Position paper featured in the 2003 International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA), Amsterdam; September 23, 2003

Observations on automation in cross-platform migration

Ben Wilson, Tony Van der Beken
Anubex
Veldkant 35C, Kontich Belgium

E-mail: ben dot wilson at anubex dot com, tony dot vanderbeken at anubex dot com

Abstract

There are numerous ways for organisations to migrate an operational information system from one deployment platform to another. This paper relates a number of experiences of applying automated techniques to cross-platform migrations of larger (> .5MLOC) information systems in real world projects. The paper examines these experiences, considering factors influencing the organisations' decision for the approach, the project-specific features and limitations of the approach, and the effects of the approach on the organisational context. This paper does not attempt to provide an exhaustive comparison of the advantages and characteristics of the different approaches that may be used, but rather to consider a single approach in more detail, based on the experience of the authors. It is our position that the use of automated methods will increase as the risk-elimination effects of the technique will ensure its rise in popularity, and information systems increasingly outlive the platforms for which they were developed.

1. Introduction

Calculating the actual number of time-proven, mission-critical information systems with over a half million lines of program code currently in operation worldwide is a nearly impossible task. The term 'legacy' is often used to describe these systems, and with the term a number of negative and subjectively sensitive attributes typically spring to mind: ([3], [4], [9], [10]) hard to understand, insufficiently documented, difficult to maintain, and unintuitively structured are some of them. In the typical industry jargon, the term 'quality' ([5], [7], [8], [11]) is often quoted as a blanket term, and can be used to refer to a combination of any number of these attributes.

The notion of 'poor quality' can be used as an argument to convince the organisation owning the information systems into undertaking a revolutionary reengineering effort ([7], [8], [9]). Typically such an effort involves understanding a program's functionality (perhaps aided with a 'legacy understanding tool,' a 'business rules extraction tool,' or a 'code slicing tool'); storing this information in some form of a repository or representing the code in an easier-to-understand format or a modelling tool; and rewriting or generating new code with a 4GL or in a modern, object-oriented architecture such as J2EE or .NET.

There are, however, many mature systems for which not all or even none of the above attributes apply, and indeed the only observation that can be made is that the systems were built with leading-edge technology. And that the leading-edge technology is old. Host-based, monolithic, character-based systems on proprietary platforms that are hard to integrate are also called 'legacy systems.' Too often, however, simply because a system is built to run on technology in its teens, it gets stigmatised with the same subjectively sensitive, negative attribute of being 'difficult to maintain.'

Organisations who either lack the monetary resources to re-engineer or rewrite large amounts of code, or who see no business benefit in doing so, typically look to the alternative of replacing the system (or parts thereof) with COTS[1]. If no suitable COTS alternative can be found to replace the system's functionality (or the remaining parts thereof) then the organisation will be stuck with finding a solution that remains their own.

This, in a nutshell, is where the industry for cross-platform software migration tools lies. These tools serve to make the transition from endangered or undesired component technologies to the ones of the organisation's choosing optimally automated and cost-effective, simultaneously enabling the organisation to retain what they see as an asset, namely the functionality of their systems. While there are many variants, the better tools enable this simultaneous transition and retention without introducing runtimes foreign to the technology being implemented and proprietary to the tool vendor. Some vendors make this possible by coupling their tools with a service to generate a 100 % functionally identical and 100 % visually equivalent copy of the original system that furthermore retains its ease of maintenance.

Anubex is a Belgian IT company, and has specialised for the past ten years in building and deploying application transformation and migration tools. During this time, we have advised over fifty organisations on migration projects of larger (.5-10 MLOC) information systems, and built over

[1] COTS: Acronym for Commercial Off The Shelf software, or a packaged system. The 'approach' that this acronym refers to can be applied to the implementation of any packaged software, however, regardless whether it is publicly or commercially available.

forty tools that automate various aspects of software transformation for specific platforms and languages. In this paper, we relate, based on our experience, our view on how the overall perception of cross-platform migration is evolving, how automated translation works in the context of a manageable project, and how semi-automated transformation and migration techniques impact organisations and developers.

2. Definitions and scope

The discourse on software transformation and migration is made difficult by the lack of a clear consensus regarding the use of terms, sometimes intermingled, to describe, variously, the business goals, the technical deliverables, and the methodologies used. The terms 'legacy transformation,' 'legacy modernisation,' 'legacy renovation,' and 'legacy reengineering' are used to describe families of these 'approaches' in which redevelopment [3], re-writing [5], EAI ([1] [11]), retro-documentation (more used in French-speaking regions of the world) [6], replacement ([1], [5], [11]), migration ([3], [10]), consolidation [1], wrapping ([3][11]), web-enablement [1], re-use [5], screen-scraping [11], domain engineering [8], and componentisation [8] (to give a few common examples) fall.

The narrower term 'migration' also suffers from a similar lack of clarity. Migration can be either a business goal in itself or a technical deliverable of a larger business goal (for example, consolidation), and is furthermore embodied in multiple reengineering methodologies that rely on automation in different ways ([3], [10]). The term is also sometimes avoided, with synonyms such as retargeting, replatforming, and rehosting [7] being used.

For the purposes of this paper, we consider migration as the *restoration of value to a software application by removing its dependency on undesired technologies or architectures, through the conversion of the application's pieces from one technology to another, creating an otherwise identical working system that uses new technologies in a native way*. This approach makes use of platform or language-specific 'models' that represent the application before the migration and afterwards.

We define a platform-specific model as one where the *bi-directional transformation between it and the source code it represents can occur an infinite number of times, without the loss of any information*. Migration is automated, then, through the transformation of one platform-specific model into another. It can rely on the use of either one or two meta-models, traditionally referred to as grammars [2], depending on whether the migration of the legacy application involves a conversion of the code from one language to another or not. This approach has the characteristics of a black box, meaning that any manual alterations made to the code occur either before the application is parsed or after the migrated code is generated.

With the exception of any additional application tuning, which may be necessitated to ensure the retention of the performance of the system in its new environment, this is where migration, as used in this paper, stops. Many legacy strategies that bear the label 'migration' attempt to go further ([3], [10]), incorporating additional steps such as the restructuring of 'spaghetti' code, the re-architecturing of an application's entities, the manual development of GUI interfaces, or the full exploitation of object-orientation. While it is not our intention to ignore the value of such additional services, our observation is that the ROI argument to follow the shortest possible path to achieve a clearly defined business goal (in this case, the decommissioning of an undesired technology component) is growing in importance. This is especially the case as concerns decisions made for larger software applications, and the approach may be pursued even when the 'quality' of the code may be questionable.

The overwhelming majority of organisations we advise (100 % of them) consider automated migration for administrative software applications. These applications run the 'core business' of banks, insurance firms, government institutions, or services companies; or the 'back office' applications for companies in the aerospace, telecommunications, or manufacturing industries. Most of these organisations started developing the applications between fifteen and twenty years before their migration, and in all cases except for one, used COBOL. (The sole exception regarded a 3MLOC application written in BASIC for a European airline company.)

At a technical level, the transformation projects being pursued fall into one of two categories:

- Actual cross-platform migration from endangered or proprietary hardware platforms to 'open' distributed systems (mainly Unix or NT), or from endangered development environments to modern development tools and deployment platforms (mainly application servers such as WebSphere, etc.);
- Retargeting applications from data access methods that pre-date RDBMS (networked databases, hierarchical databases, or (index) sequential files) to an RDBMS product (mainly Oracle or DB2).

3. Justifying migration

Other studies ([4], [9]) have investigated common 'drivers' for software transformation projects. Some [5] have gone further to analyse these drivers according to their justification by internal considerations (such as cost reduction or guaranteed operational continuity) or external considerations (such as eBusiness initiatives or more strategic 'future-proofing' of the applications).

Our experience with COBOL-heavy environments in Europe suggests that migration or transformation projects are mostly justified by a combination of these predictably recurring 'drivers' together, but that in a quarter of the cases a single overriding factor is sufficient to justify the project in its entirety. Y2K compliance, obviously, has disappeared as a driver.

Perhaps surprisingly for a business context, the driver of 'cost reduction' is rarely used to justify a migration project on its own. Examples of cost reduction drivers include eliminating administrative overheads and extra technical support costs for running processes over separate, non-integrated systems; eliminating the need for middleware to connect proprietary systems with open ones; reducing maintenance overheads by adopting cheaper hardware or development platforms; or other economies made through platform consolidation.

Examples of overriding, singular drivers that do get used to justify migrations are the following:

- The technologies used present a physical technical barrier in terms of performance, (storage) volume, or the maximum number of concurrent users. Migration is seen as urgent when these technical barriers prevent the business' natural growth;
- The technologies used by the application are outmoded and the organisation is pressured by its clients to modernise them. ISV's are of course especially vulnerable to these influences, but this driver has also been found in the B2B insurance and services sectors;
- The migration of the application is a necessary step in some other process. A common example involves organisations implementing an ERP package to replace business-generic functionality in a legacy application, and that need business-specific application functionality to be migrated in order to retain the integration of the processes and data. In these cases, migration makes the implementation of the ERP system possible;
- The supplier of the technology has announced the termination of support. This can involve both hardware and software suppliers.

Perhaps equally surprisingly, in none (0 %) of the cases have any of the following been used either as primary or supporting drivers:

- Pressure from clients or suppliers to integrate supply chains;
- eBusiness;
- Migrating away from COBOL to a 'more modern' programming language.

In well over half of the cases, organisations that have real migration needs do not initially consider automated migration as a potential solution. The most common reasons for this are the following:

- An unawareness of the availability of tools that cater for their requirements in terms of source and target technologies supported, or the belief that the creation of a tool that fits their specific environment requirements is not feasible or cost-effective;
- A belief that manual redevelopment of their systems from scratch in newer technologies is desirable or feasible, or resignation to the belief that the only cost-effective solution is to outsource the redevelopment offshore;
- The perception that a project which delivers a 100 % functionally identical piece of software "does not take the company forward;"
- A prior negative experience with a migration tool that generated unmaintainable code.

Over 50 % of attempts to migrate applications with over .5MLOC through manual redevelopment are abandoned after two to four years as failures.

4. Wanted? 100 % migration

The figure of 100 % is often referred to in the justification and planning phases of migration projects, and this in two cases:

- How to justify a project that creates a 100 % functionally identical target system;
- The evaluation of the quality of a migration tool by measuring how close to 100 % of the objects or language statements in the original technologies it migrates automatically.

Deliberately creating a target system that is 100 % functionally identical to the original system is sometimes seen as a counterintuitive milestone. This limitation nevertheless offers several benefits, all of which an organisation typically comes to recognise during project execution:

- First and foremost, perhaps, it is incontestable. When organisations undertake a tools-assisted migration of their software applications, they rarely use tools of their own making. Not having the expertise or experience to build the tools needed, they look to licensing existing ones from a tool vendor. Normally, the tool vendor provides the services that go with the tool and can be given the responsibility for guaranteeing the new system works. A discrepancy in behaviour or in output is easily demonstrated and gives the organisation procuring the service added protection;
- It is the only way to avoid the difficult, expensive, and time-consuming process of making specifications; it prevents the danger of scope creep; and it facilitates the testing phases of the project;
- It gives the organisation a clear, easy-to-understand means of tracking the progress of the project;
- It relieves the strain on users to adapt to the new system and limits the entire change management process to the IT department;
- It is the fastest way for an organisation to transition from old technology to new, and it is the fastest way for the organisation to regain autonomous control and resume normal incremental maintenance activities for the system.

5. Migration Complexity

Regardless which life extending approach an organisation pursues for its applications, at some point 'analysis' takes place. For the sake of simplicity we will limit the discussion to the approaches of continued incremental maintenance (the 'do nothing' approach), replacement with COTS, re-engineering/re-writing, or automated migration. The exception to the analysis rule involves certain language, platform, or presentation extension technologies such as emulators, wrappers, or (cross) compilers that serve at the same time as a means and an end.

With the exception of migration, all of the strategies listed above have in common that analysis is functionality-driven. With COTS, a ‘gap analysis’ may be performed to highlight original application functionality that is not supported in the commercial product, and other analyses can be performed to plan change management when internal processes of the organisation have to be revised before the commercial software package can be used. With re-engineering and re-writing, code complexity, similarity, and redundancy can be analysed in addition to the functionality of the application.

Analysis plays an important role in the planning of a migration project too, however this analysis is not driven by a need to understand either existing or intended functionality. The focus of migration is in code and object translation, and is predominantly a purely technical exercise. Because of this, an understanding of the code’s functional purpose is not needed.

Such analysis can be automated or done manually. Some tool vendors supply analysis tools as a companion part of their toolkits, which automate the process and provide a more mature solution. These analysis tools, much like the tools that do the actual migration, extract the information they need from the code itself. While there are many variants, a common denominator is the extraction of information pertaining to the size of the applications and their complexity. Vendors use this information to forecast the amount of work that the migration effort will entail and to draw up project plans. From a commercial perspective, vendors may also use this information to calculate the licence price that the migrating organisation must pay to use the migration tool.

Bearing the above in mind, automated migration is an exceptional part of application development for three reasons. First, since the calculation of the complexity and the number of lines of code in the original system is sufficient to calculate the effort required in terms of man-days, automated migration is exceptional since it uses a predictive LOC metric with accuracy. Second, automated migration is exceptional since functional or business analysis is not used to predict man-days of programming effort and function point analysis does not offer the project any direct benefits. And third, due to the purely technical nature of the exercise, automated migration is exceptional since the effort required in terms of man-days does not accelerate in function of the size of the applications, and as our experience shows, in some cases even decelerates.

The notion of complexity also warrants further clarification, since complexity in this context is not calculated on the basis of the relationships of the lines of code to each other, or from the code’s structure or lack thereof. Complexity in migrations is an *indication of the number of occurrences in the source code of the original system where a statement or object does not have a one-to-one equivalent in the target technology*. These occurrences can be simple (for instance, a variable name used in the original application is a reserved word on the target environment) or complex.

An example of a frequently recurring, COBOL-related, cross-platform incompatibility of a high complexity involves the data access methods that are used on most legacy platforms. While most basic data types like numeric or alphanumeric are prevalent and equivalent in both legacy and modern platforms, composite data types often pose difficulties. COBOL makes it possible to access and store data at the record level through powerful low-level pointers, and many COBOL programs make use of this facility to store data in a single file with the individual data elements organised inconsistently. Modern RDBMS products restrict data access to the field level, and manage the structure of records so that each is guaranteed to have a consistent organisation. When a COBOL application uses a REDEFINES clause in a File Description, the lack of a one-to-one equivalent in the target RDBMS environment prevents an automated translation of the statement. Individual instances of this cross-platform incompatibility can sometimes be dealt with fully automatically. However, when a record is redefined with incompatible data types (for instance, to store alphanumeric data at a position in a record where previously numeric data was stored) the translation must be manually prepared before the automated translation process can continue.

The issue of cross-platform incompatibility leads to the question of whether it is possible to create perfect tools that automate the migration of 100 % of the objects and language statements in the original technologies. It is perhaps good to mention at this point that such ambitions are hardly the holy grail of the automated migration industry. Typically, averages of 95-99 % are achieved, and it is worth mentioning that a tool, compatible with 95 % of the objects and language statements of a development environment could automatically migrate 100 % of one application and only 80 % of another. At the same time, it is dangerous to compare tools on the basis of coverage percentages only. Certain cross-platform incompatibilities can be solved automatically with ease, but the solution may come at the cost of being very difficult to maintain. Except in circumstances where the target technologies have been built specifically with backwards-compatibility in mind, the likelihood of being able to automate the migration process of 100 % of the objects and language statements, and at the same time generate code that is easily maintained, is low to non-existent.

From the discussion on complexity and cross-platform incompatibility, it should be clear as well that the level of automation has a direct impact on cost. This is for two reasons: less automated translation means more man-hours to implement manual solutions; and less automated translation also means more time is spent on testing as humans tend to make more mistakes than software.

But how important is this factor, and how does this weigh against the drive of organisations to embrace ‘more modern’ languages and development environments? Surely a language such as Java is more modern than COBOL, but at the same time surely a COBOL-to-COBOL migration is cheaper than a COBOL migration coupled with a language conversion to Java, since there are more incompatibilities between the two languages? And surely there must be a perception that an application written in Java is better ‘future-proofed’ than one in COBOL, but how does this weigh against the notion that the Java program will be less intuitive for the original developers to maintain than if it were kept in COBOL, due to structural changes made to the code?

Actually, none of the organisations we have dealt with have pursued the automated migration of a large-scale information system with a language conversion from COBOL to Java. However, organisations that we have dealt with who pursue a migration of COBOL applications to, say, Oracle database technology have a choice, since a variety of tools are available that perform COBOL-to-COBOL retargeting, COBOL-to-PL/SQL conversion and retargeting, and COBOL-to-Java conversion and retargeting. When migrating COBOL applications from legacy platforms to an Oracle database, and given the choice to keep the applications in COBOL or to convert them to PL/SQL or Java, 95 % of the organisations opted to keep the applications in COBOL, and the other 5 % chose to convert to PL/SQL.

How each justified their decisions is also something of a surprise. The majority who chose to keep the COBOL did so out of cost considerations. The minority who chose conversion to the 'more modern' language, on the other hand, also did so out of cost considerations.

The cost argument used by the majority of organisations opting for COBOL-to-COBOL migration was that the cost to retrain teams of COBOL developers to the 'more modern' language was greater than the potential 'future proofing' benefits of having the code in PL/SQL or Java. Coupled with the fact that most organisations had other COBOL developers in their employ who worked on other applications, these organisations took this decision with the certainty that the number of available COBOL developers, for the time being at least, was higher than the number of available PL/SQL and Java developers. The cost argument used by the minority was a licensing issue, in which paying for COBOL runtime licences for the hundreds or thousands of users of the system was higher than the cost to retrain the COBOL developers.

This evidence suggests that organisations with large-scale software applications are not capricious with their migration choices, and stresses that the business angle weighs heavily in the major investment decisions taken around them.

6. Project dynamics and fluid systems

The approach to migration as explored in this paper is able to reduce or even eliminate many project elements that involve users, such as user retraining or the analysis of user requirements. Such economies are of course inherent to the deliberate limitations of the approach, which actively seeks to ignore these and a number of other issues.

On the other hand, it is rare to find a situation in which the issues, ignored by the approach, do not surface at some point during the project. This problem introduces a new issue that is common to any approach taken to migration, and involves the way in which it can balance the need for a system 'freeze' with the need of the existing system to evolve freely during the project's course.

In accordance with the strict enforcement of the 100 % equivalence rule, all modifications must be done on the existing system in production. In this case, the application of the rule protects the tool vendor, since any modification of the functionality pursued by the migrating organisation must be proven to work on the original system prior to being taken into consideration. Through the 100 % equivalence rule, then, business disruption is not only minimized as concerns the operational context of the system's use, but also as concerns its evolutionary maintenance throughout the transitory period.

To put the problem into perspective, it is necessary to consider two facts: First, the duration of an average migration project for applications as treated here is between five and six months. Second, despite organisations recognising the complications that modifications to the existing systems will introduce to the migration project, it has been necessary in 100 % of the cases for the original system to be modified at least once while migration is in progress. Such changes can be necessitated by law or by regulation; or can be warranted by other business needs.

The need for the original system to evolve freely during the course of a project makes it impossible for all practical purposes to impose any form of freeze on the code. The only freeze that does takes place regards the system in its totality, and is limited to the very last stage of the project in which the final conversion of the data from the old environment to the new one takes place. This phase normally takes place during a weekend when the system is otherwise not in use. Since 24/7 system availability has been necessary in 0 % of the cases, such a freeze has not been the cause of business disruption, and the overhead of implementing of a real-time switchover mechanism, while possible, has not yet been justified.

When assessing the impact of a modification on the existing system during the migration, there are two parameters that are the most important to consider. These are, first, whether or not the source concerned has been converted AND manual work has been done on the converted source; and second, whether or not the modification involves a change to the data structure.

The simplest scenario is if the modification does not involve a change in the data structure and no manual work has happened yet on the converted source. In this case, the source is merely converted again.

The scenario with a slightly higher complexity occurs when the modification does not involve a change to the data structure, but manual work has already been done on the converted source. This scenario introduces a version conflict, as illustrated in the figure below:

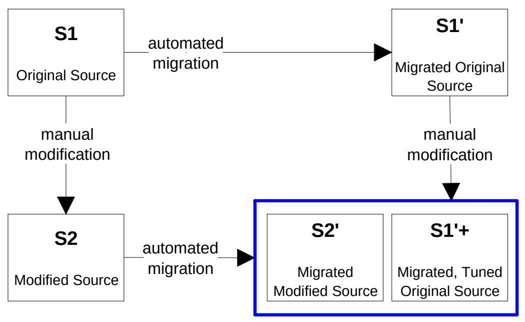

Figure 1: The version conflict in the migration of fluid systems

As shown in Figure 1, source S1 is converted to the new environment and manual work has been done, resulting in a production-ready candidate S1'+. When modifications are subsequently made to S1 on the original system, resulting in S2, the creation of S2'+ must result from the comparison of the differences between S1'+ and S2'. Very often, the modifications do not affect one another and S2'+ can be the result of the straightforward merge between S1'+ and S2'. In other cases, additional manual work and testing must be done.

Scenarios that involve the modification of the original system's data structure are significantly more complex. This is largely due to the method inherent to this form of migration, which combines incremental elements of 'chicken little' [4] during the construction and testing of the new system with a 'big bang' in the event of going live. As a result, the 'test data' being used in the tuning and testing of the migrated programs prior to going live plays an important role in the migration process, and the definition of the test data environment must always be kept up to date. For this reason, the creation of the test data environment is always one of the first steps done in any project.

When the data structure in the original system is modified, this implies that the test data on the target system together with the data dictionary and DDL statements must be updated. Any application sources that are affected by the change must be reconverted. This process can be the cause of numerous version conflicts, as depicted in Figure 1.

7. Migration impacts

Even when bearing in mind that the goal of migration is the creation of a 100 % functionally identical target system, and that doing so benefits the organisation since change management is limited to the IT department, change management in the IT department can be heavy nevertheless. While migration of course impacts the IT department, it is the transition to a new environment that causes the most disruption, and it is the automated migration approach that actually minimises the extent of it.

Or at least, it can. Our experience shows a clear correlation between the level of direct involvement of the organisation in the migration project and their overall level of satisfaction with the project's final outcome. This factor persists in all projects, and is not influenced by the involvement of third-party migration service providers. This factor is especially pronounced when the maintainers of the system play an active role in the performing of manual work. Maintainers who get involved are more autonomous and confident in their abilities to resume maintenance over the new system once the project is finished.

In environments where large, 15-20 year-old applications are maintained in-house, developers typically posses three critical competencies:

- A knowledge of the business;
- A knowledge of the application code and its structure;
- A knowledge of the development and deployment technologies used.

Armed with these three competencies, developers have the capacity to support the applications that support or enable the business. Automated migration makes it possible to economise and retain the first two of these, with both being actively used both during the course of the project and afterwards. As concerns the last point, the transition to new technologies can cause disruption since new skills must be acquired. There is rarely the luxury of time, since the migration projects of the organisations we advise normally take up to six months to complete.

Automated transformation and migration is arguably the best way for maintainers to acquire new skills and adapt to new technologies, and although organisations do not always see the benefits initially, real-world examples confirm it to be so. Some developer-related benefits of migration are the following:

- Since the bulk of the code is converted by a piece of software, the code is translated and generated consistently. This relates not only to code formatting conventions such as capitalisation and indentation, but also to the consistent translation of the statements the maintainers are already familiar with and the retention of comments. Subsequent application maintenance is easier since the code still 'belongs' to the developers;
- Learning the new environment is easier since developers can compare the code 'before and after;'
- The retraining of developers is never on the 'critical path' of the project, and developer retraining is never rushed as a result of it being a prerequisite for the project to begin, as is the case in fully manual redevelopment projects;
- Libraries and languages are pre-deployed by the migration tool. Through this, developers do not have to achieve reasonable professional proficiency in the target environment, and then go through a difficult process of agreeing on a development 'house standard' in the new technology before the project can start;
- By working with a tools vendor with extensive experience in both the source and target technologies, training materials and programs can be tailored to take advantage of the skills the developers already posses. Such targeted training is normally impossible to find externally. Our experience shows that training time of developers can be reduced by up to 50 % in comparison to following standard, vendor-approved, entry-level courses when the training can be tailored in this way.

The most lasting impact on the organisation of a migration project, of course, is that an application's anticipated lifespan is doubled, and that the applications preserved get a new lease of life. Especially if migration is partial and business-generic parts of the original legacy system are replaced by COTS, the migration to new technology of core applications can bring an organisation's appreciation of their uniqueness as a business into sharper relief.

This added realisation can impact the way that subsequent maintenance decisions of the system are handled, and mark the transition of business-specific application functionality to a new level of maturity in which an organisation makes a conscious choice to continue investing in its growth for many years to come. The awkward position the system occupied prior to the migration as being simultaneously a business-enabling asset and a technical liability is thankfully put in the past.

Conclusion

In this paper, we have examined automated migration from a number of different angles, exploring the utility of the approach through the limitations that serve as its defining characteristics. The choice to deliberately limit the target system to feature 100 % functional and visual equivalence as a deliberate milestone on the road to ultimate modernisation is perhaps the most prominent feature of this approach. When applied consistently, this limitation can affect, as explained here, the way in which organisations justify implementing the approach, the relationship between complexity and the cost of the approach, the ability of systems to evolve freely during the transitory period, and the way in which the organisational context is impacted through the approach's application.

The fact that many organisations opt for an approach that deliberately limits the scope of the project deliverables opens a number of potential relatively unexplored research directions. It is our opinion that research is lacking, which considers the viability of achieving modernisation for larger software systems in terms of sequentially applied steps that are applied to a system on the whole. Such research has the potential to benefit organisations that are under pressure to achieve clearly understood business objectives and realise ROI in increasingly shorter timeframes.

Bibliography

[1] Aberdeen Group: Legacy Applications: From Cost Management to Transformation, March 2003.

[2] J Bézivin, S Gérard: A preliminary identification of MDA components. Date unknown.

[3] J Bisbal, D Lawless, B Wu, J Grimson: Legacy Information System Migration: A Brief Review of Problems, Solutions, and Research Issues. Technical Report TCD-CS-1999-38, Computer Science Department, Trinity College Dublin, May 1999

[4] M Brodie, M Stonebraker: Migrating Legacy Systems, Morgan Kaufmann Publishers, Inc. 1995.

[5] D Good: Legacy Transformation, Club de Investigación Tecnológica, August 2002.

[6] D Good: Proceedings of the Club de Investigación Tecnológica Legacy Transformation Workshop, San Jose Costa Rica. February 2003. Available at www.cit.co.cr/Presentations/DeclanGood.ppt

[7] M Olsem: STSC Reengineering Technology Report, Document No: TRF-RE-9510-000.04, Software Technology Support Center. October 1995.

[8] A van Deursen, B Elsinga, P Klint, R Tolido: From Legacy to Component: Software Renovation in Three Steps, Cap Gemini and the CWI. 2000

[9] I Warren: The Renaissance of Legacy Systems, Method Support for Software System Evolution, Springer Verlag.1999.

[10] B Wu, D Lawless, J Bisbal, R Richardson, J Grimson, V Wade, D O'Sullivan: The Butterfly Methodology: A Gateway-free Approach for Migrating Legacy Information Systems. Proceedings of the third IEEE Conference on Engineering of Complex Computer Systems (ICECCS97), September 8-12, 1997. pp.200-205, IEEE Computer Society.

[11] F Zoufaly: Issues and Challenges Facing Legacy Systems. November 1, 2002. Available at http://www.developer.com/mgmt/article.php/1492531.