and specific application software like SAP R/3, Oracle Financials, etc.), and standard integration services (e.g. for transformation and routing of messages). Utilizing this infrastructure, these systems allowed designers to construct an integration solution using a higher level of abstraction than was provided by the traditional programming languages.

Nevertheless, our experiences with the state-of-the-art ESB product (Mederly and Pálos, 2008) as well as with some others show that even using such powerful tools some concerns remain: As programming and modeling languages of existing tools are still at a quite technical level, it is (1) hard to design, create and maintain integration solutions, (2) hard to port those solutions between different integration platforms (e.g. when – for whatever reason – the enterprise has to switch from one platform to another). As mentioned in previous section, these issues are more significant in the area of Pipes and Filters-based solutions. In making these activities easier, cheaper and less error-prone we see the big and important space for this dissertation to cover.

## *1.4 State of the art in the academia*

Technical issues of creating an integration solution or a service composition have been the subject of recent research efforts, undertaken mainly in the following two areas:

1. Model-driven integration
2. Model-driven service composition

They deal with construction of integration solutions and service compositions, respectively, 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.

But before looking at existing results in more depth let us have a look at the following research proposal: Papazoglou, Traverso, Dustdar, Leymann, and Krämer (2006) have published the *Service-Oriented Computing Research Roadmap* containing a list of "Grand Challenges", among which this one is directly applicable to our research: (Emphasis added.)

> ***Business-driven automated compositions Grand Challenge***
>
> *"One of the main ideas of service oriented applications is to abstract away the logic at the business level from its non-business related aspects, the 'system level', e.g., the implementation of transaction, security, and reliability policies. This abstraction should make easier and effective the composition of distributed business processes. However, the provision of automated composition techniques, which make this potential advantage real, is still an open problem. Business-driven automated compositions should exploit business and system level separation in service compositions. According to this view, service composition at the business level should pose the requirements and the boundaries for the automatic composition at the system level. While the service composition at the business level should be supported by user-centered and highly interactive techniques,* ***system level service compositions should be fully automated and hidden to the humans****. System level compositions should be QoS-aware, should be generated and monitored automatically, and should also be based on autonomic computing."*

Said in other words, Papazoglou et al. here call for a clear separation of requirements specification and design activities (the business and system levels) with the latter being carried out fully automatically. This directly corresponds to our goal of automating the design and implementation of integration solutions.

## *1.5 Model-driven integration*

Model-driven development (MDD) (Mellor, Scott, Uhl, and Weise, 2004) is an approach to software development that has the potential to significantly reduce human work needed to construct a software system by automating some of tasks related to its design and implementation. Nowadays it is typically used for the development of individual applications (systems), yet there are attempts to use it also in the area of application integration.

This section reviews several works in the area of model-driven application integration.

### 1.5.1 Modeling languages

First, let us focus on modeling languages used in the area of EAI. Modeling languages for software systems can be categorized with respect to how abstract, concise, and platform-independent they are. At one side there are abstract, platform-independent languages allowing the developer to concentrate on substantial features of problem and solution being developed; at the other side there is a concrete code that can be executed on a specific software/hardware platform. Between them there are modeling languages that are specific to a platform (or a set of related platforms) yet are at a higher level of abstraction than an executable code.

The promise of model-driven development is that one can start with an abstract description and then refine it step-wise, eventually coming to executable code (see Figure 4).

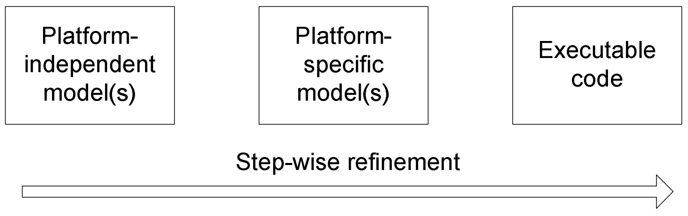


**Figure 4**. Step-wise refinement of an abstract solution specification.

### 1.5.2 Enterprise integration patterns

In the area of enterprise application integration there are many languages specific to particular EAI and ESB tools. As has been mentioned in Section 1.3, they are generally platform-specific and at a quite technical level.

When looking at platform-independent languages, an important result is Hohpe and Woolf’s (2004) book on enterprise integration patterns (or EIPs for short). It captures knowledge on architecture as well as on technical details of integration solutions, specifically in the area of integration based on – mostly asynchronous – messaging.

Authors have described 6 general patterns: Message Channel, Message, Pipes and Filters, Message Router, Message Translator, and Message Endpoint, as well as 55 more specific patterns, mainly refining the general ones. These patterns are summarized in Table 1.

**Table 1.** A list of enterprise integration patterns.

| Area | Pattern |
|---|---|
| Basic patterns | Message Channel |
| | Message |
| | Pipes and Filters |
| | Message Router |
| | Message Translator |
| | Message Endpoint |
| Messaging channels | Point-to-Point Channel |
| | Publish-Subscribe Channel |
| | Datatype Channel |
| | Invalid Message Channel |
| | Dead Letter Channel |
| | Guaranteed Delivery |
| | Channel Adapter |
| | Messaging Bridge |
| | Message Bus |
| Message construction | Command Message |
| | Document Message |
| | Event Message |
| | Request-Reply |
| | Return Address |
| | Correlation Identifier |
| | Message Sequence |
| | Message Expiration |
| | Format Indicator |
| Message routing | Content-Based Router |
| | Message Filter |
| | Dynamic Router |
| | Recipient List |
| | Splitter |
| | Aggregator |
| | Resequencer |
| | Composed Message Processor |
| | Scatter-Gather |
| | Routing Slip |
| | Process Manager |
| | Message Broker |
| Message transformation | Envelope Wrapper |
| | Content Enricher |
| | Content Filter |
| | Claim Check |
| | Normalizer |
| | Canonical Data Model |
| Messaging endpoints | Messaging Gateway |
| | Messaging Mapper |
| | Transactional Client |
| | Polling Consumer |
| | Event-Driven Consumer |
| | Competing Consumers |
| | Message Dispatcher |
| | Selective Consumer |
| | Durable Subscriber |
| | Idempotent Receiver |
| | Service Activator |
| System management | Control Bus |
| | Detour |
| | Wire Tap |
| | Message History |
| | Message Store |
| | Smart Proxy |
| | Test Message |
| | Channel Purger |

Let us shortly describe the most important patterns referenced in this dissertation.

1. *Point-to-Point Channel* is a kind of channel that ensures that each particular message will be consumed by exactly one receiver. An example of such a channel is a queue in messaging middleware implementing JMS (Java Message Service).

2. *Publish-Subscribe Channel* is a kind of channel that enables more receivers (subscribers) to attach to it and then delivers every message to each of them. This functionality is provided e.g. by topics in JMS-based middleware.

3. *Datatype Channel* is a channel that transports messages of a given type. This separation of messages of different types into individual channels is a usual, but not the only one, way of organizing channels in a messaging-based system.

4. *Message Router* is a component that has multiple output channels and routes each incoming message to a selected channel (or channels) based on a set of conditions.

5. *Content-Based Router* is a kind of router that routes messages in dependence on their content.

6. *Message Filter* is a kind of router that either forwards a message to an output channel or discards it, based on a defined condition.

7. *Recipient List* is a kind of router that routes messages to a list of recipients. (In this dissertation we use the simplest form of this component, which sends each message to a predefined list of recipients.)

8. *Wire Tap* is a simple Recipient List that copies each incoming message to the output channel as well as to a special channel intended for message content monitoring. It is often used to monitor message traffic going through a point-to-point channel.

9. *Splitter* is a component that divides a composite message into a series of individual ones.

10. *Aggregator* is a component that merges a set of related messages together, and sends them out as a single message.

11. *Resequencer* is a component that collects messages and sends them out in a defined order.

12. *Composed Message Processor* is a pattern that splits a message into its constituent parts, ensures the appropriate processing of these parts, and reaggregates responses back into a single message.

13. *Scatter-Gather* is a pattern consisting of a mechanism that broadcasts a message to multiple recipients and a mechanism that reaggregates responses back into a single message.

14. *Message Translator* is a component that translates messages from one data format into another.

15. *Content Enricher* is a special kind of Message Translator that changes a message by adding some information to it.

16. *Content Filter* is a special kind of Message Translator that removes unnecessary data from a message.

17. *Transactional Client* is a messaging client that is able to group a set of messaging-related operations into one transaction. In some cases such a transaction can also contain operations on other resources, typically a database.

18. *Idempotent Receiver* is a messaging client that can safely receive the same message multiple times.

19. *Competing Consumers* are multiple consumers reading messages off a channel concurrently, so that they are able to achieve higher processing rate and/or higher system availability in comparison with processing by single consumer.

20. *Message Dispatcher* is a consumer that read messages from a channel and dispatches them to entities that process them (typically each in its own thread).

Even if these patterns were not originally intended as a language to be used in model-driven approach (Hohpe, 2004), they provide a commonly accepted vocabulary that has been, as shown below, used in such an approach. Hohpe and Woolf's patterns have also a visual representation, and therefore they provide an effective means for modeling messaging-based integration solutions. In Figure 5 we show visual symbols for integration patterns that are used in this dissertation.

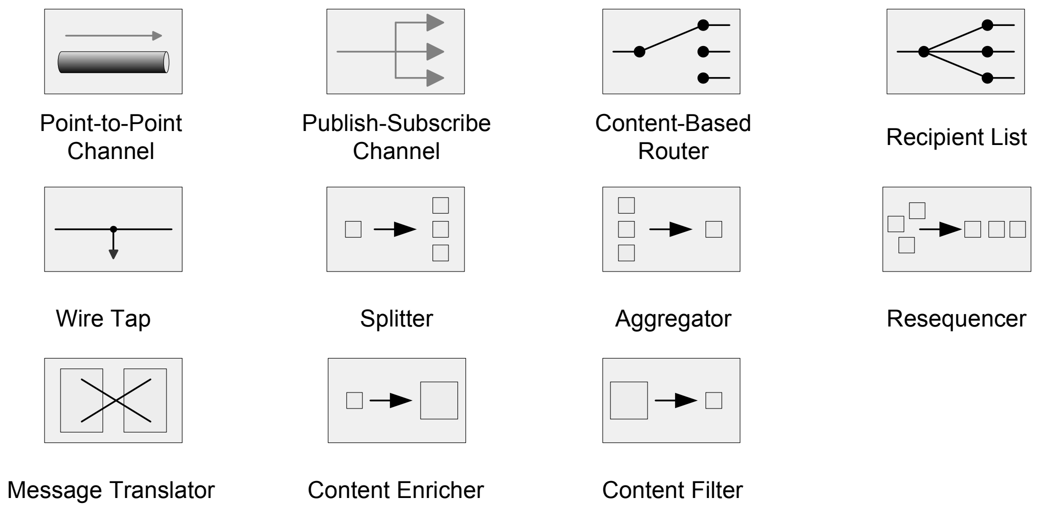


**Figure 5**. Symbols for enterprise integration patterns used in this dissertation.

### 1.5.3 Executable enterprise application integration patterns

Building on the work on enterprise application patterns, Scheibler and Leymann (2008, 2009) have proposed an idea of executable enterprise application integration patterns. 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.

Parameters that are attached to the patterns are of four categories:

1. input: a characterization of input messages,
2. output: a characterization of output messages,
3. characteristics: specifying details of a pattern implementation behavior,
4. control: a characterization of control messages, i.e. those that influence the behavior of a pattern implementation at run time.

For example, their Aggregator pattern has the following parameters that determine exactly how this component should function:

1. completeness condition – whether the aggregator will wait for all expected messages, for a specified amount of time, for an external event, or whether it will treat first message that comes as the best one;
2. timeout value,
3. specification of a criterion and a XML element in message that is used to decide what is the “best” message to pass forth (Hohpe and Woolf, 2004),
4. a channel to receive external events signaling the aggregation completion,
5. a flag indicating whether the aggregation will be realized by an external web service (along with specification of the service, its interface and operation).

Authors have provided a graphical editing environment for creating integration solution designs. The environment allows developers to pick patterns from a palette and place them into the working space, then parameterize and connect them. It checks the syntactical validity of the composition and generates executable code for a chosen platform. The target is either Business Process Execution Language (BPEL) (Druckenmüller, 2007) or a configuration language for specific integration tools: Apache ServiceMix (Mierzwa, 2008) or Apache Camel (Kolb, 2008). Service Component Architecture environment is supported as well, using BPEL as a tool (Scheibler, Mietzner, and Leymann, 2009). 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 (Scheibler, Mietzner, and Leymann, 2008).

### 1.5.4 Guaraná language

Frantz, Corchuelo, and Gonzáles (2008) have proposed Guaraná, a modeling language for EAI based on entities that are very similar to enterprise integration patterns. The principle of their work is comparable to the one described above, with the following differences:

1. Models in Guaraná are more structured than models based on traditional EIPs. Basic executable entities in Guaraná, named tasks, can be simple or composite, allowing decomposition of complex integration processes into easily understandable parts. Simple tasks correspond roughly to integration patterns. Moreover, tasks are encapsulated into building blocks with well defined interfaces (ports), connected by explicit integration links.

2. Although tasks correspond to integration patterns, these two are not exactly the same. For illustration, we list simple task types in Table 2, along with enterprise integration patterns that we have found to be the closest ones for particular tasks. (The „-“ symbol means that we have not found a correspoding pattern).

**Table 2.** A list of Guaraná simple tasks.

| **Task type** | **Task** | **Corresponding EIP** |
|---|---|---|
| Message constructors | Aggregator | Aggregator |
| | Splitter | Splitter |
| | Custom task | - |
| Transformers | Content enricher | Content Enricher |
| | Slimmer | Content Filter |
| | Translator | Message Translator |
| Routers | Filter | Message Filter |
| | Replicator | Recipient List |
| | Distributor | Recipient List |
| | Merger | Aggregator |
| | Synchronizer | - |
| Timing | Timer | - |
| | Delayer | - |
| Interfacing | Database | Channel Adapter |
| | Gateway | Channel Adapter |
| | Channel | Channel Adapter |
| | File | Channel Adapter |
| | Scrapper | Channel Adapter |

In other aspects this approach is similar to the one of Scheibler and Leymann (2008). For example, the overall process is exactly the same: A developer creates an integration solution, based on components listed in Table 2. This process is supported by a graphical editing environment. After choosing a target platform, the code for it is generated.

Currently authors claim a support for Microsoft Workflow Foundation as a target platform (Sleiman, Sultán, and Frantz, 2009) with some limitations due to conceptual mismatches between the world of messaging-based integration and workflow automation. However, their approach is independent on a 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 (Frantz, 2011).

What distinguishes works of Frantz et al. from the others in this area is a special interest in exception handling. As described in (Frantz, Corchuelo, and Molina-Jimenez, 2009), they deal with failures using a special component, named monitor, that receives notifications on failures and reacts to them in a way specified in a declarative language based on ECA (Event-Condition-Action) rules.

### 1.5.5 A critique of approaches based on integration patterns

Works of Scheibler et al. and Frantz et al. 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, these works 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. In (Mederly, 2009a) we have shown this fact on a case study. We try to address these shortcomings in our work presented in this dissertation.

### 1.5.6 Integration Designer Assistant

Generally speaking, we are looking for approaches that would allow the developer to specify solution at a higher level of abstraction, strictly separating business and technical aspects, with transformation to lower levels of abstraction being as automated as possible.

An interesting work aimed towards such a lifting of the level of abstraction has been performed by Umapathy and Purao (2007, 2008). 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.

Umapathy and Purao therefore have devised a system (called Integration Designer Assistant, or IDAssist, existing in the form of a research prototype) based on Speech Act theory that assists the designer with the mapping from models in BPMN to sets of EIPs. The tool allows a designer to depict an integration solution in the form of a BPMN diagram showing a graph of tasks, with each task annotated by one of 11 so called Action Types (these types are e.g. Request for Information, Provide Information, Invocation, and Accept/Reject with/without sending receipt). Users are then being offered suitable EIPs based on the diagram structure as well as on individual task Action Types. In order to do this, the tool uses an inference engine backed by an ontology. A weak point of this method is that the abstract specification of the solution (i.e. the annotated BPMN diagrams) captures very little information, so the proposals provided to the developer are of varying relevance and the model itself cannot be used to generate directly executable solutions.

### 1.5.7 Other model-driven integration approaches

Concerning other attempts in the area of model-driven integration, Al Mosawi, Zhao, and Macaulay (2006) proposed a general idea of platform-independent specification of integration solution, modeling it at five levels: (1) collaboration between enterprises, (2) collaboration within an enterprise, (3) services provided by individual systems, (4) supporting services, and (5) technology-specific model. Induruwana (2005) describes the idea of aspect-oriented approach to modeling of EAI solutions. Unfortunately none of these authors have provided detailed information on their work, and we have not found any follow-on work on this topic by them.

There are also some vendors, e.g. E2E Technologies Ltd. (E2E Technologies, 2010) claiming they have products implementing model-driven approach to application integration. Actually what they provide is an integration engine with a UML-based configuration language. In contrast to them, we aim to a platform-independent approach that is able to generate integration solutions targeted to many integration platforms.

Only very recently, after writing a draft of this dissertation, we have become familiar with the BIZYCLE integration process (Milanovic, Cartsburg, Kutsche, Widiker, and Kschonsak, 2009), (Agt, Bauhoff, Cartsburg, Kumpe, Kutsche, and Milanovic, 2009). It is a result of research project whose goal was to investigate the potential of model-based software and data integration methodologies.

BIZYCLE process works with models at various levels, roughly in the following order:

1. *Computation-independent model (CIM)* of the integration solution: reflects functional requirements that the solution has to fulfill. A major part of this kind of model is a diagram conceptually similar to UML activity diagram that shows control and data flows within the solution. Then there is a data model

showing structure of business objects and two more types of models that relate business objects to business functions and connectors, respectively.

2. *Platform-specific models (PSMs)* of systems that are to be integrated: these models reflect interfaces of systems at the technical level. Currently there are metamodels created for various platforms – relational databases, XML files, web services, Java Platform, Enterprise Edition, .Net components and selected ERP (Enterprise Resource Planning) systems. The idea is that interfaces of participating systems are modeled using these metamodels (or, even better, the description of interfaces is imported from a dictionary provided by systems' execution platforms) and automatically converted to a platform-independent form.

3. *Platform-independent model (PIM)* contains the description of systems that are to be integrated, but this time at a higher level of abstraction. Main reason for existence of this model is to enable conflict analysis, as described below.

Conflict analysis is a key activity in the course of creating an integration solution in the BIZYCLE process. It analyzes PIM in order to discover mismatches between component interfaces, at the semantic, behavior, property, communication and structural levels. This analysis is semi-automated; there are situations where it requires user interaction. An output of conflict analysis, along with the above mentioned models (CIM, PSMs, and PIM) is used to generate code for the integration solution. Generated code is executed in BIZYCLE Runtime Environment, based on Glassfish OpenESB product.

In comparison to our work, BIZYCLE process is much more comprehensive: it helps the integration developer at multiple levels, ranging from the questions of incompatible semantics of data down to the level of technical interoperability. Results of the research are being offered also in the commercial form (Model Labs, 2011), indicating their relevance for real integration projects. 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.

## *1.6 Model-driven service composition*

Service composition deals with creation of more complex (composite) services out of simpler (elementary) ones. It has become an important research topic in last few years: the goal is to reduce human effort needed to develop and maintain such composite services.

When speaking about service composition, services based on web services technology (i.e. SOAP and WSDL) are usually meant.

What is the connection between service composition and application integration? Application integration deals with making systems (applications) to interoperate. Service composition is concerned with creating composite services out of simpler ones, i.e. with the interoperation of services. Applications are often made available to integration using wrapping services (or adapters); their integration can be directly seen as a composition of their wrapping services. The only technical difference is in technologies used: while service composition is based almost exclusively on XML, HTTP, SOAP and WSDL (at least in the academic sphere), application integration uses various message formats and transport protocols (very often asynchronous messaging as described in Section 1.2.4). This is very important in our case, because we are interested exactly in solving these technical problems. Nevertheless, let us take a look at model-driven service composition, especially how the technical aspects are dealt with here.

Service composition can be implemented in various languages. In theory, any implementation language can be used, providing it has adequate support for calling individual services. During last few years, BPEL (Business Process Execution Language) has been established as the de facto standard in this area, given its ability to describe the composition without the need to specify too much implementation aspects. The execution of composite services implemented in BPEL is done by BPEL servers (or BPEL engines), implementing the Process Manager architectural pattern mentioned in Section 1.2.5. Other languages for composite service description are e.g. OWL-S (Web Ontology Language for Services) for semantic web services, Web Components, π-calculus, Petri Nets and Finite State Machines (Milanovic and Malek, 2004). Principles of many of these languages are very similar to principles of existing workflow languages (van der Aalst, Dumas, and ter Hofstede, 2003).

*Model-driven service composition* applies an idea of model-driven development in the area of service composition. Methods of this type generally start with a platform-independent model of the composition (created frequently in a UML-based language) and through a sequence of generation and/or refinement steps they go to platform-specific models and to an executable code.

Let us take as a representative example a recent work done by Mayer, Schroeder, and Koch (2008). It is devoted to generating orchestration code (currently in BPEL, Java, or formal language Jolie) on the basis of an abstract model written in UML4SOA. UML4SOA is a conservative extension of UML 2.0, developed with the goal of achieving minimalism, conciseness and a comfort for the developer, adding features like scopes and compensations. The transformation itself is done in two steps: in the first one the UML4SOA graph-based model is converted to an intermediate structure-oriented form (called Intermediate Orchestration Model, or IOM). Here a rule-based approach is used in order to infer how decision and merge elements in UML models should be transformed into structured concepts (like branches and loops) in IOM. IOM is then translated into platform-specific models and eventually into code. As for BPEL, along with the code, the interfaces to partner services are generated.

A number of similar works are described in a survey done by Rauf, Iqbal, and Malik (2008). What they have in common is the fact that the service composition modeling is done using UML (typically using activity or state chart diagrams complemented by class diagrams) and then transformed into an executable language, typically BPEL. Some of the works are very straightforward, using specially created UML profiles for modeling BPEL processes; the more interesting ones are those that enable transformation to various executable languages. An example of such works is (Skogan, Grønmo, and Solheim, 2004). These works differ also in the modeling constructs supported (e.g. are scopes and compensations available to the developer?), in the approach to code generation (e.g. does the method aim to generate a "nice", readable code, or just any working code?), and in the scalability of the method. For example Koehler, Hauser, Sendall, and Wahler (2005) use techniques known from compiler theory in order to partition large processes into subprocesses (to be processed more effectively) and to detect unstructured cycles and to transform them without exponential expansion of the resulting program.

These approaches, in general, aim to provide a concise modeling language for the developer and then use more or less sophisticated transformational algorithms to generate platform-specific models and/or executable code. They do not provide "intelligence" to free developer from specifying e.g. adaptation components that have to be included in the composite service (if any) – something that is crucial in the case of application integration. Generally they also do not deal with technical or quality of service (QoS) issues, forcing the developer to solve these issues "by hand", and, moreover, intermixed with essential (business) aspects of the service composition.

There were some attempts to separate technical and business aspects of service composition, however. Let us mention some of them here.

An early attempt to include *transactional aspects* and treat them separately from business aspects is provided by Schmit and Dustdar (2005). The authors have created a UML profile for modeling transactional properties of service composition. They model basic service composition using a UML state chart diagram; transactional aspects of this composition are included in a separate layer, modeled as UML class diagram. Each transaction is modeled as a class, with nested transactions as subclasses. Attributes of this class correspond to services included in the transaction, while operations (namely, constructors and destructors) correspond to transitions in the basic state chart diagram during which the transaction should start and end, respectively. The transaction has associated tagged values (like a flag whether it can be compensated, or maximum time it can be active) and stereotypes (indicating e.g. whether it is Atomic Transaction or Business Activity). Authors provided a prototype doing some rudimentary code generation, although the artifact generated is not a running code, just an example of WS-BusinessActivity SOAP message header (a coordination context). The approach seems to be promising; unfortunately we have not found any follow-on work giving more concrete results.

There are also a couple of UML profiles designed for specifying *non-functional properties* of services, e.g. the one described by Wada, Suzuki, and Oba (2006). Their

UML profile allows specifying the required properties of connectors connecting the services: delivery assurance (unspecified, at most once, at least once, exactly once; and whether the message order has to be preserved), maximum allowable message delivery time, transmission channel parameters (e.g. how to handle situations when underlying message buffers overflow), and filtering actions that have to be applied on messages flowing through that connector. Other entities that can be modeled are services themselves, messages, and message exchanges (though only the simple "request-response" ones). The authors provide a tool to map models created using their UML profile into specific technologies, namely Mule ESB – an open-source enterprise service bus implementation. Their approach concentrates on generating appropriate message delivery code (e.g. choosing suitable transport mechanism and configuring it) for applications that need to communicate; however, the generated code seems to be not directly usable for service orchestration. Comparing to our vision, this language forces the developer to specify non-functional properties at a low level of abstraction; we would like to generate these from a more abstract description automatically.

*An aspect-oriented approach to specifying and/or developing service composition* is relatively frequently present. Among first attempts to use aspects in service composition are those of Charfi and Mezini (2004, 2005, 2005a) aimed to separate various crosscutting concerns from the basic workflow/composition code. These concerns are e.g. volatile business rules and technical aspects like transactions, security, reliability and persistence. The authors have created an aspect-oriented variant of BPEL, called AO4BPEL and its implementation. The work is summarized in Charfi's dissertation (2006). Aspect-oriented service composition is dealt with also by Courbis and Finkelstein (2005), Schmidmeier (2007), and Xu, Tang, Xu, and Tang (2007). In our case, these ideas seem to be useful in the last phase of integration solution construction, namely when generating the executable code.

## 1.7 Enterprise application integration – summary

In this chapter we have described an area of enterprise application integration. We have shortly characterized messaging-based integration solutions and identified the possibility of making their creation easier as the main goal of this dissertation.

Surveying the research results available we have found no comprehensive approach to creation of messaging-based application integration solutions that would allow the developer to separate business and technical aspects of the solution, and then to automatically or semiautomatically solve these technical aspects.

We therefore plan to create such an approach, as described in the next chapter.

# 2 Dissertation goal and hypotheses

Given the situation described in Chapter 1, 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.

# 3 Methods and tools used

In this chapter we describe major methods and tools that are used in this dissertation.

## *3.1 Model-driven software development*

A general approach we use is the *model-driven software development* (Schmidt, 2006), (Mellor, Scott, Uhl, and Weise, 2004). The idea of this approach is that the software system is developed using *models* – more or less abstract representations of the problem and its software solution.

The development of software systems using this approach starts with creating a set of abstract models, comprising a high-level description of the software system being constructed. Then it continues by stepwise transforming or refining these models into lower-level ones and eventually into executable code.

Modeling languages can be graphical or textual ones. They can be based on industry standards like UML (Unified Modeling Language), possibly customized e.g. using UML profiles, or they can be created specially for the particular domain. In the latter case they are usually called *domain-specific modeling languages* (DSMLs).

In this work we use our own, textual domain-specific modeling languages. However, for the sake of understandability, we show examples of methods' inputs in Chapters 4 to 6 using two well-known graphical modeling languages: Business Process Model and Notation (BPMN) and Unified Modeling Language (UML) activity diagrams.

**Business Process Model and Notation (BPMN)**

BPMN is devised as a standard means for describing business processes by and for human users (Object Management Group, 2011). It provides five categories of modeling elements: flow objects, data, connecting objects, swimlanes and artifacts. However, we use only a small subset of this rich language, utilizing the following kinds of flow objects:

1. events:
    a. *start Message event* meaning that a message is to be received, and
    b. *end Message event* meaning that a message is to be sent;
2. *tasks*: atomic activities that have to be carried out,
3. gateways: used to influence the control flow:
    a. *exclusive gateway* that either splits the control flow based on a specified condition, or merges back multiple existing alternative paths,
    b. *parallel gateway* that either creates parallel paths or joins back multiple existing parallel paths,

c. *complex gateway* that is used to model complex synchronization behavior. It can create and merge back both alternative and parallel paths.

Symbols for these elements are shown in Figure 6.

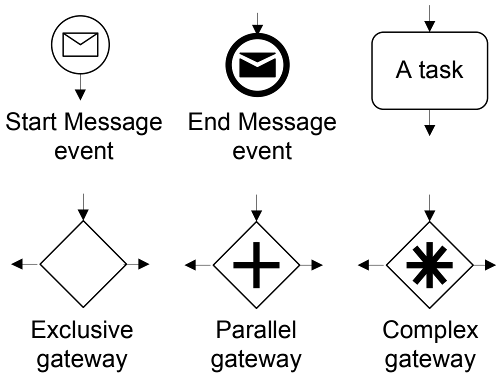


**Figure 6.** Symbols for BPMN elements used in this dissertation.

## UML Activity diagrams

Unified Modeling Language is a language primarily intended for system architects, software engineers, and software developers (Object Management Group, 2010). It provides several kinds of diagrams, from which we have chosen activity diagrams as a tool for visualizing abstract design of integration solutions. We have chosen this tool for solutions with explicit data input and output parameters for individual components and with the for-each construct, as we consider the UML notation very well suited for this purpose.

We use the following constructs in our diagrams:

1. *activities*: atomic activities that have to be carried out (like tasks in BPMN),
2. *decision* and *merge* nodes: split the flow to multiple alternative paths and merge them later back together (like exclusive gateways in BPMN),
3. *fork* and *join* nodes: fork the flow to multiple parallel paths and join them later back (like parallel gateways in BPMN),
4. *expansion region*: used to model for-each construct, i.e. an execution of a subprocess once for each element of an input collection,
5. *input* and *output pins*: used to denote input and output parameters of individual activities,
6. *start* and *final nodes*: used to model the start and end of process.

Symbols for these elements are shown in Figure 7.

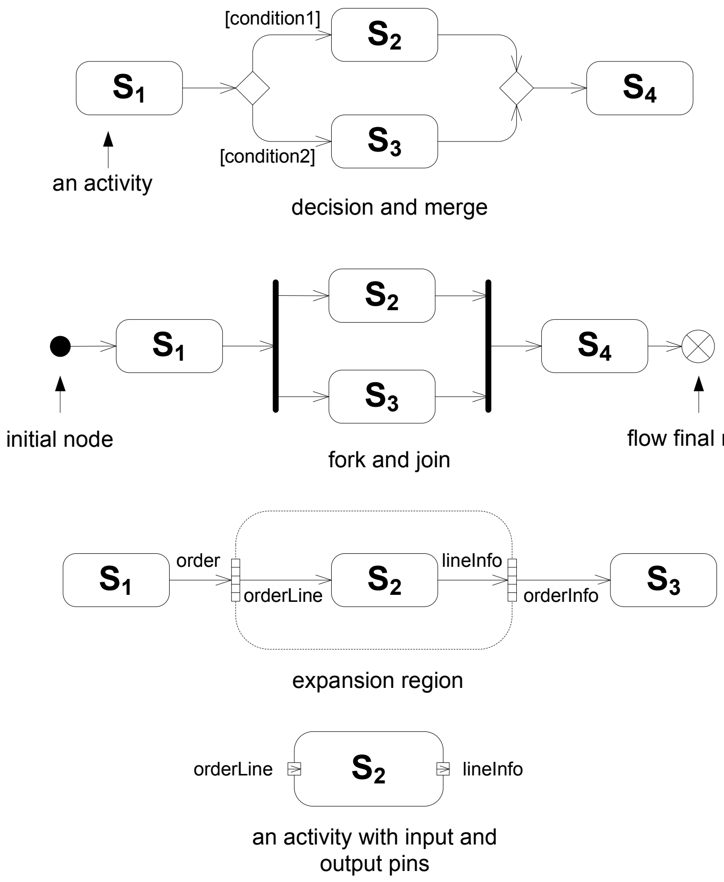


**Figure 7.** Symbols for UML activity diagram elements used in this dissertation.

## 3.2 Planning

Generally speaking, planning is an approach to problem solving whose aim is to produce a course of actions that takes a system from an initial state to a goal state (Schalkoff, 1990).

Current action-based (or STRIPS-like) planners, as descendants of the automated planner STRIPS (Fikes and Nilsson, 1971), are based on the situation calculus. States of the world (situations) are described as conjunctions of grounded first-order predicate formulas; these formulas are positive literals (atoms).

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 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.

What we have just described is the basic STRIPS-like planning. There are several extensions to this approach that we use in some of our methods. For example, an extension used in the DL/P method allows us to work with quantified preconditions and universally quantified and conditional effects.

A planning problem consists of a planning domain (a set of operators) and a definition of the initial state of the world 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 correspond to real-world actions and are usually required to have all their variables bounded. Although in most cases the plan is a sequence of actions, it is also possible to create plans with concurrent actions.

A frequently used algorithm of action-based planning works by sequential adding of operators to the plan. Plan construction is guided by operators' preconditions and effects, usually employing some kind of heuristics. (There are other methods as well, for example using a planning graph, plan space search, converting planning to a constraint satisfaction problem, and so on.) In our methods, 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 (Russel and Norvig, 2003).

The planning as a general paradigm can be used in two ways: One could either create a custom planner for his problem, or he or she can use an existing, domain-independent planner. We have chosen the second option.

Actually, we are not alone when using this approach. There have been several successful attempts to translate domain-specific problems into an input for a domain-independent planner. As an example, this approach has been promoted by organizers of ICKEPS 2009 (International Competition on Knowledge Engineering for Planning and Scheduling). Participating applications were from the domains of data mining, e-learning, business workflows (a question of resource allocation), semantic web service composition, and instructable computing. More information on this event can be found in (Bertoli, Botea, and Fratini, 2009). Other examples of using domain-independent planners for specific domains are test cases generation (Scheetz, von Mayrhauser, and France, 1999; Fröhlich and Link, 2000) or deployment of components of distributed software systems (Arshad, Heimbigner and Wolf, 2007).

### Planning Domain Definition Language (PDDL)

PDDL is a de facto standard input language for domain-independent action-based planners. It provides facilities for description of a planning *domain* and a planning *problem*. The description of a planning domain consists primarily of information on object types, predicates and operators (actions), while the planning problem is described by listing concrete objects, the initial state and a goal state or states. For simplicity, in this dissertation we use the term "planning problem" to describe both a problem and its associated domain.

Let us explain the PDDL syntax used for operator description. For example, in a domain of physical objects, an operator `MoveBriefcase` with two parameters (`from`, `to`) can be described as follows – adapted from (McDermott et al., 1998):

```
(:action MoveBriefcase
 :parameters (?from ?to – location)
 :precondition
  (and
    (at Briefcase ?from)
    (not (= ?from ?to))
  )
 :effect
  (and
    (at Briefcase ?to)
    (not (at Briefcase ?from))
    (forall (?thing)
            (when (in-briefcase ?thing)
                  (and (at ?thing ?to)
                       (not (at ?thing ?from))
                  )
            )
    )
  )
)
```

The description says that an operator `MoveBriefcase` has two parameters: `?from` and `?to`. Note that parameters in PDDL are marked by having a question mark as a prefix. In our example, these two parameters are of type `location`.

Then there comes a specification of the operator's precondition. In this case the briefcase can be moved from location `?from` to location `?to` only if it currently really *is* at the location `?from`. The fact of being physically present at some place is, in this example domain, modeled by predicate `at` having two arguments: a thing and a location. The fact that the Briefcase object is at the location `?from` is then written as `(at Briefcase ?from)`. Please note that PDDL uses a lisp-like way of writing expressions, including predicates, i.e. `(predicate argument1 argument2 ... argumentN)` instead of the more traditional form `predicate (argument1, argument2, ..., argumentN)`.

Second part of the precondition, i.e. `(not (= ?from ?to))`, denotes the requirement that the locations `?from` and `?to` must be distinct.

Finally, operator's effect is specified. Here the effect is:

1. The briefcase is present at new location – `(at Briefcase ?to)` – i.e., this predicate will be added to the state of the world.

2. The briefcase is no longer at original location – `(not (at Briefcase ?from))` – i.e., the predicate `(at Briefcase ?from)` will be removed from the state of the world.

3. Every thing that is present in the briefcase (modeled by predicate `in-briefcase`) is moved along with it: it is present at the new location

`(at ?thing ?to)` and it disappears from the original one `(not (at ?thing ?from))`. Appropriate atoms (made by replacing `?thing`, `?from`, `?to` with concrete objects) will be added to, or removed from, the state of the world.

The third item is an example of a universally quantified effect – one of extensions of basic STRIPS-like core of PDDL.

For more information on PDDL and its extensions please see (McDermott et al., 1998) and (Gerevini and Long, 2005).

**Tools used**

In this dissertation we have used the following planners:

- HSP 2.0 – a planner that combines several heuristic search algorithms based on an A* algorithm with a weight assigned to the heuristic part of the evaluation function. It provides both admissible and non-admissible heuristic functions; we have used it with a non-admissible one that does not guarantee finding optimal solution, yet, in general, it comes to a solution faster (Bonet and Geffner, 2001).

- FF 2.3 – implements a search strategy that combines hill-climbing with systematic search. Uses a non-admissible heuristic based on estimating goal distances by ignoring delete lists, a principle similar to the one used in HSP (Hoffmann and Nebel, 2001).

- Gamer – a sequential optimal planner that uses symbolic search planning with binary decision diagrams (Edelkamp and Kissmann, 2009).

- MIPS-XXL – a sequential optimal planner using an external memory to cope with large state-space that has to be searched (Edelkamp and Jabbar, 2008).

- LPG 1.2 – a planner that uses a stochastic local search procedure, supporting durative actions and numerical variables (Gerevini, Saetti, and Serina, 2003).

- SatPlan2006 – a parallel planner that works by translating a planning problem into a Boolean satisfiability problem (SAT), which is then solved using a general solver (Kautz and Selman, 2006).

- MaxPlan – translates a planning problem into satisfiability one, as SatPlan2006 does. In comparison to SatPlan2006 it provides several optimizations, e.g. instead of solving SAT problem as a whole, MaxPlan decomposes it into a series of smaller SAT subproblems, using knowledge of the structure of the original planning problem (Xing, Chen, and Zhang, 2006).

- JSHOP2 – a Java implementation of the SHOP2 planner. In contrast to action-based planners mentioned above, this one is based on a hierarchical task network (HTN) planning. Instead of trying to reach a specified goal state, HTN planners try to find a sequence of actions that allow accomplishing a specified task (Nau, Au, Ilghami, Kuter, Murdock, Wu, and Yaman, 2003). We use this planner in a mode that emulates action-based planning, as described in Chapter 5.

## *3.3 Constraint programming*

Constraint programming is an approach to problem solving that is based on formulating and solving constraint satisfaction problems (CSPs).[2] Each such problem consists of a set of variables { $x_1$, $x_2$, ..., $x_n$ } with their respective domains { $d_1$, $d_2$, ..., $d_n$ }. Each variable can be assigned a value from its domain $d_i$. Along with these two sets there is a set of constraints { $c_1$, $c_2$, ..., $c_k$ } over these variables. The constraints can be of arbitrary arity greater than zero. The process of solving a CSP means finding an assignment of a value to each of the variables, such that the assignment is consistent with all the constraints. Additionally, one of the variables can be declared to be the *cost variable* and then the overall goal is to find a solution that minimizes the value of this variable.

In the following we will briefly look at the following two approaches to solving CSPs: consistency checking and searching for a solution (Mach and Paralič, 2000).

*Consistency checking* algorithms try to transform a CSP in order to reduce its complexity, while keeping the set of its solutions unchanged. These transformations eliminate unfeasible values from domains of CSP variables, strengthen constraints, or add new constraints. There are many concrete algorithms, differing primarily in the degree of consistency they are going to achieve. The definition of the k-consistency is the following (Mach and Paralič, 2000):

> Let arbitrary *k*-1 variables have assigned such values from their domains, so that all constraints defined over this (*k*-1)-tuple of variables are satisfied. For each other (*k*-th) variable it is then possible to choose such a value from its domain, so that all constraints defined for the resulting *k*-tuple of variables will be satisfied as well.

In practice, mostly used algorithms attempt to establish the consistency of degree 1 (often called node consistency) and 2 (arc consistency). Achieving higher-level consistency is possible as well, however, these algorithms are less frequently used due to their bigger computational complexity (Mach and Paralič, 2000).

*Searching* constructs a solution by successively assigning values to CSP variables. In that context, there are two crucial questions:

1. Which variable should the solver choose (among variables that have no value yet) to assign a value?

[2] In this dissertation, we will also use terms “constraint satisfaction” and “problem solving using constraint satisfaction” to denote this approach.

2. What value (from the set of potentially suitable ones) to use?

Concerning the first question, there are well-known general strategies. Some of them are (Kuchcinski and Szymanek, 2011):

1. *Smallest domain:* the solver selects a variable with the smallest domain.
2. *Largest domain:* the solver selects a variable with the largest domain.
3. *Smallest (largest) minimal (maximal) value:* the solver selects a variable with the smallest minimal (or smallest maximal, largest minimal, largest maximal) value of their domain.
4. *Max regret:* the solver selects a variable with the largest difference between two smallest values in a variable's domain.
5. *Most constrained:* the solver selects a variable that has the biggest number of constraints assigned to it. This criterion can be evaluated either statically, i.e. counting all constraints assigned (this can be evaluated at the beginning of solving process), or dynamically, i.e. counting only constraints that are awaiting evaluation.
6. *Minimal domain over degree:* the solver selects a variable that has the smallest ratio of domain size to number of attached constraints awaiting evaluation.
7. *Weighted degree:* the solver selects a variable that has the highest weight divided by its domain size. A variable weight is a value that starts at 1 and is increased every time a failure of a constraint related to this variable is encountered (Boussemart, Hemery, Lecoutre and Sais, 2004).

Concerning the second question, there are again several general strategies, like:

1. choosing the *lowest* (or *highest*) value from the variable's domain,
2. choosing the *middle* value from the variable's domain,
3. choosing a *random* value from the domain,
4. let the user specify the *exact ordering* of values to be chosen.

Any of these value-choosing strategies can be applied globally to all variables, or one can choose to apply different strategies for individual variables.

**Tools used**

In this dissertation we have used JaCoP (Szymanek, 2011), a constraint programming library for the Java environment. It is an open-source library that allows defining finite domain variables and constraints over them. JaCoP solves constraint satisfaction problems using a combination of consistency checking and depth-first search. During searching, it is possible to apply various strategies for variable and value selection, as described above. Moreover, JaCoP allows modifying the search using so called plugins – a custom code that is executed at specified points of the search process. Plugins can be attached e.g. to the events of initializing a search, exiting a search, finding a solution, exiting a search subtree, evaluation of consistency during a search, and others.

We are considering using other solvers as well. However, in comparison to the area of action-based planning that has a well established standard input language (PDDL), here the level of portability between solvers is much lower. Only very recently a standard has been proposed, namely, the MiniZinc language (Nethercote et al., 2007).

## *3.4 Concrete software tools used*

Prototypes of our methods have been implemented in the Java programming language, using Java SDK (Software Development Kit) 1.5 and 1.6 with the Eclipse development environment in versions from 3.4 to 3.6. Other major software tools and libraries include (besides those that have been already mentioned in this chapter):

- Java tools and libraries: Xtext 1.0, JAXB 2.1, JGraphT 0.8, JUNG 2.0, Apache Velocity 1.6,
- integration platforms: Progress Sonic ESB 7.6.2 and 8.0.1, Apache Camel 2.5,
- visualization toolkit: graphviz 2.26.

## 4 Our approach: A general description

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. All of them share an approach shown in Figure 8.

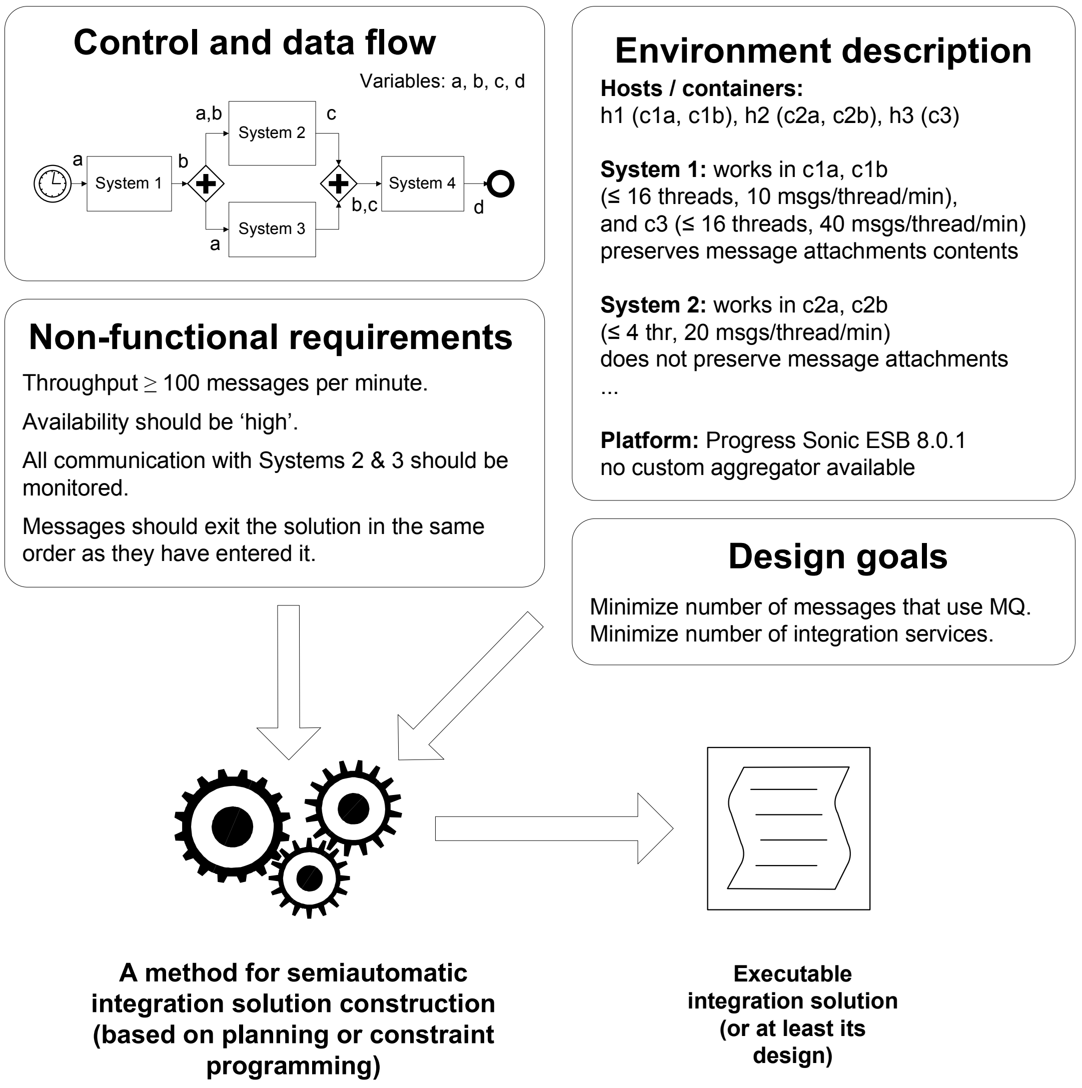

Control and data flow
Variables: a, b, c, d
a
System 1
b
a,b
System 2
c
System 3
a
b,c
System 4
d
Non-functional requirements
Throughput ≥ 100 messages per minute.
Availability should be 'high'.
All communication with Systems 2 & 3 should be monitored.
Messages should exit the solution in the same order as they have entered it.
Environment description
Hosts / containers:
h1 (c1a, c1b), h2 (c2a, c2b), h3 (c3)
System 1: works in c1a, c1b
(≤ 16 threads, 10 msgs/thread/min),
and c3 (≤ 16 threads, 40 msgs/thread/min)
preserves message attachments contents
System 2: works in c2a, c2b
(≤ 4 thr, 20 msgs/thread/min)
does not preserve message attachments
...
Platform: Progress Sonic ESB 8.0.1
no custom aggregator available
Design goals
Minimize number of messages that use MQ.
Minimize number of integration services.
A method for semiautomatic integration solution construction (based on planning or constraint programming)
Executable integration solution (or at least its design)

**Figure 8.** A schema of our approach.

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:

- *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. It was presented at CEE-SET 2009 (Mederly, Lekavý, Závodský and Návrat, 2009).

- *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 and it was presented at IIT.SRC 2010 (Mederly, 2010).

- *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. It was presented at ADBIS 2010 (Mederly and Návrat, 2010).

- *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. Its preliminary versions were presented at DATAKON 2010 (Mederly and Návrat, 2010a) and IIT.SRC 2011 (Mederly, 2011); the most current one will be presented at DATAKON 2011 (Mederly and Návrat, 2011).

In the remaining parts of this chapter we describe common features of these methods. Then in Chapter 5 we explain planning-based methods in more detail and in Chapter 6 we show methods based on constraint programming.

## 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

**Required flow of control and data**

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 mainly use Business Process Modeling Notation (BPMN) that is often utilized as a platform-independent way of modeling control and data flow between activities, carried out either by people or by computers.

As an example integration scenario used to illustrate the methods let us consider a hypothetical online retailer company "Widgets and Gadgets 'R Us" (Hohpe and Woolf, 2004). 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 Figure 9).

Handling of purchase orders in the integration solution should look like this: Orders are being placed by customers through three systems: web interface, call center and fax gateway. These systems are connected to our integration solution via adapters that send each received order in a separate message to a channel dedicated to each of these three systems. Our integration solution is responsible for picking up such a message and translating it from source system-specific data model to a common data model. After that, it ensures 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 see if the items ordered are widgets, gadgets, or something else.[3] 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.

[3] In this version of the scenario we assume that an order contains items of only one type. A generalized version of the scenario, more similar to the one presented by Hohpe and Woolf, is described in Section 6.1.1.

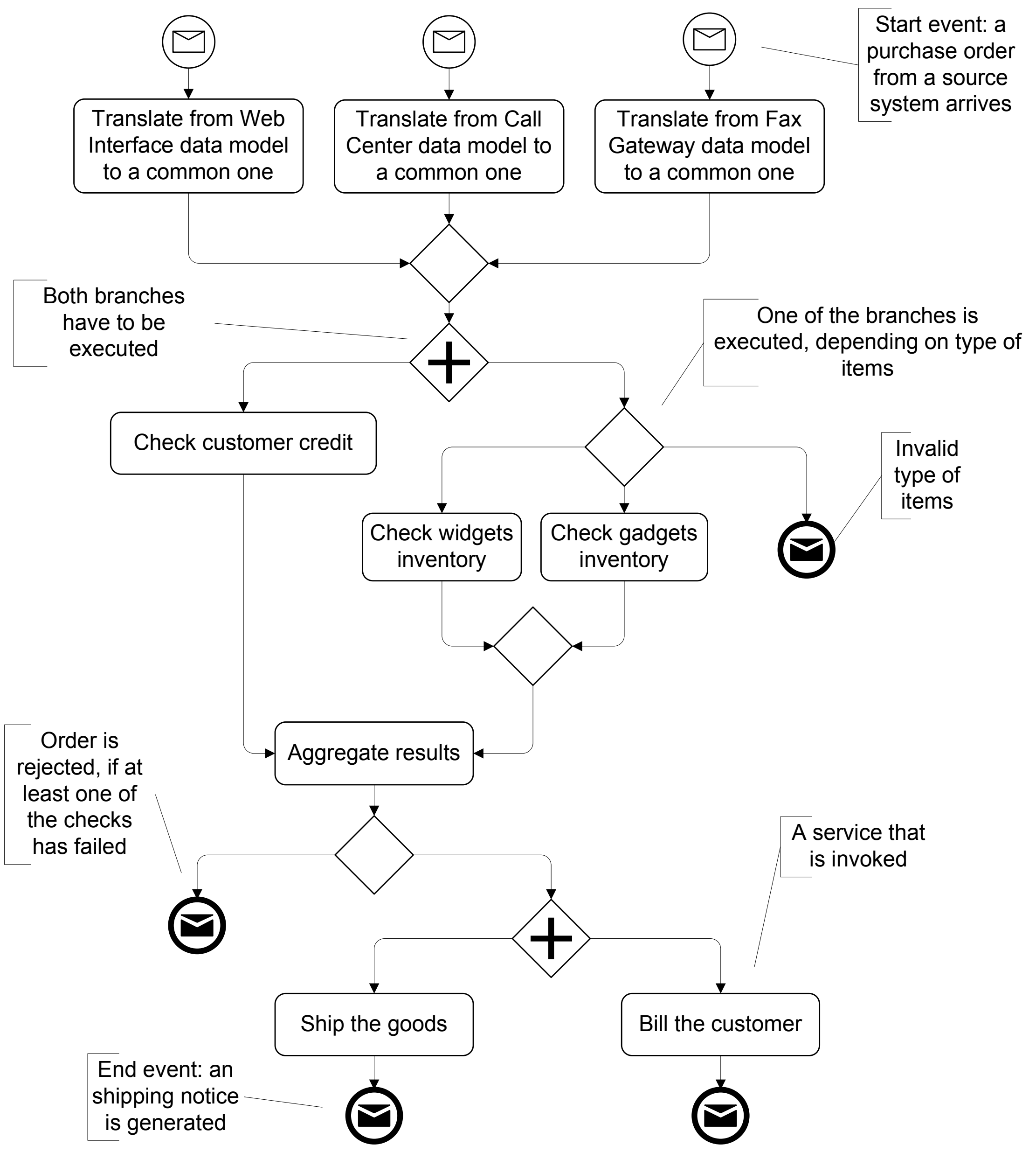

Start event: a purchase order from a source system arrives
Translate from Web Interface data model to a common one
Translate from Call Center data model to a common one
Translate from Fax Gateway data model to a common one
Both branches have to be executed
One of the branches is executed, depending on type of items
Check customer credit
Invalid type of items
Check widgets inventory
Check gadgets inventory
Order is rejected, if at least one of the checks has failed
Aggregate results
A service that is invoked
Ship the goods
Bill the customer
End event: an shipping notice is generated

**Figure 9.** An example of the flow of data to be implemented, using BPMN.

So, in Figure 9 we see an example of a control flow specification (shown using connections between model elements) as well as a data flow specification (shown using the same connections). More detailed explanation follows.

## Business and integration services

In the BPMN representation shown in Figure 9, rectangles with rounded corners correspond to business and integration services invocations. *Business services* provide business functionality; usually that means they convey access to systems that have to be integrated (in our case these services are e.g. CheckCredit, CheckWidgetInventory, and so on).[4] On the other hand, *integration services* implement technical functionality

[4] So, when we are speaking that we are connecting *systems (applications)*, we (most often) connect *business services* that provide access to these systems' functionality.

that is necessary for an integration to take place: for example, they convert messages from one format or data model into another; they log messages, route or reorder them, and so on. In our case, services for translation of orders from system-specific data model to a common one could be considered to be integration ones. Further integration services are added to the solution during the design process, as can be seen e.g. in Figure 10.

The border between business and integration services is not always clear. In particular, routing and transformation services could deal with business logic, integration logic, or both – so they can be considered to be business or integration services, depending on the situation and the viewpoint of the observer. Actually, this explanation only shifts the question of distinction between business and integration services – in case of transformation and routing ones – down to the question: "What is the difference between business and integration logic?" By business logic we mean a functionality that is specified by business users, or, at least, that has to be consulted with them. For example, a rule stating that "a student is considered to be enrolled to a faculty if his study record contains a start date, does not contain an end date, or the end date is greater than the current date, and has a special confirmation flag set" is a typical example of a business logic rule. On the other hand, simple transformations dealing with e.g. renaming the attributes (like Name → FirstName), changing encoding of values (like male/female → M/F or 0/1), and so on, are characteristic examples of integration logic.

The creation of business and integration services is currently out of scope of our methods: our task is to select, configure and connect them appropriately into a working integration solution.

## Control and data dependencies

We distinguish between two kinds of relations between services and other components[5] of the integration solution: control and data dependencies.

A *control dependency* between two components $C_1$ and $C_2$ means that $C_1$ has to be executed before $C_2$. In the diagram shown in Figure 9 such dependencies are denoted by connections (directed edges) between components (graph vertices). Of course, we show only "primary" dependencies, not those that can be derived from them using a transitive closure.

[5] By these "other components" we mean components for receiving and sending messages from and to external channels. These are shown as circles with envelopes that denote a receipt of a message and a sending of a message out, i.e. the BPMN Message Start Event, shown as an envelope not filled in, and the BPMN Message End Event, shown as a filled-in envelope, respectively.

Besides connections, the control flow is described also using diamonds. Diamonds denote BPMN gateways and mean the following (see also Section 3.1):

1. when drawn with a ‘+’ sign (i.e. the BPMN Parallel Gateway), a flow of control is split into two or more *parallel* flows, or more parallel flows are joined into one in a synchronized way,

2. when drawn without a sign (i.e. the BPMN Exclusive Gateway), a flow of control is partitioned into two or more *alternative* flows, or more alternative flows are merged back into one.

A *data dependency* between two components $C_1$ and $C_2$ means that $C_1$ produces a piece of data that $C_2$ needs.

Data dependency implies control dependency: if $C_1$ produces a piece of data that $C_2$ needs, then $C_1$ has to be executed before $C_2$, i.e. in the control flow graph there should be a path from $C_1$ to $C_2$.

In Pipes and Filters architecture (see Section 1.2.5), passing of control and data is implemented by a common facility: sending a message. Therefore, from the designer’s point of view, the simplest situation is when control and data dependencies are the same, i.e. when for each “primary” control dependency between components $C_1$ and $C_2$ the output of $C_1$ exactly matches the input of $C_2$. Both of these control and data dependencies can then be implemented simply by passing the output message from $C_1$ to the input of $C_2$.

In ML methods we have made exactly this assumption. Please note that in Figure 9 the control flow dependencies between components can be interpreted as data dependencies in general, and message flows in particular. On the other hand, DL/P and U/CP methods are more flexible in this respect, as they allow decoupling of control and data dependencies, as described (for the U/CP method) in Section 6.1.1.

As a terminological note, by *control* or *data flow specification* we mean the sum of all control or data dependencies between integration solution components, respectively, prescribed in the requirements specification.

By a *message flow* we mean the flow of messages at a particular point of the solution. A message flow is carried in a unique message *channel*, which could be an in-memory or a messaging middleware-based one. In our methods we use the Datatype Channel pattern[6] (Hohpe and Woolf, 2004) that requires the all messages in a channel (and, in our case, messages in a flow as well) to be of the same type. (The only exception to this rule is the situation when there are multiple message flows going into an aggregator service that has only one input channel. In this case there are more

[6] Hohpe and Woolf’s patterns are referenced here by using names with capitalized words.

message flows in one physical channel, making it not compliant with the Datatype Channel pattern.)

## Specification of control and data flow in our methods

Individual methods slightly differ in the way of specification of control and data flow that they expect.

Planning-based methods (ML/P, DL/P) work with *flow of control implicitly specified by the data dependencies*: each service is described by its inputs and outputs and the method tries to find a structure (more specifically, a directed acyclic graph) of services implementing the required transformation of input message flow(s) to output one(s). Therefore, before using these methods, the required flow of data such as the one described in Figure 9 has to be translated into an input/output characterization of the services. (This process is straightforward and can be easily automated, if necessary.) An example of such a characterization is shown in Table 3. It should be noted that implicitly specified control flow is more general than explicitly specified one used in later methods: it allows capturing e.g. alternative ways of achieving a business goal. It is perfectly possible to start with this kind of specification instead of the BPMN-like specification of control flow.

**Table 3.** Input/output characterization of services used in the example integration scenario.

| Service | Input | Output |
|---|---|---|
| `WebOrderTranslator` | `OrderWebNative` | `OrderWeb` |
| `CCOrderTranslator` | `OrderCcNative` | `OrderCc` |
| `FaxOrderTranslator` | `OrderFaxNative` | `OrderFax` |
| `DeclareMergedFlow_Orders`[7] | `OrderWeb`<br>`OrderCc`<br>`OrderFax` | `Order` |
| `CheckCredit` | `Order` | `OrderWithCreditInfo` |
| `ItemTypeRouter` | `Order` | `OrderWidgets`<br>`OrderGadgets`<br>`OrderInvalidType` |
| `CheckWidgetInventory` | `OrderWidgets` | `OrderWidgetsWithInvInfo` |
| `CheckGadgetInventory` | `OrderGadgets` | `OrderGadgetsWithInvInfo` |
| `DeclareMergedFlow_InvInfo`[8] | `OrderWidgetsWithInvInfo`<br>`OrderGadgetsWithInvInfo` | `OrderWithInvInfo` |
| `DeclareJoinedFlow_CrAndInvInfo` | `OrderWithCreditInfo`<br>`OrderWithInvInfo` | `OrderWithCreditOrInvInfo`[9] |
| `AggregateResults` | `OrderWithCreditOrInvInfo` | `OrderWithCreditAndInvInfo` |

[7] This service is a virtual one; it just declares that flows carrying orders coming from three sources are to be merged together.

[8] This service is again a virtual one – it marks merging flows `OrderWidgetsWithInvInfo` and `OrderGadgetsWithInvInfo` together.

[9] This flow contains both messages carrying credit information and messages carrying inventory information. In contrast, flow `OrderWithCreditAndInvInfo` contains messages carrying both credit and inventory information – it is being created by the `AggregateResults` service.

| | | |
|---|---|---|
| `OrderFeasibilityCheckRouter` | `OrderWithCreditAndInvInfo` | `OrderFeasible` `OrderRejected` |
| `Shipping` | `OrderFeasible` | `ShippingInfo` |
| `Billing` | `OrderFeasible` | `Invoice` |

Columns meaning: *Service* is the name of a service in question. *Input* and *Output* columns describe the content of the service's input and output message flows.

On the other hand, constraint programming-based methods (ML/CP, U/CP) work with *explicitly specified flow of control*: they expect a graph of control like the one shown in Figure 9. This is perhaps one of reasons they are significantly more efficient in designing the solution, as described in Chapter 1.

Another difference between our methods is that ML methods assume that control and data dependencies are the same, as we have mentioned above. They also work with the data *at the level of messages*, not looking at individual information elements (that we call *process variables* or simply variables) that are being carried in messages. In contrast, methods DL/P and U/CP allow a developer to specify data dependencies separately from control dependencies, and in more details: in terms of individual variables. They then try to find an efficient placement of variables in physical messages that flow within the integration solution.

### 4.1.2 Non-functional requirements

An important part of requirements specification is the characterization of *non-functional requirements*. Currently we deal with the following categories of requirements:

1. *Throughput*: we could require the solution to be able to continuously process a specified number of messages per time unit, e.g. per second or per minute (implemented in methods ML/P, ML/CP, and U/CP).

   Of course, buffering facilities provided by messaging middleware enable processing bursts of messages that arrive at rates higher than expected. Yet, here we require that the solution should be able to process specified amount of messages per time unit *for a long time*, without a negative impact on the processing time (latency).

2. *Availability*: we could 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 (ML methods and indirectly also U/CP).

   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 content, format, ordering, monitoring, duplication, and checkpointing* – these aspects are explained in Section 4.2.

### 4.1.3 Description of target environment

Besides requirements, design of any integration solution is driven by features of the target environment: integration platform and services that are available. In our method we deal with the following environment characteristics:

A. First of all, each business service (and in some cases also integration services) is described by the following characteristics:

   1. *Input* and *output*:

      a. *content* of messages entering and exiting the service – either at the level of whole messages (ML methods) or at the level of process variables carried in them (methods DL/P and U/CP),

      b. *format* of messages entering and exiting the service – like XML, CSV, JSON, and so on (ML methods and U/CP),

      c. *technical information* about positioning of data elements in message parts and about preserving other data elements in specified parts (methods DL/P and U/CP).

      For example, these are statements like "a service `CheckCredit2` requires an input data item `order` in message body and an input data item `creditPolicy` in message header. It produces an output data item `credit` in message body. The service keeps all items in message header and in message attachment untouched."

   2. *Throughput*: what throughput can the service provide? Of course, this depends on how it is deployed (in how many threads, processes, at what concrete hosts). Methods ML/P, ML/CP and U/CP deal with this issue, at different levels of accuracy: in methods ML/P and ML/CP we distinguish between four modes of deployment (so called 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. `CheckCredit` service in parallelism level 1 could achieve a throughput of 100 messages per minute and availability at the level of "normal". In parallelism level 4 it could achieve a throughput of 1000 messages per minute and availability at the level of "high". The U/CP method works with more precise estimates – we can specify the dependence of throughput on the number of threads the service is deployed in, with regards to specific service containers (an example is shown in Table 10 on page 100).

3. *Availability*: how available is the service – again, depending on its deployment (ML methods).

4. *Cost* or *resource usage* of the service: (1) the cost could be an abstract number (ML methods), reflecting service's use of resources like CPU, memory, network bandwidth, software licenses, and so on, usually depending on the way of deployment. (2) Another possibility is to specify resources consumed by the service individually for some or all of categories mentioned above. Then we can compute the total cost of the integration solution as the weighted sum of measures of consumption of individual resources by the services. This approach is used in the U/CP method.

B. Methods also need to have some knowledge about the implementation platform, for example: Does it allow combining Publish/Subscribe and Competing Consumers patterns (as does Progress Sonic ESB via shared subscriptions to topics)? Does it offer asynchronous in-memory channels (as does Apache Camel)? What is an effect of using some features or services on system resources, e.g. utilization of messaging middleware?

## *4.2 Output of the methods*

We work with messaging-based integration solutions that follow the Pipes and Filters pattern (see also Section 1.2.5): 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.

An example of such a solution is shown in Figure 10. This solution corresponds to a control and data flow shown in Figure 9 combined with the following requirements:

1. while all services in our scenario work with messages in XML format, inventory checking ones use JSON format instead,

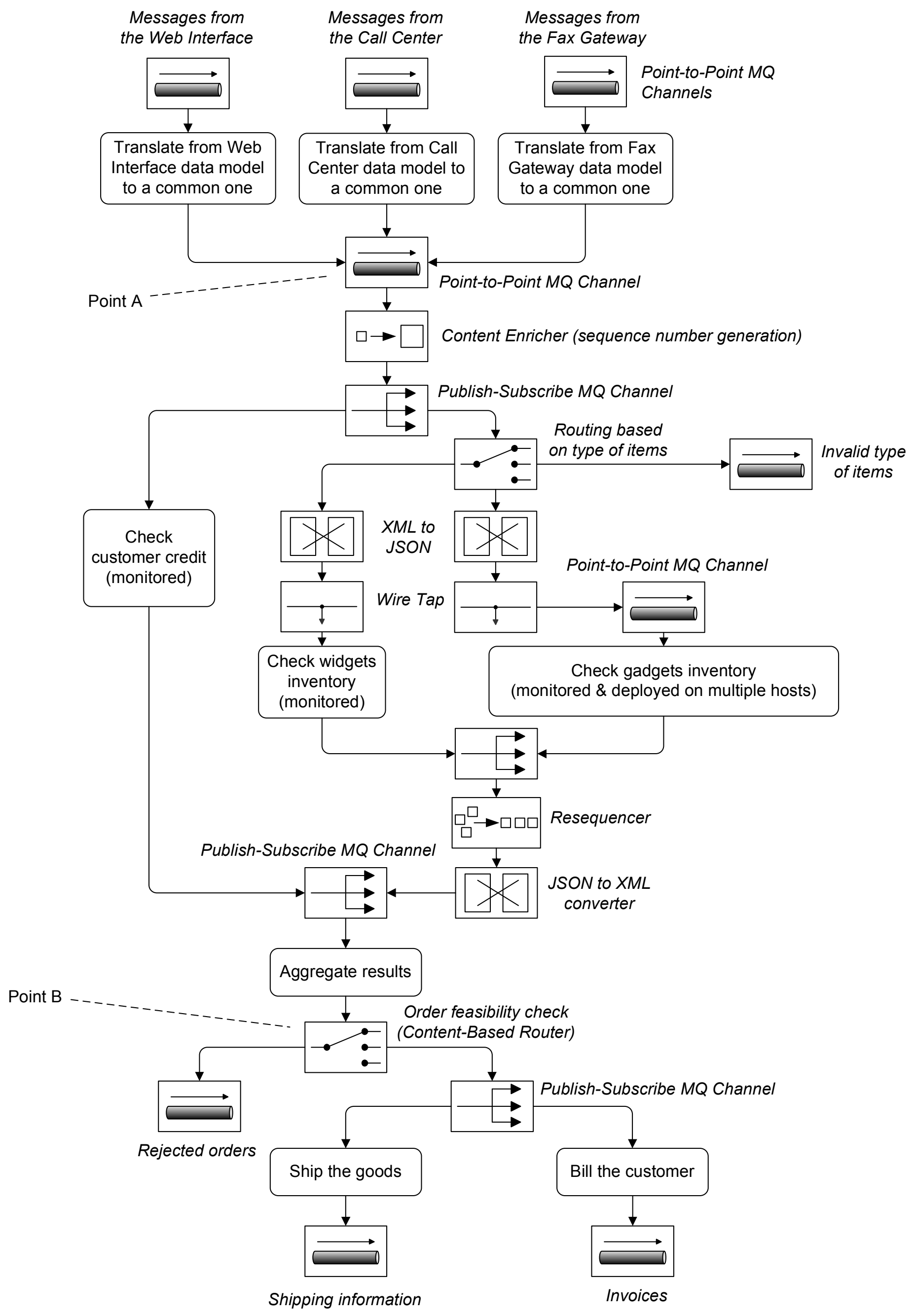
2. due to performance reasons the `CheckGadgetsInventory` service has to be deployed on multiple hosts,

3. we need to monitor correct functioning of credit checking service and both inventory checking services,

4. the order of messages arriving at `OrderFeasibilityCheck` service (point B in Figure 10) should be the same as original order of messages at the input side (at point A in Figure 10).

The solution presented is an output of the ML/P method. The method correctly determined that it should use a queue to dispatch messages to `CheckGadgets-Inventory` service executing at multiple hosts, employ format converters at appropriate places, and use sequence numbers generator just after Point A as well as the Resequencer somewhere before Point B.

As a notation we have used icons for channels and integration services suggested by Hohpe and Woolf (2004) and described in Section 1.5.2. (Connections between services without an indication of channel type are plain in-memory point-to-point channels.)



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

**Figure 10.** An example of a design produced by our method.

The remainder of this chapter is devoted to our representation of integration solutions using graphs.

### 4.2.1 Integration solution graphs

Each Pipes and Filters-based integration solution can be represented by a directed 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. In order to model this fact let us define a partial function $Service\colon V \rightarrow Services$ that, for each vertex $v \in V$ representing a service, gives this service, and for other vertices is undefined. *Services* set includes all business services (these are specific to the integration problem) as well as integration services.

As integration services we consider implementations of Wire Tap, Recipient List, Content Based Router, Splitter, Aggregator, Resequencer, Content Enricher, and Message Translator patterns. For their description please see Section 1.5.2. Last two patterns are used to fulfill a specific function: we use Content Enricher to insert sequence numbers into messages (therefore we call such a component Order Marker) and Message Translator is used to manipulate variables within a message (for example, moving them between message parts) – therefore we call it a Data Manager.

Vertices representing auxiliary components are:

1. solution's input points (their set is designated as *Input*);
2. solution's output points (their set is designated as *Output*);
3. fork points that are implemented by a Publish/Subscribe Channel (i.e. a topic), not by a service;
4. merge points – places where two or more message flows merge in one channel (again without using a real service).[10]

Please note that these vertices exist for modeling purposes only. They do not manifest themselves in an integration solution implementation.

In order to better illustrate our graph-based integration solution representation, let us redraw a part of Figure 10 into the form of the integration solution graph, shown in Figure 11. Each node in the diagram (either a box, or a small circle or small diamond symbol) is a graph vertex, and each connector between nodes is a graph edge. Vertices represented by boxes are services, the ones represented by small circles and diamonds are auxiliary components. Although not so visually appealing as Figure 10, this diagram better corresponds to our formal presentation of an integration solution.

[10] An alternative to representing channels implementing fork and merge points by vertices would be to model these channels using hyperedges. A disadvantage of this representation would be, however, that the solutions having fork point implemented using a Recipient List and using a Publish/Subscribe Channel would have different graph structure (a vertex with edges vs. a hyperedge) – something that would present a complication for methods based on constraint programming (Chapter 1).

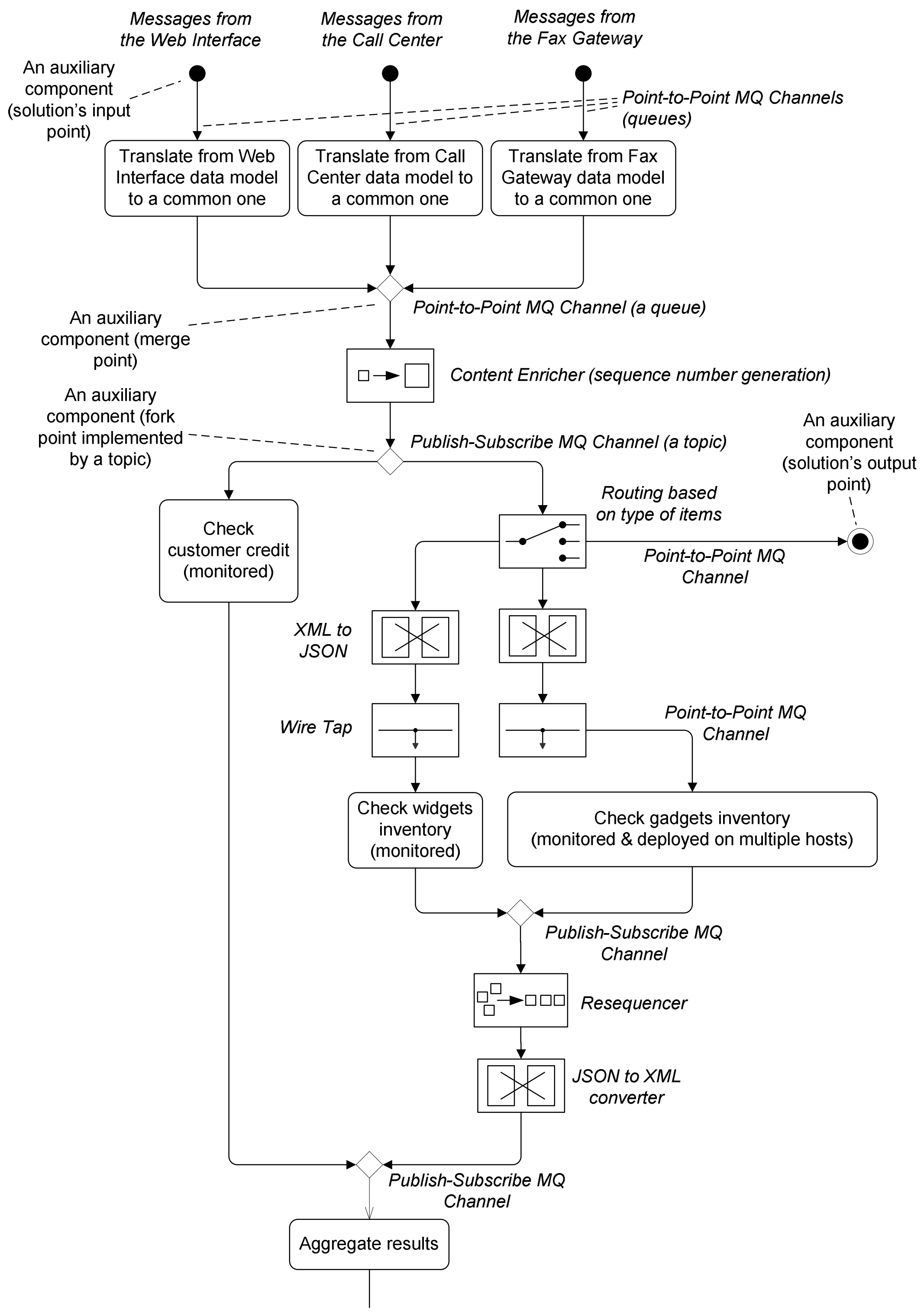

Messages from the Web Interface
Messages from the Call Center
Messages from the Fax Gateway
An auxiliary component (solution's input point)
Point-to-Point MQ Channels (queues)
Translate from Web Interface data model to a common one
Translate from Call Center data model to a common one
Translate from Fax Gateway data model to a common one
An auxiliary component (merge point)
Point-to-Point MQ Channel (a queue)
Content Enricher (sequence number generation)
An auxiliary component (fork point implemented by a topic)
Publish-Subscribe MQ Channel (a topic)
An auxiliary component (solution's output point)
Check customer credit (monitored)
Routing based on type of items
Point-to-Point MQ Channel
XML to JSON
Wire Tap
Point-to-Point MQ Channel
Check widgets inventory (monitored)
Check gadgets inventory (monitored & deployed on multiple hosts)
Publish-Subscribe MQ Channel
Resequencer
JSON to XML converter
Publish-Subscribe MQ Channel
Aggregate results

**Figure 11.** A part of an example of a design produced by our method, shown as a design graph.

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 $v$ let us denote a set of its incoming edges *In(v)* and a set of its outgoing edges *Out(v)*.

If $v_i$ and $v_j$ are services, then the meaning of $e_{ij} = (v_i, v_j)$ can be understood easily. If $v_i \in$ *Input*, then $e_{ij}$ is a solution's input channel. If $v_j \in$ *Output*, then $e_{ij}$ is a solution's output channel. If *v* is a fork point implemented using a Publish/Subscribe Channel, then all edges in $In(v) \cup Out(v)$ correspond to the same Publish/Subscribe Channel. If *v* is a merge point, then $In(v) \cup Out(v)$ correspond again to the same channel. The exact meaning of vertices representing auxiliary components and their incident edges differs in a small amount between our methods and between target integration platforms.

If *v* corresponds to a service, *In(v)* is a set of input channels of that service. Most services have only one input channel; however, some services implementing the Aggregator pattern can have more than one input channel. *Out(v)* is a set of output channels of the service denoted by vertex *v*. Again, while the usual number of output channels is one, services implementing the Content-Based Router and Recipient List patterns have typically more than one output channel.

Solution's input and output points have no input and output channels, i.e. for each $v \in Input$ and $w \in Output$ there is $In(v) = \varnothing$ and $Out(w) = \varnothing$.

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 modeling properties, along with the Service function, to be *property functions*.

In the following we discuss some of the common properties of services and channels.

**Channel content**

A very basic question is: what is being transported in a channel? Some of our methods (namely ML/P and ML/CP) treat channel content as an indivisible unit, denoted by a simple symbol. We model this using a function $Content: E \rightarrow Contents$, where *Contents* is a set of all possible types of message content. In our sample integration problem, *Contents* = { *OrderWebNative*, *OrderCcNative*, *OrderFaxNative*, *Order-Web*, *OrderCc*, *OrderFax*, *Order*, *OrderWithCreditInfo*, *OrderWidgets*, *Order-Gadgets*, *OrderInvalidType*, *OrderWidgetsWithInvInfo*, *OrderGadgetsWithInvInfo*, *OrderWithInvInfo*, *OrderWithCreditOrInvInfo*, *OrderWithCreditAndInvInfo*, *Order-Feasible*, *OrderRejected*, *ShippingInfo*, *Invoice* } (please see also Table 3). To be precise, this function reflects not only message content as such, but gives also some information about the context of the messages – for example, "this flow contains orders that have been rejected" (*OrderRejected*).

We can use such a characterization to formulate design rules: formulas that must be valid for any solution for a given integration problem. For example, as we know that the credit checking service (*CheckCredit*) takes an order (*Order*) as its input and produces an order with the customer credit information (*OrderWithCreditInfo*) we can state that

$$\forall v \in Dom(Service), Service(v) = CheckCredit, e \in In(v), f \in Out(v):$$
$$Content(e) = Order \wedge Content(f) = OrderWithCreditInfo.$$

By *Dom(F)* we mean the domain of function *F*; so *Dom(Service)* denotes all vertices $v \in V$ that correspond to services. This rule should be therefore read as follows:

For each vertex *v* that corresponds to the `CheckCredit` service, for its input channel *e*, and for its output channel *f* it holds that *e* transports messages with the content of type `Order` and *f* transports messages with the content of type `OrderWithCreditInfo`.

As we have mentioned in Section 4.1, methods DL/P and U/CP deal with the content of messages with more precision: they try to find out how to store logical data items (process variables) in physical messages.

There are two related questions:

1. What process variables to transport in an individual channel?
2. Exactly where to place these variables in messages?

Concerning the second question, a message typically consists of a header, main part (part 0) and other parts (attachments). It can transport data in any of these, although there are often reasons for choosing specific parts, depending e.g. on type and size of respective data items, on requirements and capabilities of services involved, and so on.

Let us denote a set of process variables that are available in the integration solution *Variables* and a set of possible positions for carrying data in messages *Positions*. Then we define the following functions that characterize content of messages flowing in individual channels:

1. *VariablePresence:* $E \times Variables \rightarrow Boolean$, i.e. this function determines, for each channel $e \in E$, a set of process variables transported in this channel.
2. *VariablePosition:* $E \times Variables \times Positions \rightarrow Boolean$, i.e. this function determines, for each channel $e \in E$, a set of process variables transported in this channel as well as positions they are transported at.

   We acknowledge this is a simplified model, as we cannot exactly capture the situation when a message contains a process variable twice (or more times) at a given position, e.g. in attachments. Such an exact representation would be necessary for example for more precise computation of the number of bytes transported through messaging middleware. We have considered improving this by and recording the *number of times* a variable is present in places where it can be present in more copies (typically in header and in attachments). But for now we keep only simple binary information.

As an example of related rule, let us consider again the `CheckCredit` service that is used in an order processing scenario in such a way that it expects an `Order` variable in the message body, produces a `Credit` variable in the message body (overwriting existing value of the body), discarding all message attachments but keeping all the headers:

$$\begin{gathered}\forall v \in Dom(Service), Service(v) = CheckCredit, e \in In(v), f \in Out(v):\\ VariablePosition(e, Body, Order) = true \wedge VariablePosition(f, Body, Credit) = true \wedge\\ (\forall var \in Variables: VariablePosition(e, Header, var) = true \Rightarrow\\ VariablePosition(f, Header, var) = true) \wedge\\ (\forall var \in Variables: VariablePosition(f, Attachments, var) = false)\end{gathered}$$

This rule should be read as follows: For each vertex *v* that corresponds to the `CheckCredit` service, for its input channel *e*, and for its output channel *f* it holds that:

1. messages transported in *e* contain `Order` in their body,
2. messages transported in *f* contain `Credit` in their body,
3. if a variable *var* is stored in message header in messages in channel *e*, it will be stored in message header in messages in channel *f* (i.e. the `CheckCredit` service keeps message headers intact),
4. at output of `CheckCredit` there are no attachments (i.e. `CheckCredit` discards all attachments).

Such service-specific rules (determined from the description of data flow and environment) are augmented by more general rules valid for a given environment or all environments, for example:

$$\begin{gathered}\forall e \in E, var \in Variables: VariablePresence(e, var) \Leftrightarrow\\ VariablePosition(e, Header, var) \vee VariablePosition(e, Body, var) \vee\\ VariablePosition(e, Attachments, var)\end{gathered}$$

meaning that variable can be transported either in header, body, or attachments (reading it like this: a variable is present in messages flowing through a channel if and only if it is present in headers, bodies, or attachments of these messages), or:

$$\begin{gathered}\forall e \in E, var_1, var_2 \in Variables, var_1 \neq var_2:\\ \neg\, [VariablePosition(e, Body, var_1) \wedge VariablePosition(e, Body, var_2)]\end{gathered}$$

meaning that it is not possible to put two variables into message body at once.

### Channel types

Another usual property of a channel is its *type*. 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.

So, at a general level, there is a function $ChannelType: E \rightarrow \{ InMemory, Topic, Queue \}$. The set of values is influenced by a concrete integration platform for which we create an integration solution. For example, Apache Camel provides two types of in-memory channels – synchronous and asynchronous ones – therefore, in that case, the property *ChannelType* corresponds to a function $ChannelType_{Camel}: E \rightarrow \{ InMemorySynchronous, InMemoryAsynchronous, Topic, Queue \}$.

**Service deployment in service containers**

Other typical properties are connected to the *deployment* of individual services. In order to increase throughput and/or availability of a service we often have 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 patterns.

The idea of Message Dispatcher pattern is that a specialized software component, a dispatcher, is reading messages off the channel and routes them to a set of worker threads (within one process). On the other hand, Competing Consumers pattern is based on the concept of having multiple independent consumers reading messages from the messaging infrastructure (more specifically, from a channel of type Queue) and giving them to further processing. These consumers can reside in multiple processes. However, implementation details of these patterns vary among integration platforms.

Our methods deal with this question at three levels of granularity:

1. coarse-level $DeploymentMode: V \rightarrow \{ SingleThread, MultipleThreads, MultipleProcesses, MultipleHosts \}$ (see also point A.2 in Section 4.1.3),

2. finer-level: $ThreadCount: V \rightarrow TC$, $ContainerCount: V \rightarrow CC$ and $HostCount: V \rightarrow HC$ where $TC$, $CC$, and $HC$ denote sets of possible threads counts, container counts and host counts, respectively,

3. finest-level $Deployment: V \rightarrow \{ (t_1, t_2, ..., t_n) \mid t_i \in TC_{v,i} \}$ where $t_i$ denotes a number of threads $Service(v)$ is deployed in service container $C_i$.[11]

[11] Even this concept of a service being deployed in a container using a specified number of threads can be ambiguous and platform-specific in a situation where a service is to be used simultaneously at more than one point in the integration solution. For simplicity we assume here that (1) each business service is used only once, (2) individual integration services are distinct, i.e. if we have two Wire Taps, these are, in fact, two independent services of type Wire Tap.

Concrete sets *TC*, *CC*, *HC*, and $TC_{v,i}$ ($v \in V$, $i$ = 1, ..., n, where n is the number of service containers available) and functions *DeploymentMode*, *ThreadCount*, *ContainerCount*, *HostCount* and *Deployment* are determined from the description of environment. For example, if *Service(v)* cannot be deployed in container $C_i$, then $TC_{v,i} = \{ 0 \}$.

For optimization purposes we sometimes do not allow the number of threads to have an arbitrary integer value between 0 (or 1) and a specified maximum thread count. Usually we restrict these numbers to be powers of two, i.e. *TC* or $TC_{v,i} = \{ 0, 1, 2, 4, 8, ..., max \}$ where *max* is a value determined from the environment description.

Chosen deployment restricts the choice of input channel. For example, topics are usually not allowed to be used with Competing Consumers pattern, so we can assert that

$$\forall v \in Dom(Service), e \in In(v):$$
$$DeploymentMode(v) \in \{ MultipleProcesses, MultipleHosts \} \Rightarrow ChannelType(e) \neq Topic,$$

or (equivalently, at more specific level)

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

Another example of a dependency between input channel and service deployment is derived from characteristics of synchronous in-memory channels: these can be used only if services connected by this channel are deployed in exactly the same way, i.e. in the same containers, with the same numbers of threads in them. This fact can be captured at various levels of details, depending on the exact way of representation used:

$$\forall v, w \in Dom(Service), e \in Out(v), e \in In(w):$$
$$ChannelType(e) = InMemorySynchronous \Rightarrow Q,$$

where formula Q depends on the level of abstraction concerning deployment used – i.e. Q is either

$$DeploymentMode(v) = DeploymentMode(w),$$

or

$$ThreadCount(v) = ThreadCount(w) \wedge ContainerCount(v) = ContainerCount(w) \wedge$$
$$HostCount(v) = HostCount(w),$$

or

$$Deployment(v) = Deployment(w).$$

**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. This is very easy for Publish/Subscribe Channels, i.e. topics, as they (by definition) allow a monitoring tool to subscribe to them and listen to all messages going through them. Otherwise, we can apply the Wire Tap pattern that provides a special service that copies all of its input to a dedicated monitoring channel.

Formally we can define a function *Monitored*: *E* → *Boolean* that assigns a value of *true* or *false* to each channel, subject to the following rule:

$$\begin{aligned}
&\forall e \in E:\ [\ \mathit{Monitored}(e) = \mathit{true} \Leftrightarrow \\
&\quad (\mathit{ChannelType}(e) = \mathit{Topic} \vee \\
&\quad\ \exists v \in \mathit{Dom}(\mathit{Service}): \mathit{Service}(v) = \mathit{WireTap} \wedge (e \in \mathit{In}(v) \cup \mathit{Out}(v))) \vee \\
&\quad\ \exists w \in V: \mathit{MonTransparent}(w) = \mathit{true} \wedge (e \in \mathit{In}(w) \wedge\ \exists f \in \mathit{Out}(w): \mathit{Monitored}(f) = \mathit{true} \\
&\qquad\qquad \vee\, e \in \mathit{Out}(w) \wedge \exists g \in \mathit{In}(w): \mathit{Monitored}(g) = \mathit{true}))\ ]
\end{aligned}$$

Actually this rule is slightly more general that the description above: it assumes that we allow messages to be monitored at a place that is connected to the place where monitoring was requested by a path not changing message's content, i.e. by a path consisting of vertices that are transparent with regards to monitoring, characterized by the function *MonTransparent*: V → *Boolean*, derived from the description of the environment.

**Message ordering**

There can be a situation that the original ordering of messages is lost, typically in the case of parallel or alternative processing. The designer then has to employ a Resequencer service that restores the original order using message sequence numbers. If messages do not contain such sequence numbers, it is necessary to add them using Content Enricher (of course, it has to be applied while messages are in the original order).

We have defined two functions, namely *Ordered*: *E* → *Boolean* and *OrderMarked: E* → *Boolean* (with some extensions designed to deal with alternative message flows but these are not important to cover in details). The first function describes whether messages flowing through the channel $e \in E$ are (or are not) ordered and the second one says whether the message order is marked within these messages using sequence numbers (or is not). Of course, it is important to say what the *reference point* is, to which we relate message ordering. Usually – but not necessarily – it is the entry point of the integration solution.

An example of a rule concerning message ordering: At an output of a service deployed in multithreaded mode the flow of messages is not ordered.

$$\forall v \in \mathit{Dom}(\mathit{Service}), e \in \mathit{Out}(v): \mathit{ThreadCount}(v) > 1 \Rightarrow \mathit{Ordered}(e) = \mathit{false}$$

(An analogous formulation can be written using *DeploymentMode* function.)

### Message format

Almost all services require messages to be in a specified format, e.g. comma-separated values, fixed-length records, XML, JSON (JavaScript Object Notation), or other. The integration designer has to employ specific converters appropriately. We can model this using a function $Format\colon E \rightarrow Formats$ that for each $e \in E$ describes the format of messages going through $e$ as $Format(e)$. We use this function for message-level methods (ML/P, ML/CP).

An example of a rule: If a service $S_1$ requires XML as its input and produces XML as well, we should enforce that

$$\forall v \in Dom(Service), Service(v) = S_1, e \in In(v), f \in Out(v)\colon$$
$$Format(e) = XML \wedge Format(f) = XML$$

For the U/CP method we have extended the range of the *VariablePosition* function from Boolean to the set $\{\ NotPresent\ \} \cup Formats$, so this function describes both whether a variable is present at a specified position and what format is used.

### Message duplication

When using messaging middleware, message duplication sometimes occurs. There are three basic approaches how to deal with it: (1) using the Transactional Client pattern that ensures that the duplicate messages do not arise, in the first place, (2) using a special Message Filter designed to eliminate duplicates if they are already present, or (3) using an Idempotent Receiver pattern denoting services that can accept duplicate messages without problems.

This aspect is modeled by function $Duplicates\colon E \rightarrow Boolean$ that for each $e \in E$ indicates whether in channel $e$ there can be duplicate messages present or not. Moreover, we provide a function $TransClient\colon V \rightarrow Boolean$ that signal whether, for $v \in V$, we use the Transactional Client pattern or not (and, therefore, whether it cannot produce duplicates, or it can), and a function $Idempotent\colon V \rightarrow Boolean$ indicating if a *Service(v)* is an Idempotent Receiver.

An example of a rule stating that a service not using a Transactional Client that is sending messages out to messaging middleware can produce duplicates:

$$\forall v \in Dom(Service), e \in Out(v)\colon$$
$$TransClient(v) = false \wedge ChannelType(e) \in \{\ Topic, Queue\ \} \Rightarrow Duplicates(e) = true$$

### 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. Let us call such points to be *checkpoints*. When using messaging middleware it is natural to implement them via messaging channels: such a channel can keep a message until its processing is acknowledged by the service.

A developer can specify a concrete channel to hold a checkpoint, or he or she can say that a checkpoint should be present in one channel from a defined set of channels.

Formally,

$Checkpoint_i \subseteq E$ is a set of channels that could hold a i-th checkpoint (i = 1, 2, ..., *cp*) where *cp* is the total number of checkpoints defined.

$Checkpointed: E \rightarrow Boolean$ is a function that says whether a given channel $e \in E$ holds a checkpoint.

$$\forall e \in E: Checkpointed(e) = true \Rightarrow ChannelType(e) = \{ Queue, Topic \}$$

$$\forall i \in \{1, 2, ..., cp\}\ \exists e \in Checkpoint_i: Checkpointed(e) = true$$

In Table 4 we summarize functions that we use to model the above mentioned aspects.

**Table 4.** 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) with VariablePosition(e, var, pos) |
| Channel types | - | ChannelType(e) |
| Service deployment | DeploymentMode(v), or ThreadCount(v) with 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) |

Columns meaning: *Aspect* denotes a concrete aspect of integration solution design. *Vertex-related function(s)* and *Edge-related function(s)* columns contain names of functions, with the domain of vertex set and edge set, respectively, that model the particular aspect.

Individual methods do not cover all the aforementioned aspects. In Table 5 we summarize support for these aspects by our methods.

**Table 5.** 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 | Finer & finest level |
| Monitoring | Yes | - | Yes | Yes |
| Message ordering | Yes | - | Yes | Yes |
| Message format | Yes | - | Yes | Yes |
| Message duplication | - | - | Yes | - |
| Checkpointing | - | - | - | Yes |

Columns meaning: *Aspect* denotes an aspect of integration solution design. The following four columns contain information about whether the particular aspect is supported by the individual methods and, optionally, at what level.

Let us conclude this section by recapitulating its main ideas.

1. Our methods produce designs of messaging-based integration solutions that use Pipes and Filters architectural pattern.

2. We have found a way to formalize these designs in a form of directed acyclic graph with vertices corresponding to services and auxiliary components and edges corresponding to channels carrying messages.

3. We have identified key design properties of such solutions and formalized those using functions defined on graph's vertex and edge sets. We have identified and formalized rules governing these properties so that they can be used to find suitable integration solutions for given integration problems.

Because of space constraints we have listed only selected examples of these rules here. We should also highlight that the scope of these rules varies from *very general*, for rules that are valid for almost all integration platforms, derived directly from enterprise integration patterns description given in (Hohpe and Woolf, 2004), to *platform-specific* where "platform" can mean a concrete integration platform product, its specific version, or even its specific version combined with specialized services created in order to support our methods at run-time.

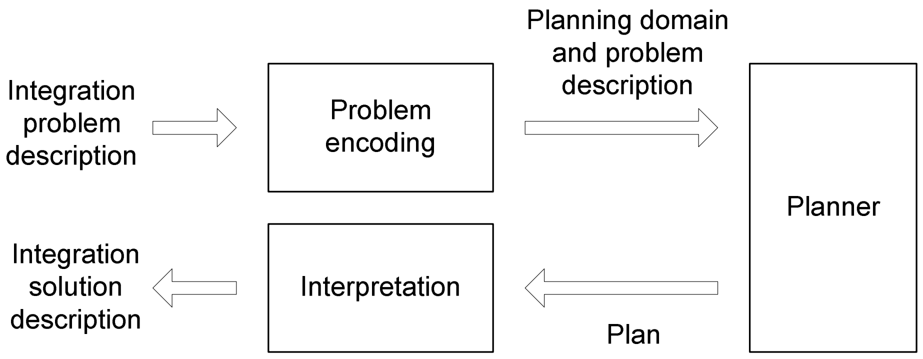
In constructing integration solutions, we did not stop at the level of their abstract design, though. The U/CP method provides an output that is sufficiently detailed so it can be directly translated into an executable code for a specified integration platform. Code generation for Progress Sonic ESB and for Apache Camel (in part) has already been successfully implemented; code generation for Mule ESB is underway, and for other platforms it is planned.

# 5 Planning-based methods

Methods ML/P and DL/P use planning 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.

The principle of these methods is following: 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 (see Section 4.2).

This approach is depicted in Figure 12.



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

**Figure 12.** Basic principle of the planning-based methods.

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 cuts of the solution graph, as described in Section 5.1.1. 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 symbol 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 symbol in the DL/P method. (More details are in sections 5.1.1 and 5.3.)

Concerning properties of solution graph vertices, these are mapped to the planning operators names (and possibly also their parameters), as described in sections 5.1.1 and 5.3 as well.

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 for the output side, the plan (a sequence of 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.

## *5.1 The ML/P method details*

In this section we show the details of the ML/P method – a mapping from an integration problem to a planning problem: how the state of the world is represented and how the operators acting upon it look like.

### 5.1.1 Mapping from an integration problem to a planning problem

First of all, for purposes of the ML/P method we have slightly modified the way of representing an integration solution using a graph: instead of creating an auxiliary vertex for each fork point implemented by a Publish/Subscribe MQ Channel (see point 3 in the list at page 48) we have decided to create an auxiliary vertex for *each* channel implemented in messaging middleware. The reason is that this provides us with a simple and powerful optimizing criterion: as actions in the plan correspond to solution graph vertices, when we optimize on the number of actions (this is the most common option implemented in planners), we are de facto trying to find a solution that has the smallest number of components and MQ channels – a very relevant optimizing criterion for practical use. More on the notion of the optimal solution is in Section 5.2.

We illustrate the mapping using our case study, as shown in Figure 9 (abstract design) and Figure 10 (resulting detailed design), part of which is repeated in Figure 13.

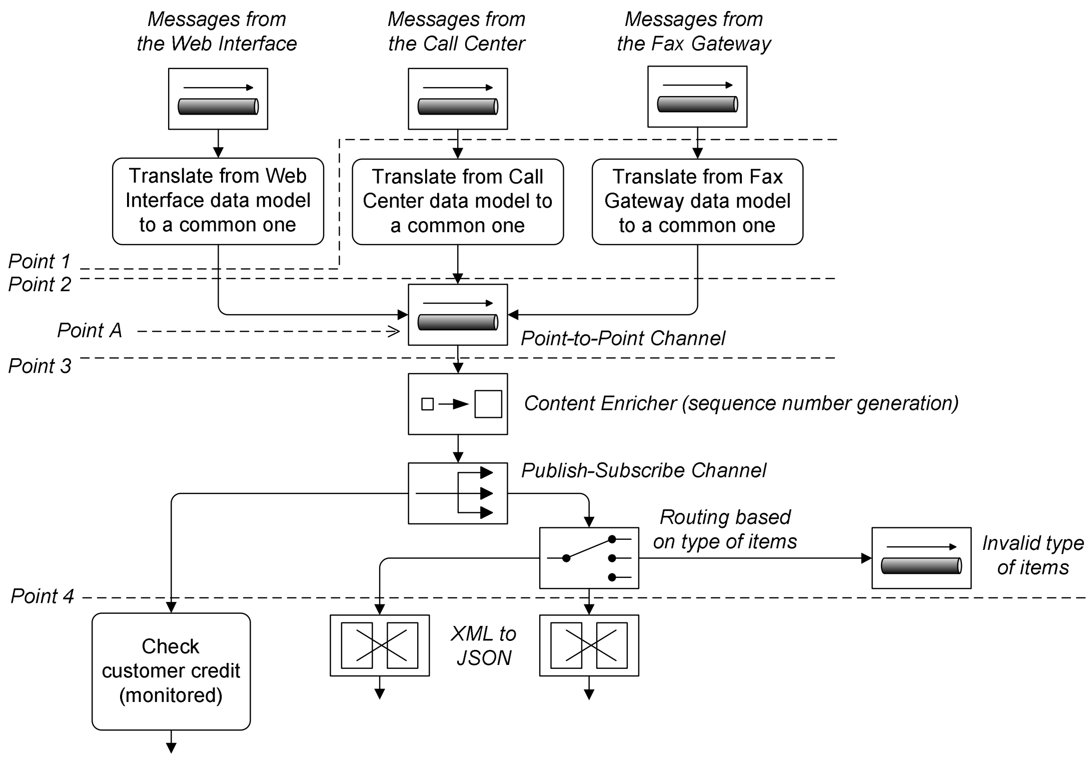

Messages from the Web Interface
Messages from the Call Center
Messages from the Fax Gateway
Translate from Web Interface data model to a common one
Translate from Call Center data model to a common one
Translate from Fax Gateway data model to a common one
Point 1
Point 2
Point A
Point-to-Point Channel
Point 3
Content Enricher (sequence number generation)
Publish-Subscribe Channel
Routing based on type of items
Invalid type of items
Point 4
Check customer credit (monitored)
XML to JSON

**Figure 13.** A part of messaging-based implementation of the sample integration scenario.

Please consider a fragment of a solution shown in Figure 13. As there are three input channels (message flows), the initial state in this case would be represented as a conjunction of these three literals:[12]

```
(message c_order_web_native xml unord ord_not_marked_partial not_mon ch_queue flow_1)
(message c_order_cc_native xml unord ord_not_marked_partial not_mon ch_queue flow_1)
(message c_order_fax_native xml unord ord_not_marked_partial not_mon ch_queue flow_1)
```

These literals capture information about the individual channels (message flows), using a predicate symbol `message` with the seven arguments. First six of them correspond to channel properties mentioned in Section 4.2, namely content of messages, format of messages, ordering of messages, whether the order is marked within messages, monitoring-related state, and channel type.

In order to simplify implementation of rules that characterize a correct solution we had to slightly modify the meaning of some of these arguments in comparison to corresponding abstract solution graph property functions, namely:

1. a monitoring-related state of messages has been extended from simple ("is monitored" / "is not monitored") to three-way distinction: whether they are not

[12] As described in Section 3.2, we use a PDDL notation where the literal contains the predicate symbol followed by its parameters. Moreover, instead of writing constants using camel case (e.g. *OrderWebNative*) we use all-lowercase (`c_order_web_native`) convention, with the prefix `c_` for domain-specific content types.

to be monitored (value `not_mon`), or they are monitored (value `mon`), or they are not monitored yet but should be (value `mon_req`),

2. an indication of the type of channel that carries these messages, with values of `ch_topic`, `ch_queue`, `ch_memory12`, `ch_memory3`, `ch_memory4`. Last three possibilities represent various versions of in-memory channels: within one container, within more containers at one host, within containers at more hosts. (Actually, this assumes asynchronous in-memory channels. In case of synchronous ones, we should also distinguish between `ch_memory1` and `ch_memory2`.)

These extensions are necessary in order to give a planner all necessary information it needs to be able to add next action to the plan without looking back at already existing actions (i.e. to have all the information in the current state of the world). For example, when the planner sees that a channel is of type `ch_memory3`, it knows that the next service has to be deployed using 3rd deployment mode (multiple processes, single host) without looking back to check the deployment mode of the previous service.

The last (seventh) parameter is used to distinguish among multiple identical message flows coming out e.g. from a topic or from a Recipient List service.

Generally, for each additional design aspect our method would need to cover, we would try to identify relevant attributes that could be attached to channels or services. Channel attributes would then be mapped to arguments of the `message` predicate symbol, as shown above. Service attributes would be mapped to operators' names and/or parameters.

As we can see, the state of the world describes not an individual message flow, but a set of message flows present at a particular point of the integration solution. By "point" here we understand a cut-set of a specific cut of the integration solution graph – informally, a cut that corresponds to a state in the message(s) processing. More formally, we are thinking of such a cut $C = (S, T)$ of a solution graph G = (V, E), so that $C$ is a partition of $V$, $Input \subseteq S$, $Output \subseteq T$, and there is no such edge $e = (u, v) \in E$ that $u \in T$ and $v \in S$.

We have decided to put all the information about a message flow into one predicate in order to make working with parallel message flows in operators' preconditions and effects easier and less demanding with respect to the expressive power of PDDL variant that has to be used.

In the example shown in Figure 13, after messages coming from web interface are processed by appropriate translation service (named "Translate from Web interface data model to a common one"), i.e. at Point 1, the state of the world would look like this (changes are shown in bold):

```
(message c_order_web xml unord ord_not_marked_partial not_mon ch_memory12 flow_1)
(message c_order_cc_native xml unord ord_not_marked_partial not_mon ch_queue flow_1)
(message c_order_fax_native xml unord ord_not_marked_partial not_mon ch_queue flow_1)
```

As we see, the content of messages in the first flow has changed from `c_order_web_native` to `c_order_web` and the transport channel has changed from `ch_queue` to `ch_memory12`. The first is an effect of a content transformation service; the second is a general effect of any service.

In a similar way, at Point 2 the situation would look like this – here we see changes in the second and the third literal:

```
(message c_order_web xml unord ord_not_marked_partial not_mon ch_memory12 flow_1)
(message c_order_cc xml unord ord_not_marked_partial not_mon ch_memory12 flow_1)
(message c_order_fax xml unord ord_not_marked_partial not_mon ch_memory12 flow_1)
```

At Point 3 we see an effect of joining all these flows in a channel of type “queue”, and the declaration that at this point we consider the message flow to be ordered:

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

At Point 4 we have again more message flows active, so the state of the world is:

```
(message c_order xml ord ord_marked monitored ch_memory12 flow_1)
(message c_order_widgets xml unord ord_marked_partial not_mon ch_memory12 flow_1)
(message c_order_gadgets xml unord ord_marked_partial not_mon ch_memory12 flow_1)
(message c_order_invalid xml unord ord_marked_partial not_mon ch_queue flow_1)
```

Finally, the flows at integration solution’s output correspond to the following goal state:

```
(message c_invoice xml unord ord_marked_partial not_mon ch_queue flow_1)
(message c_order_shipping_info xml unord ord_marked_partial not_mon ch_queue flow_1)
(message c_order_rejected xml unord ord_marked_partial not_mon ch_queue flow_1)
(message c_order_invalid xml unord ord_marked_partial not_mon ch_queue flow_1)
```

Concerning planning *operators*, the most important ones are those directly derived from services available. In the ML/P method, each service is transformed to up to four operators, one for each of the following modes of deployment (parallelism levels, PL): (1) single thread, (2) single process, multiple threads, (3) single host, multiple processes, and (4) multiple hosts. 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. 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. We provide more sophisticated treatment of these aspects in the U/CP method.

As an example, the `CheckCredit` service that expects a message with content “Order” (`c_order`) and transforms it into a “Order with credit information” (`c_order_crinfo`), deployed on multiple hosts, with monitoring required, is represented by the following operator (symbols starting with `?` depict operator parameters, symbols starting with `t-` represent object types):

```
(: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)
   )
  )
```

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. Transport of messages through messaging middleware is modeled by two special operators, `Queue` and `Topic`, which correspond to the auxiliary vertex types mentioned at the beginning of this section. Technically, if one of these operators is added after a service, it changes the default `ch_memoryX` channel type to `ch_queue` or `ch_topic` and means that the service sends its output through this kind of channel. As we have mentioned above, 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 more precisely.

Other design aspects (message formats, ordering, and monitoring) are treated by operators in a similar way. More information can be found in (Mederly, Lekavý, Závodský and Návrat, 2009).

### 5.1.2 Optimization

In order to shorten the time needed to find a solution we provide the user with an option to disable processing of design aspects he or she does not need – currently it is possible to turn off evaluation of message formats, monitoring, and message ordering. If disabled, the respective parameters are simply omitted from the `message` predicate, resulting in reduction of the state-space a planner has to work with.

## *5.2 Results*

We have implemented both planning-based methods (ML/P, DL/P) in the form of prototypes. Here we describe results provided by the ML/P method.

First of all, in order to evaluate our method we have tried several existing 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 (ICAPS Competitions, 2011) and by our previous experience. Namely, we have used the following planners: Gamer, MIPS-XXL, HSP 2.0, FF 2.3, SatPlan2006, and MaxPlan, briefly characterized in Section 3.2.

We demonstrate the evaluation results here using four selected integration problems:

- Problems 1 and 2 correspond to a part of Widgets and Gadgets order processing scenario. The part covered begins when orders from three sources are merged (Point A in Figure 10) and ends as orders enter feasibility check (Point B in Figure 10). Problem 1 takes into account monitoring and throughput/availability aspects. Problem 2 takes into account aspects of monitoring, message format, and throughput/availability.

- 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/availability into account, while Problem 4 does.

These settings are summarized in Table 6.

**Table 6.** Description of problems selected for the ML/P method evaluation.

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

Columns meaning: *Problem #* column contains the identification of the problem in question. *Scope* (reduced, full) describes whether the problem concerns a part of the scenario or the whole scenario. *Aspects* column shows which aspects are solved: monitoring (M), message formats (F), message ordering (O), and throughput/availability (TA). *Message predicate arity* column refers to the arity of the `message` predicate (this arity depends on what aspects we take into account, ranging from 3 to 7). *Parameters/operator* column presents average number of parameters of individual operators. These measures, along with *number of operators* and number of objects in the domain (*Domain objects* column), 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. *Optimal plan length* is the length – i.e. number of steps – of the optimal plan; again, we use it as an indication of the hardiness of the problem.

The results (quality of solution found and CPU time needed to find it) for some of the planners are summarized in Table 7. Please note that 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 7.** Characteristics of selected planners and results of using them with the ML/P method.

| Planner | Domains solved | Plan search algorithm | Problem 1 | Problem 2 | Problem 3 | Problem 4 |
|---|---|---|---|---|---|---|
| Gamer | cost, seq | state, opt | 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 |

Columns meaning: *Planner* column indicates the planner used. *Domains solved* column shows which kinds of problems is the planner able to solve: sequential plans (seq), parallel plans (par), and action costs (cost). *Plan search algorithm* describes how the planner works, i.e. whether it uses state-space search (state) or transformation to satisfiability problem (SAT), and whether it guarantees to generate optimal plan (opt), or it is not guaranteed to generate optimal plans (subopt). *Problem 1* to *Problem 4* columns contain information about results of solving the corresponding problems using the particular planner. Acronyms for results are: optimal plan found (O), suboptimal plan found (SO), computation failed (Error). The numeric value is the CPU time needed to find the plan.

These results show that the ML/P method is able to find solutions for practical integration problems using currently available planners. Yet, the majority of the planners had difficulties solving the most complex Problem 4. We suspect that primary reason is that they were not designed to work with such a large state-space as it was present in this problem; their execution usually halted because of exceeding available memory or because of using fixed-size structures and variables, not scaling with the problem size. These problems usually resulted in runtime errors like "memory allocation error" or "segmentation fault". Also, some of the planners have limitations concerning the arity of predicates used. In some cases we have been able to increase these limits by changing source code, but in some cases not – the latter cases are labeled as "computation failed" in Table 7 as well.

Let us include a few remarks concerning the state-space size. A state of the world is a conjunction of grounded positive literals (atoms). The size of the state-space of a planning problem is therefore equal to the number of subsets of a set of all grounded atoms. If we call this set of all grounded atoms $A$, the state-space size is then $2^{|A|}$. If the planning problem contains predicates $P_1, ..., P_n$, then $A = A_{P1} \cup A_{P2} \cup ... \cup A_{Pn}$, where $A_{Pi}$ is a set of all atoms using a predicate $P_i$. Among predicates used in our planning problems, `message` has the largest number of arguments with the largest number of objects that can be used as arguments' values, so the $|A|$ is determined primarily by $|A_{Pmessage}|$. For example, in Problem 1 `message` has four arguments (Content, Monitoring, Channel, FlowID) with 10, 3, 5, and 3 applicable objects, respectively, giving 450 atoms in $A_{Pmessage}$. For Problem 4 this predicate has seven arguments (Content, Format, Ordered, OrderMarked, Monitoring, Channel, FlowID) with 20, 2, 2, 4, 3, 5, and 3 applicable objects, respectively, leading to 14400 atoms in $A_{Pmessage}$. The main goal of a planner is to reduce the vast amount of states. The state-space search restricts the state space to states reachable from the initial or final states. Additionally, different heuristics are used to further prune states, which are not likely to be a part of the final plan. Nevertheless, even this reduced state-space may be very large. Not all planners are able to cope with such an increase in the planning problem state-space. We suppose that this is partially because of time and space complexity of individual methods and partially because many implementations are optimized for scaling of one planning problem attribute (e.g. branching factor) by constraining some other attribute (e.g. by limiting maximal predicate arity). However, this issue should be researched further in order to make a definitive statement.

Overall, if we would like to further increase the number of aspects or the size of integration problems we deal with, we would need to tackle this problem of state-space expansion in some way. It is not only a question space complexity; it is a question of time complexity as well. According to Erol, Nau, and Subrahmanian (1992), planning problems that have similar characteristics to our ones – i.e. having predicates with parameters, operators that are given in the input, and non-empty delete lists – are, in the worst case, EXPSPACE-complete (or, more exactly, telling if a plan exists, is an EXPSPACE-complete decision problem). This worst-case estimation may or may not be relevant for our situation; nevertheless, domain-independent planning is computationally difficult and often intractable (Bacchus and Kabanza, 2000), and results of our experiments are consistent with this fact. We describe two possible ways to overcome this limitation at the end of this chapter.

Returning to our experimental results, 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. Some of the possibilities are:

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, etc.).

(These and other optimization criteria are discussed in more depth in the context of the U/CP method in Section 6.1.2.)

At this moment, for planning-based methods we use the criterion 1, corresponding to the shortest sequential plan. We have also tried to incorporate the criterion 4 by assigning costs to individual actions based on resource consumption – it is possible, but it limits the set of available planners to those that are able to work with action costs and, as our experience shows, it also makes planning significantly slower. 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). We decided to stay with the goal of finding the solution with the smallest complexity and leaved the issue of exact optimality definition to be solved in methods based on constraint programming.

Given this assumption about solution optimality we see that optimal sequential planners with no other extensions are sufficient. Parallel planners, especially MaxPlan, are quick to find the optimal solution – unfortunately, they generally optimize makespan (number of steps of plan execution) instead of 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 the real-life experience of the author of this dissertation (Mederly and Pálos, 2008) 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 (Mederly and Lekavý, 2009) and is available on the attached CD-ROM media as well.

## 5.3 *Other planning-based methods*

In the DL/P method we tried to address another set of design problems: how to manage the data that flows within an integration solution. We temporarily left out other aspects, like throughput, availability, service deployment, and so on.

In a way similar to the ML/P method, as its input this method expects a *data flow specification* using input/output characterization of services in the form shown in Table 3. As an illustrative case study we use a scenario that is graphically shown in Figure 14. However, the method also needs to know relationships between data elements that flow within the solution; therefore it expects a data model as part of its input. In our case study we use the model shown in Figure 15.

The method then tries to arrange data manipulation services (Splitter, Aggregator, Content Filter, Message Translator, and services for copying and moving data elements within messages), other integration services (Content-Based Router, Recipient List), as well as ordinary business services (e.g. CheckCredit) into a comprehensive integration solution, so that each service gets its input and provides its output as needed.

An output corresponding to the above mentioned sample integration problem is shown in Figure 16.

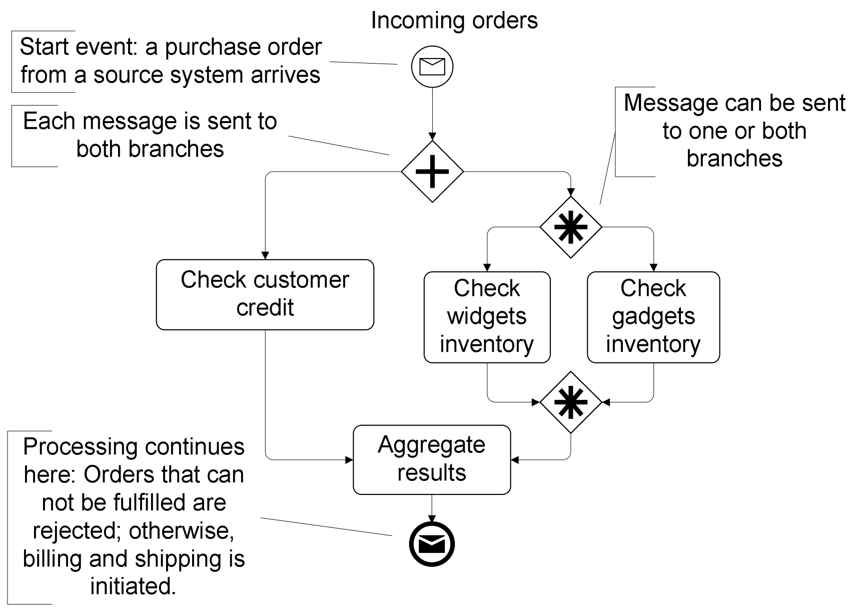

Incoming orders
Start event: a purchase order from a source system arrives
Each message is sent to both branches
Message can be sent to one or both branches
Check customer credit
Check widgets inventory
Check gadgets inventory
Aggregate results
Processing continues here: Orders that can not be fulfilled are rejected; otherwise, billing and shipping is initiated.

**Figure 14.** An example of required flow of data – an input for the DL/P method.

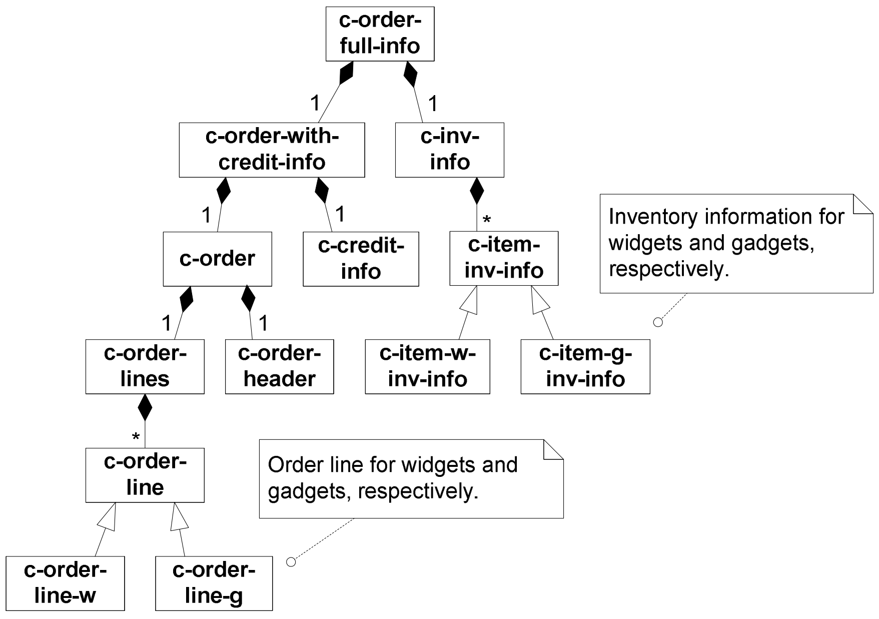


**Figure 15.** An example data model – an input for the DL/P method.

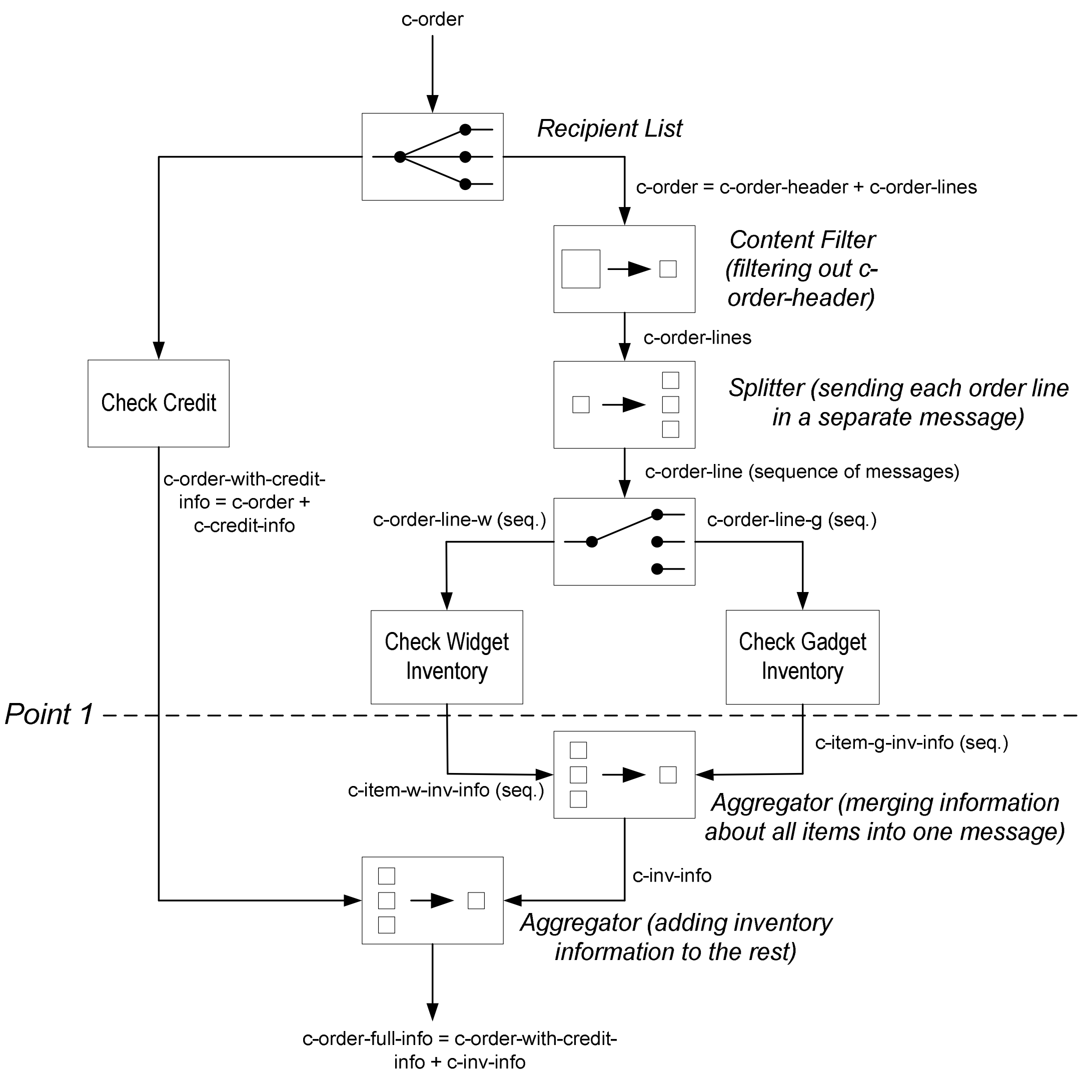

c-order
Recipient List
c-order = c-order-header + c-order-lines
Content Filter (filtering out c-order-header)
c-order-lines
Check Credit
Splitter (sending each order line in a separate message)
c-order-line (sequence of messages)
c-order-with-credit-info = c-order + c-credit-info
c-order-line-w (seq.)
c-order-line-g (seq.)
Check Widget Inventory
Check Gadget Inventory
Point 1
c-item-g-inv-info (seq.)
c-item-w-inv-info (seq.)
Aggregator (merging information about all items into one message)
c-inv-info
Aggregator (adding inventory information to the rest)
c-order-full-info = c-order-with-credit-info + c-inv-info

**Figure 16.** An example integration solution created by the DL/P method.

In order for this method to work, the state of the world should describe the content of messages in the channels in more detail than the ML/P method did, i.e. using the *VariablePresence* and *VariablePosition* functions instead of the *Content* function. We have chosen the following representation: If messages in a flow identified as `flow-id` contain data of types `content-type`$_i$ stored in message parts `message-part`$_i$, then the following literals would be present in the description of the state of the world:

```
(data content-type₁ message-part₁ flow-id)
(data content-type₂ message-part₂ flow-id)
                .
                .
                .
(data content-typeₙ message-partₙ flow-id)
```

For example, if considering the point "Point 1" in the solution shown in Figure 16, the state of the world would be the following:

```
   (data c-order part-0 flow-1)
   (data c-credit-info part-other flow-1)
   (data c-order-line-w part-0 flow-2)
   (data c-order-line-g part-0 flow-3)
```

As we can see from the description of state of the world, for its manipulation we need to have tools that are more powerful than those needed in the ML/P method. For example, in the ML/P method we model the effects of Recipient List component (the one that creates a clone or clones of an input message flow) in the following way. (We show the most relevant parts in bold. For simplicity we have omitted aspects of message formats, monitoring and ordering.)

```
(:action RecipientList_PL1
 :parameters (?cnt – t-content ?chan – t-channel ?flow – t-flowid)
 :precondition
   (and
     (message ?cnt ?chan ?flow)
     (acceptable_input_channel_for_PL12 ?chan)
   )
 :effect
   (and
     (not (message ?cnt ?chan ?flow))
     (message ?cnt ch_m12 flow_1)
     (message ?cnt ch_m12 flow_2)
    )
)
```

In the DL/P method we have to work with more literals at once using quantifiers (shown in bold):

```
; copies entire 'flow' to 'flow-new'
; - as a precondition, 'flow' is not empty, 'flow-new' is empty
; - as an effect, everything present in 'flow' is duplicated in 'flow-new'

(:action RecipientList
 :parameters (?flow ?flow-new - t-flowid)
 :precondition
  (and
    (not (= ?flow ?flow-new))
    (exists (?cnt - t-content ?p - t-part)
            (data ?cnt ?p ?flow))
    (not (exists (?cnt - t-content ?p - t-part)
                 (data ?cnt ?p ?flow-new)))
  )
 :effect
  (forall (?cnt - t-content ?p - t-part)
          (when (data ?cnt ?p ?flow)
                (data ?cnt ?p ?flow-new))))
)
```

The quantifiers are available in the ADL feature of the PDDL (Fox and Long, 2003). This feature is not widely supported, so the set of suitable existing planners was rather small. Based on our previous experiences, for the method prototype implementation we have selected the FF planner (Hoffmann and Nebel, 2001).

We have evaluated the prototype using variants of the Widgets and gadgets order processing scenario such as the one shown in Figure 14. The method has been able to find a solution, although in some cases we had to manually adjust the method’s options to help the planner to construct the plan in reasonable time. It seems that using currently available planners this method is able to solve only relatively simple problems.

Although we originally intended to optimize this method and find a more suitable planner to solve planning problems it generates (or to create a specific planner for it), we have chosen to employ constraint programming instead, as is described in Chapter 1.

For completeness we should mention one more method here: We have developed a way to construct *service interface adapters* that are able to adapt selected attributes of a service interface, namely message content, format, transport protocol, authentication and confidentiality mechanisms. We have used a simple encoding of the state of the world using a set of predicate symbols corresponding to the above attributes, with some additions – predicate symbols for message validation and authorization, and functions for message rate and total cost.

Individual integration services that could be used for the construction of an adapter were again characterized by their preconditions and effects, for example the validation service could require an input message to be in XML format, it could process up to 400 messages per minute, and (on average) rejected 4% of the input messages.

Planning problems that were generated by this method required the numeric extensions of PDDL, i.e. PDDL version 2.1 Level 2 (Fox and Long, 2003). For an evaluation we have used LPG (Gerevini, Saetti, and Serina, 2003) and JSHOP2 (Nau, Au, Ilghami, Kuter, Murdock, Wu, and Yaman, 2003) planners. (As for the latter, although being a hierarchical task network planner, we have used it in a mode that emulates action-based planning.) Concerning other planners, even if they declared they can solve numerical domains, they did not support our problems – for example, Metric FF planner could not work with operators that had non-constant effects on the metric value.

In a way similar to the ML/P method, this was an attempt to formalize some aspects of integration solution design along with a preliminary evaluation. Though, instead of continuing to more depth here, we moved to work with messaging-based integration solutions. More information on the service interface adaptation method can be found in (Mederly, Lekavý and Návrat, 2009).

## *5.4 Planning-based methods: a conclusion*

Methods that we have implemented demonstrate that it is possible to use action-based planning to solve design problems in the domain of messaging-based integration solutions.

We have identified several issues that require further attention:

1. Our current methods do not scale well with regards to problem size and/or the number of aspects employed: As we have shown, increasing the number of properties of services and channels as well as increasing the number of integration problem objects (e.g. message content types) leads to a significant increase of the state-space size, which presents a problem for many of currently available planners, and, generally, makes finding a plan difficult.

   There are some possibilities of how to cope with this issue, though. One of them could be using domain-specific knowledge, as suggested e.g. by Kautz and Selman (1998) or Bacchus and Kabanza (2000). As an example, authors of the latter work have proposed a domain-dependent search control mechanism based on first-order temporal logic representation of the search control knowledge. They have shown a significant improvement (multiple orders of magnitude) of execution time for some of the classical planning problems, in comparison to domain-independent planners.

   Yet another way of overcoming the issue of planning complexity would be to partition the problem of finding an integration solution into smaller subproblems, and solve them in a sequence – in a way similar to the approach described in Section 6.1.3.

2. The notion of "optimal" solution is quite coarse. Majority of existing planners either do not support action costs at all, or these cost values have to be constant. In the first case, the only optimization criterion is the plan length, i.e. the number of components used. In the second case, we are able to do a limited optimization (i.e. optimization taking into account component cost aspect only), at an expense of significantly slower computation.

   Nonetheless, action costs seem to be sufficient to optimize commonly used metrics (e.g. component cost, or the number of messages or bytes going through communication middleware), so they are probably adequate for the practical purposes – albeit limited to optimizing in one dimension (at a given time) only. What remains to be solved, is the time complexity, as described above.

Overall, we have shown that planning presents a possible approach to integration solutions design. Although there are some open issues, we believe it would be possible, after further research, to overcome them. However, in the time available to work on this dissertation we have decided to explore also a entirely different approach to problem solving, namely the constraint satisfaction.

# 6 Methods using constraint programming

In this section we describe two methods, namely ML/CP and U/CP, which use constraint programming in order to create messaging-based integration solutions.

These methods are based on a transformation of an integration problem into a constraint satisfaction problem (CSP) in such a way that a solution of the CSP can be transformed back into a solution for this integration problem (see Figure 17).

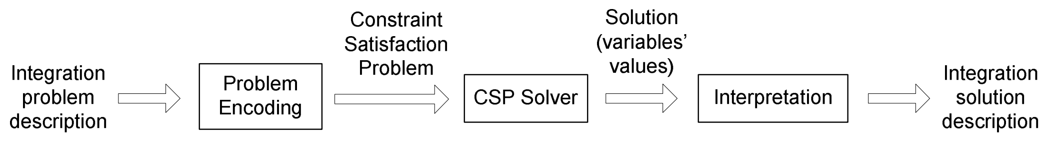

Integration problem description
Problem Encoding
Constraint Satisfaction Problem
CSP Solver
Solution (variables' values)
Interpretation
Integration solution description

**Figure 17.** Basic principle of the methods based on constraint programming.

Principles of the transformation are the following:

1. Given the integration problem, we create a skeleton of the solution graph (for a description of a solution graph please see Section 4.2.1).
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 node, is represented by one or more variables (exceptions are explained below).
3. Each design rule is represented using one or more constraints over respective CSP variables.

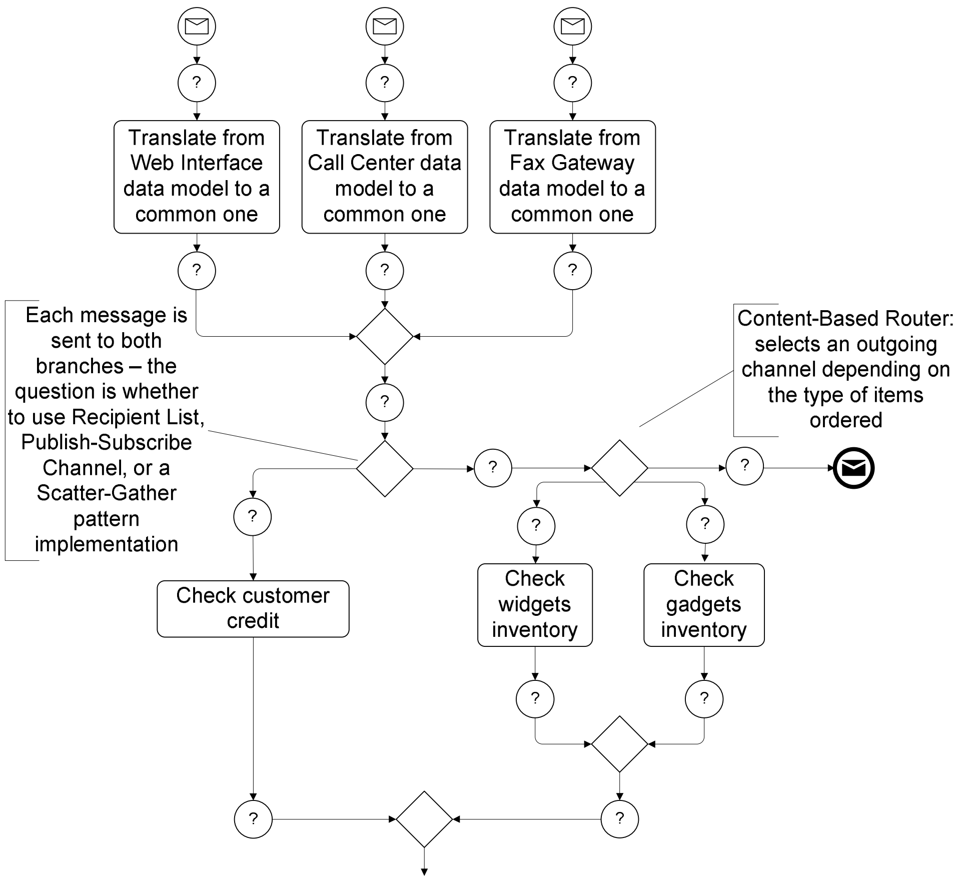
Let us now cover these principles in more details. First, how do we create a skeleton of the solution graph, based on known abstract design and non-functional requirements?

Our 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.

Therefore, the initial skeleton of the solution graph is created as a copy of the control flow graph.

We are not finished yet, though. Besides business services and integration services that implement control constructs (fork, join, decision, and so on), an integration solution contains other integration services as well (e.g. Wire Tap, Resequencer, Data Manager). The problem is that we do not know in advance how many of these services will be needed, and therefore how many vertices the solution graph should contain. In the case of planning-based methods this was not an issue, as the solution graph has been constructed by a planner. However, now we have to create it beforehand. The way out is to insert a sufficient number of empty slots that will potentially contain integration services. Concretely, we replace each edge in the control flow graph by a user-defined number of slots for integration services connected by messaging channels.

After applying this rule, the model shown in Figure 9 would be translated into an integration solution graph, part of which is presented in Figure 18. 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.



?
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 – the question is whether to use Recipient List, Publish-Subscribe Channel, or a Scatter-Gather pattern implementation
Content-Based Router: selects an outgoing channel depending on the type of items ordered
Check customer credit
Check widgets inventory
Check gadgets inventory

**Figure 18.** A fragment of an integration solution sought for.

As we have already stated, the whole process looks like this: the integration problem is transformed into a constraint satisfaction problem (CSP), which is then solved by a CSP solver. After finding a CSP solution, it is interpreted as an integration solution that we were searching for (see Figure 17).

In fact, the method allows us to use more iterations and a user involvement in this process. After getting a CSP solution (or, actually, even during the solving process), the method – either by itself or as instructed by user – can update the CSP (or create it anew) and use the solver to find its solution again. This cycle can repeat while needed.

Moreover, the user is able to select heuristics used to speed up the solution process, as described in Section 6.1.3. The situation then looks like the one shown in Figure 19.

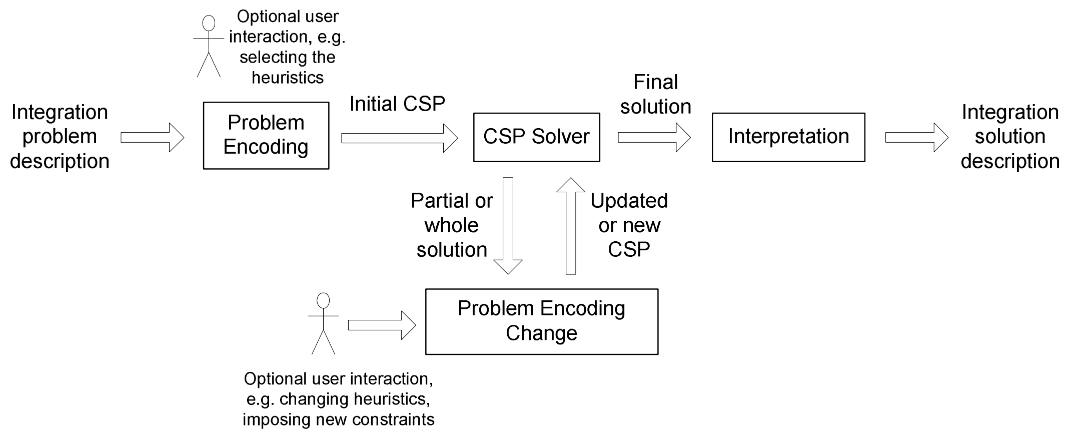

Optional user interaction, e.g. selecting the heuristics
Integration problem description
Problem Encoding
Initial CSP
CSP Solver
Final solution
Interpretation
Integration solution description
Partial or whole solution
Updated or new CSP
Problem Encoding Change
Optional user interaction, e.g. changing heuristics, imposing new constraints

**Figure 19.** Iterative use of constraint programming in our methods.

In the following we describe the details of the U/CP method. We have chosen this one, because it is basically an enhanced version of its predecessor ML/CP.

## *6.1 The U/CP method details*

### 6.1.1 Input of the U/CP method

This method allows the developer to specify a control flow between solution components more precisely than previous methods; the following constructs can be used: (In Figure 20 they are shown using UML activity diagram fragments, as well as using one of textual domain-specific languages (DSLs) developed to serve as an input for U/CP method.)

1. *sequence* – a developer can indicate that services $S_1$, $S_2$, ..., $S_n$ have to be executed in a sequence (see Figure 20a);

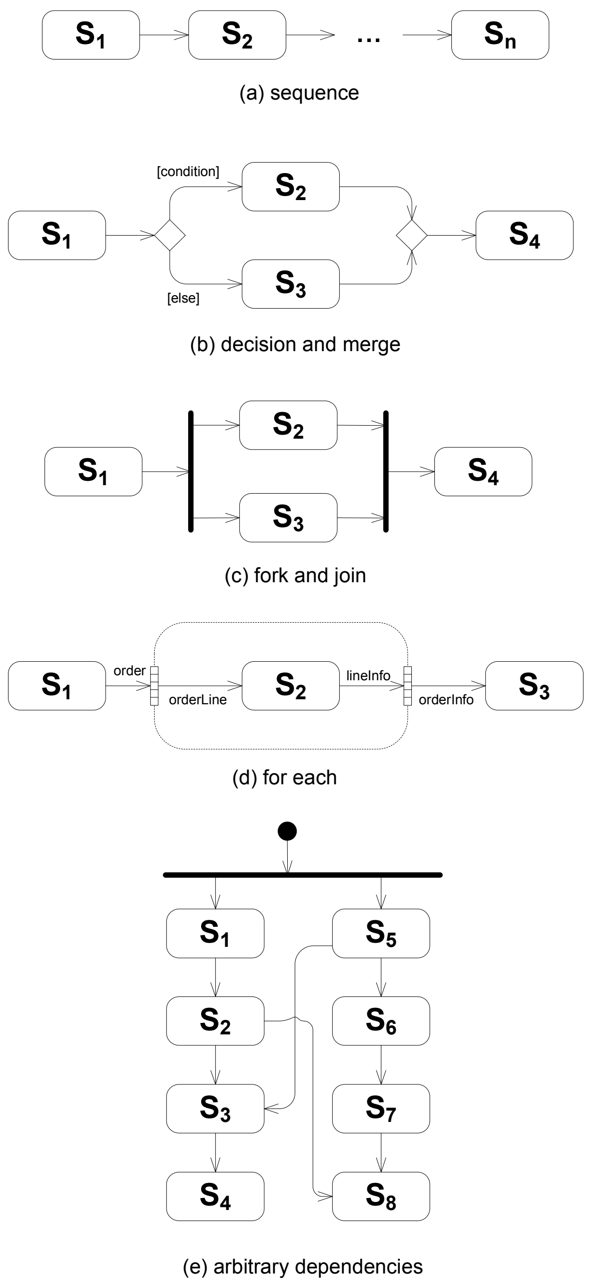
2. *decision node* – it denotes the fact that the flow of control continues using one (or more) of outgoing edges; they can be then merged back in a *merge node* (Figure 20b). Please note that the `xpath` keyword in the textual DSL denotes the language used for specifying the condition (in this case it is XPath);

3. *fork*, optionally with *a join* – these constructs are interpreted in such a way that the flow of control continues using all outgoing edges, and is (optionally) synchronized at a join node (Figure 20c) – this construct corresponds to the Scatter-Gather pattern (Hohpe and Woolf, 2004);

4. *for each* – services specified in a subsequence have to be executed once for each part of specified input variable. At the end of a subsequence, the specified output variables might have to be merged together and passed downwards as a new variable (Figure 20d); see also Composed Message Processor pattern (Hohpe and Woolf, 2004). In this case, after executing $S_1$,

the `order` has to be split into individual `orderLines` and for each one $S_2$ has to be executed. Resulting `lineInfo` values should be aggregated into `orderInfo` value. Then $S_3$ should be run. There is also a possibility of omitting the synchronization point at the end – in that case the whole process would finish inside the subsequence;

5. *arbitrary control flow dependency between services* – a developer can specify that a service $S_j$ can start only after service $S_i$ had finished its processing (Figure 20e).



```
S1();
S2();
// ...
Sn();
```

```
S1();
if (xpath("condition"))
  S2();
else
  S3();
S4();
```

```
S1();
fork-and-join
{
  S2();
  S3();
}
S4();
```

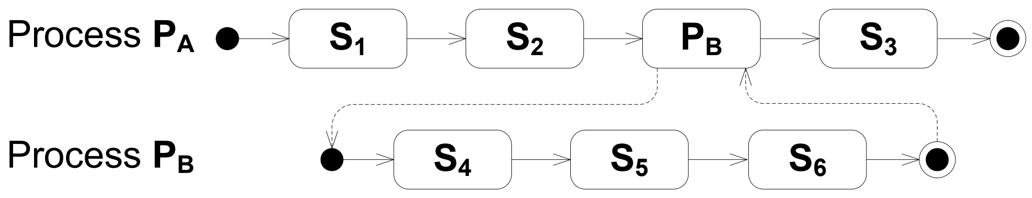
```
S1();
for-each (orderLine in order)
{
  lineInfo = S2(orderLine);
}
aggregate (lineInfo into orderInfo);
S3();
```

```
fork
{
  {
    S1();
    S2();
    wait-for "S5";
    S3();
    S4();
  }
  {
    S5();
    S6();
    S7();
    wait-for "S2";
    S8();
  }
}
```

**Figure 20.** Main elements of abstract control flow between services in the U/CP method.

Please note that, except for the for-each case, we have omitted the data flow description from our example, in order to concentrate on the control flow description.

Moreover, this method allows structuring an integration solution into a set of *processes*. Each process has its own control and data flow specification. Processes can be connected using an *execute process* control construct that has the semantics of "invoking" the child process from the parent one. In Figure 21 we can see an example of process $P_A$ invoking process $P_B$, again using UML as well as the textual DSL. In this case an integration solution has to execute services in the following order: $S_1$, $S_2$, $S_4$, $S_5$, $S_6$, $S_3$. In current version of the method there cannot be cycles in process invocation. (In future we plan to relax this restriction. However, also today it is possible to model cycles in process invocation graph using communication via named messaging channels.)



```
process Pa()            process Pb()
{                       {
  S1();                   S4();
  S2();                   S5();
  call Pb();              S6();
  S3();                   return;
}                       }
```

**Figure 21.** An example of subprocess invocation in the U/CP method.

Another important aspect of this method is that we have decoupled data flow from the control flow: a developer declares a set of *variables* in a process and then specifies how these variables are used, usually as input and output parameters of service and process invocations, and in some control constructs (decision, for each). The method then tries to determine what variables will have their values carried within individual message flows, and at what positions – in message header, body, or attachments.

In current version of the method we have decided the variable scope to be one process. Processes exchange data using formal and actual parameters, in a way similar to the traditional programming model.

An example of control and data flow specification represented using UML activity diagram is shown in Figure 22. The same specification written using our textual DSL is shown in Figure 23.

This diagram shows an extended version of Widgets and gadgets orders processing scenario originally introduced in Section 4.1 (see Figure 9).

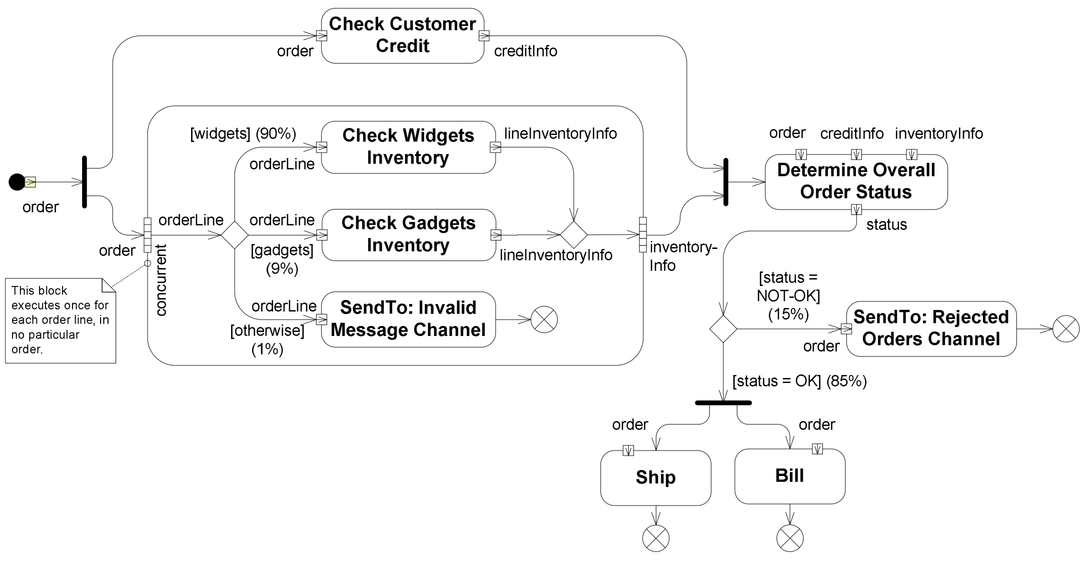


**Figure 22.** An example of specification of control and data flow for U/CP method, using UML.

```
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($orderLine//wg:ProductId,1,1) = 'W'") ratio 0.9:
            lineInventoryInfo = CheckWidgetInventory(orderLine);
          case xpath("substring($orderLine//wg:ProductId,1,1) = '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 23.** An example of specification of control and data flow for U/CP method, in the textual form.

The version of the scenario presented here differs from the original one in that it has been complemented by the specification of data flow between services. In textual DSL representation these are self-explanatory; in UML activity diagram they are expressed in the form of input and output *pins* of individual activities. Each pin is annotated by a name of variable or variables that should be present as an input or an output of the service. (So, for example, service `CheckCustomerCredit` takes a variable `order` and produces a variable `creditInfo`. Service `ComputeOverall-Status` takes variables `order`, `creditInfo`, and `inventoryInfo` and produces a variable `status`.)

The scenario differs from the original one also in that it allows an order to contain order lines of more types (for both widgets and gadgets). Therefore, the integration solution has to split an order into individual lines corresponding to products ordered, evaluate the availability of each product individually, and then join the resulting information together. This split-and-join feature was not present in ML methods.

On the other hand, in the U/CP method we expect each process in an integration solution to have exactly one input. That is why the above mentioned process starts in a place where orders coming from three sources are put together (marked as Point A in Figure 10 at page 47). In order to model complete original integration scenario we would have to add three auxiliary processes that would implement initial processing of messages coming from three sources up to their conversion to the common data model; each of these processes would then invoke the main process (`ProcessOrder`).

### 6.1.2 Finding integration solutions by the U/CP method

Let us describe how the U/CP method finds integration solutions in more detail. In this section we discuss the basic version of the method as shown in Figure 17 at page 75. In the following section we will concentrate on the iterative use of constraint satisfaction problem solving along with user interaction (Figure 19 at page 77).

**Algorithm 1.** Finding an integration solution in the U/CP method (basic version).

```
FindIntegrationSolution()
begin
  CreateControlFlowGraph();
  CreateSlotsForIntegrationServices();
  CreateAuxiliaryDataStructures();
  CreateCSP();
  SolveCSP();     // in parallel thread: DisplaySolutions();
end.
```

`CreateControlFlowGraph` transfers block-structured specification of the control flow like the one shown in Figure 24a into unstructured (graph-oriented) form of a directed acyclic graph shown in Figure 24b. As described at the beginning of this chapter, each edge of this graph will be ultimately transformed into a channel – so, basically, what we see here is the skeleton of the future integration solution.

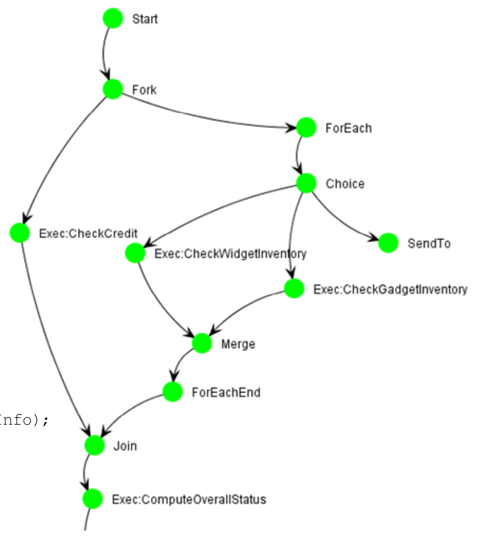
Vertices of this graph correspond to:

1. service invocations (vertex type Execute and, in later phases, IntService),
2. individual control flow constructs (vertex types Choice, Merge, ForEach, ForEachEnd, Fork, Join),
3. other actions (vertex types Start, End, SendTo, NoOp): Start and End are auxiliary vertices per definition of integration solution graph in Section 4.2, SendTo and NoOp are services (although in the code generation phase they can be implemented in a way different from regular services).

Arbitrary dependencies (WaitFor constructs, see Figure 20e) are converted into Fork – Join vertices pairs.

```
process ProcessOrder(Order order)
{
  fork-and-join
  {
    {
      Credit creditInfo = CheckCredit(order);
    }
    {
      for-each (Line orderLine in order) using "..."
      {
        LineInventoryInfo lineInventoryInfo;
        exclusive choice
        {
          case xpath("... = 'W'") ratio 0.9:
            lineInventoryInfo = CheckWidgetInventory(orderLine);
          case xpath("... = '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);
```

(a) block-structured specification of the control flow


Start
Fork
ForEach
Choice
SendTo
Exec:CheckCredit
Exec:CheckWidgetInventory
Exec:CheckGadgetInventory
Merge
ForEachEnd
Join
Exec:ComputeOverallStatus

(b) graph-oriented skeleton of an integration solution

**Figure 24.** An example of a transformation from block-structured specification of the control flow into graph-oriented skeleton of the integration solution (the U/CP method).

The second step in Algorithm 1, namely `CreateSlotsForIntegrationServices`, represents replacing each edge in the graph by a sequence of edges and vertices of type IntService (integration service). The number of integration services that should be used is configurable; usually, the default value of 1 is sufficient.

At this point we have created an integration solution graph as defined in Section 4.2. The only two differences are that (1) some vertices will not be used in the final solution (those of Integration Service type that the method will find unnecessary), and (2) in the final solution some more edges will be present (see the Data flow issue). And, of course, we do not know the values of service and channel properties.

`CreateAuxiliaryDataStructures` deals with pre-computing some properties of the solution, for example:

1. message rates that are expected to be present in individual channels,
2. process variables that could be (potentially) present in individual channels,
3. solution regions that (potentially) need to be interconnected by additional channels needed to transport process variables (see Data Flow design issue),
4. pairs of components that could not be interconnected by Split-and-Join services (as these should be well-nested).

`CreateCSP` is the core of the method; here we create CSP variables and then impose constraints over them. We create a set of CSP variables for:

1. each solution graph vertex, i.e. business or integration service or auxiliary component,
2. each solution graph edge, i.e. a channel,
3. some other entities, e.g. servers hosting service containers or potential channels (see data flow design issue below).

Variables and constraints creation is done by traversing the integration solution graph (in no particular order) and creating these entities as defined by the method.

In this step we also add supplementary user-specified constraints to the CSP.

`SolveCSP` is done by executing a selected CSP solver. The solver we have used looks for the optimal solution iteratively: after finding a solution with a cost C, it adds a constraint “cost < C” and continues with finding other solutions. So, the solver produces a sequence of solutions with gradually decreasing costs. This is very convenient for us, as we can present the developer a solution as soon as it is found, without knowing that other (better) solutions exist. He or she can decide to accept a solution or to wait for other ones or for a conclusion that no better solutions exist.

As soon as any solution is found, it can be displayed to the developer (`DisplaySolutions`). After examining a solution, the developer can choose to generate executable code for the specific integration platform (or to store a solution for later use, but these details are not important here).

Now, let us describe how the method creates CSP variables and constraints.

First of all, we have divided the scope of the method into individual *design issues*. A design issue is a more or less independent aspect that a designer would take into account when creating an integration solution.

We distinguish between two kinds of design issues. The first category comprises issues that directly influence the appearance of an integration solution – for example, the Channel types issue deals with selecting an appropriate type for each of channels. In the following we call them structural design issues, or design *aspects*. They approximately correspond to issues we have mentioned in Section 4.2. Into the second category belong those issues that provide information about properties of the solution – for example, the Throughput issue indicates how many messages per time unit can be processed. These will be called *metrics*.

Among design aspects there are:

1. *General issues:* these are issues that need to be tackled in design of any integration solution. The main one is: how to implement control structures (for each, fork, join, and so on) and what integration services to use.

   *Solution graph representation:* Each of these control structures is represented by one or two vertices in the solution graph. (Structures having a start and end nodes, i.e. fork with join, for-each with for-each-end, choice with merge are represented by two vertices, others by one vertex.) Each place for an integration service is represented by a vertex as well.

   *CSP mapping realization:* We use a CSP variable `V.ImplementationType`[13] analogous to the function *Service*: $V \rightarrow$ *Services*. We define such variable for all vertices that have an ambiguity with respect to their implementation, i.e. for majority of control structures and integration services. A domain of this variable is a set of all possible implementations of a given vertex determined by the target platform, by constraints stated by the designer, and by the other design aspects we are currently dealing with. (For example, the "fork" part of the fork-and-join construct can be implemented using Recipient List, Split and Join Parallel[14], or Publish/Subscribe Channel. These options are mapped to three values in the domain of the corresponding CSP variable. However, the last value is available only when we are dealing with Channel types aspect – otherwise, only the first two values are present.)

2. *Message content* – what process variables will be transported in physical messages flowing in individual channels?

   *Solution graph representation:* Using a function *VariablePresence:* $E \times$ *Variables* $\rightarrow$ *Boolean.*

[13] CSP variables are show in `Courier` font and are almost always related to some object – typically to a vertex or an edge of the solution graph. We show this fact by using a notation `V.VariableName` or `E.VariableName`.

[14] Progress Sonic ESB-specific component implementing the Scatter-Gather integration pattern.

*CSP mapping realization:* For each edge we create a vector of CSP variables denoting whether a given process variable is being transported in messages in the corresponding channel. We assume a Datatype Channel pattern is used, i.e. that messages in this channel are of the same type – each message contains exactly the same process variables. So, for each edge `E` we create a vector `E.VariablePresence[Var]` that has values of 1 for variables (`Var`) that are carried in messages flowing through `E`, and values of 0 for variables that are not.

3. *Positions and formats:* where exactly to put each variable in messages? In many platforms, there are three basic possibilities: to a message body, to a message header, or to a message attachment. This design decision has some consequences with respect to e.g. preserving the variable value in some situations, as described below.

   *Solution graph representation:* Using a function *VariablePosition:* $E \times$ *Variables* $\times$ *Positions* $\rightarrow$ *Boolean.*

   *CSP mapping realization:* In a way similar to the mapping for variable presence, for each edge `E` we create a vector of CSP variables `E.Variable-Position[Var,Pos]` denoting whether a given process variable `Var` is being transported in messages in this channel at a position `Pos`. Positions are platform-specific; for Apache Camel we use “header”, “body”, “attachments” while for Sonic ESB we use “header”, “bodypart”, “context” with an additional variable indicating which of the process variables is known to reside in a special bodypart numbered 0, if any.

   Domain for these CSP variables is { 0, 1 }; however, if we take into account alternative data formats for one process variable (e.g. CSV, XML, fixed-length, and so on for a variable of type RecordList or XML, DOM, java.util.Map, and so on for a variable of type Map) the domain is defined as { 0, 1, 2, ..., *f* } where *f* is the number of possible data formats for this particular variable.

4. *Data transformations:* in order to make integration solution specification concise it is sometimes useful to be able to leave out auxiliary data transformation services from the model. The method then solves their positioning for the developer.

   *Solution graph representation:* Using places for integration services.

   *CSP mapping realization:* We extend the domain of integration services’ types to contain one value for each of possible transformation services and bind these values to `VariablePresence` and/or `VariablePosition` CSP variables using appropriate constraints.

5. *Data flow* is about ensuring that data are present at places where they are needed. For an example please consider Figure 25a: if we have a sequence of services $S_1$ and $S_2$, where $S_1$ requires variable $v_1$ and produces variable $v_2$, and $S_2$ requires both variables $v_1$ and $v_2$, how to achieve this? If we would use a naive approach to message-oriented communication, the message entering $S_1$ would contain variable $v_1$ and message exiting it would contain variable $v_2$. If this message flow would be directly connected to $S_2$, this service would be unable to find value of $v_1$ in the input message.

   The solution is either to use an auxiliary data manipulation service DM to put $v_1$ into some place of message that is preserved (i.e. copied from input to output) by $S_1$, as shown in Figure 25b. Many services do keep intact values stored in message header and/or in message attachments. If that is not possible, we should send value of $v_1$ via separate message flow with the help of fork-and-join construct, as displayed in Figure 25c.

   *Solution graph representation:* In current version of the method we solve this problem outside solution graph (as described below).

   *CSP mapping realization:* We create a network of possible data flow connections between candidate vertices. As candidate vertices we consider all integration services and all fork and join points. For each such possible connection one CSP variable (`ConnectionPresence`) is created. This variable indicates whether this connection is used and if so, how exactly. Technically, these variables are assigned to a candidate flow destination vertex and indexed by candidate flow source component: `Dest.ConnectionPresence[Source]` (Dest = destination vertex, Source = source vertex). These variables can have a value of "none" (no connection), or one value for each possible implementation – in current version these are "Splitter-Aggregator" (general) and "Split and Join Parallel" (specific for Sonic ESB).

6. *Channel types:* what channel type to choose for individual message flows? Basic possibilities are: in-memory, MQ topic, MQ queue, as discussed in Section 4.2.

   *Solution graph representation:* Using a function *ChannelType*: $E \rightarrow$ *ChannelTypes*.

   *CSP mapping realization:* For each edge `E` we create a CSP variable `E.ChannelType` that has a domain of all channel types possible for a given platform.

7. *Threads:* how to deploy individual services – into what containers and into how many threads in each? This is crucial to achieve goals in the area of throughput and availability.

*Solution graph representation:* Using a function *Deployment*: $V \rightarrow \{ (t_1, t_2, ..., t_n) \mid t_i \in TC_{v,i} \}$ (see Section 4.2).

*CSP mapping realization:* For each vertex `V` we create a vector of CSP variables `V.ThreadsPerContainer[Cont]` indicating the number of threads in which the respective component executes in the container `Cont`.

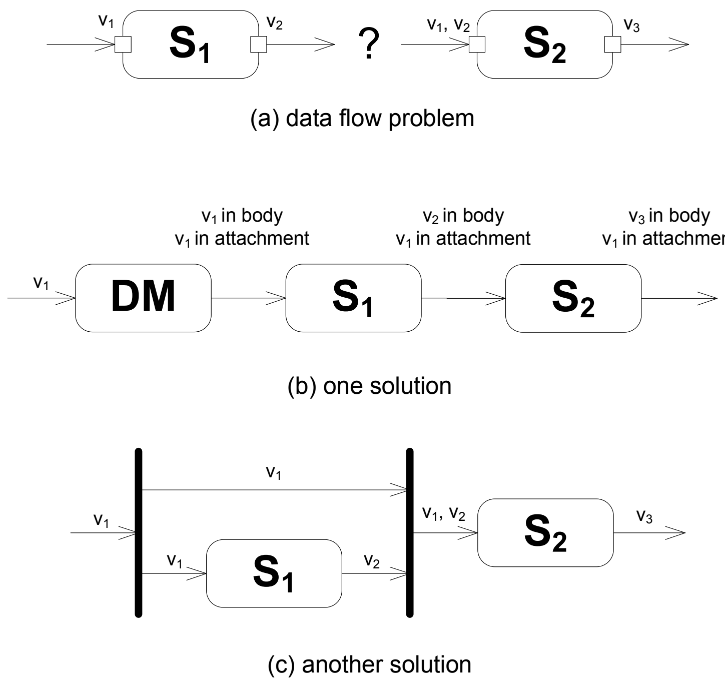


**Figure 25.** An example of a problem and its solutions for the data flow design aspect.

8. *Containers:* a "lightweight version" of the preceding aspect – instead of capturing the exact number of threads per container, here we only work with the total number of threads, containers and hosts.

   *Solution graph representation:* Using functions *ThreadCount*: $V \rightarrow TC$, *ContainerCount:* $V \rightarrow CC$ and *HostCount:* $V \rightarrow HC$ (see Section 4.2).

   *CSP mapping realization:* For each vertex `V` we create the following CSP variables: `V.ThreadCount`, `V.ContainerCount`, and `V.HostCount`, representing the total number of threads, containers, and hosts, respectively, in which this component executes. If using along with the Threads design issue, these variables are computed from vectors `V.ThreadsPerContainer[Cont]`.

9. *Monitoring:* how to ensure monitoring of message flow at certain points of the integration solution?

   *Solution graph representation:* Using a function *Monitored*: $E \rightarrow$ *Boolean* and *MonTransparent*: $V \rightarrow$ *Boolean.*

*CSP mapping realization:* In a way similar to ML methods, each edge `E` is assigned a variable `E.Monitored` that specifies whether the respective channel is monitored by a Wire Tap upstream (value `WT_UP`), or it should be monitored by a Wire Tap downstream (value `WT_NEEDED`), or it need not be monitored by Wire Tap at all (value `NO_NEED`). This differs from a mathematically abstract point of view presented in Section 4.2 in order to make CSP solving more efficient. Also, the *MonTransparent* function is regarded directly in the constraints, and is not modeled using CSP variables.

10. *Message ordering:* how to ensure that messages arrive at specified point of the integration solution in the same order as they were at other specified point?

    *Solution graph representation:* Using functions *Ordered*: *E → Boolean* and *OrderMarked: E → Boolean*.

    *CSP mapping realization:* Again, in a way similar to ML methods, each edge `E` is assigned a variable `E.Ordered` that specifies whether the message flow in this channel is ordered with respect to a defined point in the solution, or not. We have simplified the situation in such a way that we assume each message has a natural sequence number (e.g. order identification number) already present. However, it would be easy to include explicit `E.OrderMarked` variable in this method as well, if necessary.

11. *Checkpoints:* in case of service container failure the processing would restart from last checkpoint – usually the place where the message flow is taken from the messaging middleware. The method allows us to specify concrete places or wider intervals that should contain a checkpoint.

    *Solution graph representation:* Using a function *Checkpointed*: *E → Boolean*.

    *CSP mapping realization:* For each edge `E` there is a variable `E.Checkpointed` that specifies whether this channel will serve as a checkpoint.

Concerning metrics, the current version of the method supports these ones:

1. *Number of messages and/or bytes transferred through messaging middleware:* as a throughput of message broker is limited, this is a usual measure we want to optimize. In-memory channels are generally preferred over MQ-based ones in situations where their functionality suffices.

2. *Number of bytes transferred in all channels:* when deciding about the placement of process variables in channels we usually want to optimize the total number of bytes transferred.

3. *(Weighted) number of integration services used:* generally we want to design simple solutions, i.e. solutions that contain as few components as possible. One of the reasons is the maintainability of such solutions, although in an ideal

situation the developer would not need to work with generated solution in any way. Another reason is that in the current version of the method we work with the CPU usage of business services only – but integration services have an impact on CPU as well, so it is reasonable to minimize their use.

4. *Throughput:* we can make an estimate of the overall throughput of the integration solution provided we know approximate values of throughput of individual services in concrete deployments (containers/threads). We can then use this value for specifying requirements, typically in the form: “solution throughput must be greater or equal to a given value”.
5. *Host load:* in a way similar to throughput estimation, we can make an attempt to predict the load that the integration solution would impose on individual computers hosting service containers. Then we can formulate constraints like e.g. “no host should be utilized to more than 10% of its capacity”, or “all hosts should be utilized in a balanced way”.
6. *Number of threads in containers:* we could try to minimize this value in some situations, but we do not expect this metric to be as important as previous ones.

*CSP mapping realization:* For these metrics we usually create appropriate CSP variables, like `Host.HostLoad` for each host whose load we want to measure. Then there is a general `Cost` variable that is bound to be equal to a user-specified weighted sum of metric variables.

This list of issues can be further extended, for example by adding elimination of duplicate messages (implemented in the ML/CP method), measuring and optimizing message processing latency, using product-specific features for fault tolerance, like backup containers in Sonic ESB, and so on. Another list of candidate design issues is mentioned at the end of Section 6.4.1 where we summarize experiences of applying the method to a set of real-life integration problems.

In Table 8 we summarize major types of CSP variables used currently in the U/CP method, arranged according to a primary design issue they are related to. Please note that CSP variables can be shared between design issues, e.g. `ImplementationType` of a vertex (issue General) is used virtually in any other issue. Likewise, `ThreadsPerContainer` variables are used in the Channel types, Threads, Containers, Throughput, Host load, and Threads in containers issues.

Please also note that some design issues do not bring any new CSP variables (at this level of abstraction), because they just use existing ones. For example, issue “Component cost” just adds a connection between vertices’ `ImplementationType` variables and the total cost of the solution.

*Design rules* described in Section 4.2 are translated into constraints over these variables. In principle, these constraints are very similar to the design rules; their formulation is influenced by the need to efficiently solve the CSP.

**Table 8.** The most important CSP variables used in the U/CP method.

| Design issue | Edge | Vertex | Other |
|---|---|---|---|
| General | - | ImplementationType | |
| Message content | VariablePresence[Var] | - | |
| Positions & formats | VariablePosition[Var, Pos] | - | |
| Transformations | - | - | |
| Data flow | - | ConnectionPresence[V] | |
| Channel types | ChannelType | - | |
| Containers | - | ThreadsCount, ContainersCount, HostsCount | |
| Threads | - | ThreadsPerContainer[C], ThreadsPerHost[H] | |
| Monitoring | Monitored | - | |
| Ordering | Ordered | - | |
| Checkpoints | Checkpointed | | |
| Messages/bytes | MessagesMQCost, BytesMQCost, BytesCost | - | |
| Host load | - | LoadPerHost[H] | H.HostLoad |
| Throughput | - | Throughput | - |
| Threads in containers | - | - | - |
| Component cost | - | - | - |

Columns meaning: The *Design issue* column contains the design issue to which the information about CSP variables is related. *Edge* and *Vertex* columns contain CSP variables pertaining to this issue, attached to solution graph edges and vertices, respectively. *Other* column contains other CSP variables, i.e. those that are neither attached to edges nor to vertices. Use of brackets represents a vector of CSP variables, e.g. `VariablePresence[Var]` denotes the situation that `VariablePresence` is a vector of CSP variables indexed by process variable. Acronyms used here are: *Var* = variable, *Pos* = position, *V* = vertex, *C* = container, *H* = host.

### 6.1.3 Optimization

A naive approach to using constraint programming would transform an integration problem into a CSP and then let a solver solve it, without any further considerations. This leads to good results for small problems, yet for practically-sized ones (like our case study) such an approach could take an unacceptably long time to produce expected solutions.

In order to shorten the time needed we had to think about the solving process more carefully. We have done the following:

For *variable selection* (the first question) we have decided to combine selected general heuristics for solving CSPs mentioned in Section 3.3 with our own situation-specific ones. For example, when solving the channels design issue there is an option of starting with `ChannelType` variables and then to consider all the others, according to selected heuristic. When solving data flow design issue there is a possibility to start with the variable denoting the count of non-zero `ConnectionPresence` variables and continue with their concrete values.

For *value selection* (the second question) we generally use the *lowest value* strategy. We have constructed domains in such a way that the lower values are – when possible – the "cheapest" ones. For example, in-memory channels are assigned the value of 0, and MQ-based ones got values of 1 and 2. Therefore the solver first explores cheaper solutions before trying more expensive ones.

As the results described in Section 6.4 show, for complex cases these heuristics alone are not sufficient to shorten the time needed for the computation to an acceptable value. Therefore we have decided to allow solving individual design issues independently, or in related groups – i.e. in a way similar to how would a human designer approach such a problem.

Dependencies between individual design issues are summarized in Figure 26.

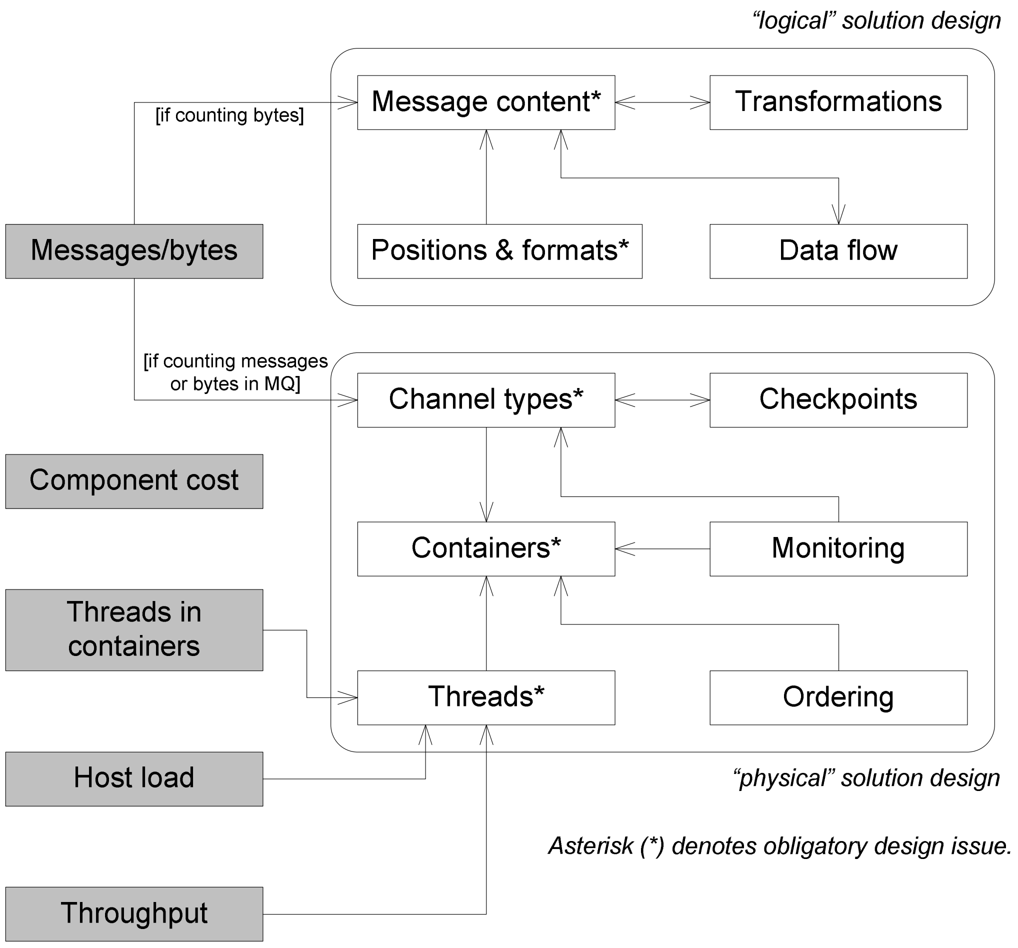


**Figure 26.** Dependencies between design issues solved by the U/CP method.

White boxes represent structural design issues, gray ones represent metrics. (Please note that the General design issue is not shown: it should be solved in any case and virtually every other design issue depends on it.) An oriented line between two boxes denote a dependency: the line source is dependent on the line target. For example, the Positions and formats issue cannot be solved without the Message content issue. Some issues are obligatory, i.e. they have to be solved in all integration scenarios in order to get an executable solution: for example, we always have to decide about message content or threads used for services' execution. Some other issues are present only in certain integration scenarios – there are situations that do not require e.g. implicit data transformations, data flow among non-adjacent components, or message monitoring features. In some cases these non-obligatory issues have to be solved along with their obligatory peers, for example if an integration problem requires implicit data transformations and/or advanced data flow, it is not possible to solve the Message content issue without considering these two ones.

Moreover, in Figure 26 we can see a rough division between "logical" and "physical" structural issues. The former deal with the content of messages that flow within the solution, while the latter deal with the deployment of the solution components to concrete containers and the choice of individual channels implementations.

So, how would a developer proceed through solving a complex problem, i.e. a problem that cannot be reasonably quickly solved at once? Typically, he or she would start with the logical set of structural issues. Even among them it is possible to postpone solving the positions issue to the second phase (i.e. solve Message content + optionally Transformations + Data flow first, and Positions and formats afterwards).

Then one can try to solve physical issues, namely Containers, Threads, Channel types and Checkpoints together. If necessary, also this can be split and Containers can be solved first, then Channel types with Checkpoints or Threads or Ordering. However, if one needs to evaluate throughput (that is the usual reason for parallelizing the processing) he or she has to know exact numbers of threads: so, generally, we advise to compute Containers + Threads + Channel types together. Monitoring can be solved along with Channel types, or later. These possibilities are summarized in Figure 27.

A developer influences the solution construction process by specifying a sequence of solution *construction steps*. Each step is characterized by a number of options, most important ones being:

1. a *domain* (scope) of the construction step, i.e. which parts of the integration solutions we are dealing with in this step – the basic unit of construction is one process, so each domain is defined as a set of processes;
2. design issues (aspects and metrics) we are taking care of in this step;
3. additional options for the current step, like how many integration services to create, which heuristics to choose, and so on.

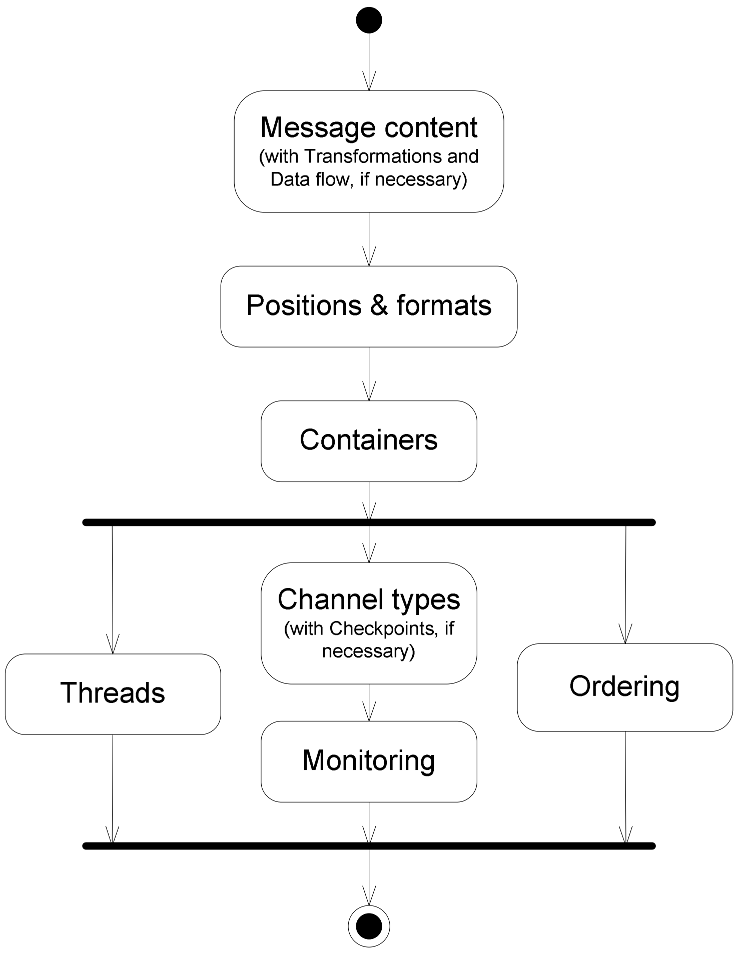

Message content
(with Transformations and Data flow, if necessary)
Positions & formats
Containers
Threads
Channel types
(with Checkpoints, if necessary)
Monitoring
Ordering

**Figure 27.** Possible pathways through the solving process.

The overall algorithm then looks like this:

**Algorithm 2.** Finding an integration solution in the U/CP method (advanced version).

```
FindIntegrationSolution()

begin
  CreateControlFlowGraph();
  CreateAuxiliaryDataStructures_GlobalLevel();
  for each construction step:
    CreateSlotsForIntegrationServices();
    CreateAuxiliaryDataStructures();
    CreateCSP();
    SolveCSP();            // in parallel thread: DisplaySolutions();
    RemoveUnusedIntegrationServices();
    RecordSolution();
  end for
  GenerateCode();
end.
```

`CreateControlFlowGraph` is basically the same as in Algorithm 1. We create a graph for the whole integration solution.

`CreateAuxiliaryDataStructures` has been split into two parts: selected activities (namely computing message rates) are executed at a global level; others are done at the level of individual steps.

`CreateCSP` is in principle the same as in Algorithm 1. However, it is technically more complex because we must take into account results from previous steps as well as the fact that some parts of an integration solution are in scope of the current step, while others will be solved only in the future. Our approach is:

1. We create a CSP only for those parts of an integration solution that are in the scope – with an exception of CSP variables that characterize inter-process interfaces.
2. We define a set of key design decisions – roughly speaking, these correspond to CSP variables listed in Table 8 connected to structural design issues – and after solving a CSP, we store these decisions for later use (the `RecordSolution` step in Algorithm 2). When creating any subsequent CSP, we strictly impose any design decisions made before; we do so within `CreateCSP`.

`RemoveUnusedIntegrationServices` is a step necessary to keep the size of the design problem from unnecessarily growing.

Besides working with heuristics and design aspects isolation we have done a number of attempts to make computation faster. Generally these are dependent on the concrete CSP solving algorithms and their implementations, so they are not much important at the conceptual level:

As an example of these attempts, we have undertaken a significant effort to optimize computations related to the Data flow aspect by looking for adequate formulations of relevant constraints. The resulting formulation can be characterized by a certain level of redundancy.

Another example is the computation of various service deployment characteristics (total number of threads, containers, hosts, number of threads per host, load per host, and throughput) given information about threads per container this service executes in. One possibility is to use a set of numerical constraints, like

```
V.ThreadsCount = Sum(forall Cont : V.ThreadsPerContainer[Cont])
V.ContainersCount =
  TotalNumberOfContainers – Count(forall Cont : V.ThreadsPerContainer[Cont], 0)
V.ThreadsPerHost[Host] = Sum(forall Cont @ Host : V.ThreadsPerContainer[Cont])
V.HostsCount = TotalNumberOfHosts – Count(forall Host : V.ThreadsPerHost[Host], 0)
```

These constraints are shown using a simple language that we have devised to describe constraints generated by U/CP method. `V.`*`variablename`* denotes a CSP variable with a given name attached to a vertex `V` of the solution graph. `Sum(forall` *`Entity`* `:` *`variablename`*`[`*`Entity`*`])` describes a new CSP variable that is constrained to be equal to a sum of variables `variablename[Entity]` for all instances of Entity, where Entity is typically a host (Host) or a container (Cont). `Cont @ Host` symbolically means that we have to take into account only those containers that are deployed on a given host. `Count(forall Entity : variablename[Entity], value)` means a new CSP variable that is constrained to be equal to a number of variables among `variablename[Entity]` vector (for all instances of Entity) that have the value of `value`. Finally, `Expression1 = Expression2` signifies a constraint that tells that these two expressions must be equal.

We tried to replace all these computations by simple enumeration of all acceptable values of a vector containing these variables: `V.ThreadsPerContainer[Cont]`, `V.ThreadsPerHost[Host]`, `V.ThreadsCount`, `V.ContainersCount`, `V.HostsCount`, `V.LoadPerHost[Host]`, `V.Throughput`.

The latter approach allows us to provide more accurate results for host load computations and is more efficient in some cases. However, its use is limited to situations with relatively smaller number of containers and threads (e.g. only problems P1.1 and P1.3 from the set shown in Table 14 on page 111) and in some situations it leads to significantly slower solution search process. For example, results in Table 12 have been computed with the first approach, as the second's use has led to approximately 3-5 times worse computation times.

## *6.2 Implementation*

We have implemented both methods (ML/CP, U/CP) in the form of prototypes.

For solving CSPs we have used the JaCoP (Java Constraint Programming) solver in versions 2.4.1, 2.4.2 and 3.0 (Szymanek, 2011). Methods themselves are implemented in Java.

## *6.3 Evaluation of the ML/CP method*

In this section we shortly compare the performance of the ML/CP method with the ML/P method that has a similar scope. We use the same integration problems as in Section 5.2 and (Mederly and Lekavý, 2009), namely:

- Problem A here corresponds to Problem 2 in Section 5.2.
- Problem B here corresponds to Problem 4 in Section 5.2.
- Problem C here corresponds to Problem 4 in Section 5.2, enhanced with the message duplication elimination aspect (not covered by the ML/P method, though).
- Problem D here corresponds to Problem J in (Mederly and Lekavý, 2009). It deals with the transfer of data between Student Records and Human Resources information systems and Central Database of Persons at Comenius University in Bratislava.

Results of solving these problems using our method are shown in Table 9.

**Table 9.** Results of the evaluation of the ML/CP method.

| ID | Domain | Aspects | Business services | Int. comp. | Optimal solution (sec) | All solutions (sec) | Optimal solution in ML/P (sec) |
|---|---|---|---|---|---|---|---|
| A | W&G-part | M, F, TA | 5 | 10 | 0.17 | 0.44 | 0.43 |
| B | W&G-full | M, F, TA, O | 11 | 20 | 0.33 | 30.04 | 15.47 |
| C | W&G-full | M, F, TA, O, D | 11 | 29 | 2.80 | 7,012.36 | n/a |
| D | University | M, F, TA | 11 | 20 | 0.20 | 1.87 | 55.00 |

Columns meaning: The *ID* column contains the identification of the integration problem in question. The *Domain* column denotes the domain of the respective problem. The *Aspects* column shows which of the design issues the problem deals with, using the following acronyms: monitoring (M), message formats (F), throughput/availability (TA), message ordering (O), and duplicate messages elimination (D). The *Business services* column shows the number of business services that constitute the problem. The *Int. comp.* (integration components) column gives the number of integration services, messaging middleware channels, and messaging transactions used in the optimal solution. The next two columns characterize performance of the method: the first one describes how many seconds of CPU time it took the method to produce the optimal solution (*Optimal solution* column); the second describes how many seconds of CPU time it took the method to conclude that no better solution exists (*All solutions* column). Although the second time interval is the most important one, from the practical point of view the first one is also relevant, as after this time the optimal solution (albeit without knowing for sure that it is really the optimal one) can be displayed to the user.

For comparison purposes we include also the CPU time needed by the ML/P method to find a solution, where applicable. Although the processors that have been used for computations are slightly different, the numbers provide an indicative comparison of these methods' performance.

These results show that this method can design defined aspects of integration solutions for practically-sized integration problems. (Although the numbers of services and integration components can be seen as small, many real integration solutions are not of much larger scale.) What should be – and, actually, was – extended, however, was the set of aspects being tackled by the method, as implemented in the U/CP method.

## 6.4 Evaluation of the U/CP method

For U/CP we have created three prototypes. Basic Algorithm 1 has been implemented in Prototype 1 (called also P1), and more advanced Algorithm 2 in Prototype 2 (called P2). While the core of the method, i.e. integration problem encoding, is the same, the P2 allows decomposing the solution creation: both horizontally – to sets of integration processes, and vertically – to sets of design issues. A support for data formats has been added in this prototype as well.

Recently, a third prototype (P3) has been created as well. It aims to demonstrate a practical usability of the method for constructing real-world integration solutions. The implementation of the method core responsible for creating integration solutions design is in P3 exactly the same as in P2. The main difference is that P3 uses a custom textual domain-specific language (DSL) to formulate integration problems, instead of XML used in P2. Also, it is integrated with Eclipse IDE (Integrated Development Environment)[15] in order to be more programmer-friendly, providing an editor with syntax coloring, error checking while typing, content assistant, hyperlinking and more (provided by the Xtext language development framework). Due to time constraints, P3 lacks some features that are present in P2, for example the Transformations aspect and some of the metrics. However, it is only a question of implementing their support in the new DSL; after that is done they will be fully functional in P3.

In order to thoroughly evaluate this method we are trying to answer the following questions:

Q1. Is the method *feasible* (usable in practice), that means:

   a. Is it universal enough, i.e. can it be applied to a class of integration problems that is sufficiently wide?

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

   c. Is it correct, i.e. do integration solutions created by it meet specified functional and non-functional requirements?

Q2. Is the method an *improvement* over existing approaches?

We will answer *Q1* by selecting a set of integration problems taken from the literature, i.e. mainly (Hohpe and Woolf, 2004), and from real-life integration projects undertaken at Comenius University in years 2007-2011. For these integration problems we will create input data for the method (thus answering *Q1a*), execute a method prototype implementation to measure the time needed to find a solution (answering *Q1b*), manually inspect the solutions found (partially answering *Q1c*) and finally deploy selected solutions to real integration platform and execute them (answering *Q1c*). Details are given in sections 6.4.1 to 6.4.3.

In answering *Q2*, we will compare the method to (1) the development of integration solutions using product-specific graphical editing environment (Progress Sonic Workbench) that could be seen as a simple model-driven approach to integration and to (2) existing model-driven approaches published in academic literature (Scheibler

[15] Eclipse is an open source, extensible development platform, originally started as an environment for developing software in Java (Eclipse Foundation, 2011).

and Leymann, 2009; Sleiman, Sultán, and Frantz, 2009). This is described in Section 6.4.4.

### 6.4.1 Formulating integration problems

First we prepare a set of *integration scenarios*, i.e. general descriptions of situations for which we want to create integration solutions. Each scenario is then enhanced with concrete details (complete non-functional requirements, environment description and design goals) in order to create one or more *integration problems*. We check whether the input language of the method is powerful enough to capture all the details needed to construct a working integration solution for a given problem.

We have selected the following eight integration scenarios:

1. Widgets and gadgets order processing (Hohpe and Woolf, 2004)
2. Loan broker (Hohpe and Woolf, 2004)
3. Transfer of data from academic information system (AIS) and SAP Human Records module (SAP) to central database of persons (Comenius University integration project)
4. Canteen menu web presentation (Comenius University integration project)
5. Transfer of data about thesis and dissertations from AIS to external plagiarism detection system (Comenius University integration project)
6. Transfer of data about defended thesis and dissertations as well as fulltexts of these works from AIS to university library information system (Comenius University integration project)
7. Transfer of personal data from Central database of persons to the information system of dormitories and canteens (Comenius University integration project)
8. Transfer of students' admission confirmations from Academic information system to Central database of persons (Comenius University integration project)

In the following we describe these scenarios more closely.

**Scenario S1: Widgets and gadgets**

Our first integration scenario is based on the extended version of Widgets and Gadgets order processing system, as described in Section 6.1.1. (Its original version has been explained in Section 4.1.)

The flow of control and data is shown in Figure 22 at page 78, repeated here for convenience.

The first integration problem (P1.1) then has the following characteristics.

*Flow of control and data* is as specified above.

*Non-functional requirements* for the integration solution are:

1. The solution has to provide a throughput of at least 60 orders per minute.
2. The following services have to be deployed at no less than two hosts in order to achieve required availability: `CheckCredit`, `CheckWidgetInventory`, and `ComputeOverallStatus`.
3. Input and output of the following services have to be monitored: `Check-Credit`, `CheckWidgetInventory`, `CheckGadgetInventory`, and `Compute-OverallStatus`.
4. Messages arriving at input endpoints of `Bill` and `Ship` services have to be in the same order as messages that are present at the beginning of the process.

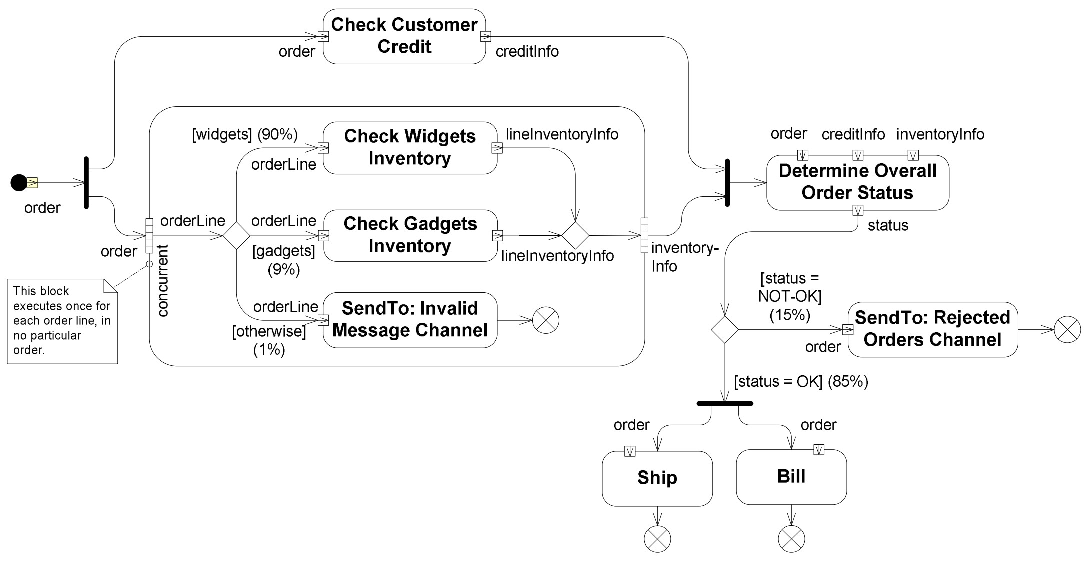


**Figure 28.** Specification of control and data flow for Widgets and gadgets order processing scenario.

We want to provide an integration solution for the following *environment*:

1. As an integration platform, Progress Sonic ESB 8.0.1 has to be used.
2. When implementing Fork and Join and ForEach constructs, we should use only built-in “Split and Join Parallel/ForEach” services; we have no custom Aggregator service available.
3. For some reasons, in order to distribute processing into more containers we want to use queues (not Sonic-specific shared topic subscriptions).
4. We can use 5 containers running at 3 hosts: Host *H1* with containers *WG-C1a* and *WG-C1b*, host *H2* with containers *WG-C2a* and *WG-C2b*, and host *H3* with a container *WG-C3*.

5. All services except `ComputeOverallStatus` require their input and output to be present in the message body. Moreover, they do not preserve any context in message header nor in attachments. (Except standard JMSReplyTo and JMSCorrelationID header fields, of course.) In contrast, the `Compute-OverallStatus` service is implemented in such a way that it can accept its input in message body as well as in attachments, and preserves values stored in message header, body, and attachments.

6. Business services can be deployed into containers as shown in Table 10.

**Table 10.** Parameters of business service deployment for integration problem P1.1.

| Service | Container | Maximum # of threads | Messages per minute per thread |
|---|---|---|---|
| CheckCredit | WG-C1a | 8 | 10 |
| | WG-C1b | 8 | 10 |
| | WG-C3 | 8 | 20 |
| CheckWidgetInventory | WG-C2a | 8 | 60 |
| | WG-C2b | 8 | 60 |
| | WG-C3 | 8 | 120 |
| CheckGadgetInventory | WG-C3 | 8 | 20 |
| ComputeOverallStatus | <any> | 8 | 600 |
| Ship | WG-C3 | 8 | 20 |
| Bill | WG-C3 | 8 | 20 |

Table description: For each service (*Service* column) we see the list of containers the service can be deployed in (*Container* column). Deployment is a subject of technical constraints that are beyond the scope of this method, for example platform dependencies, availability of required software, co-location with external resources like database management systems, and so on. For each service and container there is (1) a maximum number of threads the service can be deployed in the respective container (*Maximum # of threads* column), and (2) the number of messages that can be processed per time unit (in this case per minute) in one thread (*Messages per minute per thread* column).

Please note that although this implies a linear model of performance, the method in its present form can work (with some limitations related to the number of containers and threads) with any function that maps the number of threads in concrete containers to service throughput. What remains, though, is an assumption of the independence of performance of individual service/container pairs – e.g. that a throughput provided by a service $S_1$ in container $C_1$ is independent on the deployment of a service $S_2$ in container $C_1$ (or $C_2$, or any other). In reality, however, these services often use shared resources (CPU and memory of hosts, external resources like database servers, and so on), so their performance characteristics can be more complex. We leave such dependencies to be resolved by a human designer or to a future research.

7. Integration services can be deployed in any container, using up to 64 threads.

*Design goals* are:

1. Minimize the number of messages flowing through messaging middleware (counted with weight 1, i.e. the cost increase factor is the number of messages flowing through MQ per minute).
2. Minimize the weighted number of integration services, with the weights summarized in Table 11.

Other integration problems (P1.2-P1.8) are derived from this one by changing the environment description, namely the number of containers and service threading constraints, as described in Section 6.4.2.

**Table 11.** Costing weights for integration services for integration problem P1.1.

| Service | Cost |
|---|---|
| Wire Tap | 1 |
| Data Management | 1 |
| Fanout (fixed Recipient List) | 2 |
| Split and Join Parallel | 20 |
| Split and Join ForEach | 20 |
| Resequencer | 100 |

Columns meaning: The *Service* column contains a name of a service and the *Cost* column contains the cost of its use within the integration solution.

### Scenario S2: Loan broker

This integration scenario is again taken from the book on enterprise integration patterns (Hohpe and Woolf, 2004, p. 361).

In this case we want to create an integration solution for a loan broker that would allow it to provide a potential customer with the best loan quote. The process that has to be implemented starts when the broker receives a loan request from the customer. First the credit bureau is to be contacted to get a credit score for the client. Next a list of banks that should be inquired is created using a rule base. Then individual banks are contacted in order to get their offers. As the last step, the best offer is selected and returned to the customer.

The flow of control and data for this scenario is shown in Figure 29.

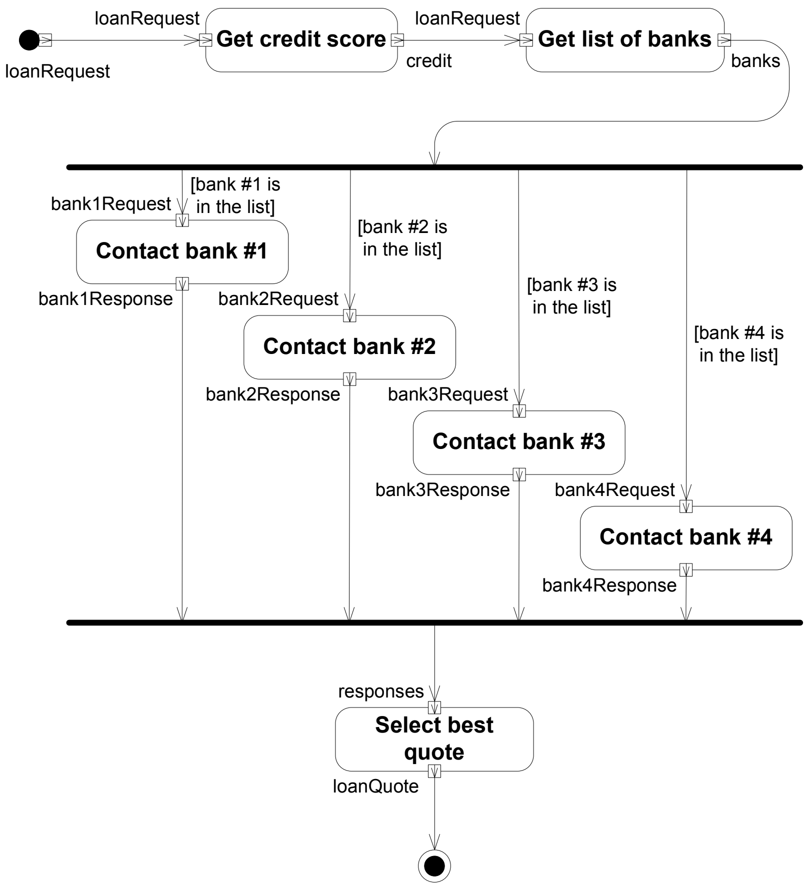


**Figure 29.** Specification of control and data flow for Loan broker scenario.

Please note that there are data transformations not shown in the control flow diagram – i.e. transformations from `loanRequest` and `credit` to bank requests (`bank1Request`, ..., `bank4Request`) that are specific for individual banks and a transformation from bank responses (`bank1Response`, ..., `bank4Response`) to aggregated variable `responses`. These transformations should be prepared by the developer and listed in the environment description. The method is free to place them anywhere it considers suitable, without requiring the developer to clutter the control flow specification with these technical details.

### Scenarios S3: Students and employees of a university

A major integration project at Comenius University in Bratislava, the largest Slovak university having more than 27,000 students and 4,200 employees (full-time equivalents) has been dealing with the transfer of the data about students and employees from source systems, namely Students Records and Human Resources, into more than 20 applications that need personal data in order to effectively provide their services. This project runs from 2004, continually adding new “client” systems, new features, and adapting to changes in system landscape and integration requirements. There were two major changes, the first in January 2008 when the 17-years old Human Records system was replaced by a modern SAP R/3 Human Records (HR) module; and the second in summer 2009 when the 18-years old Student Records was replaced by a modern Academic information system (AIS). What is the most important from an architecture point of view is that these changes did not force any change at the side of “client” systems: they were completely absorbed by two key integration applications: Central database of persons (CDO) and University Personal ID (UOČ[16]) generator.

Scenario S3 deals with the transfer of students’ and employees’ data from AIS and SAP, respectively, into central database of persons, along with generating personal IDs. This scenario is quite complicated, having 23 processes with 125 components and 85 variables in total; it is executing in 6 containers. The scenario description consists of more than 450 lines of text in 4 XML documents. Execution dependences among its processes are shown in Figure 30.

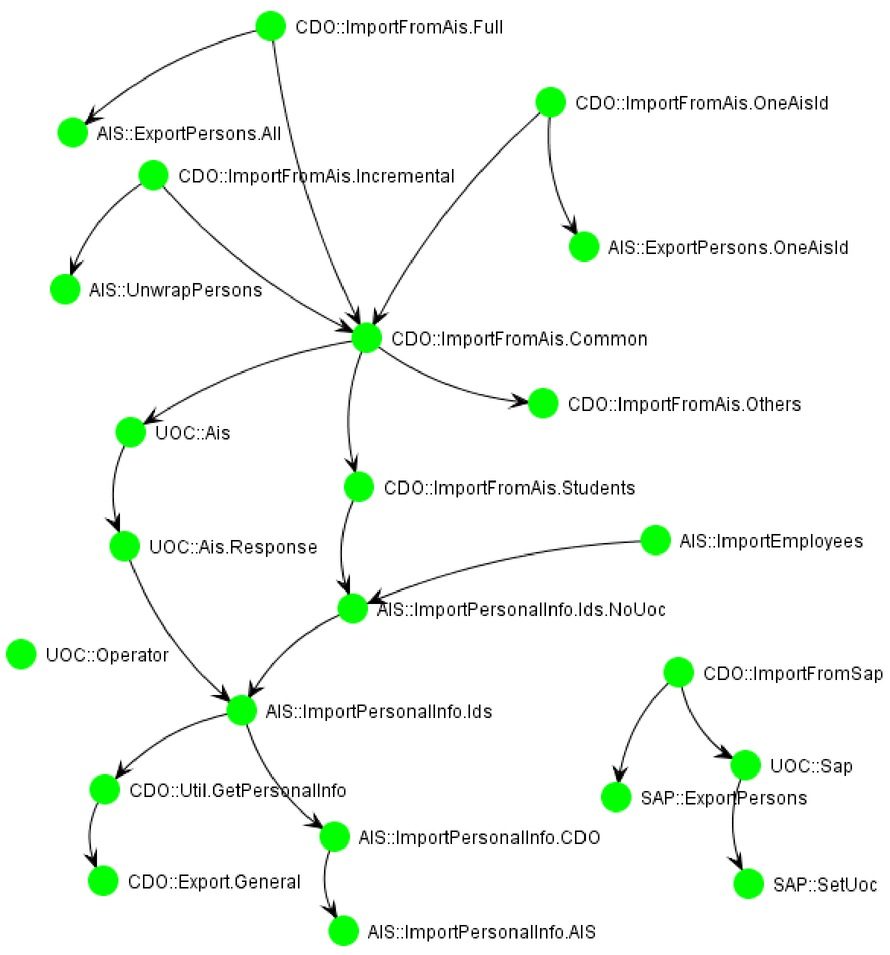

CDO::ImportFromAis.Full
AIS::ExportPersons.All
CDO::ImportFromAis.OneAisId
CDO::ImportFromAis.Incremental
AIS::ExportPersons.OneAisId
AIS::UnwrapPersons
CDO::ImportFromAis.Common
CDO::ImportFromAis.Others
UOC::Ais
CDO::ImportFromAis.Students
AIS::ImportEmployees
UOC::Ais.Response
AIS::ImportPersonalInfo.Ids.NoUoc
UOC::Operator
CDO::ImportFromSap
AIS::ImportPersonalInfo.Ids
UOC::Sap
CDO::Util.GetPersonalInfo
SAP::ExportPersons
AIS::ImportPersonalInfo.CDO
CDO::Export.General
SAP::SetUoc
AIS::ImportPersonalInfo.AIS

**Figure 30.** Dependencies among integration processes in scenario S3.

Complexity of this solution is given by significant differences between data models, data formats, and data transfer technologies used by AIS at one side and central

[16] These abbreviations are determined by official Slovak names of these applications; they are used in the integration problem specification and integration solution design, so we will use them in the text as well.

database of persons and personal ID generator at the other. Moreover, at many places, there are technical restrictions like the necessity to communicate with AIS in batches, as it is obviously not able to import or export all the data about 32,000 persons (about 4 gigabytes) in one web service execution. Also the functionality provided by the integration solution is significant: it transfers data from AIS to central database of persons – both in “full” nightly batch synchronization mode as well as in “incremental” mode, i.e. when “person change” event is signaled by AIS. It also transfers data about students’ university cards, logins, passwords, photographs, and status from database of persons to AIS. It does a rough business-level validation of the transferred data and signals any errors to the responsible persons by e-mail. It transfers data from SAP to central database of persons as well. The solution also generates new or assigns existing personal IDs to all persons at the university.

Besides the control and data specification that forms an input of the U/CP method there are a lot of programming components of this solution that are (at least for now) created manually by the integration designer. In this case there are more than 25 XSLT (Extensible Stylesheet Language Transformations) stylesheets, 5 web service invocation or receive scripts, 7 custom Java ESB services and a couple of JavaScript functions. Automating the creation of some of these components is the focus of our future work.

More information about an older version of this integration scenario can be found in (Mederly and Pálos, 2008).

**Scenario S4: Canteen menu web presentation**

In comparison to the previous scenario, this one is very simple: it implements a solution that converts an existing SOAP/HTTP interface giving information on a daily menu (provided by an application installed at one of Comenius University hostels) into a plain HTTP-based interface that provides aggregate information on a weekly menu. A simple PHP script (outside of this solution) then takes this information and displays it to users of the hostel’s web site.

**Scenario S5: Theses and dissertations (plagiarism check)**

We have implemented a small integration project at Comenius University, connecting AIS to an external plagiarism detection system (Theses). The functionality of this solution is similar to the scenario S4 above: it converts existing SOAP/HTTP interface having two operations (“Get a list of works”, “Get data on a work in CRZP[17] format”) provided by AIS into a new, much simpler, HTTP-based interface that provides information on the works, but this time using a format specific to Theses.

[17] CRZP = national registry of thesis and dissertations in Slovak Republic

**Scenario S6: Theses and dissertations (transfer of defended works)**

This integration solution is used to fetch metadata and full texts of works from AIS and to store them into library information system. Besides this main process, there is a couple of auxiliary ones e.g. a process for processing a selected work or a process for fetching metadata about all processes in two supported formats (CRZP and ISO 2709 ones).

**Scenario S7: Personal data for dormitories**

After processing data from AIS, SAP and other sources (see scenario S3), the central database of persons emits personal data change notifications. These are used for keeping selected destination systems' data up-to-date. One of such systems is ISKaM ("Informační systém pro koleje a menzy") – a system for managing dormitories and canteens. The goal of integration scenario S7 is to receive such a change notification, translate it to specific format for ISKaM (requests for web services used to update person and update its card), execute those services, and report failures if there are any.

**Scenario S8: Students' admission confirmations**

As a part of the process of students' enrollment at Comenius University the data about their admission confirmations (including photographs that are uploaded by students) are transferred from AIS into the central database of persons. This is implemented by a dedicated simple integration solution.

**Formulating integration problems – a conclusion**

Concerning the formulation of the method's input for scenarios S1 to S8, we were able to prepare it without significant restrictions.

Yet there were a couple of issues: some were of a technical character, and others were more conceptual. Here is a list of the most important ones:

1. *Multiple usage profiles:* In scenario S3 we are looking for an integration solution whose parts will be used under two different regimes: nightly transfer of all persons' data (raw size about 150 megabytes in one message; sent once per night) and continual transfer of individual persons' records (small messages of about 10-50 kilobytes each, but more frequent; in peak times of students' enrollment they come as fast as hundreds per hour with bursts counting tens of messages per seconds).

   One possibility how to deal with this is to create two different integration solutions, each optimized for one of these scenarios. Yet, mainly because of maintainability and manageability reasons, we want to have one solution for both cases. The method should therefore be modified to allow specifying more usage profiles (message peak rates, message sizes, and so on) and to compute solution cost based on weighted sum of costs related to these profiles.

A related point to improve is that in the future the method should distinguish between *hard limits* (e.g. message sizes, number of iterations in for-each cycles, and so on) and *usual (average) values*. While the former should be used for deriving some solution properties (e.g. whether a process variable can be placed to a header limited to 64 KB in size), the latter should be used for determining various solution cost attributes, like the average number of messages transferred through MQ per time unit. This could be modeled using two usage profiles: (1) a "worst" case, (2) an "average" case.

Both these changes are of a technical nature. They would mean changing today's scalar values (e.g. channel's message rate or variable's size) to be vectors (e.g. channel's message rate and variable's size for various usage profiles). Until that time the developer has the following possibilities: (1) to generate separate integration solutions for individual usage profiles, (2) to specify non-functional requirements (e.g. variable sizes and message rates) and design goals (e.g. weights of costs related to MQ usage) in such a way that they would reasonably cover all usage profiles, or (3) to make the most important design decisions (e.g. deployment of key services into containers/threads) himself or herself and let the method compute the rest. In our scenario S3 we used a combination of the second and the third approach.

2. *Aggregator timeout value computation.* In current version of the method the developer has to specify timeout values for fork-and-join and for-each constructs (i.e. how long has the system wait for messages to come in) by hand. However, these values are quite important to be determined correctly, because they influence (among other things) the number of threads necessary for Split and Join constructs.

   We will be able to estimate these values when the method will be enhanced to compute message processing latency – it is the subject of our future work.

3. *Intricate message sequencing issues.* In scenario S3 we have a situation in which we want to serialize processing of "full" and "incremental" data coming from AIS. The reason is that we would like to disallow a person change event to "outrun" full data batch produced before that event, because in this way the event would be lost (overwritten) by the full data batch.

   Current version of the method is not able to deal with this issue, and it is our intent to add it in the future – it would require us to slightly generalize the Ordering design aspect, e.g. to work across processes and choice-merge component pairs. Until that time the developer has to review a generated solution and modify it by hand to ensure that the message sequencing requirements are met. Or, he could specify custom constraints that will ensure that a solution generated by the method would meet these requirements.

4. *Message compression*: In scenario S3 there are channels that need to transport very large messages – namely, XML messages containing all personal data from AIS are up to approximately 200 megabytes in size. While there is no problem transporting such large messages in memory, in case of MQ-based channels it is more appropriate to compress them.

   Although this could be theoretically solved using Transformations aspect (modeling compressing and uncompressing services as implicit data transformations), applying this approach would require solving Channel types and Transformations issues together. For large scenarios this could present a performance problem.

   For the time being we have decided to solve this issue in the code generation phase by applying message compressing and uncompressing services to each MQ-based channel transporting messages of a size that exceeds a defined threshold. However, this can be inefficient in certain situations, as it could lead to repeated and unnecessary compressing and uncompressing message content. Therefore in the future we plan to create a design aspect dedicated to this issue and solve it along with (or after) Channel types aspect.

Overall, we were able to formulate method's input for the chosen set of integration problems. This leads us to a conclusion that the method is sufficiently universal, and has a potential to be applied to a wide range of integration problems. Obviously, it cannot presuppose all peculiarities of messaging-based integration solution design that can occur; what is important is that it should enable the developer to solve these special cases 'by hand' and to include his or her solution into the generated solution.

In the future we plan to apply the method to other real-life integration projects in order to more thoroughly check its universality and to continue its development, e.g. towards eliminating the issues discovered as well as increasing its practical usability.

### 6.4.2 Creating and checking integration solutions

**Results for the scenario S1 (Widgets and gadgets)**

We executed the method for the problem P1.1, with the following settings:

1. We solved these design issues: Message content, Channels, Threads, Containers (always), and Data flow, Positions, Monitoring, Ordering (optionally, see Table 12). We skipped Data transformations, as they are not needed in this case. Also Checkpoints issue was not used. Each combination of design issues formed a test case, i.e. we had 16 test cases in total. For each test case we tried to find out how successful was the method in finding optimal (or near optimal) solutions, and with which heuristics.

2. Concerning heuristics, we alternated between using generic "Weighted degree" and "Most constrained (static)" variable-choosing strategies because

in preliminary tests these two had shown to be suitable for our purposes. We optionally combined them with our custom “channels first” strategy.

3. The CPU time consumed was limited to 3, 10, 60, and 600 seconds. This would correspond to various modes of use by the developer – e.g. “quick solution preview”, “generating a ‘good enough’ solution”, and “having the time to create optimal solution”.

4. Between each two business services and control flow elements, the method inserted 1 place for an integration service. For this problem this value was sufficient.

We statically checked the correctness of the generated solutions, namely:

1. whether the message routing logic was correct with respect to the control and data flow that had to be implemented,

2. whether non-functional requirements concerning the expected throughput, redundancy of deployment, monitoring, and message ordering listed on page 99 were met.

All checks were successful. The solutions found and the CPU time needed for the computations are shown in Table 12.

**Table 12.** Results of solving problem P1.1 by the U/CP method Prototype 1.

| ID | DF | Pos | Mon | Ord | 3s | 10s | 60s | 600s | Opt |
|---|---|---|---|---|---|---|---|---|---|
| wg0 | - | - | - | - | **1459** / 1.3s<br>WD | **1459** / 1.3s<br>WD | **1459** / 1.3s<br>WD | **1459** / 1.3s<br>WD | 314.2s<br>WD+Ch<br>WD/MCS+Ch |
| wg1 | - | - | - | Y | 1731 / 0.2s<br>WD/MCS+Ch | 1731 / 0.2s<br>WD/MCS+Ch | 1726 / 22.1s<br>MCS+Ch<br>MCS/WD+Ch | 1725 / 155.6s<br>MCS+Ch<br>MCS/WD+Ch | - |
| wg2 | - | - | Y | - | 1468 / 2.1s<br>MCS+Ch<br>MCS/WD+Ch | 1466 / 3.4s<br>MCS+Ch<br>MCS/WD+Ch | **1463** / 21.1s<br>WD/MCS+Ch<br>WD+Ch | **1463** / 21.1s<br>WD/MCS+Ch<br>WD+Ch | 53.9s<br>WD+Ch<br>WD/MCS+Ch |
| wg3 | - | - | Y | Y | 1888 / 0.1s<br>MCS+Ch<br>MCS/WD+Ch<br>WD/MCS+Ch | 1798 / 7.8s<br>MCS/WD+Ch | 1798 / 7.8s<br>MCS/WD+Ch | 1798 / 7.8s<br>MCS/WD+Ch | - |
| wg4 | - | Y | - | - | 1462 / 1.5s<br>WD | 1462 / 1.5s<br>WD | 1462 / 1.5s<br>WD | **1461** / 360.9s<br>WD/MCS+Ch<br>WD+Ch | 388.7s<br>WD+Ch<br>WD/MCS+Ch |
| wg5 | - | Y | - | Y | 1733 / 0.3s<br>WD/MCS+Ch | 1733 / 0.3s<br>WD/MCS+Ch | 1727 / 28.0s<br>MCS+Ch<br>MCS/WD+Ch | 1726 / 197.9s<br>MCS+Ch<br>MCS/WD+Ch | - |
| wg6 | - | Y | Y | - | 1470 / 2.6s<br>MCS+Ch<br>MCS/WD+Ch | 1468 / 4.3s<br>MCS+Ch<br>MCS/WD+Ch | **1465** / 26.2s<br>WD/MCS+Ch<br>WD+Ch | **1465** / 26.2s<br>WD/MCS+Ch<br>WD+Ch | 66.9s<br>WD+Ch<br>WD/MCS+Ch |
| wg7 | - | Y | Y | Y | 1889 / 0.1s<br>MCS+Ch<br>MCS/WD+Ch<br>WD/MCS+Ch | 1889 / 0.1s<br>MCS+Ch<br>MCS/WD+Ch<br>WD/MCS+Ch | 1799 / 14.8s<br>MCS/WD+Ch | 1799 / 14.8s<br>MCS/WD+Ch | - |
| wg8 | Y | - | - | - | **1459** / 1.4s<br>WD | **1459** / 1.4s<br>WD | **1459** / 57.9s<br>WD | **1459** / 1.4s<br>WD | 370.1s<br>WD+Ch<br>WD/MCS+Ch |

| ID | DF | Pos | Mon | Ord | 3s | 10s | 60s | 600s | Opt |
|---|---|---|---|---|---|---|---|---|---|
| wg9 | Y | - | - | Y | 1650 / 2.1s WD/MCS+Ch | 1649 / 9.3s WD/MCS+Ch | 1595 / 34.8s WD/MCS+Ch | 1593 / 301.7s WD/MCS+Ch | - |
| wg10 | Y | - | Y | - | 1468 / 2.2s MCS+Ch MCS/WD+Ch | 1466 / 3.6s MCS+Ch MCS/WD+Ch | **1463** / 23.4s WD/MCS+Ch WD+Ch | **1463** / 23.4s WD/MCS+Ch WD+Ch | 59.9s WD+Ch WD/MCS+Ch |
| wg11 | Y | - | Y | Y | 1653 / 2.5s WD/MCS+Ch | 1652 / 9.4s WD/MCS+Ch | 1598 / 34.2s WD/MCS+Ch | 1595 / 271.7s WD/MCS+Ch | - |
| wg12 | Y | Y | - | - | 1462 / 1.5s WD | 1462 / 1.5s WD | 1462 / 1.5s WD | **1461** / 382.3s WD/MCS+Ch WD+Ch | 412.2s WD+Ch WD/MCS+Ch |
| wg13 | Y | Y | - | Y | 1733 / 0.3s WD/MCS+Ch | 1733 / 0.3s WD/MCS+Ch | 1727 / 29.2s MCS+Ch MCS/WD+Ch | 1726 / 206.0s MCS+Ch MCS/WD+Ch | - |
| wg14 | Y | Y | Y | - | 1470 / 2.7s MCS+Ch MCS/WD+Ch | 1468 / 4.5s MCS+Ch MCS/WD+Ch | **1465** / 27.2s WD/MCS+Ch WD+Ch | **1465** / 27.2s WD/MCS+Ch WD+Ch | 69.0s WD+Ch WD/MCS+Ch |
| wg15 | Y | Y | Y | Y | 1799 / 0.6s MCS+Ch | 1799 / 0.6s MCS+Ch | 1790 / 51.8s WD/MCS+Ch | 1742 / 565.1s WD/MCS+Ch | - |

Columns meaning:

- *ID* = test case identification;
- design issues: *DF* = data flow, *Pos* = positions, *Mon* = monitoring, *Ord* = ordering ("Y" means that in the test case we have tackled this design issue);
- *3s*, *10s*, *60s*, *600s* = results obtained when limiting CPU time to particular value – we show here:
  - the best solution cost achieved[18]; if the solution cost was proved to be optimal for the test case, it is shown in bold;
  - CPU time needed to get the best solution,
  - heuristics that were used to achieve this result[19], using abbreviations: *MCS* = Most Constrained (Static), *WD* = Weighted Degree, *Ch* = our own "channels first" strategy; notation of *a/b+Ch* means that heuristic "a" was used for discriminating between Channel type variables and heuristic "b" was used for discriminating between all the other ones;
- *Opt* = if a proof of optimality was obtained within 600 seconds, then we show the CPU time needed and heuristics that were successful in this respect.

Discussion: From Table 12 we can make the following observations:

[18] Please note that the cost value in *cannot* be compared across rows, as each set of aspects requires different integration services in order to satisfy the requirements – however, the cost value in any row should gradually decrease left-to-right, as the method progressively discovers better solutions.

[19] Of course various heuristics did not lead to computing the solution using exactly the same CPU time. In order to list heuristics here we require the time needed was no worse than 110% of the best result.

1. The method has been able to find an integration solution (although not always the optimal one) for the problem P1.1 even within the strictest time limit of 3 seconds. It means that from the time complexity viewpoint it is suitable for the interactive use (performance for other integration problems is described later).

2. It is much easier to find a near-optimal solution than the optimal one. Moreover, even that is much easier than to confirm that no better solution exists. This is not a big limitation for practical use, because the developer is often satisfied with a solution that is "good enough".

3. There is no single "best" combination of heuristics.

    a. For 6 test cases there was a single combination of heuristics that was among the best for all four columns (wg0, wg8: WD, wg3, wg7: MCS/WD+Ch, wg9, wg11: WD/MCS+Ch) but for remaining 10 cases there was no single "best" combination.

    b. If we count the times a combination of heuristics was among the best for all 64 situations (i.e. combinations of a test case and a CPU limit – there are 64 of them in total), we get results shown in Table 13. Generally we can say that "channels first" strategy is the useful one, yet there are 14 situations when it was better not to use it.

**Table 13.** Results of individual combinations of heuristics in integration problem P1.1.

| **Heuristics** | **Usefulness** |
|---|---:|
| WD/MCS+Ch | 29 |
| MCS/WD+Ch | 22 |
| MCS+Ch | 19 |
| WD | 14 |
| WD+Ch | 10 |
| MCS | 0 |

Columns meaning: The *Heuristics* column contains a combination of heuristics we have used to solve a set of 64 situations and *Usefulness* column shows the number of situations this combination was among the "best" ones, as described above.

This means that either we should give the developer the possibility to try different heuristics combinations, or we should run the method using more heuristics combinations in parallel to be able to use the best result found.

4. Also interesting (although not much surprising) is the fact that relatively small changes in constraint templates often lead to significant changes in the time needed for the computation. This is not visible in this evaluation, but can be observed when repeating these tests on various versions of a method prototype implementation. Even a change as small as modifying the names of CSP variables has lead to different order of variables to be assigned their values, resulting in significant change in computation time.

**Changing the deployment possibilities**

The size of CSP solution space that has to be searched is influenced by many factors. Besides integration problem size and design aspects employed we suspected that the number of deployment possibilities could play a significant role.

In order to check this hypothesis we have prepared a set of integration problems P1.2-P1.8 that differ from P1.1 in the number of containers and the number of threads business services can be deployed in.

Problems P1.1, P1.3, P1.5, P1.7 and P1.8 allow 8 threads per business service in a container, while problems P1.2, P1.4, and P1.6 allow 64 of them. Problems P1.1 and P1.2 work with the default of 5 containers, while problems P1.3 and P1.4 use a reduced number of containers (WG-C1b and WG-C2b are removed) and problems P1.5 and P1.6 use an increased number of them (for each container we add its copy to get 10 containers in total). P1.7 and P1.8 are special versions of P1.1 that contain 5 and 15 additional containers, respectively, that cannot be used for any business service (only for integration services).

Then we run an analogy of test case wg14 (see Table 12) to find a solution. Results are shown in Table 14.

**Table 14.** Characterization and results for problems P1.1-P1.8 (using Prototype 1).

| Problem | Max. threads per business service in container | Number of containers | Optimal solution's cost | Optimal solution (seconds) | All solutions (seconds) |
|---|---|---|---|---|---|
| **P1.1** | **8** | **5** | **1465** | **26.6** | **67.7** |
| P1.2 | 64 | 5 | 327 | 0.9 | 1.8 |
| **P1.3** | **8** | **3** | **1465** | **24.9** | **63.6** |
| P1.4 | 64 | 3 | 327 | 0.8 | 1.6 |
| P1.5 | 8 | 10 | 447 | 7.6 | 10.6 |
| P1.6 | 64 | 10 | 327 | 1.1 | 2.1 |
| **P1.7** | **8** | **10 (5 + 5 unused)** | **1465** | **30.3** | **77.2** |
| **P1.8** | **8** | **20 (5 + 15 unused)** | **1465** | **37.0** | **93.8** |

Columns meaning: The first three columns characterize an integration problem, as described above. *Optimal solution's cost* is the cost of the optimal solution. *Optimal solution* denotes the CPU time needed to find the optimal solution. *All solutions* is the CPU time needed to conclude that no better solution exists. Both of these times are measured in seconds.

Discussion: As the results show, increasing the number of threads and/or containers made the integration problem significantly easier to solve – the cost of the solution (in this case determined primarily by MQ channels used) is generally lower than the cost of the baseline problem P1.1. It is then of no surprise that problems P1.2, P1.4, P1.5 and P1.6 took much less time to solve, despite the larger search space in the dimension of threads in containers. This means we cannot use them to measure an effect of increased deployment complexity on computation time.

Therefore, problems P1.7 and P1.8 have been introduced in order to increase deployment complexity without simplifying the integration problem.

When we compare the results for problems of similar complexity, i.e. P1.3, P1.1, P1.7 and P1.8 (these are problems with the solution cost of 1465 having 3, 5, 10, and 20 containers, respectively; they are shown in Table 14 in bold), we see a slight increase in processing time (see also Figure 31).

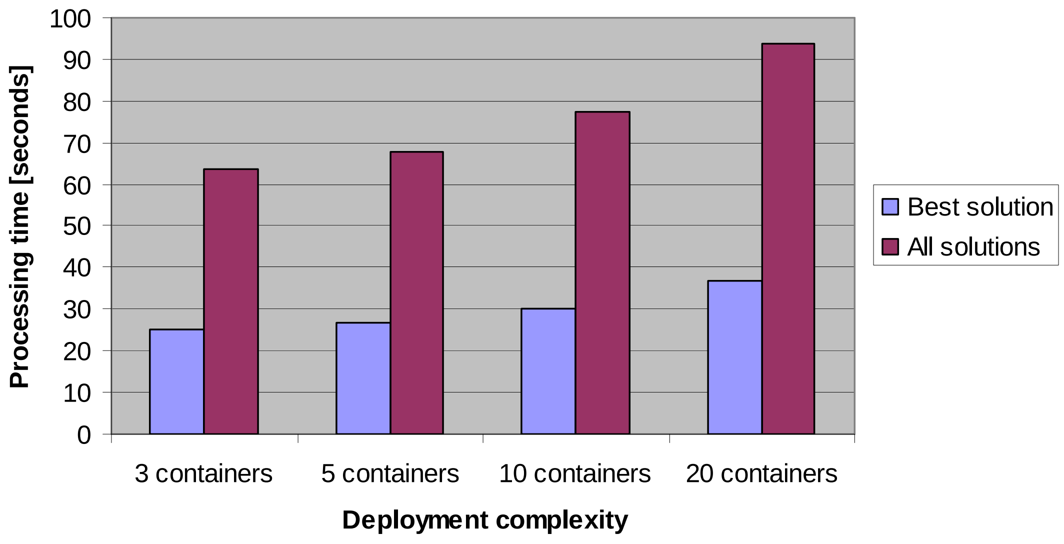


**Figure 31.** Dependency of processing time on the deployment complexity.

The effect of the number of containers on the processing time is therefore not as strong as we originally expected, and the method is able to work with relatively large numbers of containers. (For completeness we note that this test was done with Prototype 1.)

## Effect of design problem partitioning

In order to assess the effectiveness of problem partitioning, described in Section 6.4.3 and implemented in Prototype 2, we have solved the above mentioned integration problems in a sequence of construction steps. We have executed test cases summarized in Table 15.

**Table 15.** Ways of design problem partitioning used for the evaluation.

| Partitioning symbol | Content | Positions | Threads | Channels | Monitoring |
|---|---|---|---|---|---|
| CoPTChM (baseline) | Step 1 | | | | |
| CoP-TChM | Step 1 | | Step 2 | | |
| Co-P-TChM | Step 1 | Step 2 | Step 3 | | |
| CoP-TCh-M | Step 1 | | Step 2 | | Step 3 |
| Co-P-TCh-M | Step 1 | Step 2 | Step 3 | | Step 4 |

Columns meaning: *Partitioning symbol* is an abbreviation of the way of problem partitioning that is described using the remaining five columns. In a particular row, when a set of columns is merged together under the name “Step N” it means that the corresponding aspects are solved together in the N-th solution step. For example, a row containing *CoP-TCh-M* should be read like this: In the first step, Content and Positions aspects are solved. Then, in the second step, Threads and Channels are solved. Finally, in the third step, Monitoring is solved.

Results are shown in Table 16.

**Table 16.** Effects of design problem partitioning on the integration solution creation.

| Problem | Partitioning | Best solution | All solutions | Cost of the best solution found |
|---|---|---|---|---|
| P1.1 | CoPTChM (baseline) | 6.5 | 48.8 | 1465 |
| P1.1 | CoP-TChM | 6.5 | 50.1 | 1465 |
| P1.1 | Co-P-TChM | 7.5 | 53.5 | 1466 |
| P1.1 | CoP-TCh-M | 0.3 | 0.3 | 1465 |
| P1.1 | Co-P-TCh-M | 0.3 | 0.3 | 1466 |

Columns meaning: The first column contains an identification of an integration problem. Second column symbolically describes the partitioning used (see Table 15). Last three columns contain the CPU time used to find the best solution and all solutions, respectively, and a cost of the best solution. CPU times are shown in seconds.

In this particular case we can see that separating “logical aspects” (Content, Positions) from “physical ones” (Threads, Channel types, Monitoring) was not as helpful as one could expect – it did not even lead to lower computation times. The reason is that these logical aspects are, in this case, quite simple to solve. However, separating Monitoring from Threads + Channel types reduced time needed to find the best solution almost 22 times (6.5 vs. 0.3 seconds) and the time needed to find all solutions almost 163 times (48.8 vs. 0.3 seconds). In other integration problems with more complex logical aspects is the separation of Content and/or Positions aspects more important – for example, when solving P2.1 and P2.2 with Content and Positions together, the method was not able to find any solution in 600 seconds, while when solving these aspects in separation it could find the optimal solution in 21.9 seconds.

It is natural that when solving individual aspects in isolation, the method is sometimes unable to find the optimal solution. In the above experiment we can see that when we separated Content from Positions, we got a suboptimal solution with the cost 1466 instead of 1465. The difference is in one superfluous data management integration service (cost 1) – when solved the Content aspect, the method made several decisions whose cost manifested itself only in subsequent steps (in this case, when solving Positions issue). However, when creating the method, we have tried to arrange individual aspects in such a way that these “unknowingly expensive decisions” would be minimized. First results indicate that we were successful in this respect.

### Results for the other scenarios

In a way similar to the scenario S1 we executed the method to find solutions for integration scenarios S2-S8. The results are summarized in Table 17.

**Table 17.** Results of the U/CP method for scenarios S1 to S8.

| Problem | Aspects | Proc. | Vertices | Edges | Var. | Cont. | Best solution (seconds) | All solutions (seconds) |
|---|---|---|---|---|---|---|---|---|
| P1.1-WG | CoP-ChT-M | 1 | 19 (25) | 20 (26) | 6 | 5 | 0.3 | 0.3 |
| P2.1-LB | CoTr-P-ChT | 1 | 23 (47) | 29 (53) | 13 | 3 | 21.9 | Timeout |
| P3.1-Uni | CoP-ChT | 23 | 125 (142) | 129 (146) | 85 | 6 | 30.4 | 30.5 |
| P4.1-Uni | CoPChT | 1 | 8 (9) | 7 (8) | 7 | 1 | 0.8 | 0.8 |
| P5.1-Uni | CoPChT | 1 | 9 (9) | 8 (8) | 9 | 1 | 0.9 | 0.9 |
| P6.1-Uni | Co-P-ChT | 9 | 74 (89) | 91 (106) | 67 | 1 | 4.4 | 4.4 |
| P7.1-Uni | CoPChT | 1 | 22 (24) | 23 (25) | 9 | 1 | 0.3 | 0.3 |
| P8.1-Uni | CoD-P-ChT | 1 | 18 (25) | 19 (26) | 10 | 1 | 0.6 | 0.6 |

Columns meaning: The first column contains integration problem identification. Second column shows design aspects as well as their partitioning (Co = Content, D = Data flow, P = Positions, Tr = Transformations, Ch = Channel types, T = Threads, M = Monitoring). *Proc.* is the number of processes within the integration problem. *Vertices* and *Edges* describe the size of the control flow graph (the first number, i.e. before parentheses) and the size of the solution graph (the second number, i.e. in parentheses) and roughly correspond to the number of control constructs and control flow dependencies (the first numbers) and the number of services and channels within the solution (the second numbers). *Var.* is the number of process variables. *Cont.* is the number of containers. *Best solution* and *All solutions* are CPU times necessary to find the best solution and all solutions, measured in seconds. *Timeout* means that the solution finding process did not finish in allotted time (600 seconds), however, by a manual inspection we have found that the solution it has produced in this case is indeed the optimal one.

As we can see, our method is able to provide integration solutions in an acceptable time. For bigger problems it is necessary to employ problem partitioning in order to achieve a time that is short enough to be used in an interactive development environment.

### 6.4.3 Executing created integration solutions

In order to verify that a generated solution really meets its specification we prepared the following testing environment for scenario S1:

1. We created business services according to Table 10 on page 100. The services perform only a simulation of their real function, e.g. `CheckCredit` service returns a predefined value depending on a customer identifier. However, we implemented their throughput limitations by inserting appropriate delay instructions into them.

2. We prepared five containers as specified in point #4 in environment description on page 99. We put them at one testing machine, along with the messaging broker – because the services just simulated the throughput

limitation by delaying the processing without actually using the CPU, there was no need to actually distribute the processing to different hosts.

3. We created a testing client that sent a specified number of order requests per minute (60 in this case, as given by non-functional requirement #1 on page 99), with the characteristics corresponding to assumptions given in the requirements specification – probabilities of individual choice branches and the number of times each “ForEach” cycle is to be executed. It counted responses and measured the time needed to get them.

Then we chose a problem wg14 (see Table 12) and created a solution using the method. We deployed the solution into our testing environment and started it. We executed the testing client and verified:

1. whether the replies were correct with respect to the requests (as described in the functional specification),
2. whether the solution was able to process the generated load.

The test lasted for 20 minutes in order to find whether the integration solution was able to sustain the prescribed load for a longer period of time. During that time the test client generated 1200 requests, in exactly 1-seconds interval, i.e. 60 requests per minute.

The test was successful: all the replies were correct, and the solution was able to process the load, as described in the following.

First, we have observed that messages in channels did not built up, meaning that the solution was able to process them continually.

As for processing times, we expected the following distribution: We had three classes of requests:

1. orders with invalid product types: 120 of them[20],
2. orders rejected due to insufficient credit and/or inventory: 180 of them[21],
3. orders accepted: the rest, i.e. 900.

For each of these classes we expected the following processing times – assuming a message is processed by an idle system and taking into account the control flow

[20] The specification assumes that 1% of all *order lines* were of invalid type, and each of testing orders consisted of up to 10 order lines. We constructed testing input so each order had exactly 10 order lines and 10% of orders contained 1 invalid order line, making for 1% of all order lines being invalid.

[21] The specification assumes that 15% of orders are rejected due to insufficient credit and/or inventory and 85% are accepted. 15% out of 1200 is 180.

between business services shown in Figure 28 and their throughput characteristics listed in Table 10:

1. 120 orders with invalid product types should be processed very quickly, because they take a very short part through the overall process without being processed by a business service with limited throughput.
2. 180 rejected orders: from 3 to 6 seconds.
3. 900 accepted orders: from 7 to 10 seconds.

Real distribution of processing times is shown in Figure 32. Although in the figure we do not distinguish orders by the above categories, we can see that 120 orders were processed under 1,000 milliseconds and the rest took from 4,000 to 22,000 ms to process, with the distribution roughly corresponding to the expected processing times mentioned above. Variations are due to the fact that the system was not idle at all: the threads were busy servicing orders, so some of messages had to wait, generally for a short time. Only 18 orders (1.5% of all orders) took more than 16 seconds to process – this is probably due to burst conditions causing temporary accumulation of a small number of messages at the entry channels of individual business services. Overall, processing times were close to our expectations.

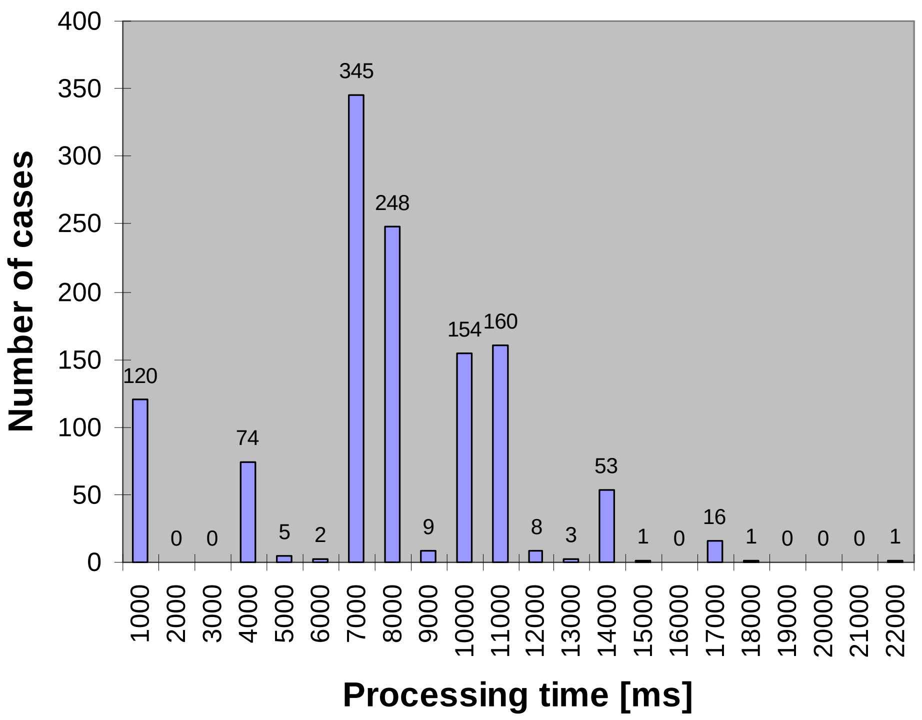


**Figure 32.** Distribution of time needed to process an incoming order.

We also measured the number of messages that went through the messaging middleware. The total number of in-process messages received by MQ was 28,140, what means 1,407 per minute. This corresponds roughly to a prediction given by the method (1,419 messages per minute). The difference is due to the fact that the method cannot predict the actual distribution of invalid items in orders, so it cannot know exactly how many messages are rejected in the middle of the process. Anyway, we consider this calculation to be sufficiently precise for the practical purposes.

Executable code generated for scenarios S4, S5, and S7 has been put into routine use at Comenius University in Bratislava. Scenarios S6 and S8 are in operation as well; they will enter routine use in few weeks. Several parts of scenario S3 have been successfully executed in test environment; its full implementation using U/CP is expected in the near future.

### 6.4.4 Comparison to existing approaches: A product-specific graphical editing environment

We have an intensive four-year experience with developing integration solutions using a professional tool, namely Progress Sonic Workbench (various versions ranging from 7.0 to 8.0), so we can try to compare the development process using this tool and our method.

Similar to other commercial integration tools, Progress Sonic Workbench provides a graphical editing environment for composing integration solutions. Due to the features of underlying execution platform (Progress Sonic ESB), the need to write concrete code is significantly reduced – for most of the time the developer just picks business and integration services, configures them appropriately and connects them together. Yet, as we have also partially described in (Mederly and Pálos, 2008), some drawbacks of this environment are:

1. The complexity of integration solutions created is a significant factor that limits their understandability and therefore maintainability.

2. Concepts of service types, services, endpoints, channels (topics and queues), threading, deployment of services in containers, and ESB processes are not easy to learn. Author of this dissertation had to take four days of intensive on-the-job training (mentoring) combined with a couple of months of study and practicing in order to become productive as a developer in this environment.

3. The negative effects of solutions' complexity are amplified by the lack of adequate visualization and browsing capabilities. Moreover, although it is possible to write comments and notes directly into integration solutions, it is not easy to display them in a visually convenient manner while editing the solutions.

4. Some tasks, like creation of a new process and deploying it in a container, take more developer's actions (mostly clicking and filling-in forms) than would be necessary.

Points 3-4 above could be considered not essential and rather easy to overcome. As for point 3, it would be possible to create an add-on tool for visualization and browsing of existing integration solutions. Concerning point 4, the situation is getting better as the product evolves and can be improved further by creating a specialized add-on tool as well.

Yet the integration solutions' complexity (point 1) is perhaps the most significant drawback. The developer has to deal with many technical details, like how to transport pieces of data to particular points of the solutions e.g. by storing them at appropriate places of messages, whether to use Java Message Service (JMS) queues, topics, or in-memory channels, how to direct message flow into appropriate container or containers[22], how to ensure message validation, logging and auditing, and so on. These details are not hard to solve per se, but when combined, they quickly conceal the core integration logic that has been implemented and make whole integration solution hard to understand and maintain. For a simple case study on this point, please see (Mederly, 2009a).

Our method directly attacks this problem of complexity. By taking care of technical details, it allows the developer to concentrate to the abstract control and data flow model. Moreover, these abstract models are much simpler than the final implementation, so they can be more easily comprehended and modified, if necessary.

**Quantitative measurements**

In order to determine how "much simpler" are these abstract models we have tried to compare the complexity of development of integration solutions using Progress Sonic Workbench and U/CP using a quantitative measurement.

Our first experiments were based on measuring the volume of source code needed to create such solutions. Their results are published in part in (Mederly and Návrat, 2011). The most important findings are summarized here.

First of all, it is not possible to compare the amount of code needed directly. The reason is that the integration solutions are created in Sonic Workbench using graphical editors, while U/CP uses a textual language. Therefore we have decided to estimate the amount of code using the *number of symbols* that the programmer has to employ.

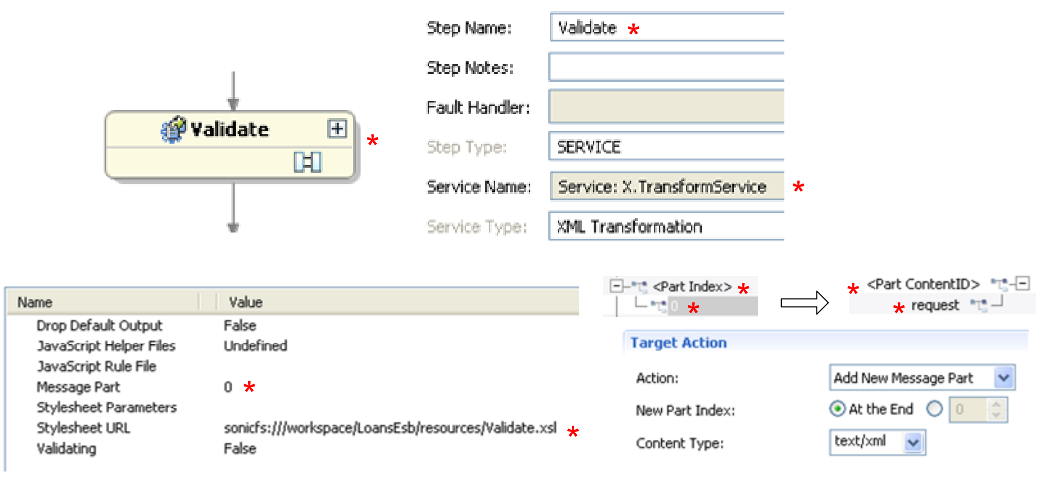
In the textual language of U/CP we count as symbols for example keywords, identifiers (names of processes, services, data types, or variables), and strings (e.g. network addresses, file names, and so on). We omit auxiliary symbols like semicolon, brackets, '=' sign (except cases where it stands for an assignment command), and so on.

[22] Although Sonic ESB has a mechanism for dynamically choosing between in-memory and MQ-based channels (called intra-container messaging), there are situations when its use makes the behavior of the solution harder to understand correctly.

In the graphical language we count as symbols identifiers and strings as well, but also making a choice from presented options, dragging a component from a palette, etc.

In both cases we do not count comments; although in the graphical language we count the step names, as without using them the code would be unintelligible for a programmer.

An example of symbols counting is shown in Figure 33; asterisks indicate symbols counted.



| | |
|---|---|
| Step Name: | Validate * |
| Step Notes: | |
| Fault Handler: | |
| Step Type: | SERVICE |
| Service Name: | Service: X.TransformService * |
| Service Type: | XML Transformation |

| Name | Value |
|---|---|
| Drop Default Output | False |
| JavaScript Helper Files | Undefined |
| JavaScript Rule File | |
| Message Part | 0 * |
| Stylesheet Parameters | |
| Stylesheet URL | sonicfs:///workspace/LoansEsb/resources/Validate.xsl * |
| Validating | False |

Target Action

| | |
|---|---|
| Action: | Add New Message Part |
| New Part Index: | [✓ At the End] ○ 0 |
| Content Type: | text/xml |

(a) Progress Sonic Workbench

```
 *     *     *     *           *          *
Xml  status = execute  ("Validate.xsl",  request);
```

(b) U/CP

**Figure 33.** An example of counting the number of symbols used to invoke a XSLT Validate service in Progress Sonic Workbench and U/CP.

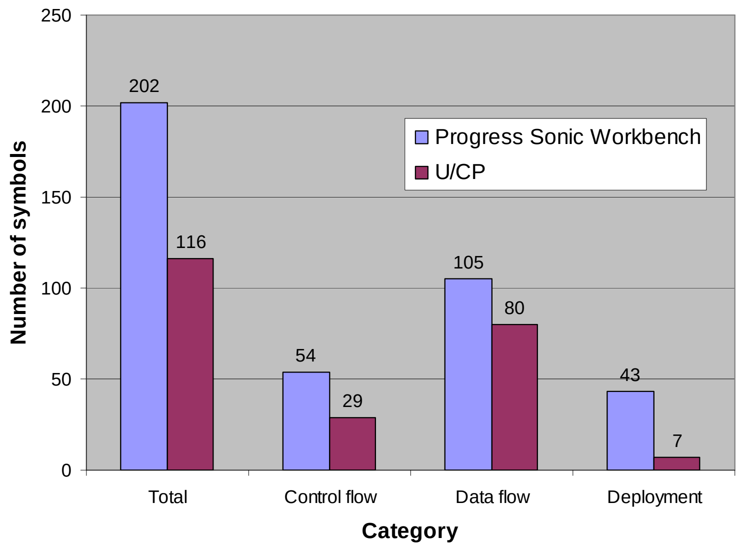
We divided the symbols used into three categories:

1. *Control flow* – these are symbols used to describe the flow of control in the integration solution. Here come e.g. service invocation, branching, looping, and so on. Auxiliary symbols (used in e.g. a module declaration) are counted here as well.

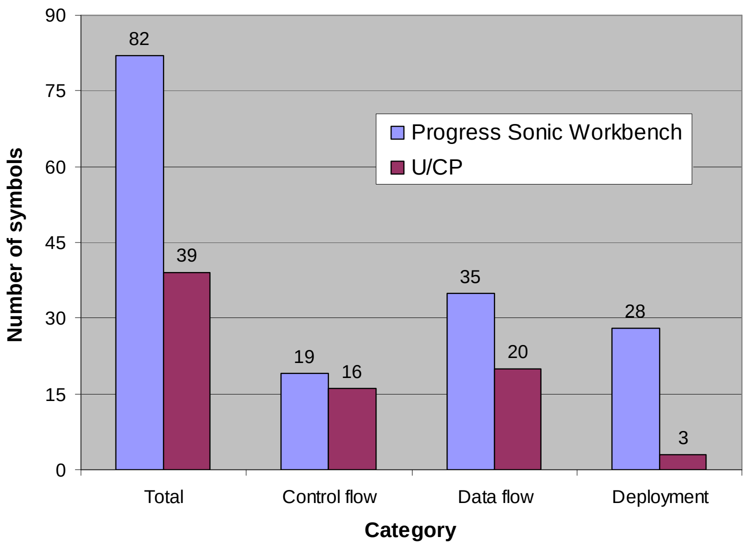
2. *Data flow* – these are symbols used to implement working with process variables (U/CP) and messages and message parts (Sonic).

3. *Deployment* – these symbols are used to specify the deployment of the integration solution into the execution environment. That means specifying e.g. concrete containers, threads, communication endpoints, channels, and so on.

In Figure 34 there are results of analyzing the amount of code required to implement two sample scenarios described in (Mederly and Návrat, 2011). The first one is a modified version of the loan broker scenario[23] (S2) and the second one is the canteen integration scenario (S4).



(a) Loan broker scenario



(b) Canteen menu presentation scenario

**Figure 34.** Number of symbols necessary to implement two sample scenarios using Progress Sonic Workbench and U/CP.

We acknowledge that comparing the number of symbols used is only a very rough estimation of the development complexity. The effort needed to write a program using

[23] The reason for modifying the scenario for (Mederly and Návrat, 2011) was to be able to better compare our run-time performance results with the results of (Scheibler, Leymann and Roller, 2010), as described in the paper.

a given number of symbols depends strongly on what kind of symbols they are and how complicated is their determination (when creating or changing a program) and understanding (when debugging, or before making a change during maintenance).

Nevertheless, as we can see, U/CP brings a significant reduction of the number of symbols used to describe a program – from 202 to 116 (i.e. a reduction by 43%) and from 82 to 39 (by 52%), respectively. Most significant is the reduction that concerns the deployment. What is not visible from the graph, however, are the characteristics of the code that has to be created. In our opinion, the most important change concerns the way in which data is treated. As can be seen also in Figure 33a, in Pipes and Filters architecture (represented here by Progress Sonic Workbench) we work with the content of messages flowing in the system: we have to know what is stored in which part of messages at a service's input, and we must be careful to put appropriate content to suitable parts to be present at service's output. In contrast to it, in the abstract design for U/CP, we just declare what process variables should serve as a service's input and output (see Figure 33b). The method then takes care of the appropriate placement of the data in messages and generates all the necessary code to manage it. So, even if data-management code is reduced by only 24% (from 105 to 80 symbols) and 43% (from 35 to 20 symbols), what is important is the change of the *character* of the code.

In order to get more objective results we are planning an experiment using two programmers developing the same solution using these two environments, measuring the time needed to create and modify such a solution.

**Other aspects**

As a by-product, the method addresses points 2-4 mentioned at the beginning of this section (learning curve, documentation and visualization, and deployment, respective-ly) as well:

- Although it is still useful for the developer to know the details of ESB concepts, it is no longer strictly required.
- As for point 3, this method accepts multiple representations of input models, namely using domain-specific languages that are XML-based (implemented in Prototype 2), Java-like (Prototype 3) and graphical (planned). It is now very easy to write comments directly into models. Furthermore, in Prototype 2 we have implemented a design documentation generator that is able to graphically show both abstract and concrete designs, i.e. both input and output of our method. An example of such a graphical representation of a part of concrete design for scenario S1 is shown in Figure 35.
- Finally, concerning point 4, our implementation contains a solution deployment module that automatically creates all the necessary artifacts,

including all the ESB processes, endpoints, MQ queues, and all configuration files, reducing unproductive "clicking" by the developer.

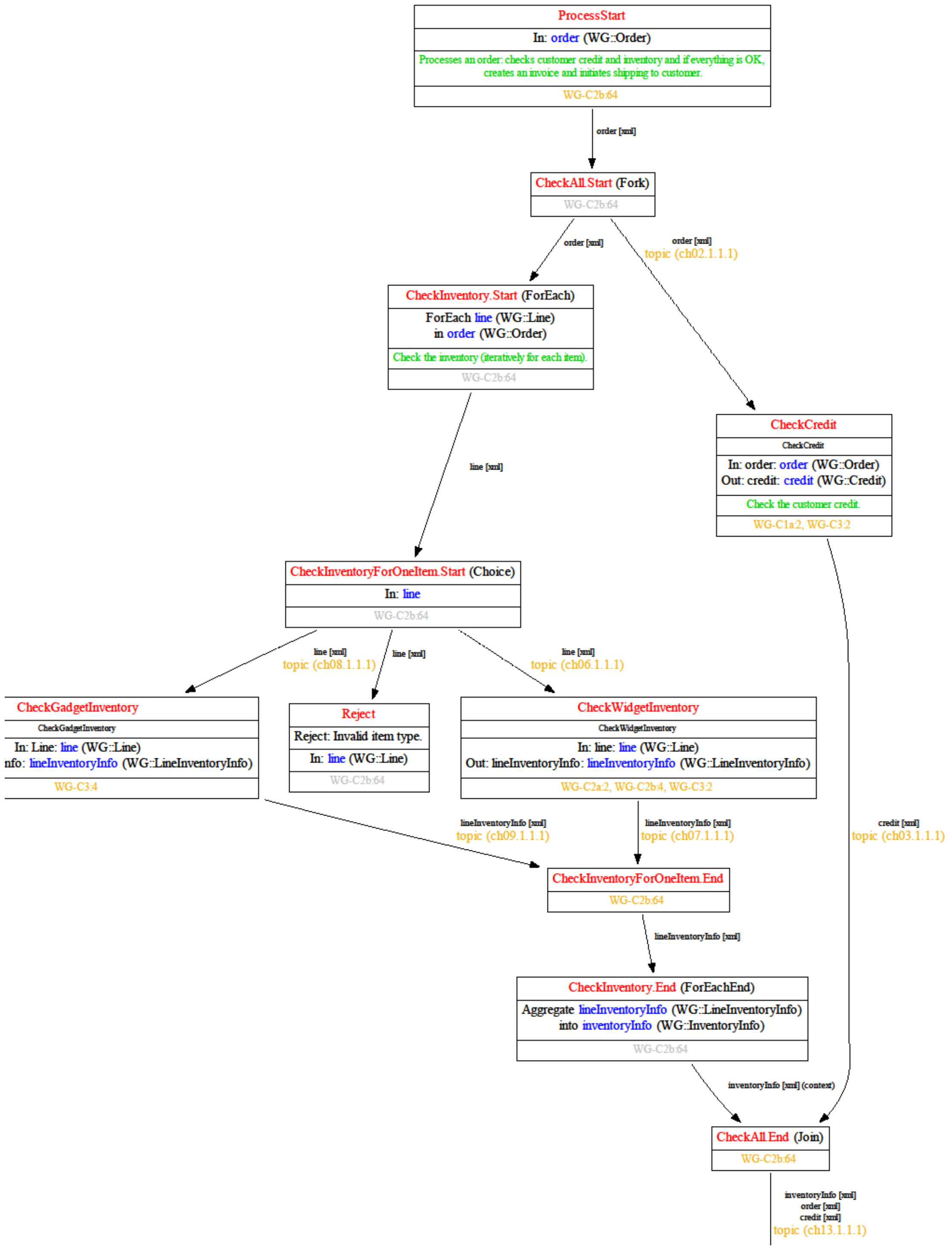

ProcessStart
In: order (WG::Order)
Processes an order: checks customer credit and inventory and if everything is OK, creates an invoice and initiates shipping to customer.
WG-C2b:64
order [xml]
CheckAll.Start (Fork)
WG-C2b:64
order [xml]
order [xml]
topic (ch02.1.1.1)
CheckInventory.Start (ForEach)
ForEach line (WG::Line)
in order (WG::Order)
Check the inventory (iteratively for each item).
WG-C2b:64
CheckCredit
CheckCredit
In: order: order (WG::Order)
Out: credit: credit (WG::Credit)
Check the customer credit.
WG-C1a:2, WG-C3:2
line [xml]
CheckInventoryForOneItem.Start (Choice)
In: line
WG-C2b:64
line [xml]
topic (ch08.1.1.1)
line [xml]
line [xml]
topic (ch06.1.1.1)
CheckGadgetInventory
CheckGadgetInventory
In: Line: line (WG::Line)
nfo: lineInventoryInfo (WG::LineInventoryInfo)
WG-C3:4
Reject
Reject: Invalid item type.
In: line (WG::Line)
WG-C2b:64
CheckWidgetInventory
CheckWidgetInventory
In: line: line (WG::Line)
Out: lineInventoryInfo: lineInventoryInfo (WG::LineInventoryInfo)
WG-C2a:2, WG-C2b:4, WG-C3:2
lineInventoryInfo [xml]
topic (ch09.1.1.1)
lineInventoryInfo [xml]
topic (ch07.1.1.1)
credit [xml]
topic (ch03.1.1.1)
CheckInventoryForOneItem.End
WG-C2b:64
lineInventoryInfo [xml]
CheckInventory.End (ForEachEnd)
Aggregate lineInventoryInfo (WG::LineInventoryInfo)
into inventoryInfo (WG::InventoryInfo)
WG-C2b:64
inventoryInfo [xml] (context)
CheckAll.End (Join)
WG-C2b:64
inventoryInfo [xml]
order [xml]
credit [xml]
topic (ch13.1.1.1)

**Figure 35.** An example of graphical design documentation produced by the U/CP method implementation.

## Other integration platforms

If we would try to generalize this comparison to other messaging-based integration solutions development environments or platforms (like Apache Camel or FUSE Integration Developer), the most important point 1 and (partially) point 2 is still true. As for the other points, we cannot say for certain, as we have not enough experiences with these tools yet.

In the context of considering various platforms, we should mention a strong point of our method: its platform independence. From the abstract design it is easily possible to generate integration solutions for diverse integration platforms. Of course, this assumes that the business services that are composed into integration solutions do already exist on such platforms. This is true for some of the services, for example, services implemented in JavaScript or XSLT can be ported with almost no changes; achieving this kind of portability of other services (for example, Java-based ones, or external web services) is a topic of our further research and implementation work.

## Drawbacks of using the method

There are certain drawbacks of using our method, of course. The main one is a drawback common to majority of model-driven approaches: Although being quite universal, our method and/or its implementations use a limited number of control constructs and make concrete assumptions about the solutions being created, e.g. how the messaging variables are transported within multipart messages. There could be situations where this method would not find a solution as efficient or elegant as a developer would create "by hand".

In (Mederly and Návrat, 2011) we compare the performance of two integration solutions using (1) native Progress Sonic ESB implementations, and (2) Sonic ESB implementation created by the U/CP method. Each solution has been tested using messages of two sizes: 1 KB and 10 KB. Results are summarized in Figure 36.

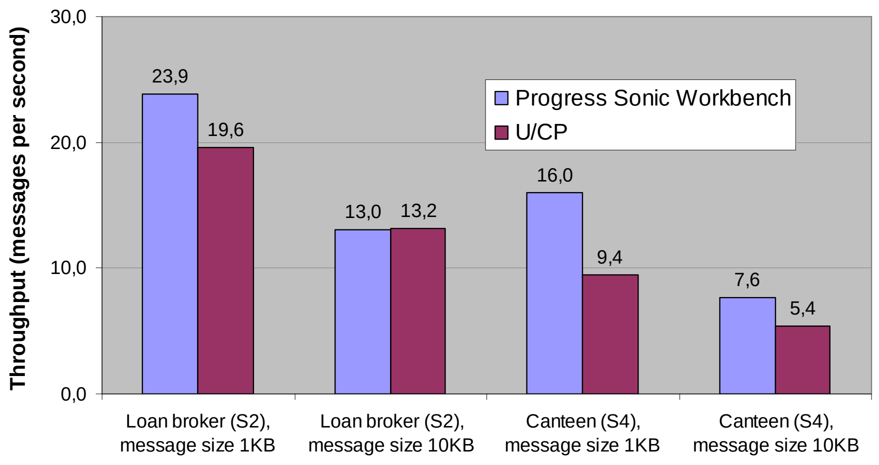


**Figure 36.** Comparison of performance of integration solutions using native and U/CP-generated implementations.

The observed decrease in the run-time performance in the third and fourth case is caused by the fact that in the native Progress Sonic implementation of S4 we have used a feature that is – for the time being – not available in U/CP. As part of further development of U/CP we plan to include it, along with several other features, directly in the method. Other observed slight decrease in the performance (23.9 vs. 19.6 messages per second) is supposedly caused by not optimal implementation of some run-time support components for U/CP-generated solutions. Again, we plan to improve that in the future. More details concerning this comparison can be found in (Mederly and Návrat, 2011).

### 6.4.5 Comparison to existing approaches: Model-driven approaches published in academic literature

Unfortunately, we did not have a possibility to work with implementations of two published approaches to model-driven creation of messaging-based integration solutions, namely (Scheibler and Leymann, 2009) and (Sleiman, Sultán, and Frantz, 2009). However, after careful studying these publications, as well as (Scheibler and Leymann, 2008), (Scheibler, Mietzner, and Leymann, 2008), (Scheibler, Mietzner, and Leymann, 2009), (Scheibler, Leymann, and Roller, 2010), (Frantz, Corchuelo, and Gonzáles, 2008), and (Frantz, 2011), we can state the following:

1. Methods described in the publications above allow the developer to design a solution using abstract components related to enterprise integration patterns.
2. These methods then generate an executable solution based on the given design, for a chosen integration platform.

The developer is therefore freed from the need to write platform-specific code for integration solutions. However, he or she has to provide a detailed design of the solution, so his or her situation is similar to the situation of a designer using product-specific graphical editing environment like Progress Sonic Workbench. There is a difference in that the solution is – at least in theory – platform independent. Unfortunately, when changing the platform, any platform-specific design decisions the developer has made must be revised.

Our method, in contrast, is able to make a number of (potentially platform-specific) design decisions by itself. Moreover, it allows to eliminate some auxiliary services (currently transformation, logging, and validation ones) from the model altogether. This has a positive effect in that a model of the integration solution is much more concise and, at the same time, truly platform-independent. Key benefits for the developer include easy creation of new solutions, quick understanding and good maintainability of existing ones.

### 6.4.6 Other follow-on implementation projects

All three prototypes evaluated in this chapter generate code for Progress Sonic ESB integration platform. Besides that, a separate implementation of the U/CP method for

Apache Camel platform has been created as well. It combines slightly modified design-creating module from Prototype 2 with a newly created code generator. It has been created as a master thesis of Peter Bradáč (Bradáč, 2011) under a supervision of the author of this dissertation.

As part of three U/CP prototype implementations we have created a simple graphical user interface intended for displaying the abstract design, the process of solving CSPs (showing the search tree as well as proposed values of CSP variables), and resulting solutions. Two bachelor-level students have created a more advanced graphical user interface for Prototype 2 (Maršalek, 2011), (Michalko, 2011), again under a supervision of the author of this dissertation. Its overall functionality is similar to the original one; however, it is more comfortable and provides several additional functions that make the navigation through abstract design and concrete solutions much easier. An example of this interface is shown in Figure 37.

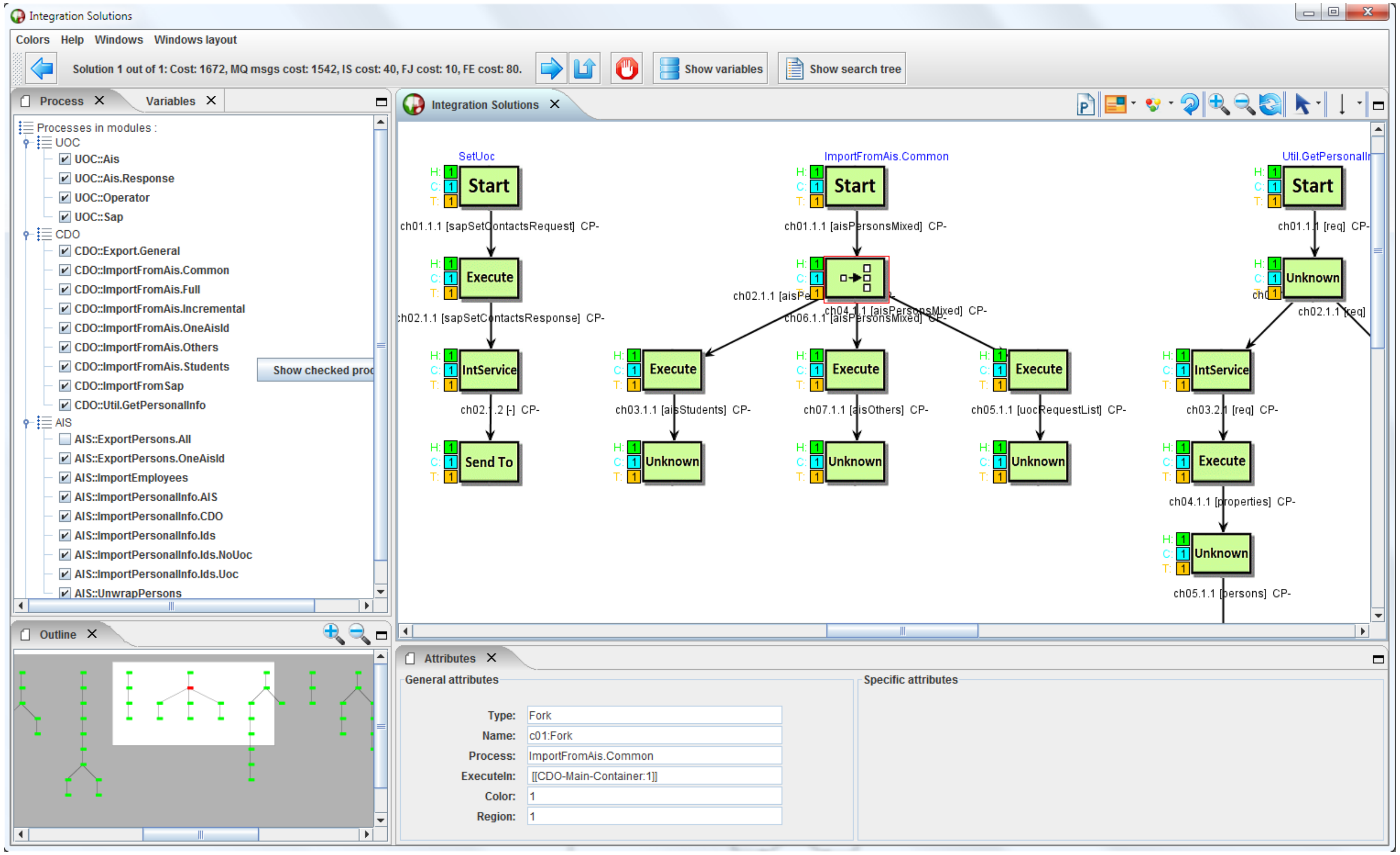
## *6.5 Methods using constraint programming: a conclusion*

As can be seen from the evaluation of methods based on constraint programming, namely the U/CP method, this approach is very effective in creating designs of integration solutions. We observed that this method (and, to some extent, ML/CP as well) is able to solve more complex design problems with more design aspects than our methods based on planning, and generally does it in a shorter time.

We have found these additional facts, partly explaining the above observation:

1. Formulating an integration problem using CSP (i.e. in terms of variables and constraints) is, in some way, easier than formulating it as a planning problem.

   Let us consider abstract design rules, such as those shown in Section 4.2.1, expressed as first-order logic formulas over variables corresponding to solution graph vertices and edges, and functions corresponding to vertex and edge property functions. Our experience suggests that they are much easier to convert to the language of CSP variables and constraints than to the language of planning problem predicates, operators and objects – at least for simple (and, therefore, effective to work with) variants of PDDL language we have used. We have identified two major reasons for this: (1) In variants of PDDL, which we used, we had only a limited set of constructs available, comparing to the set of constraint types we could use in constraint programming-based methods, and (2) in planning-based methods in general we are forced to express properties of the solution as properties of a current state of the world (i.e. a “current” cut of the solution graph), while in CSP we can easily reference any part of an integration solution at any time.



Solution 1 out of 1: Cost: 1672, MQ msgs cost: 1542, IS cost: 40, FJ cost: 10, FE cost: 80.

Processes in modules :
- UOC
  - [✓] UOC::Ais
  - [✓] UOC::Ais.Response
  - [✓] UOC::Operator
  - [✓] UOC::Sap
- CDO
  - [✓] CDO::Export.General
  - [✓] CDO::ImportFromAis.Common
  - [✓] CDO::ImportFromAis.Full
  - [✓] CDO::ImportFromAis.Incremental
  - [✓] CDO::ImportFromAis.OneAisId
  - [✓] CDO::ImportFromAis.Others
  - [✓] CDO::ImportFromAis.Students
  - [✓] CDO::ImportFromSap
  - [✓] CDO::Util.GetPersonalInfo
- AIS
  - [ ] AIS::ExportPersons.All
  - [✓] AIS::ExportPersons.OneAisId
  - [✓] AIS::ImportEmployees
  - [✓] AIS::ImportPersonalInfo.AIS
  - [✓] AIS::ImportPersonalInfo.CDO
  - [✓] AIS::ImportPersonalInfo.Ids
  - [✓] AIS::ImportPersonalInfo.Ids.NoUoc
  - [✓] AIS::ImportPersonalInfo.Ids.Uoc
  - [✓] AIS::UnwrapPersons

| | |
|---|---|
| Type: | Fork |
| Name: | c01:Fork |
| Process: | ImportFromAis.Common |
| ExecuteIn: | [[CDO-Main-Container:1]] |
| Color: | 1 |
| Region: | 1 |

**Figure 37.** Advanced version of the graphical user interface for the U/CP method (Maršalek, 2011).

2. Probably due to facts listed above, we have been able to implement a rich set of design aspects and metrics within the U/CP method (11 and 6, respectively), and we strongly believe that further aspects and metrics can be added as necessary.

3. As metrics can be used to reflect designers' preferences by binding their values to the cost variable (potentially in a form of weighted sum), we have a very flexible tool for defining what we consider to be an optimal solution. In planning, our possibilities for optimization criteria are more limited (see point 2 in the list in Section 5.4).

4. In case of constraint satisfaction we have found a way of partitioning the design problem, described in Section 6.1.3. As a result, the U/CP method can successfully solve bigger problems, with more design aspects, than the other methods. Moreover, further increasing the size of problems and the number of design aspects and metrics, which are used for a given integration problem, seems to be feasible. As for planning approach, our experiences suggest that implementing such a partitioning within the frame of planning-based methods would be harder, probably significantly harder, but without further research we cannot state anything for certain about it.

# Conclusion

In this dissertation we have tried 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.*

*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.*

As for 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 that we have introduced in Chapter 4.

By evaluating these methods in Chapters 5 and 6 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 (Chapter 5). 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 (see Section 5.4).

Third, constraint satisfaction can be used for designing integration solutions as well (Chapter 6). 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 formulated in Section 4.2.1 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, described in Section 6.1.3. 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. Details are described in Section 6.4.4. 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, namely:

> *"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,"*

has been fulfilled.

## 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:

1. Evaluating the benefits of using the U/CP method more exactly, utilizing other source code metrics as well as quantitative measurements of the effort needed to create and maintain integration solutions. We plan to make these results more general by carrying out experiments on several integration platforms.

2. Implementing additional design aspects, including those that we have identified in Section 6.4.1, e.g. message latency issue, more complex message sequencing issues, data compression, multiple usage profiles, and so on.

3. Enhancing the method to be able to work with platform-independent form of business and transformation services (for example, by generating platform-specific wrappers to incorporate such services into a solution).

Among conceptual questions there are:

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. If we add domain-specific information to planning problems generated by ML/P and DL/P methods, how much will it help the planners to find solutions (i.e. plans) more quickly?

3. What other techniques could be used besides planning and constraint programming alone? Would it be helpful to try to combine these two techniques? Or, would techniques known from operations research (e.g. mixed integer linear programming) be useful?

4. Software development companies often have a kind of design guidelines that describe standard solutions for typical design problems. Integration solution development is no exception – for example, there are design manuals showing how to create a process with specified characteristics (e.g. synchronous or asynchronous, query or update, and so on). Would it be possible and beneficial to extend our methods so that they will be able to include such guidelines when proposing solution designs? Would it be also possible to use our methods to verify that the application of particular guidelines is indeed the optimal way to go in a given situation, and to propose their change, if necessary?

5. 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 (Agt, Bauhoff, Cartsburg, Kumpe, Kutsche, and Milanovic, 2009)?

## References

van der Aalst, W. M. P, Dumas, M., & ter Hofstede, A. H. M. (2003). Web Service Composition Languages: Old wine in new bottles?. In *Proceedings of the 29th EUROMICRO Conference "New Waves in System Architecture" (EUROMICRO'03)* (pp. 298-307). IEEE Computer Society.

Agt, H., Bauhoff, G., Cartsburg, M., Kumpe, D., Kutsche, R., & Milanovic, N. (2009). Metamodeling foundation for software and data integration. In *Information Systems: Modeling, Development, and Integration: Third International United Information Systems Conference, UNISCON 2009, Sydney, Australia, April 21-24, 2009, Proceedings* (pp. 328-339). Springer.

Al Mosawi, A., Zhao, L., & Macaulay, L. (2006). A model driven architecture for enterprise application integration. In *Proceedings of the 39th Hawaii International Conference on System Sciences* – 2006 (p. 181c). IEEE Computer Society.

Arshad, N., Heimbigner, D., & Wolf, A. L. (2007). Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Journal, 15*(3), 265-281. Springer.

Bacchus, F., & Kabanza, F. (2000). Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2), 123-191. Elsevier.

Bernstein, P. A. & Haas L. M. (2008). Information integration in the enterprise. *Communications of the ACM*, 51(9), 72-79. ACM.

Bertoli, P., Botea, A., & Fratini, S. (2009). Third international competition on knowledge engineering for planning and scheduling - Report of the board of judges. Retrieved August 11, 2011, from http://kti.mff.cuni.cz/~bartak/ICKEPS2009/download/report.pdf.

Bonet, B., & Geffner, H. (2001). Heuristic Search Planner 2.0. *AI Magazine*, *22*(3), 77-80. AAAI Press.

Boussemart, F., Hemery, F., Lecoutre, C., & Sais, L. (2004). Boosting systematic search by weighting constraints. In *ECAI 2004: 16th European Conference on Artificial Intelligence, August 22-27, 2004, Valencia, Spain* (pp. 146-150). IOS Press.

Bradáč, P. (2011). Model-driven application integration. (Master's thesis). (in Slovak).

Britton, C. (2000). *IT architectures and middleware: Strategies for building large, integrated systems.* Boston, MA: Addison-Wesley Professional.

Chappell, D. A. (2004). *Enterprise service bus*. Sebastopol, CA: O'Reilly Media.

Charfi, A. & Mezini, M. (2004). Hybrid web service compositions: Business processes meet business rules. In *Proceedings of Second International Conference on Service Oriented Computing (ICSOC’04)* (pp. 30-38). ACM.

Charfi, A. & Mezini, M. (2005). Middleware services for web service compositions. In *Proceedings of WWW 2005* (pp. 1132-1133). ACM.

Charfi, A. & Mezini, M. (2005a). An aspect-based process container for BPEL. In *Proceedings of the 1st workshop on Aspect oriented middleware development*. ACM.

Charfi, A. (2006). Aspect-oriented workflow languages: AO4BPEL and applications. (Doctoral dissertation).

Cook, M. (1996). *Building enterprise information architectures: Reengineering information systems.* Upper Saddle River, NJ: Prentice Hall.

Courbis, C., & Finkelstein A. (2005). Towards aspect weaving applications. In *Proceedings of 27th International Conference on Software Engineering (ICSE 2005)* (pp. 69-77). ACM.

Cummins, F. A. (2002). *Enterprise integration: An architecture for enterprise application and systems integration*. New York, NY: Wiley.

Czarnecki K., & Eisenecker U. (2000). *Generative programming: Methods, tools, and applications.* Boston, MA: Addison-Wesley Professional.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., . . . Wilkins, D. (1998). PDDL - The Planning Domain Definition Language Version 1.2. Yale Center for Computational Vision and Control, Tech Report CVC TR-98-003/DCS TR-1165

Druckenmüller, B. (2007). Parameterization of EAI patterns. (Master’s thesis). (in German).

E2E Technologies (2010). E2E | Bridging business and IT. Retrieved August 11, 2011 from http://www.e2ebridge.com/

Eclipse Foundation (2011). Eclipse newcomers FAQ. Retrieved August 11, 2011 from http://www.eclipse.org/home/newcomers.php

Edelkamp, S., & Jabbar, S. (2008). 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*.

Edelkamp, S., & Kissmann, P. (2009). Optimal symbolic planning with action costs and preferences. In *Proc. of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)* (pp. 1690-1695). San Francisco, CA: Morgan Kaufmann Publishers.

Erol, K., Nau, D. S., & Subrahmanian, V. (1992). On the complexity of domain-independent planning. In *Proceedings of the Tenth National Conference on Artificial Intelligence* (pp. 381–386). AAAI Press.

Fikes, R. E., & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*(3-4), 189-208.

Fox, M., & Long, D. (2003). PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, *20*, 61-124.

Frantz, R. Z., Corchuelo, R., & Gonzáles, J. (2008). Advances in a DSL for application integration. In *Actas del Taller de Trabajo Zoco'08 / JISBD Integración de Aplicaciones Web* (pp. 54-66).

Frantz R, Corchuelo R, & Molina-Jimenez C. (2009). Towards a fault-tolerant architecture for enterprise application integration solutions. In *On the Move to Meaningful Internet Systems: OTM* (pp. 294-303). Springer.

Frantz, R. Z. (2011). *Runtime System*. Retrieved August 11, 2011, from http://www.tdg-seville.info/rzfrantz/Runtime+System

Fröhlich, P., & Link, J. (2000). Automated test case generation from dynamic models. In *ECOOP 2000—Object-Oriented Programming* (pp. 472-491). Springer.

Gerevini, A., Saetti, A., & Serina, I. (2003). Planning through stochastic local search and temporal action graphs in LPG. *Journal of Artificial Intelligence Research*, *20*(1), 239-290. AI Access Foundation.

Gerevini, A., & Long, D. (2005). *BNF Description of PDDL3.0*. Retrieved August 11, 2011, from http://zeus.ing.unibs.it/ipc-5/bnf.pdf.

Hammer, M., & Champy, J. (1993). *Reengineering the corporation: A manifesto for business revolution.* New York, NY: HarperCollins.

Hentrich, C., & Zdun, U. (2006). Patterns for process-oriented integration in service-oriented architectures. In *Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPlop 06)* (pp.141–189). ACM.

Hoffmann, J., & Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, *14*(1), 253–302. AI Access Foundation.

Hohpe, G., Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Boston, MA: Pearson Education.

Hohpe, G. (October 15, 2004). *Are "Pattern" and "Component" antonyms?*. Retrieved August 11, 2011, from http://www.eaipatterns.com/ramblings/16_patternscompo-nents.html

*ICAPS Competitions.* (January 20, 2011). Retrieved August 11, 2011, from http://ipc.icaps-conference.org/

Induruwana, C. D. (2005). Using an aspect oriented layer in SOA for enterprise application integration. In *Proceedings of the IBM PhD Student Symposium at the 3rd International Conference on Service Oriented Computing (ICSOC 2005)* (pp. 19-24). CEUR Workshop Proceedings.

Kautz, H., & Selman, B. (1998). The Role of domain-specific knowledge in the planning as satisfiability framework. In *Proceedings of International Conference on Artificial Intelligence Planning* (pp. 181-189).

Kautz, H., & Selman, B. (2006). SatPlan: Planning as satisfiability. In *5th International Planning Competition*.

Koehler, J., Hauser, R., Sendall, S., & Wahler, M. (2005). Declarative techniques for model-driven business process integration. *IBM Systems Journal*, *44*(1), 47-65.

Kolb, P. (2008). Realization of EAI patterns in Apache Camel. (Student Research Project.)

Kuchcinski, K., & Szymanek, R. (2011). JaCoP library user’s guide. Retrieved August 11, 2011, from http://jacopguide.osolpro.com/guideJaCoP.html

Linthicum, D. S. (2003). *Next generation application integration: From simple information to web services.* Boston, MA: Addison-Wesley.

Mach, M. & Paralič, J. (2000). *Problems with constraints: From theory to programming*. Košice, Slovak Republic: Elfa. (in Slovak).

Maršalek, M. (2011). User interface for methods for integration solution generation. (Bachelor’s thesis). (in Slovak).

Mayer, P., Schroeder, A., & Koch, N. (2008). MDD4SOA: Model-driven service orchestration. In *Proceedings of 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC 2008)* (pp. 203-212). IEEE Computer Society.

Mederly, P., & Pálos, G. (2008). Enterprise service bus at Comenius University in Bratislava. In *Proceedings of EUNIS 2008 VISION IT - Vision for IT in higher education* (p.129). University of Aarhus. Available at: http://eunis.dk/papers/p98.pdf.

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

Mederly, P. (2009a). Towards a model-driven approach to enterprise application integration. In *5th Student Research Conference in Informatics and Information*

*Technologies Proceedings* (pp. 46-53). Slovak University of Technology in Bratislava.

Mederly, P., Lekavý, M., Závodský, M., & Návrat, P. (2009). 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* (pp. 37-50). Krakow: AGH University of Science and Technology.

Mederly, P., Lekavý, M., & Návrat, P. (2009). 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* (pp. 56-59). Los Alamitos, California: IEEE Computer Society.

Mederly, P., & Lekavý, M. (2009). Report on evaluation of the method for construction of messaging-based enterprise integration solutions using AI planning. Retrieved August 11, 2011, from http://www.fiit.stuba.sk/~mederly/evaluation.html

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

Mederly, P., & Návrat, P. (2010). 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* (pp. 579-582). Springer.

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

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

Mederly, P., & Návrat, P. (2011). Pipes and Filters or Process Manager: which integration architecture is “better”?. In *Datakon 2011* (to appear) (in Slovak).

Mellor, S.J., Scott, K., Uhl, A., & Weise, D. (2004). *MDA distilled: Principles of model-driven architecture.* Boston, MA: Addison-Wesley Professional.

Michalko, P. (2011). User interface for methods for integration solution generation. (Bachelor’s thesis). (in Slovak).

Mierzwa, Ch. (2008). Architecture of ESBs in support of EAI patterns. (Master's thesis). (in German).

Milanovic, N., & Malek, M. (2004). Current solutions for web service composition. *IEEE Internet Computing*, november-december 2004, 51-59. IEEE Computer Society.

Milanovic, N., Cartsburg, M., Kutsche, R., Widiker, J., & Kschonsak, F. (2009). Model-based interoperability of heterogeneous information systems: An industrial case study. In *Model Driven Architecture-Foundations and Applications: 5th European Conference, ECMDA-FA 2009, Enschede, the Netherlands, June 23-26, 2009. Proceedings* (pp. 325–336). Springer.

Model Labs (2011). Welcome to Model Labs (company homepage). Retrieved August 11, 2011, from http://www.modellabs.de/

Nau, D., Au, T. C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., & Yaman, F. (2003). SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research, 20*(1), 379-404. AI Access Foundation.

Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., & Tack, G. (2007). Minizinc: Towards a standard CP modelling language. *Principles and Practice of Constraint Programming – CP 2007* (pp. 529–543). Springer.

Object Management Group (2010). *OMG Unified Modeling Language (OMG UML), Superstructure. Version 2.3.* Retrieved August 11, 2011, from http://www.omg.org/spec/UML/2.3/Superstructure/PDF/

Object Management Group (2011). *Business Process Model and Notation (BPMN) Version 2.0.* Retrieved July 28, 2011, from http://www.omg.org/spec/BPMN/2.0/PDF

Pan, A., & Viña, Á. (2004). An alternative architecture for financial data integration. *Communications of the ACM, 47*(5), 37-40. ACM.

Papazoglou, M., & van den Heuvel, W.-J. (2007). Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal, 16*, 389-415.

Papazoglou, M. P., Traverso, P., Dustdar, S., Leymann, F., & Krämer, B.J. (2006). 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.

Rauf, I., Iqbal, M. Z. Z., & Malik, Z. I. (2008). UML based modeling of web service composition – a survey. In *Proceedings of Sixth International Conference on Software Engineering Research, Management and Applications* (pp. 301-307). IEEE Computer Society.

Russell, S. J., & Norvig, P. (2003). *Artificial intelligence: A modern approach* (2nd ed.). Upper Saddle River, NJ: Prentice Hall.

Schalkoff, R. J. (1990). *Artificial intelligence: An engineering approach.* Hightstown, NJ: McGraw-Hill.

Scheetz, M., von Mayrhauser, A., & France, R. (1999). Generating test cases from an OO model with an AI planning system. In *Software Reliability Engineering 1999 Proceedings 10th International Symposium on* (pp. 250-259). IEEE Computer Society.

Scheibler, T., & Leymann, F. (2008). A Framework for executable enterprise application integration patterns. In Mertins, K. et al. (eds.): *Enterprise Interoperability III* (pp. 485–497). Springer.

Scheibler, T., & Leymann, F. (2009). From modelling to execution of enterprise integration scenarios: the GENIUS tool. In *Kommunikation in Verteilten Systemen (KiVS)* (pp. 241-252). Springer.

Scheibler, T., Leymann, F., & Roller, D. (2010). Executing pipes-and-filters with workflows. In *2010 Fifth International Conference on Internet and Web Applications and Services* (pp. 143-148). IEEE Computer Society.

Scheibler, T., Mietzner, R., & Leymann, F. (2008). EAI as a service – combining the power of executable EAI patterns and SaaS. In *Proceedings of 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC 2008)* (pp. 107-116). IEEE Computer Society.

Scheibler, T., Mietzner, R., & Leymann, F. (2009). EMod: platform independent modelling, description and enactment of parameterisable EAI patterns. *Enterprise Information Systems*, *3*(3), 299-317. Taylor & Francis.

Schmidt, D. C. (2006). Model-driven engineering. *Computer*, *39*(2), 25-31. IEEE Computer Society.

Schmidt, M.-T., Hutchison, B., Lambros, P., & Phippen, R. (2005). The enterprise service bus: making service-oriented architecture real. *IBM Systems Journal*, *44*, 781-797.

Schmidmeier, A. (2007). Aspect oriented DSLs for business process implementation. In *Proceedings of the 2nd workshop on Domain specific aspect languages.* ACM.

Schmit, B. A., & Dustdar, S. (2006). Systematic design of web service transactions. In *Technologies for E-Services, 6th International Workshop, Revised Selected Papers* (pp. 23-33). Springer.

Skogan, D., Grønmo, R., & Solheim, I. (2004). Web service composition in UML. In *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004)* (pp. 47-57). IEEE Computer Society.

Sleiman, H., Sultán, A., & Frantz, R. (2009). 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* (pp. 134-145).

Szymanek, R. (May 30, 2011). *JaCoP - Java constraint programming solver.* Retrieved August 11, 2011 from http://www.jacop.eu/

Tayi, G. K., & Ballou, D. P. (1998). Examining data quality. *Communications of the ACM, 41*(2), 54-57.

Trowbridge, D., Roxburgh, U., Hohpe, G., Manolescu, D., & Nadhan E. G. (2004). *Integration patterns.* Microsoft Corporation.

Umapathy, K., & Purao, S. (2007). Exploring alternatives for representing and accessing design knowledge about enterprise integration. In *Proceedings of 26th International Conference on Conceptual Modeling* (pp. 470–484). Springer.

Umapathy, K., & Purao., S. (2008). Representing and accessing design knowledge for service integration. In *Proceedings of IEEE International Conference on Services Computing (SCC 2008)* (pp. 67-74). IEEE Computer Society.

Wada, H., Suzuki, J., & Oba, K. (2006). Modeling non-functional aspects in service oriented architecture. In *Proceedings of IEEE International Conference on Services Computing (SCC'06)* (pp. 222-229). IEEE Computer Society.

Wang, Y., & Taylor, K. (2008). A model-driven approach to service composition. In *Proceedings of 2008 IEEE International Symposium on Service-Oriented System Engineering* (pp. 8-13). IEEE Computer Society.

Xing, Z., Chen, Y., & Zhang, W. (2006). 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)* (pp. 53-56).

Xu, Y., Tang, S., Xu, Y., & Tang, Z. (2007). Towards aspect oriented web service composition with UML. In *Proceedings of 6th IEEE/ACIS International Conference on Computer and Information Science* (pp. 279-284). IEEE Computer Society.

# Appendix A: About the author

Pavol Mederly was born in Bratislava, Slovak Republic on May 17th, 1974. In 1997 he received master's degree in informatics at Faculty of Mathematics and Physics, Comenius University in Bratislava. After that, he worked as a lecturer at Faculty of Mathematics and Physics, Comenius University (until 2008) as well as a software and systems engineer and integration specialist at Information Technology Center at the same university. Presently he is a PhD student at the Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava in the field of software engineering. His research interests are integration of information systems, integration patterns, messaging technologies, service oriented architectures, and software engineering in general.

## *A.1 Publications*

**International conferences**

Mederly, P., Lekavý, M., Závodský, M., & Návrat, P. (2009). 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* (pp. 37-50). Krakow: AGH University of Science and Technology.

Mederly, P., Lekavý, M., & Návrat, P. (2009). 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* (pp. 56-59). Los Alamitos, California: IEEE Computer Society.

Mederly, P., & Návrat, P. (2010). 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* (pp. 579-582). Springer.

**Book chapters**

Kišac, I, Kuzár, T., Mederly, P., Tvarožek, J., Kapustík, I., & Habudová, N. (2009). Software architectures. In: Bieliková, M., & Návrat, P. (eds.) *Selected studies on software and information systems 4. The Edition of Research Texts in Informatics and Information Technologies* (pp. 73-113). Slovak University of Technology in Bratislava. ISBN 978-80-227-3139-3. (in Slovak).

Habudová, N., Kišac, I, Kuzár, T., Mederly, P., Šimko, M., & Tvarožek, J. (2009). Design patterns. In: Bieliková, M., & Návrat, P. (eds.) *Selected studies on software and information systems 4. The Edition of Research Texts in Informatics and Information Technologies* (pp. 3-35). Slovak University of Technology in Bratislava. ISBN 978-80-227-3139-3. (in Slovak).

Habudová, N., Kuzár, T., Mederly, P., Šimko, M., Tvarožek, J., & Kapustík, I. (2009). Software components. In: Bieliková, M., & Návrat, P. (eds.) *Selected studies on software and information systems 4. The Edition of Research Texts in Informatics and Information Technologies* (pp. 37-72). Slovak University of Technology in Bratislava. ISBN 978-80-227-3139-3. (in Slovak).

**Regional and local conferences**

Mederly, P., & Návrat, P. (2011). Pipes and Filters or Process Manager: which integration architecture is “better”?. In *Datakon 2011* (to appear) (in Slovak).

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

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

**Student research conferences**

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

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

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

**Other**

Mederly, P., & Pálos, G. (2008). Enterprise service bus at Comenius University in Bratislava. In *Proceedings of EUNIS 2008 VISION IT - Vision for IT in higher education* (p.129). University of Aarhus. Available at: http://eunis.dk/papers/p98.pdf.

Mederly, P. (2010). Towards semi-automated design of enterprise integration solutions, In Bieliková, M., & Návrat, P. (eds.) *Workshop on the Web-Science, Technologies and Engineering: 7th Spring 2010 PeWe Ontožúr Smolenice Castle, Slovakia April 18, 2010 Proceedings* (pp. 75-76). Slovak University of Technology in Bratislava.

# Appendix B: Content of the attached electronic media

**Table 18.** Content of the attached electronic media.

| **File or directory** | **Content** |
|---|---|
| dissertation.pdf | Text of this dissertation. |
| mlp | Artifacts related to the ML/P method |
| - evaluation | Detailed evaluation of the method – a copy of (Mederly and Lekavý, 2009) |
| ucp | Artifacts related to the U/CP method |
| - grammar | Grammar of the input language for U/CP implementation prototype 3 |
| - schemas | XML schemas for U/CP implementation prototypes 2 and 3 |
| - - common.xsd | - input language and common elements |
| - - design.xsd | - language used for describing concrete solution design (output of the U/CP method) |
| - - sonic.xsd | - language used to describe concrete design specific for Progress Sonic ESB (output of the first phase of code generation for Progress Sonic ESB) |
| - scenarios | Artifacts related to individual scenarios: input and output of the U/CP method. |