independIT Integrative Technologies GmbH
Bergstraße 6
D–86529 Schrobenhausen

BICsuite!focus

Migration Methodology

Ronald Jeninga

January 2, 2024

Migration methodology

Introduction

When evaluating the possibility of migrating an implementation of workflows from a source system to BICsuite, a common misconception is to believe that the effort depends on the number of objects within the source system.

Obviously it sounds plausible to assume that if the source system contains one million objects, it'll take more work to migrate than if the source system only contains ten thousand objects. And yes, to a certain extent this is true. In a large source system, the probability of unsound solutions will be higher. It'll also be more difficult to test the results of a migration. A very important aspect will be the presence of dependency cycles, which are much harder to resolve in a large system compared to a small one. Still, the required effort certainly doesn't grow linearly. It's more like a logarithmic scale.

To understand this, it is important to realize that while the number of objects grows, the number of special solutions doesn't, or at least grows to a far lesser extent. Since the basic strategy during the migration will be to develop a system that will automate the migration, the entire project can be compared to writing a compiler or an interpreter. A compiler for very small programs like the famous "hello world" would be very easy to write. If the programs grow larger, more syntactical constructs will be used which again requires a more complex compiler. Large systems might even require additional features such as code optimization, which will also add to the complexity of the compiler. The compiler code will still grow at a significantly lower rate than the size of the software systems it is able to compile.

The difference between a migration and building compiler software is that the input is not well known. In the case of a compiler, there exists a formal description of the syntax of the programming language. In the case of workload automation systems, a comparable rigid description of the data that can be used as input for a migration does not exist or is at least hard to find.

Outline of the migration procedure

It is assumed that the source system has the ability to create a kind of export of the definition layer. The definition layer contains all the data that describes the implemented system. Basically it describes which jobs run at what time, what the dependencies between the jobs are, and much more besides.

The migration is an iterative process that looks like this:

1. **Export&Load** The export is loaded into a relational database. The schema definition is inferred from the data itself but doesn't need to be 100% accurate as long as the structure of the data is preserved.
2. **Analysis** The next step is to analyze the resulting database and to create a kind of documentation of the used tables and columns. It is important to find

the primary keys and, of course, the corresponding foreign keys.

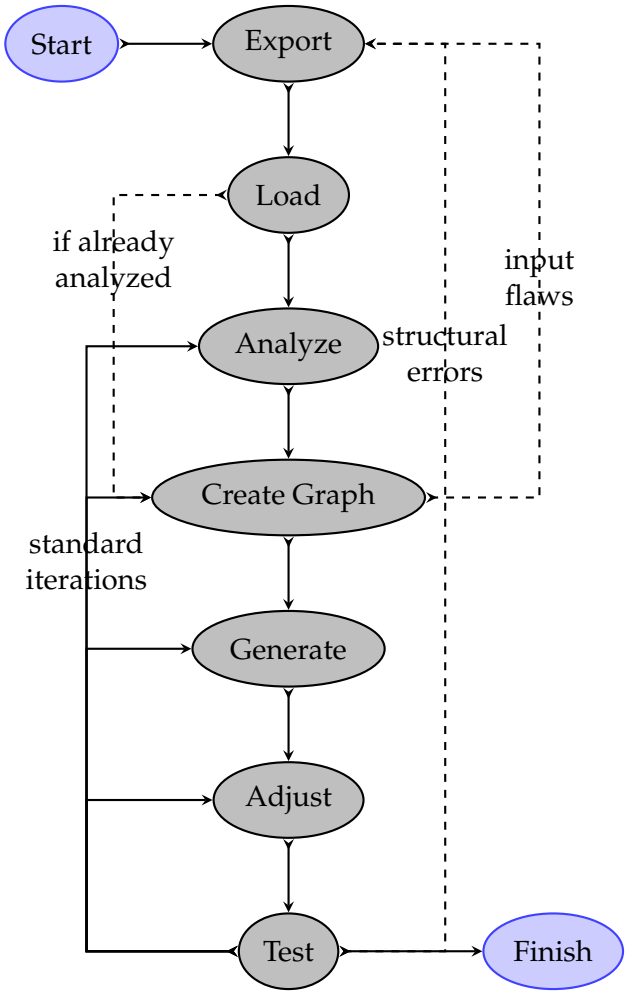
3. **Graph representation** The data in the database is then (partially) loaded into an intermediate data structure. It has proven to be useful to use a kind of graph representation of the database. It isn't necessary to load everything at once. At first only the main structural information such as the jobs and dependencies is loaded. In subsequent steps, this information is refined with more details from the original data basis.

4. **Code generation** > BICsuite source code is generated from the intermediate representation and loaded into a BICsuite system. During the code generation, it is possible that flaws in the input, effectively the source system, are revealed. In this case they must be repaired within the source system (without changing the semantics), and the process restarts with a new export. Naturally the analysis step doesn't have to be repeated. Compared to the analogy of building a compiler, this would match the situation of syntax errors in the code to be compiled.

5. **Manual Adjustments** Sometimes rare complex situations or exceptions have to be migrated manually because covering it within the generic engine would require a very complex, hard-to-implement logic. These situations each require a dedicated script.

6. **Test** The system that has been migrated so far is now tested. In the event of errors, the code generation and/or the manual correction step is refined and repeated. If everything works as expected, more details from the original database are taken into account. Unless there are no more details to consider, in which case the migration has finished, the procedure returns to step 3. Sometimes it turns out that the original comprehension of the data is not correct, in which case the procedure returns to step 2. It is also possible that structural errors such as cyclic dependencies are found. In this case, the input, i.e. the source system, will have to be modified (while retaining the semantics) in order to eliminate the errors. The process then restarts with a new export.

The picture below shows a graphical representation of the process



Hard and soft requirements

The obvious hard requirement is that the system after the migration has to behave exactly the same as the system before the migration. Hence, given the same input, both systems must produce the same output. This criterion is what is tested during the test phase of the workflow. Any discrepancies have to be analyzed and eliminated.

A number of soft requirements exist as well, however. The operating team must be able to work with the new system. The development team must be able to modify or extend the system. And maybe a monitoring team must be able to interpret the progress during the daily processing.

While this can be covered with education, it is still desirable that the old workflows remain intact. Every change in the workflows increases the risk of making mistakes.

Another aspect is the use of a strong naming convention so that objects in the migrated system can be easily mapped to the originating object in the source system. This will greatly improve both the comprehension and the acceptance of the migrated system.
Since a 1:1 copy of the source system usually can't be achieved, these requirements are called soft requirements. It'll be an as-much-as-possible approach. On the other hand, it isn't prohibited to modify workflows if this leads to less or easier work, making it harder to make mistakes.
The bottom line is that there won't be a right or wrong way. In many cases, decisions will have to be made based on a dialog with the users of the system. It is very well possible that several different representations will have to be tested in order to find the representation that pleases the users most.

Export&Load

The result of an export of the source system is extremely dependent on the source system itself, and can even substantially differ between different releases of the same source system. A Control-M will generate an entirely different format than an Autosys, a CA Workload or an Automatic. Sometimes an export is in the form of an XML document, sometimes it consists of a list of statements that can be executed within the source system, or it could be a bunch of CSV files. It is even possible that the source system doesn't offer the possibility of an export, but stores its data in a database or in a well-defined location in the file system.
Some of the export formats are easy to work with (XML or CSV format, as well as data stored in a database), while others require tools to extract the information from the export (statements, raw data).
Theoretically, it would be possible to develop libraries that allow random access to the exported information. But that would be a complex task and the access would have a poor performance. The better idea is to create tools that extract the information and load it into a relational database. The data is subsequently accessed using standard SQL, which provides an easy-to-use interface. Another advantage of this approach is that the data dictionary of the used database system provides a condensed list of data items.

Analysis

It is precisely this list of data items that is analyzed in the next step. Some items will have an obvious meaning, while other items might require further investigation. This analysis cannot be done without the help of an experienced user of the system.
At the same time, it is this list of data items that is basically a TO-DO list. If every data item in the list is treated correctly in the subsequent steps, the result must be correct. This also proves that the migration is a finite process and its duration largely depends on the complexity of the source system.

Even with the input of an experienced user of the source system, mistakes will be made. And this is why the analysis step is part of the iterative process.

Graph representation

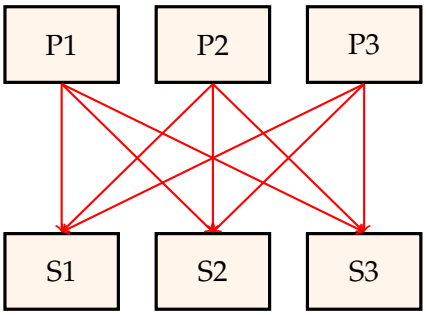
Due to the very nature of the problem, each implementation of a system in a workload automation tool can be represented as a graph, in which the nodes correspond to Jobs or entire sets of Jobs (maybe even sets of sets) and the edges correspond to predecessor-successor relationships between the nodes.

In BICsuite, the predecessor-successor relationship is called a dependency, where the successor requires the predecessor to have finished with a certain exit state. A Job (Definition) is an object that will execute a command line, while a Batch is a kind of container of Jobs and/or other Batches. Other workload automation tools have similar concepts that only differ in detail.

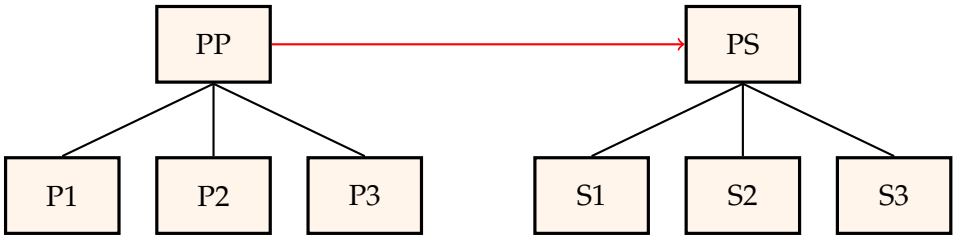
The Batches and Jobs hierarchy is also a graph, a tree in fact. This means that in the entire graph there exist two types of edges. The first type is the dependency and the second type is the parent-child relationship.

The graph without the dependencies is always a forest (a collection of trees). The edges are directed from the parent to the children. There are no cycles (otherwise it wouldn't be a collection of trees). The graph without the parent-child relationships looks like a network. Again, the edges are directed from the predecessor to the successor.

The parent-child hierarchy influences the dependency graph. Dependencies at parent level are implicitly inherited at child level. This can be nicely illustrated with the (non-planar) Utility Graph. Three predecessor jobs P1, P2 and P3 are all required by three successor jobs S1, S2 and S3:



When using the implied dependencies that are a result of the dependencies at parent level, the picture looks a lot better:

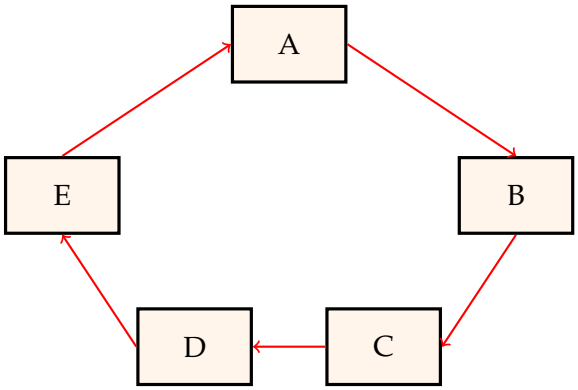


It can be seen at a glance that the jobs S1, S2 and S3 wait for the jobs P1, P2 and P3 to complete. This is not so evident in the first graph. Hence the parent-child hierarchy makes it easier to understand (and to define) complex relationships. The downside is that cycles aren't that easy to detect. An arrow pointing from S1 to P3 looks pretty harmless in the second picture. But in fact it creates a cycle, which becomes immediately obvious if we draw the same arrow in the first picture.

Still, the advantages of a parent-child hierarchy outperform the disadvantages significantly. This is probably a reason why all workload automation tools offer them in one way or another.

The implied dependency graph should be acyclic, otherwise a deadlock would be the consequence. If A waits for B and vice versa, not much will happen. The definition of the implied dependencies allows the mixed graph with the two types of edges to be converted into a unique sole dependency graph. The graph library can then be used to find cycles.

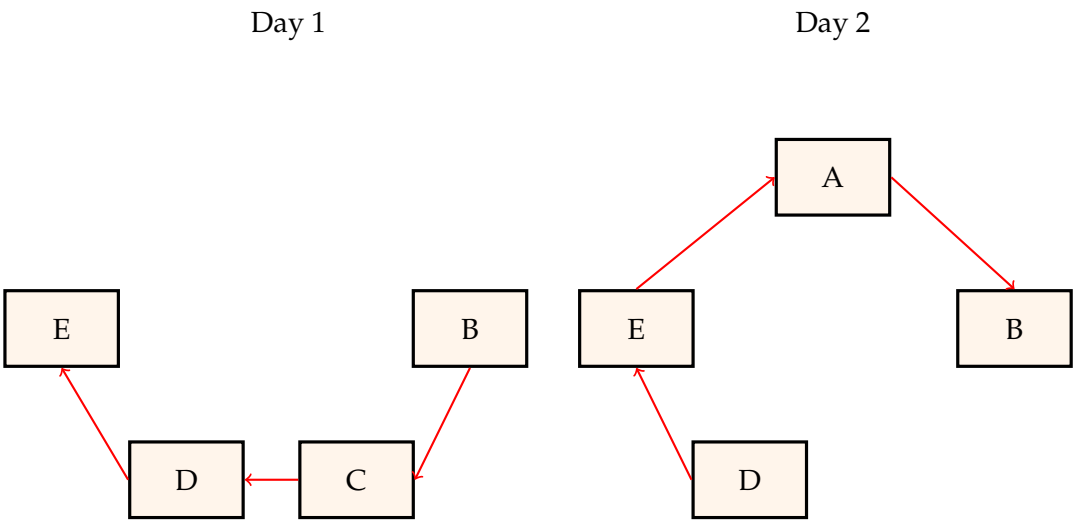
If cycles are found, they'll have to be analyzed and eliminated. Ultimately, workload automation is the goal, and a cycle doesn't allow the application to run without ending up in a deadlock that will have to be resolved manually. Let's assume that a cycle like in the picture below was found:



One possible reason for this graph is that A in fact waits for an E from the previous day. In this case, it is likely that A doesn't actually need the results of E, but it

is simply undesirable that the daily processing starts before the processing of the previous day has finished (or at least a certain milestone within the processing has been reached). Such a condition is modeled differently within BICsuite. A resource can be used to signal that the run of the previous day hasn't finished yet. This way, the dependency between A and E loses its meaning and can be eliminated. An acyclic graph is the result.

It also happens that such a definition in the source system never ends up in a deadlock because at least one of the jobs isn't run every day. Although this resolves the problem within the source system, it leaves a bitter aftertaste. The meaning of the dependencies is poorly defined.



If A is missing on day 1, then we have a transitive dependency from B to E, and E is a successor of B. If C is missing on the next day, then we have a transitive dependency from E to B. It is hard to believe that this is logically correct. On one day we need data in order to produce a report, and on the next day we produce the report in order to be able to generate the underlying data?

In many cases, these predecessor-successor definitions don't define dependencies, but are (randomly) added to prevent concurrent execution or to define a preferred order of execution. And again BICsuite offers a different method, again resource-based, to prevent concurrent execution. If necessary, priorities can be used to let the system strive towards a preferred order of execution.

As a matter of fact, using dependencies to define a preferred order of execution will degrade the overall performance. If B has to wait for A to complete because it is preferred that A runs first, B could start waiting for A even long before A is allowed to run in the first place. B could run during the time that A is still waiting for its predecessors. With the use of priorities, the preferred order is documented and the

system will act accordingly if both A and B are ready to run, but it wouldn't let B wait if A is not yet ready.
A graph library like the widely-used Python NetworkX library offers a large and strong tool set. Especially if the graphs grow to a huge size, it is important to be able to use tools that are both powerful and correct. A large number of predefined, mathematically proven algorithms are available. Especially when problems like cycles pop up, such algorithms can be effectively used to investigate the graphs.
It is important to note that the NetworkX library is capable of storing arbitrary data structures as properties of nodes and edges. This means that all information found in the export can be stored within the graph representation of the data.

Code Generation

From the graph representation it is easy to generate statements that will create the corresponding objects within BICsuite. Since every object within BICsuite can be created and/or modified using the command language, it is possible to address every feature of the system.
The remaining task here is therefore to find a translation of the source data items into a representation within BICsuite that shows the same semantics as in the original system. Previous migration projects have shown that it was always possible to find such a translation. In many cases there were even multiple alternatives. The criteria for selecting a specific implementation were given by the soft requirements. In some cases, some of the alternatives were evaluated by implementing and testing them.
Due to differences in the source systems and the soft requirements, the code generation step is specific to each project. Although parts of the code generators of previous projects could be used, it is often more work to remove project specific code than to start from scratch. The implementation of the basic skeleton (create jobs and dependencies) costs less than one day. The advantage of starting from scratch is that there won't be any unwanted side effects.
Obviously, if an entire series of similar systems has to be migrated ()such as a development system, a test system and a production system), it'll make sense to reuse the generator. But in this case, both the source system and the soft requirements will either be the same or at least very similar. The best approach will be to start with the simplest system, in this example most likely the development system, and migrate the rest in the order of complexity. The code generation will then follow the same path. The base generator that is used to migrate the initial system is then step-wise refined to cover the requirements of the subsequent systems.

Manual adjustments

There's no rule without an exception. If the code generator is the tool of choice to implement the rule based mapping from the source system to BICsuite, it won't

make sense to implement the treatment of every special case as well. Often it is faster and easier to create a piece of code that treats the exceptions after the code generation than it is to integrate this into the generator.

Again, the comparison with a compiler comes to mind. The compile process is often performed in multiple separate steps. First the input is parsed and checked for syntax errors. The result of this step is a parse tree. The next step is to generate intermediate code that is independent of the target hardware architecture. This intermediate code is then translated into machine code for the target architecture. Last but not least, this machine code is optimized to guarantee the best possible performance.

If this approach in compiler technology is compared with the migration procedure, the parallels are eye-catching:

Compiler	–	**Migration**
Parser	–	Export&Load
Intermediate code	–	Graph representation
Machine code	–	Generator
Optimized code	–	Manual adjustments

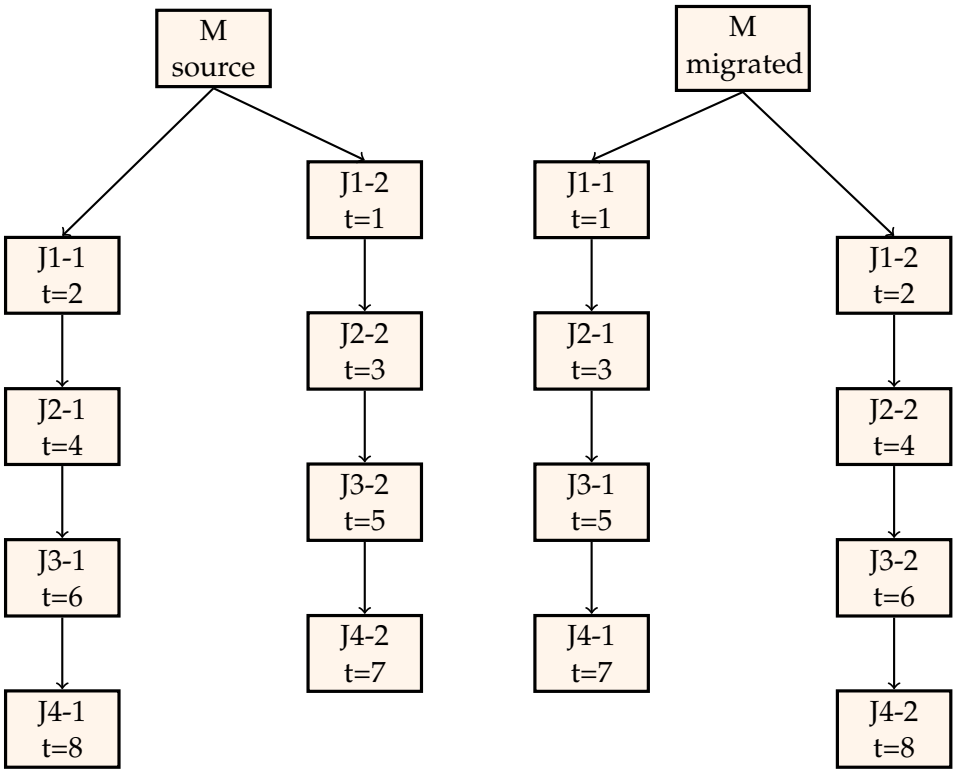
If some of the manual adjustments turn out to follow generic rules, it will make sense to integrate them into the generator code. This will be particularly advantageous when migrating multiple similar systems. In the end, the more code that is generated, the less error-prone will be the migration. The big advantage of generated code is that it tends to be always right or always wrong. The former is fine, but more importantly the latter will be easier to detect.

Test

It is safe to assume that if the executed command lines and the orders of execution are equal, with the same input the output must be the same. Now, since the command lines executed by the jobs will be copied within the process, the first condition is met. Hence the only thing to show during testing is that the orders of execution are the same in both systems.

It naturally also has to be shown that no jobs are left out and no additional jobs are executed. Furthermore, the order of execution isn't 100% strict because of the parallel execution of jobs. The picture below illustrates this claim.

The order of execution in the source system is J1-2, J1-1, J2-2, J2-1, J3-2, J3-1, J4-2, J4-1, whereas the order of execution in the migrated system is J1-1, J1-2, J2-1, J2-2, J3-1, J3-2, J4-1, J4-2.



Still, both orders of execution should be considered to be the same because each chain is executed in the same, correct order. If there were any dependencies between the jobs in both chains, those dependencies should have been modeled in the source system, and would then also be part of the migrated system. In fact, these kinds of differences could arise if two runs within the source system were compared. Too many uncontrollable factors such as the operating system process scheduling, concurrent processing, network throughput and latency play a role in the exact timing of the process starts and their duration.

The larger part of the tests don't actually have to execute the "real" command lines. Dummy programs that just sleep for a while or simply return a Success will do in many cases. This will both speed up the tests and allow the migration team to perform the tests in a kind of simulated environment, such as on the generator developer's computer.

Only when all the features have been implemented and all other tests are satisfactory do some real tests have to be performed in a serious test environment. And if those tests are successful, the go-live can be scheduled.

Time scheduling

In each migration, the time scheduling ()which job should run when) is a topic in itself. The BICsuite time scheduling module is extremely powerful and strongly supports rule-based scheduling.

Instead of creating a list of execution dates (and times) on a yearly basis, it is far better to create a general rule. If the employees' salaries are transferred on the 2nd-to-last business day of the month, it is of course an option to figure out which days these are, and to create a list of them. But BICsuite allows the user to create a rule that does exactly that. The only work left in BICsuite is to create a new list of holidays every year. Based on this list of holidays and the weekends, BICsuite is able to calculate which day happens to be the 2nd-to-last business day of the month.

The capabilities of the time scheduling module naturally aren't limited to this. Even seemingly hard requirements like the 10th day of the month unless it is a weekend or a public holiday, in which case it has to be the next business day, don't represent a challenge for the system.

At the same time, the system allows the user to override the general rule without affecting the rule itself.

The advantage of this approach is that it will be practically maintenance-free. The manual process of creating lists of execution dates is pretty error-prone. Apart from that it is a lot of work to get it right. Furthermore, the deeper logic behind the lists might get lost.

But just like the list of public holidays, any other list of dates (and times) can be created as well. This means that the user has the option to either use the old system with lists of dates or to convert to a more rule-based approach. Which option is best depends on the soft requirements.

Whichever option is chosen, this part of the migration is usually performed as a separate step. It is basically the separation of *what* has to be done and *when* it is done. It is this separation which causes two perspectives of the workload automation. One set of systems is interested in what to do when, and the other systems are interested in when to do what. In other words, one set is driven by time and the others are driven by tasks. It is this shift in perspective which often results in disbelief or at least confusion.

Utilities

To cover more of the soft requirements, it is often necessary to develop utilities. The reasons can vary from simplifying operator actions, over mimicking old workflows, through to adding non-standard functionality (non-standard in the sense that it is not part of the BICsuite system).

Since BICsuite has a powerful and easy-to-use API, it is usually no real effort to create powerful tools that make people happy. In many cases the accurate specifi-

cation of the tool will be the hard part, and the subsequent implementation is often very easy.

Education

Educating the team members is important for two reasons. The first reason is obvious. Since they plan to work with the new system in the future, they need to know how to work with it. Although BICsuite isn't hard to use, it has a lot of concepts that differ from the concepts of the source system. It'll be important to understand the new concepts.

The second reason for education is related to the migration process itself. A lot of the communication between the users and the migration team is crucial for the success of the project. In many cases, solutions that map a functionality in the source system to an implementation in BICsuite have to be discussed. This will be a lot easier if the BICsuite terminology is (at least) understood.

Mainly the second reason indicates that the project will have to begin with education. At the same time, the users will be able to acquire know-how while working with the system during the migration, with the effect that they are already skilled at the time of the go-live.

Conclusion

This document describes a methodology that has proven itself in practice and which follows the principles of the well-known field of compiler construction. It also shows that the required effort is mainly determined by the complexity of the source system and the workflows in use, and not by the size of the implemented application.

The list of input data items guarantees the finiteness of the process, both in time and money, and gives an indication of the progress that has been made. The iterative nature of the process allows the correctness of each refinement to be proved, which ultimately guarantees the correctness of the whole.

A large part of the migration, including the tests, can be undertaken on a single isolated machine. A more sophisticated setup is only required for the real-life tests. This reduces the costs in the early phase of the project. The risks for the project are therefore limited and canceling the project, for whatever reason, always remains an option.

Because the entire migration is implemented as an automated process, concurrent changes to the source system are possible at all times. It'll make sense, though, to freeze the source system between the final successful real-life test and the go-live of the migrated system.