independIT Integrative Technologies GmbH
Bergstraße 6
D–86529 Schrobenhausen

BICsuite!focus

Migrationsmethode

Ronald Jeninga

6. September 2024

Migrationsmethode

Einführung

Bei der Bewertung der Möglichkeit der Migration einer Implementierung von Workflows in einem Quellsystem nach BICsuite, besteht ein häufiges Missverständnis in der Annahme, dass der Aufwand von der Anzahl der Objekte im Quellsystem abhängt.

Man könnte annehmen, dass die Migration von einem Quellsystem mit einer Million Objekte aufwändiger ist, als von einem Quellsystem mit nur zehntausend Objekten. Und ja, bis zu einem gewissen Grad trifft das auch zu. In einem großen Quellsystem ist die Wahrscheinlichkeit von fehlerhaften Lösungen höher. Es wird auch schwieriger sein, die Ergebnisse einer Migration zu testen. Ein sehr wichtiger Aspekt wird das Vorhandensein von Abhängigkeitszyklen sein, die in einem großen System im Vergleich zu einem kleinen System viel schwieriger zu lösen sind. Dennoch wächst der erforderliche Aufwand sicherlich nicht linear. Es ist eher eine logarithmische Skala.

Um dies zu verstehen, ist es wichtig zu sehen, dass die Anzahl der Objekte zwar wächst, die Anzahl der Sonderlösungen jedoch nicht oder zumindest in einem weit geringerem Maße. Da die grundlegende Strategie während der Migration darin besteht, ein System zu entwickeln, das die Migration automatisiert, kann das gesamte Projekt mit dem Schreiben eines Compilers oder eines Interpreters verglichen werden. Ein Compiler für sehr kleine Programme wie die berühmte "Hallo Welt"Meldung wäre sehr einfach zu schreiben. Wenn das Programm größer wird, werden mehr syntaktische Konstrukte verwendet, was wiederum eine höhere Komplexität des Compilers erfordert. Große Systeme erfordern möglicherweise sogar zusätzliche Funktionen wie Code-Optimierung und dergleichen, die wiederum die Komplexität des Compilers erhöhen. Der Compiler-Code wächst jedoch mit einer erheblich geringeren Geschwindigkeit als die Größe der Softwaresysteme, die er kompilieren kann.

Der Unterschied zwischen einer Migrations- und einer Compiler-Software besteht vor allem darin, dass über den Input wenig oder nichts bekannt ist. Im Falle eines Compilers gibt es eine formale Beschreibung der Syntax der Programmiersprache. Bei Workload-Automatisierungssystemen existiert üblicherweise keine vergleichbar starre Beschreibung der Daten, die als Eingabe für eine Migration verwendet werden können, oder zumindest ist sie schwer zu finden.

Überblick über das Migrationsverfahren

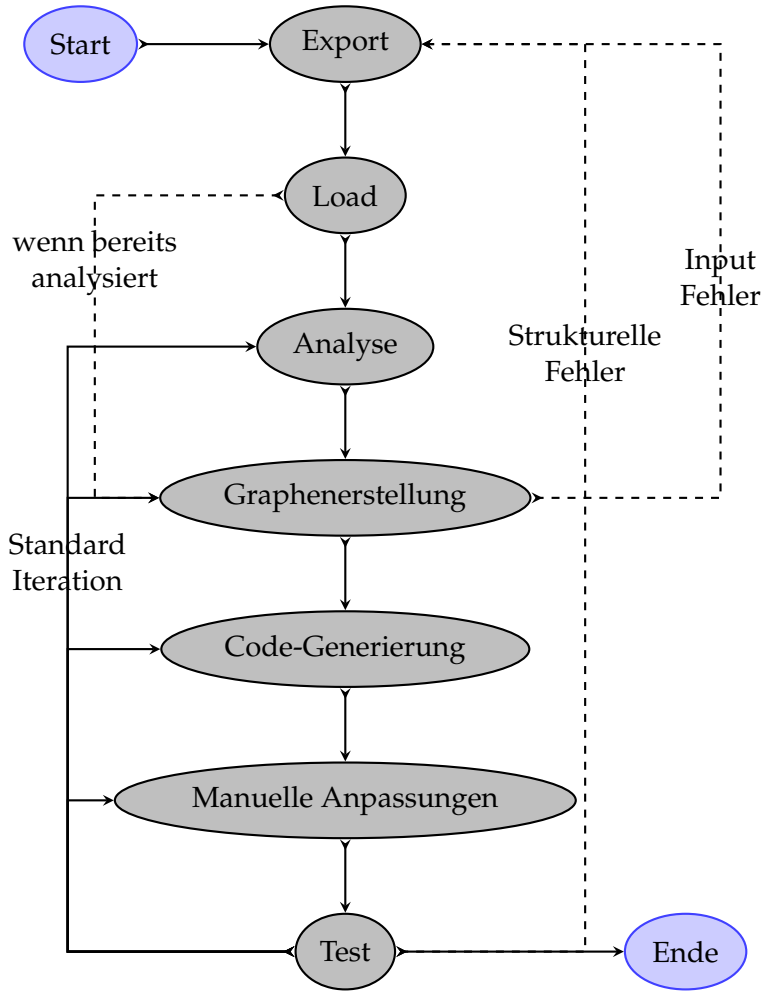
Es wird angenommen, dass das Quellsystem die Möglichkeit hat, eine Art Export der Definitionsschicht zu erstellen. Die Definitionsschicht besteht aus allen Daten, die das implementierte System beschreiben. Grundsätzlich wird beschrieben, wel-

che Jobs zu welcher Zeit ausgeführt werden, welche Abhängigkeiten zwischen den Jobs bestehen und vieles mehr.

Die Migration ist ein iterativer Prozess, der so aussieht:

1. **Export & Load** Der Export wird in eine relationale Datenbank geladen. Die Schemadefinition wird aus den Daten selbst abgeleitet, muss jedoch nicht zu 100% genau sein, solange die Struktur der Daten erhalten bleibt.
2. **Analyse** Der nächste Schritt besteht darin, die resultierende Datenbank zu analysieren und eine Art Dokumentation der verwendeten Tabellen und Spalten zu erstellen. Es ist wichtig, die Primärschlüssel und natürlich die entsprechenden Fremdschlüssel zu finden.
3. **Graphenerstellung** Die Daten in der Datenbank werden dann (teilweise) in eine Zwischendatenstruktur geladen. Es hat sich als nützlich erwiesen, eine Art Graphendarstellung der Datenbank zu verwenden. Es ist nicht notwendig, alles auf einmal zu laden. Zunächst werden nur die wichtigsten Strukturinformationen wie Jobs und Abhängigkeiten geladen. In den folgenden Schritten werden diese Informationen mit weiteren Details aus der ursprünglichen Datenbasis verfeinert.
4. **Code-Generierung** Aus der Zwischendarstellung wird BICsuite-Quellcode generiert und in ein BICsuite-System geladen. Während der Code-Generierung können Fehler in der Eingabe, die aus dem Quellsystem resultieren, aufgedeckt werden. In diesem Fall müssen sie im Quellsystem repariert werden (ohne die Semantik zu ändern) und der Prozess wird mit einem neuen Export neu gestartet. Natürlich muss der Analyseschritt nicht wiederholt werden. Im Vergleich mit der Entwicklung eines Compilers, würde dies Fehler in der zu kompilierenden Code entsprechen.
5. **Manuelle Anpassungen** Manchmal müssen seltene komplexe Situationen oder Ausnahmen manuell migriert werden, da das Abdecken innerhalb der generischen Engine eine sehr komplexe, schwer zu implementierende Logik erfordern würde. Diese Situationen erfordern jeweils ein dediziertes Skript.
6. **Test** Das bisher migrierte System wird nun getestet. Im Fehlerfall wird der Code-Generierungs- und / oder der manuelle Korrekturschritt verfeinert und wiederholt. Wenn alles wie erwartet funktioniert, werden weitere Details aus der ursprünglichen Datenbank berücksichtigt. Sofern keine weiteren Details zu berücksichtigen sind, was dann das Ende der Migration bedeutet, kehrt die Prozedur in diesem Fall zum Schritt 3 zurück. Manchmal stellt sich heraus, dass das ursprüngliche Verständnis der Daten nicht korrekt ist. In diesem Fall kehrt die Prozedur zum Schritt 2 zurück. Es ist auch möglich, dass strukturelle Fehler wie z.B. zyklische Abhängigkeiten gefunden werden. In diesem Fall muss die Eingabe, also das Quellsystem, unter Beibehaltung der Semantik geändert werden, um die Fehler zu beseitigen. Der Prozess wird dann mit einem neuen Export neu gestartet.

Das Bild unten zeigt eine grafische Darstellung des Prozesses:



Harte und weiche Anforderungen

Die offensichtlich wichtigste Anforderung besteht darin, dass sich das System nach der Migration genauso so verhalten muss wie das System vor der Migration. Das bedeutet, dass beide Systeme bei gleicher Eingabe dieselbe Ausgabe erzeugen müssen. Dieses Kriterium wird in der Testphase des Workflows getestet. Unstimmigkeiten müssen analysiert und beseitigt werden.

Es gibt aber auch eine Reihe von weichen Anforderungen. Das Betriebsteam muss mit dem neuen System arbeiten können. Das Entwicklungsteam muss in der Lage sein, das System zu ändern oder zu erweitern. Und vielleicht muss ein Überwachungsteam in der Lage sein, den Fortschritt in der täglichen Verarbeitung zu

interpretieren. Dies kann zwar mit Schulung abgedeckt werden, es ist jedoch wünschenswert, dass die alten Arbeitsabläufe intakt bleiben. Jede Änderung der Workflows erhöht das Fehlerrisiko.

Ein weiterer Aspekt ist die Verwendung einer starken Namenskonvention, sodass Objekte im migrierten System leicht dem Ursprungsobjekt im Quellsystem zugeordnet werden können. Dies wird sowohl das Verständnis, als auch die Akzeptanz des migrierten Systems erheblich verbessern.

Da eine 1:1-Kopie des Quellsystems normalerweise nicht erreicht werden kann, werden diese Anforderungen als weiche Anforderungen bezeichnet und werden so weit wie möglich umgesetzt. Allerdings können Abläufe auch geändert werden, wenn das die Arbeit reduziert oder vereinfacht und das Risiko, Fehler zu machen, dadurch reduziert wird.

Das Fazit ist, dass es kein Richtig oder Falsch geben wird. In vielen Fällen müssen Entscheidungen auf der Grundlage eines Dialogs mit den Benutzern des Systems getroffen werden. Es ist sehr gut möglich, dass mehrere verschiedene Darstellungen getestet werden müssen, um die Darstellung zu finden, die den Benutzern am besten gefällt.

Export & Load

Das Ergebnis eines Exports des Quellsystems hängt stark vom Quellsystem selbst ab und kann sich sogar zwischen verschiedenen Versionen desselben Quellsystems erheblich unterscheiden. Control-M generiert ein völlig anderes Format als Autosys, CA-Workload oder Automic. Manchmal erfolgt ein Export in Form eines XML-Dokuments, manchmal besteht er aus einer Liste von Anweisungen, die im Quellsystem ausgeführt werden können, oder es kann sich um eine Reihe von CSV-Dateien handeln. Es ist sogar möglich, dass das Quellsystem nicht die Möglichkeit eines Exports bietet, sondern seine Daten in einer Datenbank oder an einem genau definierten Ort im Dateisystem speichert.

Einige der Exportformate sind einfach zu verarbeiten (XML- oder CSV-Format, sowie in einer Datenbank gespeicherte Daten), andere erfordern Tools zum Extrahieren der Informationen aus dem Export (Anweisungen, Rohdaten).

Theoretisch wäre es möglich, Bibliotheken zu entwickeln, die einen wahlfreien Zugriff auf die exportierten Informationen ermöglichen. Dies wäre jedoch eine komplexe Aufgabe, und das Ergebnis würde eine schlechtere Perfomance aufweisen. Der bessere Ansatz ist die Erstellung von Tools, die die Informationen extrahieren und in eine relationale Datenbank laden. Der anschließende Datenzugriff erfolgt mit Standard-SQL, das eine einfach zu bedienende Schnittstelle bietet.

Ein weiterer Vorteil dieses Ansatzes besteht darin, dass das Data Dictionary des verwendeten Datenbanksystems eine komprimierte Liste von Datenelementen bereitstellt.

Analyse

Genau diese Liste von Datenelementen wird im nächsten Schritt analysiert. Einige Elemente haben eine offensichtliche Bedeutung, andere erfordern möglicherweise weitere Untersuchungen. Diese Analyse kann nicht ohne die Hilfe eines erfahrenen Benutzers des Systems durchgeführt werden.

Gleichzeitig ist diese Liste von Datenelementen im Grunde eine To-do-Liste. Wenn jedes Datenelement der Liste in den folgenden Schritten korrekt behandelt wird, muss das Ergebnis korrekt sein. Dies beweist auch, dass die Migration ein endlicher Prozess ist und ihre Dauer weitgehend von der Komplexität des Quellsystems abhängt.

Selbst mit der Eingabe eines erfahrenen Benutzers des Quellsystems werden Fehler gemacht. Aus diesem Grund ist der Analyseschritt Teil des iterativen Prozesses.

Graphenerstellung

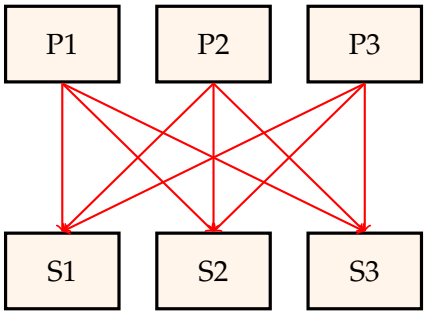
Aufgrund der Natur der Sache kann jede Implementierung eines Systems in einem Workload-Automatisierungssystem als Graph dargestellt werden, wobei die Knoten Jobs oder ganzen Mengen von Jobs (möglicherweise sogar Mengen von Mengen) entsprechen und die Kanten die Beziehung zwischen Vorgänger und Nachfolger aufzeigen.

In BICsuite wird die Vorgänger-Nachfolger-Beziehung als Abhängigkeit bezeichnet, wobei der Nachfolger verlangt, dass der Vorgänger mit einem bestimmten Exit State abgeschlossen wurde. Ein Job (Definition) ist ein Objekt, das eine Befehlszeile ausführt. Ein Batch ist eine Art Container, der Jobs und / oder anderen Batches enthält. Andere Workload-Automatisierungstools haben ähnliche Konzepte, die sich nur im Detail unterscheiden.

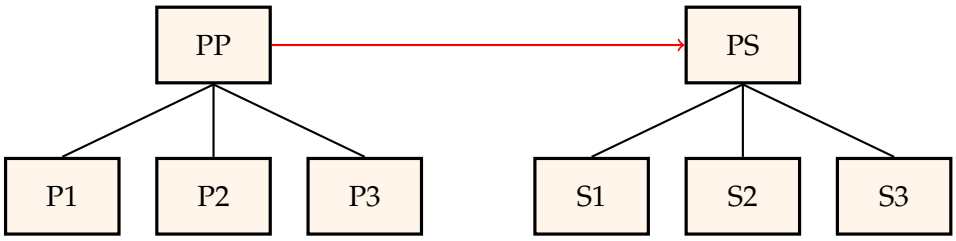
Die Hierarchie der Batches und Jobs lässt sich ebenfalls in einem Graphen (tatsächlich als Baum) darstellen. Das bedeutet, dass im gesamten Graphen zwei Arten von Kanten vorhanden sind. Der erste Typ ist die Abhängigkeit und der zweite Typ ist die Eltern-Kind-Beziehung.

Der Graph ohne Abhängigkeiten ist immer ein Wald (eine Ansammlung von Bäumen). Die Kanten sind vom Elternteil zu den Kindern gerichtet. Es gibt keine Zyklen (sonst wäre es keine Ansammlung von Bäumen). Der Graph ohne Eltern-Kind-Beziehungen sieht aus wie ein Netzwerk. Auch hier sind die Kanten gerichtet, vom Vorgänger zum Nachfolger.

Die Eltern-Kind-Hierarchie beeinflusst den Abhängigkeitsgraphen. Abhängigkeiten auf übergeordneter Ebene werden implizit auf die untergeordnete Ebene vererbt. Dies kann mit dem (nicht planaren) Utility-Graphen gut veranschaulicht werden. Alle drei Vorgängerjobs P1, P2 und P3 werden von drei Nachfolgejobs S1, S2 und S3 benötigt:



Wenn die impliziten Abhängigkeiten verwendet werden, die sich aus den Abhängigkeiten auf übergeordneter Ebene ergeben, sieht das Bild viel besser aus:

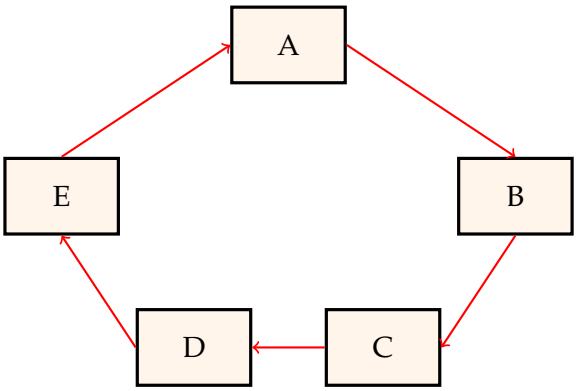


Es ist auf den ersten Blick zu erkennen, dass die Jobs S1, S2 und S3 auf den Abschluss der Jobs P1, P2 und P3 warten. Dies ist im ersten Graphen nicht so offensichtlich. Daher erleichtert die Eltern-Kind-Hierarchie das Verstehen (und Definieren) komplexer Beziehungen. Der Nachteil ist, dass Zyklen nicht so einfach zu erkennen sind. Ein Pfeil von S1 nach P3 sieht im zweiten Bild ziemlich harmlos aus. Tatsächlich entsteht jedoch ein Zyklus, der sofort deutlich wird, wenn wir im ersten Bild denselben Pfeil zeichnen. Dennoch übertreffen die Vorteile einer Eltern-Kind-Hierarchie die Nachteile erheblich. Das ist wahrscheinlich ein Grund, warum alle Tools zur Workload-Automatisierung diese auf die eine oder andere Weise anbieten.

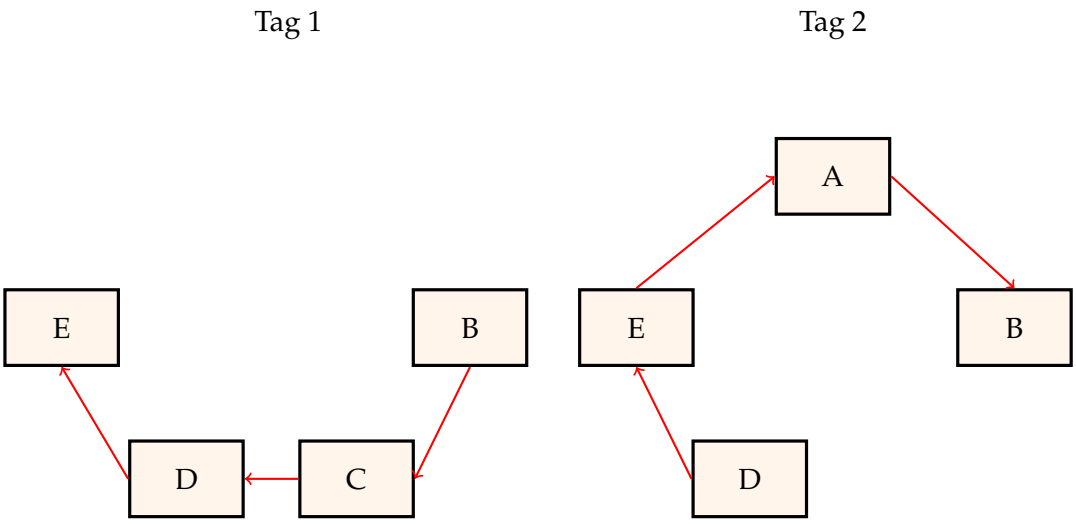
Der implizite Abhängigkeitsgraph sollte azyklisch sein, sonst wäre ein Deadlock die Folge. Wenn A auf B wartet und umgekehrt, wird nicht viel passieren. Die Definition der implizierten Abhängigkeiten ermöglicht die Konvertierung des gemischten Graphens mit den beiden Kantentypen, in einen eindeutigen Abhängigkeitsgraphen, in dem jede Kante eine Abhängigkeit darstellt. Die Graphenbibliothek kann dann verwendet werden, um Zyklen zu finden.

Wenn Zyklen gefunden werden, müssen sie analysiert und eliminiert werden. Am Ende ist Workload-Automatisierung das Ziel und ein Zyklus führt unter Umstän-

den in einen Deadlock, der manuell behoben werden muss. Nehmen wir an, dass ein Zyklus wie im Bild unten gefunden wurde:



Eine mögliche Ursache für die im Graphen aufgezeigte Situation wäre, dass A tatsächlich auf ein E des Vortages wartet. In diesem Fall ist es wahrscheinlich, dass A die Ergebnisse von E nicht wirklich benötigt, aber es ist einfach unerwünscht, dass die tägliche Verarbeitung beginnt, bevor die Verarbeitung des vorherigen Tages abgeschlossen ist (oder zumindest ein bestimmter Meilenstein innerhalb der Verarbeitung erreicht wurde). Eine solche Bedingung wird in BICsuite anders modelliert. So kann etwa eine Ressource verwendet werden, um zu signalisieren, dass der Lauf des vorherigen Tages noch nicht beendet ist. Auf diese Weise verliert die Abhängigkeit zwischen A und E ihre Bedeutung und kann beseitigt werden. Ein azyklischer Graph ist das Ergebnis.



Es kommt auch vor, dass eine solche Definition im Quellsystem niemals zu einem Deadlock führt, da jeden Tag mindestens einer der Jobs nicht ausgeführt wird. Obwohl sich so das Problem innerhalb des Quellsystems auflöst, hinterlässt es einen bitteren Nachgeschmack. Die Bedeutung der Abhängigkeiten ist schlecht definiert.

Wenn A an Tag 1 fehlt, haben wir eine transitive Abhängigkeit von B nach E, E ist ein Nachfolger von B. Wenn C am nächsten Tag fehlt, haben wir eine transitive Abhängigkeit von E nach B. Es ist kaum anzunehmen, dass dies logisch korrekt ist. An einem Tag benötigen wir Daten, um einen Bericht zu erstellen, und am nächsten Tag erstellen wir den Bericht, um die zugrunde liegenden Daten generieren zu können?

In vielen Fällen definieren diese Vorgänger-Nachfolger-Definitionen keine Abhängigkeiten, sondern werden (zufällig) hinzugefügt, um eine gleichzeitige Ausführung zu verhindern oder eine bevorzugte Ausführungsreihenfolge zu definieren. Und wieder bietet BICsuite eine andere Methode, wiederum ressourcenbasiert, um die gleichzeitige Ausführung zu verhindern. Bei Bedarf können Prioritäten verwendet werden, damit das System eine bevorzugte Ausführungsreihenfolge anstrebt.

Tatsächlich verschlechtert die Verwendung von Abhängigkeiten zum Definieren einer bevorzugten Ausführungsreihenfolge die Gesamtleistung. Wenn B warten muss, bis A abgeschlossen ist, weil es bevorzugt wird, dass A zuerst ausgeführt wird, kann B schon lange auf A warten, bevor A überhaupt ausgeführt werden darf. B könnte während der Zeit laufen, in der A noch auf seine Vorgänger wartet. Bei Verwendung von Prioritäten wird die bevorzugte Reihenfolge dokumentiert und das System wird entsprechend handeln, wenn sowohl A als auch B betriebsbereit sind, aber es würde B nicht warten lassen, wenn A nicht bereit ist.

Eine Graph-Bibliothek wie die weit verbreitete Python NetworkX-Bibliothek bietet ein großes und starkes Toolset. Insbesondere wenn die Grafiken sehr groß werden, ist es wichtig, leistungsstarke und korrekte Tools verwenden zu können. Eine große Anzahl vordefinierter mathematisch bewiesener Algorithmen ist verfügbar. Insbesondere wenn Probleme wie Zyklen auftreten, können solche Algorithmen effektiv zur Untersuchung der Graphen verwendet werden.

Es ist wichtig zu beachten, dass die NetworkX-Bibliothek beliebige Datenstrukturen als Eigenschaften von Knoten und Kanten speichern kann. Dies bedeutet, dass alle beim Export gefundenen Informationen in der Graphendarstellung der Daten gespeichert werden können.

Code-Generierung

Aus der Graphendarstellung lassen sich leicht Anweisungen generieren, mit denen die entsprechenden Objekte in BICsuite erstellt werden. Da jedes Objekt in BICsuite

mithilfe der Befehlssprache erstellt und / oder geändert werden kann, können alle Features des Systems angesprochen werden.

Daher besteht die verbleibende Aufgabe hier darin, eine Übersetzung der Quelldatenelemente in eine Darstellung in BICsuite zu finden, die dieselbe Semantik wie im ursprünglichen System aufweist. Frühere Migrationsprojekte haben gezeigt, dass es immer möglich war, eine solche Übersetzung zu finden. In vielen Fällen gab es sogar mehrere Alternativen. Die Kriterien für die Auswahl einer bestimmten Implementierung wurden durch die weichen Anforderungen vorgegeben. In einigen Fällen wurden einige der Alternativen durch Implementierung und Test bewertet.

Aufgrund von Unterschieden in den Quellsystemen und Unterschieden in den weichen Anforderungen ist der Code-Generierungsschritt für jedes Projekt spezifisch. Obwohl Teile der Code-Generatoren früherer Projekte verwendet werden könnten, ist es oft mehr Arbeit, projektspezifischen Code zu entfernen, als von vorne zu beginnen. Die Implementierung des Grundgerüsts (Anlegen von Jobs und Abhängigkeiten) kostet weniger als einen Tag. Der Vorteil, von vorne zu beginnen, ist, dass es keine unerwünschten Nebenwirkungen gibt.

Wenn eine ganze Reihe ähnlicher Systeme wie ein Entwicklungssystem, ein Testsystem und ein Produktionssystem migriert werden müssen, ist es natürlich sinnvoll, den Generator wiederzuverwenden. In diesem Fall sind jedoch sowohl das Quellsystem als auch die weichen Anforderungen entweder gleich oder zumindest sehr ähnlich. Der beste Ansatz besteht darin, mit dem einfachsten System zu beginnen, im Beispiel des Entwicklungssystems, und den Rest in der Reihenfolge der Komplexität zu migrieren. Die Code-Generierung folgt dann demselben Weg. Der Basisgenerator, der zum Migrieren des ursprünglichen Systems verwendet wird, wird dann schrittweise verfeinert, um die Anforderungen der nachfolgenden Systeme abzudecken.

Manuelle Anpassungen

Es gibt keine Regel ohne Ausnahme. Wenn der Code-Generator das Werkzeug der Wahl ist, um die regelbasierte Zuordnung vom Quellsystem zu BICsuite zu implementieren, ist es nicht sinnvoll, auch die Behandlung jedes Sonderfalls zu implementieren. Oft ist es schneller und einfacher, einen Code zu erstellen, der die Ausnahmen nach der Code-Generierung behandelt, als diesen in den Generator zu integrieren.

Auch hier trifft der Vergleich mit einem Compiler zu. Oft wird der Umwandlungsprozess in mehreren separaten Schritten ausgeführt. Zuerst wird die Eingabe analysiert und auf Syntaxfehler überprüft. Das Ergebnis dieses Schritts ist ein Analysebaum. Der nächste Schritt besteht darin, Zwischencode zu generieren, der unabhängig von der Zielhardwarearchitektur ist. Dieser Zwischencode wird dann in Maschinencode für die Zielarchitektur übersetzt. Last but not least wird dieser Maschinencode optimiert, um eine optimale Aufführungsgeschwindigkeit zu gewähr-

leisten.

Wenn dieser Ansatz in der Compilertechnologie mit dem Migrationsverfahren verglichen wird, sind die Parallelen auffällig:

Compiler	–	**Migration**
Parser	–	Export & Load
Intermediate Code	–	Graphische Darstellung
Maschinen Code	–	Generator
Optimierter Code	–	Manuelle Anpassungen

Wenn sich herausstellt, dass einige der manuellen Anpassungen allgemeinen Regeln folgen, ist es sinnvoll, sie in den Generatorcode zu integrieren. Das ist insbesondere für den Fall vorteilhaft, dass mehrere ähnliche Systeme migriert werden. Je mehr Code generiert wird, desto weniger fehleranfällig ist die Migration. Der große Vorteil des generierten Codes besteht darin, dass er immer richtig oder immer falsch ist. Ersteres ist erfreulich, aber was noch wichtiger ist: Letzteres ist leichter zu erkennen.

Test

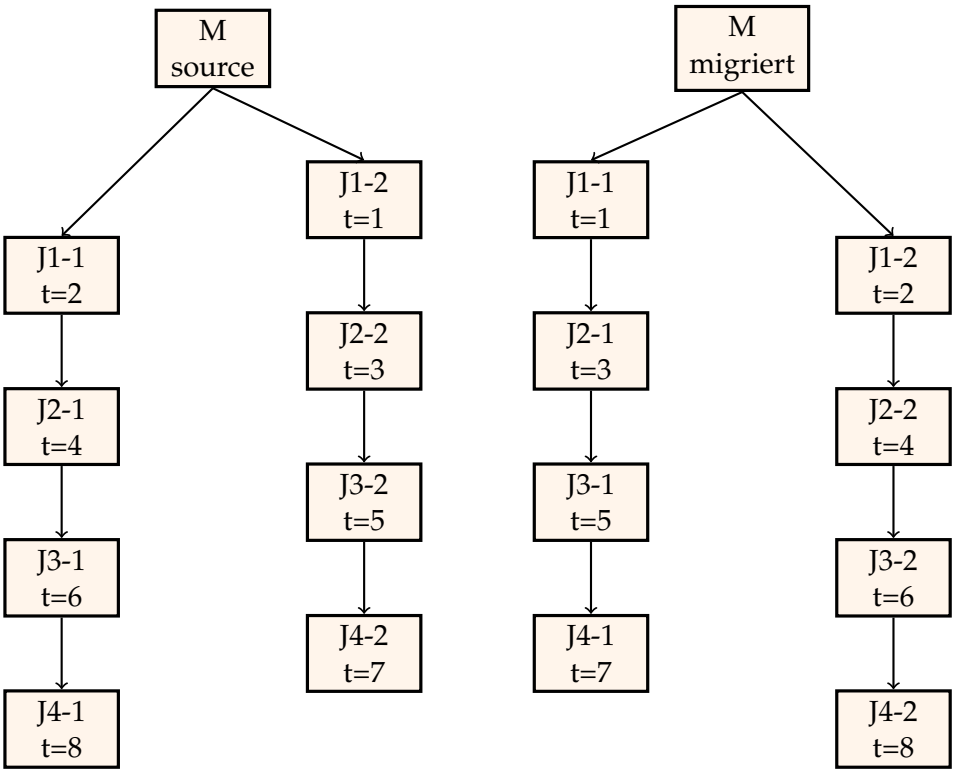
Es ist davon auszugehen, dass bei gleicher Ausführung der Befehlszeilen und gleicher Ausführungsreihenfolge die Ausgabe bei gleicher Eingabe identisch sein muss. Da nun die von den Jobs ausgeführten Befehlszeilen innerhalb des Prozesses kopiert werden, ist die erste Bedingung erfüllt. Daher muss beim Testen nur gezeigt werden, dass die Ausführungsreihenfolge in beiden Systemen gleich ist.

Natürlich muss auch sicher gestellt sein, dass keine Jobs ausgelassen und keine zusätzlichen Jobs ausgeführt werden. Die Ausführungsreihenfolge aufgrund der parallelen Ausführung von Jobs muss allerdings nicht so streng beibehalten werden, es sind durchaus sogar Optimierungen möglich. Das nächste Bild veranschaulicht diese Behauptung.

Die Ausführungsreihenfolge im Quellsystem ist J1-2, J1-1, J2-2, J2-1, J3-2, J3-1, J4-2, J4-1, während die Ausführungsreihenfolge im migrierten System J1-1, J1-2, J2-1, J2-2, J3-1, J3-2, J4-1, J4-2 ist.

Dennoch sollten beide Ausführungsreihenfolgen als gleich angesehen werden, da jede Kette in derselben und korrekter Reihenfolge ausgeführt wird. Wenn es Abhängigkeiten zwischen den Jobs in beiden Ketten gab, sollten diese Abhängigkeiten im Quellsystem modelliert worden sein und dann auch Teil des migrierten Systems sein. Tatsächlich könnte diese Art von Unterschieden auftreten, wenn zwei Läufe innerhalb des Quellsystems verglichen werden. Zu viele unkontrollierbare Faktoren wie die Prozessplanung des Betriebssystems, die nebenläufige Verarbeitung, der Netzwerkdurchsatz und die Latenz spielen eine Rolle für den genauen

Zeitpunkt des Prozessstarts und deren Dauer.



Der größte Teil der Tests muss nicht die ëchten"Befehlszeilen ausführen. Dummy-Programme, die nur eine Weile schlafen oder einfach einen Erfolg zurückgeben, reichen in vielen Fällen aus. Dies beschleunigt die Tests und ermöglicht es dem Migrationsteam, die Tests in einer Art simulierter Umgebung durchzuführen, wie z.B. dem Computer des Entwicklers des Generators.

Wenn alle Funktionen implementiert sind und alle anderen Tests zufriedenstellend sind, müssen zunächst einige echte Tests in einer seriösen Testumgebung durchgeführt werden. Und wenn diese Tests erfolgreich sind, kann die Inbetriebnahme geplant werden.

Time Scheduling

Bei jeder Migration ist die Zeitplanung, welcher Job wann ausgeführt werden soll, ein Thema für sich. Das BICsuite-Zeitplanungsmodul ist äußerst leistungsfähig und unterstützt nachdrücklich die regelbasierte Planung.

Anstatt jährlich eine Liste mit Ausführungsdaten (und -zeiten) zu erstellen, ist es weitaus besser, eine allgemeine Regel zu erstellen. Wenn die Gehälter der Mitarbeiter am vorletzten Geschäftstag des Monats überwiesen werden, kann natürlich

ermittelt werden, um welche Tage es sich handelt, und dann eine Liste erstellt werden. Mit BICsuite kann der Benutzer jedoch eine Regel erstellen, die genau das tut. Die einzige Arbeit in BICsuite besteht darin, jedes Jahr eine neue Liste von Feiertagen zu erstellen. Basierend auf dieser Liste von Feiertagen und Wochenenden berechnet BICsuite selbständig, welcher Tag der vorletzte Geschäftstag des Monats ist.

Natürlich sind die Funktionen des Zeitplanungsmoduls nicht darauf beschränkt. Selbst scheinbar schwierige Anforderungen wie "der 10. Tag des Monats, es sei denn, es ist ein Wochenende oder ein Feiertag, in dem Fall muss es der nächste Geschäftstag sein", stellen keine Herausforderung für das System dar.

Gleichzeitig ermöglicht das System dem Benutzer, die allgemeine Regel zu überschreiben, ohne die Regel selbst zu beeinflussen.

Der Vorteil dieses Ansatzes besteht darin, dass er nahezu wartungsfrei ist. Das manuelle Erstellen von Listen mit Ausführungsdaten ist ziemlich fehleranfällig. Abgesehen davon ist es eine Menge Arbeit, es richtig zu machen. Außerdem könnte die tiefere Logik hinter den Listen verloren gehen.

Aber genau wie die Liste der Feiertage kann jede andere Liste von Daten (und Zeiten) erstellt werden. Dies bedeutet, dass der Benutzer die Möglichkeit hat, entweder das alte System mit Datumslisten zu verwenden oder auf einen regelbasierten Ansatz umzustellen. Welche Option am besten ist, hängt von den weichen Anforderungen ab.

Unabhängig von der gewählten Option wird dieser Teil der Migration normalerweise als separater Schritt ausgeführt. Es ist im Grunde die Trennung von *was* getan werden muss und *wann* es getan wird. Diese Trennung führt zu zwei Perspektiven für die Workload-Automatisierung. Eine Gruppe von Systemen interessiert sich dafür, was zu einem bestimmten Zeitpunkt zu tun ist, und die anderen Systeme interessieren sich dafür, wann etwas zu tun ist. Mit anderen Worten, ein Teil der Systeme wird aus Sicht der Zeit und der andere aus Sicht der Aufgaben gesteuert. Es ist dieser Perspektivwechsel, der oft zu Unglauben oder zumindest Verwirrung führt.

Utilities

Um mehr von den weichen Anforderungen abzudecken, müssen häufig Dienstprogramme entwickelt werden. Die Gründe können von der Vereinfachung von Bedieneraktionen über die Nachahmung alter Workflows bis hin zum Hinzufügen nicht standardmäßiger Funktionen reichen. (Nicht standardisiert in dem Sinne, dass es nicht Teil des BICsuite-Systems ist).

Da BICsuite über eine leistungsstarke und benutzerfreundliche API verfügt, ist es normalerweise kein wirklicher Aufwand, leistungsstarke Tools zu erstellen, die Menschen glücklich machen. In vielen Fällen ist die genaue Spezifikation des Tools der schwierige Teil, die anschließende Implementierung ist oft sehr einfach.

Schulung

Die Schulung des Teams ist aus zwei Gründen wichtig. Der erste Grund liegt auf der Hand. Da sie in Zukunft mit dem neuen System arbeiten möchten, müssen sie wissen, wie sie damit arbeiten sollen. Obwohl BICsuite nicht schwer zu bedienen ist, gibt es viele Konzepte, die sich von den Konzepten des Quellsystems unterscheiden. Es wird wichtig sein, diese neuen Konzepte zu verstehen.

Der zweite Grund für Schulung hängt mit dem Migrationsprozess selbst zusammen. Viel Kommunikation zwischen den Benutzern und dem Migrationsteam ist entscheidend für den Erfolg des Projekts. In vielen Fällen müssen Lösungen diskutiert werden, die eine Funktionalität im Quellsystem einer Implementierung in BICsuite zuordnen. Dies wird viel einfacher, wenn die BICsuite-Terminologie zumindest verstanden wird.

Hauptsächlich der zweite Grund deutet darauf hin, dass das Projekt mit der Schulung beginnen muss. Gleichzeitig können die Benutzer während der Migration bereits Know-how mit dem System sammeln, so dass sie zum Zeitpunkt der Inbetriebnahme bereits über essentielle Fähigkeiten verfügen.

Fazit

Dieses Dokument zeigt eine Methodik, die sich in der Praxis bewährt hat und den Prinzipien des bekannten Bereichs der Compilerkonstruktion folgt. Es zeigt sich auch, dass der erforderliche Aufwand hauptsächlich von der Komplexität des Quellsystems und der verwendeten Workflows abhängt, nicht von der Größe der implementierten Anwendung.

Die Liste der ursprünglichen Datenelemente garantiert die zeitliche und finanzielle Endlichkeit des Prozesses und gibt einen Hinweis auf die erzielten Fortschritte. Der iterative Charakter des Prozesses ermöglicht es, die Richtigkeit jeder Verfeinerung zu beweisen, was letztendlich die Richtigkeit des Ganzen garantiert.

Ein großer Teil der Migration, einschließlich der Tests, kann auf einem einzelnen isolierten Computer durchgeführt werden. Erst die Tests in der tatsächlichen Umgebung erfordern einen breiteren Aufbau. Das reduziert die Kosten in der frühen Phase des Projekts erheblich. Die Risiken des Projekts sind daher begrenzt und eine Stornierung des Projekts, aus welchem Grund auch immer, bleibt immer eine Option.

Da die gesamte Migration als automatisierter Prozess implementiert wird, sind jederzeit gleichzeitige Änderungen am Quellsystem möglich. Es ist jedoch sinnvoll, das Quellsystem zwischen dem letzten erfolgreichen Test in der realen Umgebung und dem Go-Live des migrierten Systems einzufrieren.