Legacy Systems and Web Services[1]

Michal Žemlička, Jaroslav Král

Charles University, Prague
Malostranské nám. 25, 118 00 Praha 1
Czech Republic
{michal.zemlicka,jaroslav.kral}@mff.cuni.cz

Technical report KSI MFF UK No. 2004/1

[1]Supported by Czech Science Foundation, grant No. 201/02/1456.

Abstract

It is more and more important for the flexibility of companies to have their services available through web or just as web services. There are also many properly working legacy systems. It is easy to come to the idea to make the legacy system a kernel of a web service via adding new interface to the legacy system. This paper gives an overview of basic problems that the developers in such cases usually face and gives also some hints how to overcome them.

Chapter 1

Introduction

Current trend is "faster and globally", sometimes "faster and everywhere". It induces that services that were available locally or indirectly only should be available permanently online by web interface (for human use) or as web service(s) (also for automatic use). The development speed is very significant – who is too slow cannot be successful (is on the market too late to get interesting share; except investing large amount of money into very strong advertisement or illegal pushing of the product) and therefore it is no time to redevelop the service. It is a strong reason why to use legacy systems. There is also other reason why to use a legacy system: there can be some know-how hidden in legacy sytem LS that can be lost if LS must be redeveloped.

The idea to use legacy system has more advantages that are visible at a the first sight. The legacy system is usually proven that it matches some area of user needs and does there what user wants and/or expects. There are no costs related to the requirement specification – it is only needed to state whose services will be made available through a new interface of a legacy system. The new interface of given legacy system should be specified in a problem-oriented/user-oriented way – it should hide the implementation details of the legacy system. Such interface is usually very stable, as the problems that people have to solve tend to be stable for longer time than the methods used to solve it.

The legacy systems are, however, usually not written to enable future addition of new interfaces. Even if the source code of an application A is available (and it is not always the case), it can be quite difficult to provide a new gate of A. The source code can be lost or was not delivered at all (especially if the application has been developed by some other company).

If we want to transform a system having no available source code into a web service, we must apply special technical turns discussed below.

Legacy systems must often keep functions and interfaces specified in their original requirements specification, their 'local' users should use them as before – it is, the users should not to mention that 'their' application has been converted into a web service or generally a service in the sense of [KŽ04a].

The below discussed techniques are not new. We attempt to give an overview of the techniques for the development of new legacy systems interfaces and the practical experience with some of them. We also discuss some techniques of overcoming some obstacless of the development of new interfaces of the legacy systems having no available/maintenable source code.

Chapter 2

Integration of Service-Oriented Systems

There are many very large and complex projects. The tasks of the projects can be very often divided into smaller tasks. The smaller tasks are already supported by some software solution (information systems). Then the support the complex task can be implemented by a system integrating the supporting subsystems via a proper interface.

Implementation of complex systems like the systems in *e*-government takes a lot of time, effort, and other resources. Such system must be in use for a long time – at least until a new system is developed. But the world is changing very fast. New requirements occur, other can become obsolete. The system is facing to many changes during which it should be able to work. The changes should be therefore kept as local as possible. There is a way: the interfaces between the local solutions should be kept as stable as possible. It is possible if the interfaces are problem-oriented. It can be achieved if we use two-level interfaces as described e.g. in [KŽ03]. The system then can be integrated from 'autonomous components' having the properties of autonomous real world like services forming a (virtual) peer-to-peer network. Examples of such services are web services.

Such an architecture has many advanatages discussed in detail in [KŽ04a, KŽ04b]. One of the most important advantages is that such a system can be modified continuously and can integrate a legacy system into quite modern systems. The legacy system can have 'old interface' (as is had before the integration) as well as a new interface integrating it into the new system.

Such systems (eventually integrating legacy systems) must have the architecture of a

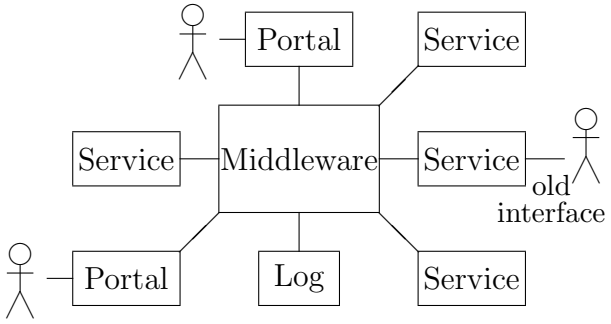

Figure 2.1: Structure of a confederation, the gates are not known

peer-to-peer network of peers behaving like autonomous real-world services. The resulting system has then the form from Fig. 2.1.

We say that such systems are *service-oriented* and that they have a *service-oriented architecture*. We shall discuss the techniques how to integrate a legacy system as a service in Fig. 2.1.

Chapter 3

Message-Driven or Data-Driven Integration?

The integration of almost independent subsystems (let us call them *autonomous components* or *autonomous services*) can be performed via communication based on data or message interchange. Both solutions have their pros and cons and their optimal application area.

Data-level interconnection is good if we need an access to the applications (raw) data for tools like OLAP. Interconnection at data level has also some drawbacks: it is dangerous for the security and safety of data. It is difficult to control what data the partner application is accessing especially in writing and updating. Then there is a quite high danger of spoiling the data integrity. Another drawback is that all the applications sharing the data should respond to all the changes of the shared underlying data structures. There can be response time problems in real-time systems.

Despite these drawbacks the data-oriented interface can be necessary for the communication between services. It is always in the case that some of the services of the system require as its input analysis of the data from a database (datastore). An example is in [KŽ04a] where the interface between Scheduler and Flexible Manufacturing System is implemented via a datastore [KD79]. The datastore enables an intelligent communication between services.

Message-oriented communication is used in the majority of cases. It allows to use the service by other applications without its prior (re)development (that is expensive and can be a source of many failures of the system) and, what is also very important, it allows to hide the implementation details of the service to the developers of the new system. It makes its analysis, design, and development simpler. This type of communication is also understandable to the majority of the system users and therefore it can be probably better specified and analyzed.

As there are two different types of application communication, we discuss both cases. – although in many cases only one variant of the communication is needed and must be implemented.

3.1 Starting Conditions and Their Consequences

The presence of the legacy system source code is crucial for the gates implementation. Having the sources we can add a new interface to the system. The interface can transform the legacy system into a (web) service. If there is no source code, the existing interface

should be used as it is.

The direct use of existing inputs and outputs is usually no simple task. We discus a technical turn how to successfully extend a legacy system to be a (web) service.

It is sometimes possible to redirect the input and output streams. Unfortunately, it can be used quite rarely. Generally we must redirect all the inputs and outputs of the legacy system user interface to a piece of software called *primary gate* (Fig 3.1). We usually must 'hack' the screen to catch all the outputs (inclusive direct writes to the screen memory and other tricks). Similarly keyboard and mouse must be 'hacked' to allow us to enter our requests and data [Žem]. It can be done by writing a special application that can be run as a sandbox for the legacy system [Žem]. It follows the changes on the screen making from them messages that are sent for processing by other parts of the newly developed (web) service. Similarly, the messages from outer world can be used (after some modifications) as inputs of the legacy system.

If the legacy system has a structure (e.g. it integrates several components) recognizable from outside (and the properties are documented or easy to recognize), our task is simplified as further information can be used.

The presence of source code need not always be an advantage: if the source code of the legacy application is changed to implement the interface, some new errors can be introduced. Therefore not only the newly added parts but whole legacy system should be tested (compare regression testing – see e.g. [Pre97]).

3.2 Decomposition of the Added Interface

If a newly created (web) service has to be in use for a longer time, it is advantageous to decompose it into three components with the following functions (see Fig. 3.1): Legacy system, its primary gate, front-end gate, web service (if desirable).

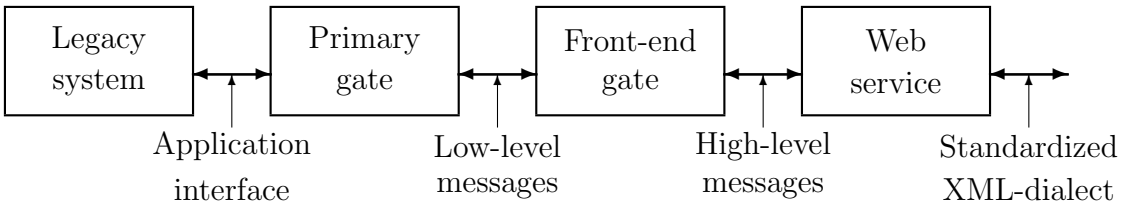

Figure 3.1: 3-level wrapping

3.2.1 Primary Gate.

The software component enabling the 'direct' access to the legacy system is called *primary gate*. It can be integrated into the legacy system (provided its source code is available); it can be a special part – a client tier (in the case of client-server or three-tier architecture), or it can be a software component (service) capturing and/or redirecting inputs and outputs of the legacy system.

The purpose of the primary gate is to transform any kind of communication with the legacy system into messages for communication via a middleware (Fig. 3.2).

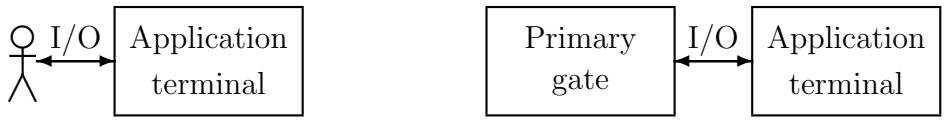

Figure 3.2: Redirection of application I/O

Let us note that the primary gate can be (as the whole web service) data-driven or mesage driven. Message driven communication is typical in *e*-commerce.

The creation of primary gate(s) may be close to hacking and therefore people with hacking abilities are good in writing this part of the gate. In the case when source code is available, the people from a maintenance team can be the best choice.

3.2.2 Front-End Gate.

The component/service converting sequences of messages/commands close to the logic (or interface) of the legacy system into sequences of messages/commands having the form required by the problem-oriented (high-level) web service interface is called *front-end gate*. The name is derived from the observation that from outside this part makes the real high-level interface to the legacy system. The web service can be seen as a wrapper enabling the application current communication standards.

This task is very close to the situation known from the compiler construction where there are also sequences of commands in one language converted into sequences of commands in other language. This topic has been deeply studied and the results are published in many publications (let us notice at least a few of them: [AU72, AU73, ASU86, WM95]).

Compiler construction was – and often is – one of the popular subjects for computer science students. Hence it is possible to expect, that there is a lot of people able to solve such tasks.

In some special cases it can happen that the languages on both sides of the front-end gate are similar enough that standard XML-based tools like XSLT [W3 99] can be used. The problem with XSLT is that XSLT machines are error prone and very inefective [Ř03, Ric02].

The most important reasons for developing front-end gates are:

- the interface stability (well-designed high-level problem-oriented interface between front-end gate and its web interface should work as boundary where the propagation of most changes of the system can be stopped – as from the partner side as from the legacy application side),
- the interface accuracy (if there are more partners with different access rights, it is advantageous to build for each such group a specific front-end gate exactly matching the rights and needs; it increases the security and simplifies the interface for the partners),
- an easier development and testing (as the task is divided into several smaller parts it can be easier to develop it and test – at least the errors tend to be more local and easier to detect).

- a tool for converting applications to web services.

Authors' experience says that this part is the hardest pill to swallow not only for many students, but also for many graduated people (in both cases especially for the ones using object-orientation only).

3.2.3 Web Service.

Web services change and develop very quickly. Their basic interface is therefore not stabilized yet. There are many standards defining communication protocols and related services (SOAP [W3 00], UDDI [UDD03], WSDL [W3 01], etc.) but they are cumbersome, changing very fast, and sometimes they are only partially backwards compatible.

Separation of activities dependent on quickly changing standards allows us to change the smallest possible part of the whole web service. It is easier to maintain and there is less new errors introduced.

Under some circumstances it is also possible to integrate web services based on different standards.

3.3 Legacy System Integration Depends on its Structure

The legacy system can have different structure. If we want to apply the principles from Fig. 3.1, we often must develop the primary gate. It depends on the properties of the legacy systems. There are the following possibilities.

- monolithic application,
- structured application
  - client-server with thin clients,
  - client-server with thick clients,
  - three-tier architecture.

3.3.1 Interactive Single-User Application.

Interactive single user applications are usually built as one executable module. If the program is a legacy system, or if it is bought from a third party, the documentation and source code are only rarely available. The only possibilities of connecting other applications to the given one are typically accessing application's data files or redirection of its terminal inputs and outputs to a software component serving a primary gate. If the system file/stream redirection is not possible, the last (emergency) possibility to attach to the legacy system is to hook its terminal inputs and outputs by monitoring screen changes and simulation of keyboard and mouse activities.

In such a situation it is better if the application interface is based on text mode (character stream) rather than on GUI. If the application is GUI-based (like X-Window, MS Windows or Workplace Shell), such a version of the desktop management allowing to capture messages sent by the application to the desktop library should be used. Such

messages can be afterwards transformed into messages in a format usable by front-end gates.

Sent messages follow the changes on the screen (error and status messages, menu selections, etc.). Accepted messages are counterparts of keyboard or mouse activities (like menu selections).

If the application source code is available and if it is internally well structured, we can try to create a primary gate in the form from Fig. 3.3.

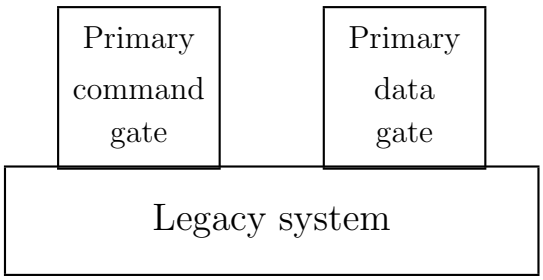

Figure 3.3: Tightly bounded primary gates to a single-user monolithic interactive legacy system

3.3.2 Interactive Multi-User Client-Server Application.

The client-server architecture has two basic variants: thin and thick clients. Each of the variants has its advantages and disadvantages in both their development and use. The possibilities of such systems to be integrated into larger entities, respective to be added to other applications or to add new application are influenced accordingly to the client-server variant.

Thick clients.

The application is divided into two parts: data-oriented part is on the server, application logic and user interface is performed by the client part of the software. In such case the interface between server and client can be used to attach the data gate. The command gate must be developed similarly to the solution known from single-user monolithic applications: input/output redirection and/or caption (see 3.3.1).

Thin clients.

The client-server scheme with thin clients concentrates on the server both data and application logic. Therefore the client-server interface can be used for a primary gate working as a special kind of client. The possibility to use standard clients and redirection of its inputs and outputs to primary gate is also in this case left open.

The "special client" case seems to be advantageous: the communication between client and server part of the legacy application is already done by sending messages. The task of primary gate is simplified to conversion the legacy system protocol to the one used by middleware used for communication between primary gate, frond-end gate(s), and the web service part.

The construction of primary data gate is quite easy if the data are stored in a standard database and the data definitions are accessible. Otherwise a source code of the server programs must be accessible.

3.3.3 Three-Tier Architecture.

Applications based on the three-tier architecture (data tier, application logic, user interface) can use all the methods mentioned above: redirection of client's inputs and outputs, special clients, data gate as special "application" connected via the application tier – data tier interface, and legacy system improvement(s) by the primary gate(s). The solution using special clients is shown in Fig. 3.4, the solution with code improvement is shown in Fig. 3.5.

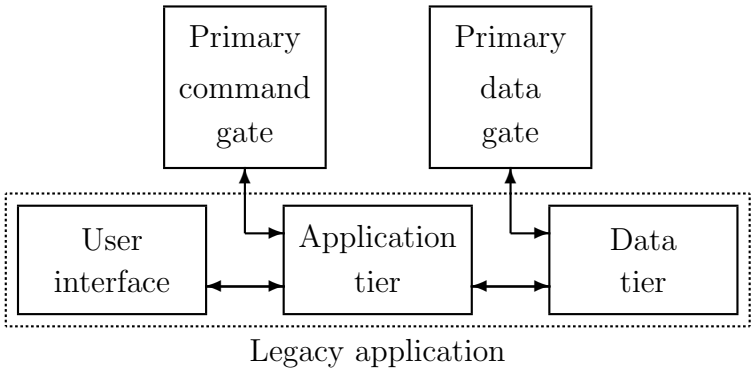

Figure 3.4: Loosely attached primary gates to a three-tier legacy application. Gates are separate applications.

3.4 Comparison of Data and Command Primary Gates

Until now it has been expected that there would be used as command access as data access to the application; in some cases even its user interface. Let us compare the

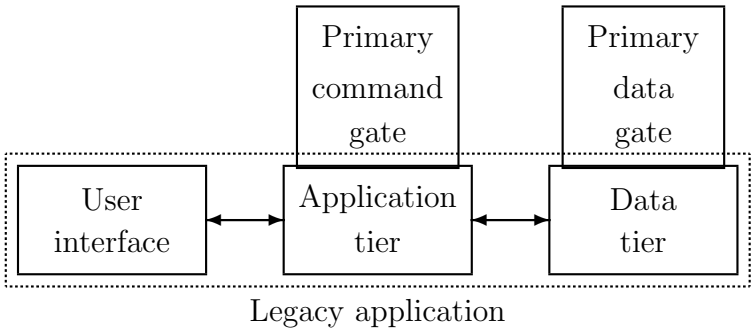

Figure 3.5: Tightly bounded primary gates to a three-tier legacy system. The gates are parts of the tiers.

different solutions.

3.4.1 Data Access.

Accessing the data part of the application makes the massive data accesses faster (there is no overhead caused by the application tier or application logic part of the legacy system). In order to keep the data consistent we need, however, to redevelop some parts of the application logic again. As a "bonus" we obtain also higher risk that the integrity and security of the application data can be broken.

Giving full access to the application data gives foreign users (users of the partner applications) full access to our data that probably are preserved also to our users. Therefore the data gate and/or its front-end gate(s) should be equipped with identification and access rights.

Sometimes using direct access to data part of the application can destroy all the advantages that we have expected from the integration of the legacy system.

3.4.2 Command Access.

Accessing application logic part (through legacy user interface or through newly developed primary gate) offers us all the "public" services of the original application. Using commands sometimes may also hide the situation that some part of the commands is physically processed – by some officers, machines, etc. Solving such situation on purely data level is without special tricks impossible.

High-level command interface that should be accessible through the couple (front-end gate, primary gate) is close to the way the most people think and talk.

3.5 Integration in Service-Oriented Systems

If there are more applications equipped with primary and front-end gates, it is possible that the applications can use services supported by other applications via their front-end gates or web service gates. Such group of permanently running autonomous applications is called service oriented system.

The principles of service-oriented systems can (should) be applied especially by huge, complex, or modifiable systems (see e.g. [KŽ00, KŽ01, KŽ03]).

One of the basic features of the service-oriented systems is that they preserve the original user interface of the applications – therefore it is possible to use them both as usual or newly using web interface.

We have shown that the resulting (integrated) system should have the logical structure of a peer-to-peer network of permanently active processes (called services) provided by peers, i.e. by the applications with gates. It is preferable to implement the user interface of the network as a specific service (peer).

Chapter 4

Conclusions

Building web services based on legacy systems is technically feasible. There are many problems related to it: the legacy system can change, or the interface of the just created service can change, or both. Reason for a change in the interface can be the change in the needs of clients as well as the change in the web service standards.

Evolution of web service standards (and, more generally, all standards having anything common with web) is so dynamic that within at least next few years the changes in web standards would belong to the most frequent reasons (probably with the exception of error fixing) reason for changes in more complex web services.

The main idea of longer-existing web services based on existing applications is a decomposition of newly developed part of the service into several (typically three) partitions. Each partition would solve task of other type and would demand from the developers other knowledge and techniques:

1. A connection to the legacy application must be established. It is implemented as a primary gate that enables access to the services of the legacy system for the other parts of the newly developed web service. People able to program at low-level (capturing messages for the screen, mouse movements and clicks simulation, and "pressing" the keys, etc.) are expected to be successfully used here.

   If several legacy systems should be integrated as web services into one new whole, we can develop a primary gate template. This template can solve the problem of the communication with front-end gates. It simplifies the development of primary gates as the hackers can concentrate on just one problem (communication with the legacy application) only.

2. It is further needed to convert the communication based on user interface of the legacy system to the communication further usable by a web service. It will be typically a conversion between implementation-oriented communication with the application (via primary gate) and the problem-oriented communication with the web service. This part is the role of front-end gate. There can be more front-end gates interfacing one legacy system – supporting as many different interfaces or access rights levels as required.

   If there are more applications cooperating for longer time, they can access the services of the partners via front-end gates This interface does not need to follow all the changes of the web standards and therefore can be more stable.

   The compiler developers (especially the ones handling back-ends) can be the best choice for the front-end gate(s) development. Compiler development is supported

by many tools. Often it suffices to specify a formal description of input and output languages, translation rules and some transformations and optimizations and the tool generates significant part of the front-end gate.

3. For every front-end gate the last part providing standardized web service interface must be developed. It is possible to use here a template for a typical web service that can be adapted by descriptions of the service and its communication protocol. This part can be integrated with its front-end gate to one application/process but we assume that it is better to develop these parts separately.

Above described solutions are applicable using arbitrary powerful enough middleware – like in the grid-computing.

Further research will be oriented toward as precise as possible specification and creation of templates simplifying development of web services based on legacy applications.

Bibliography

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*, volume 10194 of *World Students Series Edition*. Addison-Wesley, 1986.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume I.: Parsing. Prentice-Hall, Englewood Cliffs, N.J., 1972.

[AU73] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume II.: Compiling. Prentice-Hall, Englewood Cliffs, N.J., 1973.

[KD79] Jaroslav Král and Jiří Demner. Towards reliable real time software. In *Proceedings of IFIP Conference Construction of Quality Software*, pages 1–12, North Holland, 1979.

[KŽ00] Jaroslav Král and Michal Žemlička. Autonomous components. In Václav Hlaváč, Keith G. Jeffery, and Jiří Wiedermann, editors, *SOFSEM'2000: Theory and Practice of Informatics*, volume 1963 of *LNCS*, pages 375–383, Berlin, 2000. Springer.

[KŽ01] Jaroslav Král and Michal Žemlička. Electronic government and sofware confederations. In A M. Tjoa and R. R. Wagner, editors, *Twelfth International Workshop on Database and Experts System Application*, pages 125–130, Los Alamitos, CA, USA, 2001. IEEE Computer Society. ISBN 0-7695-1230-5, ISSN 1529-4188.

[KŽ03] Jaroslav Král and Michal Žemlička. Software confederations – an architecture for global systems and global management. In Sherif Kamel, editor, *Managing Globally with Information Technology*, pages 57–81, Hershey, PA, USA, 2003. Idea Group Publishing.

[KŽ04a] Jaroslav Král and Michal Žemlička. Service orientation and the quality indicators for software services. In Robert Trappl, editor, *Cybernetics and Systems*, volume 2, pages 434–439, Vienna, Austria, 2004. Austrian Society for Cybernetic Studies.

[KŽ04b] Jaroslav Král and Michal Žemlička. Towards design rationales of software confederations. In Isabel Seruca, Joaquim Filipe, Slimane Hammoudi, and Jos Cordeiro, editors, *ICEIS 2004: Proceedings of the Fifth International Conference on Enterprise Information Systems*, volume 1, pages 105–112, Setúbal, Portugal, 2004. EST Setúbal.

[Pre97] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, fourth edition, 1997.

[Ric02] Karel Richta. Using XSL in IS development. In *New Perspectives on Information Systems Development: Theory, Methods, and Practice*, pages 309–319, New York, 2002. Kluwer Academic / Plenum Publishers.

[UDD03] UDDI Initiative. Universal definition, discovery, and integration, version 3, 2002–2003. An industrial initiative, http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv3.

[Ř03] Kamil Řezáč. Xslt as a tool of the translation of xml dialects (in czech). Master's thesis, Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic, December 2003.

[W3 99] W3 Consortium. XSL transformations (XSLT), 1999. W3C Recommendation. http://www.w3c.org/TR/xslt.

[W3 00] W3 Consortium. Simple object access protocol, 2000. A proposal of W3C consortium. http://www.w3.org/TR/SOAP.

[W3 01] W3 Consortium. Web service definition language, 2001. A proposal of W3 consortium. http://www.w3.org/TR/wsdl.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. International Computer Science. Addison-Wesley, Wokingham, England, 1995.

[Žem] Michal Žemlička. IDOSgrab. Wrapper to a thrird party application to access its data.