A Dependency Management and Impact Assessment Framework for Business Process Management

Adrian Troy Christie

Directed By:
Prof. Liam Peyton

Thesis

Submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of

Master of Science, Electronic Business Technologies

uOttawa

University of Ottawa

Ottawa, Ontario, Canada

April 23, 2019

Abstract

Business Process Management (BPM) is a relatively new development paradigm that takes a high-level approach to coding by leveraging a graphical, “flow chart” aesthetic that allows users to assemble modular tasks into a larger process. The resulting diagrams effectively enshrine the organizations processes into an executable model that provides an objective and transparent view of the process and the activities contained within. In doing so, BPM models serve as both a system to guide employees through proper business procedures as well as documentation of the businesses processes.

Similar to most other software development environments, BPM development platforms possess several features intended to address the needs of code versioning, dependency management and impact assessment. However, due to the unique way that development is done in BPM platforms, the more traditional functionality of these features sometimes renders them ineffective and ill suited to the task of BPM development. Changes to lower level reusable components in these BPM models can result in impacts to diverse processes across an organization that are difficult to predict and onerous to locate.

There is much room for improvement in BPM development tools. This thesis proposes a new framework for dependency management and impact assessment to improve the usability, effectiveness and efficiency. The framework is composed of a Business Process Component Architecture, a Dependency Data Model and an Upgrade Algorithm which are all used to provide increased visibility over dependent processes and superior guidance during upgrade operations.

Several example case scenarios are be used to evaluate our proposed framework. The cases represent progressive degrees of complexity to test the capabilities and robustness of the framework. Overall, the framework was able to appropriately handle the case examples used and showed promise in terms of providing practical effort, time and cost savings for BPM developers. The framework can also provide developers assistance in locating circular dependencies, but is subject to the same limitations as developers when attempting to upgrade these relationships.

Acknowledgements

I would like to start by thanking everyone, not specifically mentioned in these acknowledgements, whose support and understanding contributed immensely to the eventual completion of this thesis and the achievement that it represents to me. You know who you are and I thank you sincerely.

To my parents, Greg Christie and Suzanne Dionne, your unending support, encouragement, patience and generosity were more than I deserved or could have asked for. I would not have had the courage to start a graduate degree without your influence, let alone have the persistence to finally finish it. I truly owe everything that I am and that I have to you.

To my thesis supervisor, Prof. Liam Peyton, your knowledge and guidance were what made it possible for me to finish writing this thesis. I greatly appreciate all of the effort and patience that you invested in me, and I am grateful for the times you took off the velvet gloves and cracked the whip to keep me moving forward. It is not an understatement for me to say that I would have been completely lost in this process without you. You went above and beyond to help me succeed. Thank you, Liam.

I would like to thank Kavya Mallur and Sepideh Bahrani for your continual encouragement and for directing me towards the Electronic Business Technologies program for my graduate studies. I would like to thank my boss, Arzina Pandjou, for affording me the understanding and flexibility to complete this thesis even when it conflicted with team priorities and work deadlines. You helped make this possible.

To my late grandfather, Jean-Paul Dionne, your passion for education and your endless pursuit of knowledge have and will always be a great source of inspiration to me. I imagine that the prospect of me getting a Masters degree would have thrilled and delighted you and that thought has kept me motivated on many occasions throughout the nearly five years that I managed to stretch this degree into.

Thanks are also in order for Craig Kuziemsky, Hussein Al Osman and Tet Yeap. Thank you for generously volunteering your time to take part in my thesis evaluation. It was a very positive experience. The feedback you provided was interesting and constructive. I am sure the thesis would have turned out even better if I had been aware of your suggestions back when I was doing the bulk of the writing.

Finally, last but definitely not least, I need to thank my wonderful girlfriend and partner Katherine Potts. The time and resources that I had to invest writing this thesis impacted you most of all and yet you consistently showed me understanding and provided me support. You sat with me to keep me company while writing. You brought me food and drink to help me avoid distractions and maintain my momentum. You sacrificed your own time and activities for my academic achievement. You even brandished a literal riding crop to discourage my perpetual procrastination with the threat of pain. I recognize the deep and powerful love behind these acts and I thank you profusely from the bottom of my heart. Without you and your help, I would still be a long way from finishing this thesis and degree. I can only hope to provide you such valuable assistance some day.

To all of you, for everything you have done, thank you.

Table of Contents

Abstract ii
Acknowledgements iv
Table of Contents vi
List of Figures ix
List of Acronyms x
Chapter 1. Introduction 1
1.1. Problem Statement 1
1.2. Thesis Motivation 3
1.3. Thesis Contributions 5
1.4. Thesis Methodology 7
1.5. Thesis Organization 12
Chapter 2. Background 13
2.1. BPM 13
2.1.1 Business Processes 13
2.1.2 Business Process Management 14
2.1.3 Business Process Modeling Notation 15
2.1.4 BPM Suites 17
2.2. BPM Components Relevant to Dependency Management 19
2.2.1 Process Applications 19
2.2.2 Toolkits 19
2.2.3 Snapshots 20
2.2.4 Dependency/Toolkits Panel 20
2.2.5 Where Used lists 21
2.3. Dependency Management & Impact Assessment 23
2.3.1 BPM Dependency Version Mismatch 23
2.3.2 Circular Dependencies 23
2.3.3 Change Propagation 24
2.3.4 Total Impact 24

2.3.5 Dependency management tools 24
2.4. Related Work 26
2.4.1 IBM BPM 8.5 26
2.4.1 Maven 27
2.5. Chapter Summary 29
Chapter 3. Dependency Management and Impact Assessment Framework 30
3.1. Gap Analysis and Evaluation Criteria 30
3.1.1 Visual 33
3.1.2 Low Learning Curve 33
3.1.3 Accurate Dependency Analysis 34
3.1.4 Recursive Dependency Discovery 35
3.1.5 Robust Upgrade Algorithm 35
3.1.6 Development Effort 36
3.1.7 Setup and Maintenance Costs 36
3.1.8 Standardization 37
3.2. Framework Overview 39
3.2.1 User Interaction 41
3.2.2 Abstract Communication Module 43
3.3. Business Process Component Architecture 45
3.4. Dependency Data Model 50
3.5. Upgrade Algorithm 56
3.6. Chapter Summary 61
Chapter 4. Case Studies 62
4.1. Simple toolkit example 62
4.1.1 Overview 62
4.1.2 Business Process Component Architecture 63
4.1.3 Dependency Data Model 64
4.1.4 Upgrade Algorithm 69
4.1.5 Results 72
4.2. Common toolkit upgrade example 73
4.2.1 Overview 73
4.2.2 Business Process Component Architecture 73

4.2.3 Dependency Data Model 74
4.2.4 Upgrade Algorithm 79
4.2.5 Results 86
4.3. Flawed dependency structure upgrade example 88
4.3.1 Overview 88
4.3.2 Business Process Component Architecture 89
4.3.3 Dependency Data Model 90
4.3.4 Upgrade Algorithm 97
4.3.5 Results 100
4.4. Irredeemable dependency structure example 101
4.4.1 Overview 101
4.4.2 Business Process Component Architecture 101
4.4.3 Dependency Data Model 104
4.4.4 Upgrade Algorithm 113
4.4.5 Results 113
4.5. Chapter Summary 114
Chapter 5. Evaluation 115
5.1. Related Works/Dependency Management Tools 115
5.2. Case Evaluation 120
5.3. Assumptions, Limitations and Threats to Validity 126
5.3.1 Assumptions 126
5.3.2 Limitations 127
5.3.3 Threats to Validity 127
Chapter 6. Conclusions and Future Work 129
6.1. Conclusions 129
6.2. Future Work 131
References 133

List of Figures

Figure 1-1: DSRM (Design Science Research Methodology) Process Model (Peffers et al., 2007) 7
Figure 2-1: BPMN Language Elements (Wohed, van der Aalst, Dumas, ter Hofstede, & Russell, 2005). 16
Figure 2-2: IBM BPM Toolkits Panel (Kolban, 2014) 21
Figure 3-1: Framework for Dependency Management and Impact Assessment 39
Figure 3-2: Communication Model for Abstracting Specific BPM APIs 44
Figure 3-3 Example Business Components 46
Figure 3-4: Example of project hierarchy impacted by change 48
Figure 3-5: Sample of Change Dictionary 49
Figure 3-6: Dependency Data Model Table 51
Figure 3-7 Upgrade Algorithm 60
Figure 4-1: Call introductions Process App dependency tree 64
Figure 4-2: Change Selection Screen 65
Figure 4-3: Example 1 Dependency Data Model Table 67
Figure 4-4: General Account Deficiency Check v1.3 68
Figure 4-5: General Account Deficiency Check v1.4 69
Figure 4-6: Critical Account Checks before upgrade 70
Figure 4-7: Critical Account Checks after upgrade 71
Figure 4-8: Call Scripting Process App dependency tree 74
Figure 4-9: Change Selection Screen 75
Figure 4-10: Upgrade Tiers 76
Figure 4-11: Example 2 Upgrades Section of Dependency Data Model 78
Figure 4-12: Check Connection Status v1.0 80
Figure 4-13: Check Connection Status v1.1 80
Figure 4-14: General Account Deficiency Check v1.4 81
Figure 4-15: General Account Deficiency Check v1.5 82
Figure 4-16: Display Connection Details v1.0 83
Figure 4-17: Display Connection Details v1.1 84
Figure 4-18: Critical Accounts Checks 85
Figure 4-19: Status Verification 86
Figure 4-20: Call Scripting Process App dependency tree 89
Figure 4-21: Change Selection Screen 91
Figure 4-22: Upgrade Tiers 92
Figure 4-23: Upgrades and Project Sections of Dependency Data Model 96
Figure 4-24: Upgrade Tiers with snapshot markers 99
Figure 4-25: Call Scripting Process App dependency tree with circular dependency 102
Figure 4-26: Change Selection Screen 104
Figure 4-27: Upgrade Tiers 105
Figure 4-28: Upgrades and Project Sections of Dependency Data Model 112
Figure 5-1: Framework and Related works evaluation grid 117
Figure 5-2: Case evaluation grid 122

List of Acronyms

Acronym	Definition
ADP	*Acyclic Dependencies Principle*
API	*Application Programming Interface*
BPD	*Business Process Definition*
BPM	*Business Process Management*
BPMN	*Business Process Modelling Notation*
DB	*Database*
DM	*Dependency Management*
DSRM	*Design Science Research Methodology*
DUA	*Dependency Upgrade Agent*
GUI	*Graphical User Interface*
IA	*Impact Assessment*
IAE	*Impact Assessment Engine*
IDE	*Integrated Development Environment*

Chapter 1. Introduction

1.1. Problem Statement

In any organization, business processes are pervasive. A business process is a defined collection of linked structured tasks, activities, and decisions performed together to produce a desired set of results in order to contribute towards the achievement of business goals on behalf of the organization. Companies are increasingly moving their business processes online using Business Process Management (BPM) tools and technologies (Olga Levina, 2010).

The advantages of BPM systems for businesses are numerous. Managed online processes help reduce training time and costs. Defined roles allow complex processes to be addressed in a coordinated and expedient fashion. Tracking of tasks can be leveraged for quality control and to fairly distribute task workload among members of larger teams. The central value proposition of BPM systems, however, is in the creation of visual models that map the organization's processes. Though the discovery and modeling of these processes can be challenging, understanding these allows businesses to target their efforts towards changes that will bring real improvement and open up new opportunities.

The ultimate motivation for adoption of BPM is to improve business performance as a whole (Sánchez, Ruiz, García, & Piattini, 2013). In theory, businesses can use this approach to make continued improvements to processes throughout their organizations, but the reality often does not live up to this expectation. BPM platforms such as IBM BPM 8.5 are typically quite agile when they are first installed and configured. They allow for

developers to make great progress in the mapping and programming of processes, but this momentum rapidly slows down as more and more processes are defined, stored and managed in the BPM platform. Increasing interconnectedness between processes and a variety of BPM component dependencies can lead to complicated and fragile dependency relationships that are challenging to manage.

It is possible to mitigate some of these challenges by maintaining a strict set of development best practices pertaining to the BPM environment, but the controls offered in the BPM suites for versioning and dependency control are very basic and do not provide batch functionality or a holistic view of dependency relationships between projects. Increased difficulty with regular development and maintenance activities ultimately reduces the life span of the BPM platform as there comes a time when the benefit of further development is outweighed by the difficulty/risk of making a change.

1.2. Thesis Motivation

The lack of advanced dependency management and impact assessment tools for BPM systems presents an obvious and significant obstacle to widespread adoption and large-scale implementations. Improving the support for dependency management and impact assessment is essential to increasing the viability of BPM as an enterprise development platform. Tackling all limitations to development in BPM environments is too broad a task, but the workplace experience of the author using BPM technology points to dependency management as being a severe bottleneck on development activities and one that is applicable to the large majority of BPM teams/shops.

Software developers who do more traditional coding have access to a variety dependency management features built into the integrated development environments (IDE) in which they work. These assists are also offered as standalone frameworks to be used independently of the choice of IDE. One such tool, used to manage dependencies for projects written in Java, is Maven. This idea of a framework that works for a coding language across development environments is precisely the concept that could bring the most benefit to the field of BPM as a whole. Could a similar tool be built around the common principles of business process management (BPM) and the notation (BPMN 2.0) just as Maven is built around the language and object-oriented structure of Java? Can the logic necessary to manage and upgrade dependencies in BPM environments be converted to an algorithm? What would such a tool look like, in what ways and to what extent could it assist developers to manage dependency relationships surrounding their BPM projects?

The intent of this thesis is to answer these questions with the goal of improving development activities surrounding dependency management, and by extension impact

assessment, in BPM environments. The same assists that can be found and leveraged by programmers of other coding languages must be made available to BPM developers in a way that is adapted to the business process management paradigm in order to make BPM development cost competitive in relation to other software platforms that could be used.

1.3. Thesis Contributions

The main contributions of this thesis are as follows:

1. ***A Dependency Management and Impact Assessment Framework for Business Process Management*** that ensures changes to a business process component are propagated to all other components and processes that are dependent upon it. The framework includes:

    a. *A Business Process Component Architecture* that explicitly defines the components relevant to business process management technology for the purposes of Impact Assessment (IA) and Dependency Management (DM). The architecture clearly defines and describe the required, core components while remaining general enough to be applied to any of the common BPM platforms in use by business and industry today.

    b. *A Dependency Data Model* that possesses the informational specificity needed to accurately describe the dependency structure of whatever BPM project/application is being analyzed. The model stores any information that is necessary for the purposes of traversing the BPM project structure and upgrading affected dependencies in an ordered fashion. The model also allows for the detection of dependency errors commonly encountered in BPM applications such as circular dependencies, convoluted dependency hierarchies and version mismatches.

c. *An Upgrade Algorithm* that describes the steps necessary to successfully complete a dependency upgrade across all dependencies in response to a change in a BPM project. The algorithm accepts the dependency data model and several user defined parameters as inputs and is detailed enough to use these inputs and BPM process data to automate the upgrade process without requiring user interaction except when handling exceptional circumstances or pausing for development to accommodate changes.

1.4. Thesis Methodology

This thesis is based on Design Science Research Methodology (DSRM) as outlined by (Hevner, March, Park, & Pam, 2004) which defines DSRM as a methodology adapted to the creation of some artifact "… intended to solve identified organizational problems". Although the absence of a tangible artifact can make this thesis' contributions difficult to quantify under the traditional DSRM approach, later work by Gregor & Hevner address the difficulty of fitting information technology research into the DSR methodology and propose making knowledge a viable contribution under DSR. Under this revised methodology, this thesis aims to produce prescriptive knowledge by converting industry experience and academic research in dependency management in BPM suites into an operational model for a dependency management framework for BPM (Gregor & Hevner, 2013). At a high level, DSRM is an iterative process of observation, goal definition, solution development and evaluation/re-evaluation. The precise steps and sequence of these activities is described by Figure 1-1 below (Peffers, Tuunanen, Rothenberger, Chatterjee, 2007).

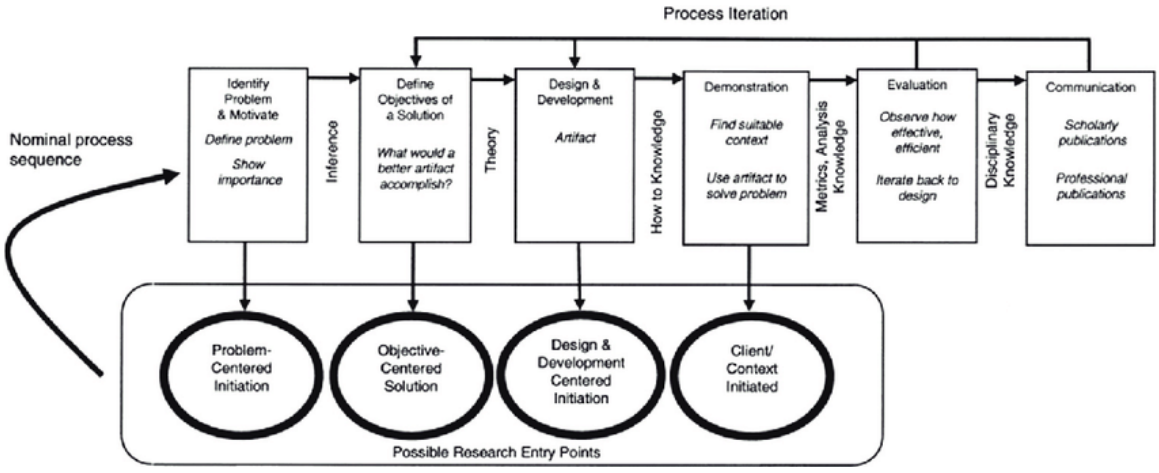

Figure 1-1: DSRM (Design Science Research Methodology) Process Model (Peffers et al., 2007)

The work described in this thesis began at the problem identification step which is indicated in the diagram as the optimal starting point for the iterative DSRM process. Initial work on the proposed solution went through several quick iterations in response to a series of informal thought experiments and some hands-on experience with the IBM BPM 8.5.5 development suite. Further iterations then resulted from the 4 cases proposed in Chapter 4. The progressively more challenging situations presented in the cases each resulted in modifications being made to the dependency data model and the upgrade algorithm defined in Chapter 3.

The thesis activities corresponding to the steps of the DSMR methodology are as follows:

Iteration 1 – Initial Inspiration

1. Identify Problem & Motivate: A problem was observed in BPM development teams regarding the resource intensive nature of managing dependencies when upgrading projects with new changes. Propagating changes throughout the dependent projects would take a team of minimum five people and an entire day to complete. A gap was observed between the basic dependency functions available to BPM developers and the advanced assists and tools available to programmers of other computer languages.

2. Defining Objectives of a Solution: The goal was to reduce man-hour costs and productivity losses resulting from such onerous dependency management activities by providing assists that could either speed up the steps of dependency management activities, reduce the human resource input or both.

3. Design & Development: A framework was designed that offers two important assists to developers. These are a holistic impact assessment feature that shows the impact of an new development change across all levels of project dependencies to the N^{th} degree and a dependency upgrade feature that introduces as much automation as can be delegated to a computer system and guides the developers down the complete upgrade path from start to finish.

4. Demonstration: A series of four simplified, but realistic cases were devised to test the capability of the proposed framework in progressively more challenging upgrade scenarios.

5. Evaluation: The framework was compared to native BPM development functionality as well as to Maven, a dependency management tool for Java projects. The framework was also evaluated by assessing its performance in the four cases representing different development scenarios.

Iteration 2 – Changes in response to Case 1

1. Identify Problem & Motivate: Case 1 prompted the realization that even the simplest upgrades cannot be fully automated if the changes being brought in cause the need for additional development in the dependent project.

2. Defining Objectives of a Solution: Framework must detect types of changes made and flag those that may require additional development for developer inspection.

3. Design & Development: To correct this issue an analysis step was introduced into the upgrade algorithm and the Project[n].Build[n].Dep[n].Changes[n] data

structure was added in order to display the results of the differential analysis. This information is used to pause the automated portions of the dependency upgrade process and draw the developer's attention to the area that may need additional development.

4. Demonstration: Framework re-tested against the same 4 cases.

5. Evaluation: Evaluation found new solution to work for Case 1, but be insufficient for Case 2.

Iteration 3 – Changes in response to Case 2 and Case 3

1. Identify Problem & Motivate: Cases 2 and 3 revealed that several upgrade paths may converge on the same point at the same or different times. This causes 2 issues. It increases the difficulty in tracking all the dependencies requiring upgrades when the framework reaches one of these convergence point projects. It also can mean that not all dependencies of a project may be ready for upgrading the first time a project is encountered in the upgrade algorithm.

2. Defining Objectives of a Solution: Framework must track all upgrade dependency relationships leading to a particular project and must store all required upgrades for each project as well as the order in which they must be performed.

3. Design & Development: To correct this issue the Upgrades[n] data structure was introduced into the Dependency Data Model. This allows the framework to track and save each dependency that must be upgraded within every affected project.

4. Demonstration: Framework re-tested against the same 4 cases.

5. Evaluation: Evaluation found new solution to work for Cases 1, 2 and 3 but be insufficient for Case 4.

Iteration 4 – Changes in response to Case 4

1. Identify Problem & Motivate: Case 4 confirmed that there are some instances of bad development practices that simply cannot be resolved in their current state. This was primarily the case for circular dependencies which cannot be resolved without a rather drastic changes to the dependencies of surrounding projects.

2. Defining Objectives of a Solution: The framework should have the ability to detect these few "fatal flaw" cases and warn developers about where they are located should any be encountered.

3. Design & Development: Upgrade algorithm logic was amended to check for the known fatal cases and to exit out gracefully avoiding unnecessary work and further confusion.

4. Demonstration: Framework re-tested against the same 4 cases.

5. Evaluation: Evaluation found new solution to work for Cases 1, 2, 3 and 4.

6. Communication: Results were published in this thesis.

1.5. Thesis Organization

This thesis is organized as follows:

- **Chapter 1** presents an introduction, together with the context and the scope of the research.

- **Chapter 2** provides a background to our research. The chapter provides context for the research problem that we are solving in this work by giving a detailed background on business processes, BPM development and dependency management. The chapter also provides literature review from related works. The technologies and tools used in these related works are analysed and compared.

- **Chapter 3** elaborates the research problem, gap analysis and evaluation criteria. The chapter then presents our framework for dependency management and impact assessment.

- **Chapter 4** presents our case studies and provides details of our experiments.

- **Chapter 5** presents the evaluation of framework against the identified criteria and comparison with related work.

- **Chapter 6** briefs the conclusions drawn from our work. And the chapter also speaks about the future work.

Chapter 2. Background

This chapter begins by explaining the key concepts and components related to Business Process Management, then summarizes the relevant background related to development and dependency management in BPM environments. Finally, related works for comparison are identified.

2.1. BPM

Business Process Management is a modern paradigm for establishing and governing operation within a business organization. This section defines business processes, business process management, business process management notation and BPM suites.

2.1.1 Business Processes

Business processes are explicit definitions of the work required to perform core functions within an organization. Often through visual means such as flowcharts, business processes express "the combination or set of activities within an enterprise with a structure describing their logical ordering and dependence, whose objective is to produce a desired result" (Aguilar-Saven, 2003).

These business process activities can be orchestrated among multiple participants including employees from across the organization and system resources available on the organization's internal network or the internet (Leymann, Roller, & Schmidt, 2002).

2.1.2 Business Process Management

Business Process Management (BPM) refers to a category of applications that support a particular software development paradigm geared towards the documentation, execution and monitoring of business processes. BPM enhances the speed and quality of organizational processes in order to improve outcomes and, ultimately, deliver better business results by reducing waste and improving the value of performed operations (Radovan, 2012). It is defined as "a generic software system that is driven by explicit process designs to enact and manage operational business processes" (van der Aalst, 2013). BPM tools support the development of business processes incorporating automated services, human participants, applications and various other data sources. BPM borrows from the fields of business and information technology to orchestrate and support business processes (van der Aalst, 2004).

According to Wetzstein et.al, BPM is designed to organize, manage, analyze and reengineer an organization's business processes (Wetzstein, et al., 2007). The typical BPM life cycle consists of four stages (Mallur, 2015):

Design and model, where requirements are gathered and analyzed to produce a business process model. **Deploy and Execute**, where mapped and modelled processes are developed into executable process models that are then deployed into active use within the organization. **Monitor**, where the data generated by running business platforms is fed into a monitoring engine in order to detect and address any problems that may arise during execution. And **Analyze and Optimize**, where the results of deployed business processes

collected and analyzed in order to locate current issues or potential avenues of future process improvement.

Upon completion of the four steps, the iterative cycle begins again with the current process model being used as the starting point for the next Design and Model phase. The proposed framework aims to help primarily with the Deploy and Execute stage of the BPM lifecycle.

2.1.3 Business Process Modeling Notation

Business Process Modeling Notation (BPMN) is central to the practice of BPM in its various forms (Owen & Raj, 2003). Developed by the Business Process Management Initiative, BPMN serves as a common visual notation to bridge the communication gap between business users, business analysts and BPM development teams. The object symbols, flow chart visual style and the presence of swim lanes can convey complex and explicit information across project roles, software platforms and BPM practitioners.

The elements of the visual notation are defined as follows:

- **Events**: Events indicate the start and stop of processes as well as the intervention of other processes or external events on a process.
- **Activities**: Activity blocks represent work performed during a process. An activity block contains either instructions for an individual task or they represent another BPMN diagram inserted into a higher-level diagram.
- **Gateways**: Gateways represent logical controls to make dynamic decisions about the flow of the process diagram. Various types of logic gates allow for

explicit control over process flow behavior with regards to parallelization and requirements for processes to move forward or follow alternate paths.

- **Connectivity Objects**: Connectivity objects link other symbols in process diagrams to one another. They are used to dictate the sequence of activities, the flow of messages or associations between elements of the process diagram.

- **Swimlanes**: Swimlanes indicate the roles involved in the process diagram and the participants who fill those roles.

- **Artifacts**: These can serve as supplementary annotations to a business process diagram. They can describe data inputs/outputs or provide additional comments on elements/sections of the diagram.

Figure 2-1 depicts the BPMN visual notation (Wohed, van der Aalst, Dumas, ter Hofstede, & Russell, 2005).

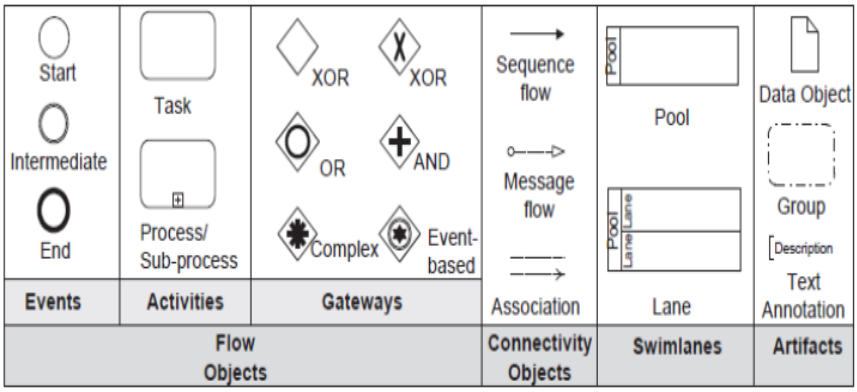

Figure 2-1: BPMN Language Elements (Wohed, van der Aalst, Dumas, ter Hofstede, & Russell, 2005).

Different combinations of these diagram elements can be grouped into general patterns that describe their collective function within a process diagram (van der Aalst, ter

Hostede, & Workflow Pattern Initiative, 1999). For the sake of brevity and to avoid digressing into another rich topic of discussion, this thesis omits an explanation of workflow patterns and the associated terminology, but these patterns can be used to assess the impact of development changes on dependent process diagrams. They could also form the basis for extending the functionality of the proposed framework upstream from the developers currently being targeted in the **Deploy and Execute** phase of the BPM lifecycle to the business analysts who plan upcoming process changes in the **Design and Model** phase.

2.1.4 **BPM Suites**

Business Process Modeling suites are software products that are intended to assist the development and execution of business process models not unlike Integrated Development Environments do for text-based coding in various computer languages. "BPM tools are software products supporting the acquisition, modeling, analysis, simulation, evaluation, and deployment of business processes" (Strobl, Mullner, & Rausch, 2009). BPM suites contain tools to assist developers with flow chart modeling, interface design for user interaction and connecting to services available within and outside the organization. They also contain a business process execution engine to run the modeled business processes based on the information encoded in the visual model as well as the technical specifics placed in the activity blocks (Reynolds, et al., April 2014).

BPM suites currently available on the market include IBM BPM, Kissflow, Nintex Promapp, Pega Infinity and Oracle Business Process Management Suite. IBM Business Process Manager 8.5.5 was used for the case work in this thesis due to familiarity of the

authors with the product. Though all BPM products are unique in some sense, the key concepts of how processes are modeled and reused within other processes generally remain present.

2.2. BPM Components Relevant to Dependency Management

BPM suites are complicated applications with many features and components. Only a limited subset of these are applicable to the dependency management focus of this thesis.

2.2.1 Process Applications

"A Process Application is the container for a solution. You can loosely think of it as a project." (Kolban, 2014) They are intended to house complete business processes. These processes are the end-product of the BPM development cycle and are used directly by users or business units to guide and monitor their activities. "Process Applications commonly contain one or more Business Process Definitions (BPDs)." (Kolban, 2014) These BPDs can also be used as sub-processes, but only within other BPDs inhabiting the same process application. This type of basic dependency relationship avoids many structural and versioning pitfalls as the most recent versions will always be present in the same snapshot.

2.2.2 Toolkits

"Similar to Process Applications, a Toolkit can also be thought of as a container for artifacts used in solutions." (Kolban, 2014) The BPDs and artifacts within a toolkit are not designed to be complete, end-to-end processes. They are pieces that have the potential to be re-used across several other projects for the sake of efficiency and effective management of the code base. In contrast to process applications, "…the contents of the Toolkit can be "included" or "used" by one or more Process Applications." (Kolban, 2014) Only Toolkit snapshots are eligible to appear in the

dependency panel of a project and their BPDs can be used as parts of processes in process applications and other toolkits.

2.2.3 **Snapshots**

At any point in time, the state of a process application or toolkit including all their artifacts, content AND dependencies can be saved as a "snapshot". (Kolban, 2014) Snapshots are the objects in BPM suites that can be deployed to run-time environments and executed. Snapshots also serve as the versioning system in BPM suites allowing code to be rolled back and versions to be managed in the run-time environment.

2.2.4 **Dependency/Toolkits Panel**

The dependency panel is where toolkits snapshots are registered as dependencies for a process app or toolkit. "To add a Toolkit as a dependency to a Process Application, the Toolkit must first have a snapshot associated with it." (Kolban, 2014) This ensures that the dependency is static and is associated to one specific version of the toolkit.

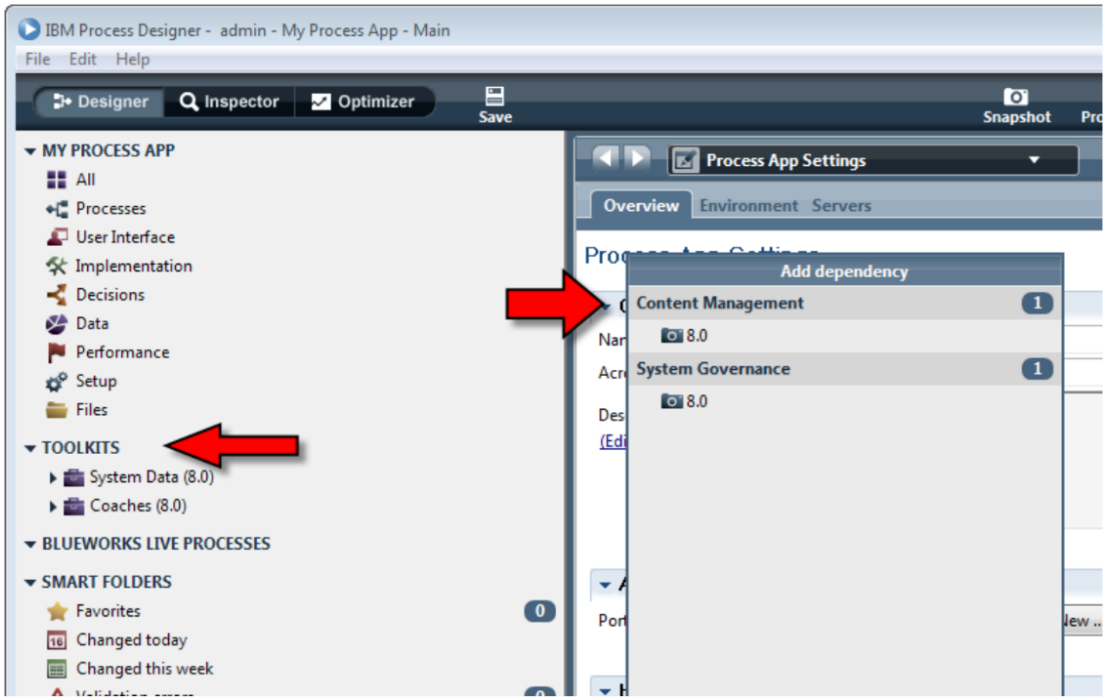

Figure 2-2: IBM BPM Toolkits Panel (Kolban, 2014)

This panel lists all the toolkit dependencies and allows for their contents to be explored. It is also the interface that allows the user to manage the versions of the project's dependencies. This functionality is essential to the topic of this thesis.

2.2.5 **Where Used lists**

The "Where Used" lists are present at the process application/toolkit level as well as the process/service/object level within projects. For projects, the where used list is tucked away as a separate tab within the snapshot information screen and offers nothing in the way of advanced features or customizable options. Similarly, the where used list for internal items does not offer any options or features and is present in the bottom portion of the designer screen where object configuration options and validation errors can also be

viewed. The “Where Used” list functionality is limited to viewing only the first level of directly dependent projects/processes and it is not exposed as an API call. (Paier, 2013)

2.3. Dependency Management & Impact Assessment

This section describes some concepts of dependency management and impact assessment that are particularly applicable to this thesis topic.

2.3.1 BPM Dependency Version Mismatch

Version mismatch is an issue that can occur within the dependencies of a BPM project when one project is present as a dependency in many other projects. It is possible for each of these projects to have a different version of the dependency resulting in differing definitions of the elements contained within. This is what is known as a version mismatch and it can result in hard stop errors for a business process at run time. Two different versions of the same service can be called depending on execution context and deployed applications may end up being much larger due to the inclusion of multiple redundant toolkit versions (Paier, 2014). These errors are a large part of the problems that the proposed dependency management framework is trying to avoid.

2.3.2 Circular Dependencies

Circular Dependencies are another problem that can occur within the dependencies of a BPM project. They occur when a project is present in the dependencies of another project while, in-turn, being dependent on that other project. This dependency structure can be formed of 2 or more projects and leads to a permanent version mismatch that can never be fully remedied as the updating of one project makes its dependency entry in other projects outdated. This perpetual mismatch poses a risk to any automated update process as there is the possibility of getting stuck in an infinite update cycle. Research has shown that the presence of circular dependencies in a dependency structure can dramatically

increase change proneness of projects surrounding the circularly dependent projects. (Oyetoyan, Falleri, Dietrich, & Jezek, 2015)

2.3.3 Change Propagation

Change propagation is the act of ensuring that a change made in a particular BPM project is pushed to all of the projects that use it as a dependency. This activity can involve several iterations as the projects that inherit the new change must then pass it on to any projects dependent on them until a point is reached where no further projects are dependent on the final recipients of the change. Throughout propagation, it is necessary to assess changes to toolkit contents and perform any required development to adapt projects to changes in dependencies. (Weidlich, Weske, Mendling, 2009)

2.3.4 Total Impact

"Total Impact" is the complete list of projects affected by the propagation of a change. It includes all of the projects impacted, all of the upgrades that must be performed in these projects and all of the snapshots that must be taken in order to complete the propagation of the change. In other words, total impact encompasses the impact of a change on other projects as well as the development effort required to perform the identified upgrades (Alam, Ahmad, Akhtar, 2014).

2.3.5 Dependency management tools

Dependency management tools are software tools that are specifically designed to assist developers in maintaining a clean and ordered list of dependencies in their projects (Seun, 2017). These tools assist with many functions and can help assess the impact of

making a change to a project dependency, alert the developer of a new available version of an outside dependency or automate the process of propagating a change to all the projects dependent on it.

2.4. Related Work

This section discusses works related to BPM development or to dependency management in software projects.

2.4.1 IBM BPM 8.5

IBM released the first version of BPM, WebSphere Lombardi Edition v7.1, in June of 2010 shortly after acquiring the TeamWorks product from Lombardi. In June of 2011, IBM renamed the product suite to Business Process Manager (BPM) starting with the release of version 7.5 (Retesh, 2015).

IBM BPM 8.5 is a full development/deployment/execution package. It includes a development interface for business process modeling, tools for testing and deployment, and a process server that functions as a business process execution engine. The testing capabilities are rather robust and afford developers the ability to intuitively step through their code to find and correct errors. The deployment features are also quite powerful and can perform highly automated project deployments with relative ease. The development environment, called process designer, is a feature rich application that offers developers numerous options and capabilities to accomplish their process modeling goals in a variety of methods. The only exception, with respect to the latter, are the tools and features geared toward dependency management and impact assessment which seem to be lacking relative to the rest of the software package.

In fact, the only features designed to assist developers with dependency management are the Dependency Panel and the Where Used lists mentioned in the previous BPM Components section (Chapter 2.2). Both of these features seem rather

limited in scope and force developers to manually deal with many aspects of dependency management, much of which could be automated or otherwise expedited. With regards to the dependency panel, the key deficiencies are the rather uninformative new-version warnings and the lack of batch processing abilities that could allow developers to efficiently make upgrades when common toolkits are updated. The where used lists suffer from a lack of customizable view options, but mostly, the problem lies in the lack of ability to view dependency usage beyond directly dependent projects. This limitation forces developers to click through many separate pages of information and manually track/compare the contents. This process is labor intensive and prone to producing errors due to inattention and complexity.

2.4.1 **Maven**

In the wider world of software development, it has long been a commonly acknowledged that "circular dependencies between programming artefacts should be avoided" (Oyetoyan et al., 2015). This long established and rarely questioned principle is known as the Acyclic Dependencies Principle (ADP). This along with the difficulties in maintaining consistency among the versions of dependencies used within a project, known as dependency mediation, are two of the primary issues associated with dependency management during the development process. These problems are only becoming more relevant as increasingly complex software applications rely more and more upon dependencies to outside code libraries. These dependencies can then, in turn, be reliant upon further dependencies. Beyond the exacerbation of dependency management issues, large number of dependencies also increases the development overhead devoted to maintenance and upgrade activities. Maven is a tool with

dependency management capabilities that can address these problems as they relate to Java programming (Jezek & Dietrich, 2014).

Maven offers several advantages when maintaining the dependencies of a software project. It has built in dependency mediation functionality for detecting version mismatches within project dependencies. This feature also extends to the detection of circular dependencies which is a variation of version mismatch that is much more difficult to solve. Not only does Maven address these two common issues, but it also offers visibility over all levels of dependencies through its use of the concept of transitive dependencies. This approach acknowledges that dependencies can, in turn, have their own dependencies. Maven offers the ability to gather dependencies from any number of levels within the dependency hierarchy (Hernandez, 2016). Removing the need for developers to manually trace the path of each dependency results in an enormous time savings and greatly reduces the likelihood of errors and oversight.

BPM, as an alternate development platform, is strongly based on the same principles of code reuse and dependency relationships. As such, it is subject to many of the same issues and eventual limitations when dependency relationships reach a certain level of complexity. However, unlike our example of the Maven application assisting Java developers with these dependency management problems, no such advanced tool is available for BPM environments. Maven is a highly relevant example of the type of assistance that the framework proposed in this thesis aims to provide to BPM developers.

2.5. Chapter Summary

This chapter has presented they key concepts, terminology and related work relevant to the subject matter of this Thesis. It has reviewed the definition of BPM and some of the early concepts leading up to its inception. It also covers the dependency management and core BPM components and how they relate to BPM development.

Chapter 3. Dependency Management and Impact Assessment Framework

This chapter will begin by discussing the gaps in current BPM suites relating to dependency management and impact assessment. Several criteria will be presented to evaluate the framework proposed in this thesis against the existing utilities present in BPM suites and their more developed analogues in traditional programing IDEs. Then, an overview of our proposed dependency management and impact assessment framework is provided. Following this overview, key aspects of the framework will be described in detail: business process components architecture, dependency data model, and upgrade algorithm.

3.1. Gap Analysis and Evaluation Criteria

BPM suites are development environments in which users develop executable business processes based on visual models built using some variant of BPMN 2.0 notation. Though some pure coding elements may be present, these BPM suites often consist, in large part, of a proprietary graphical interface through which users control the content and sequence of activities in the business processes they are designing. Along with these proprietary user interfaces, the storage of developed business processes is also accomplished through the use of a proprietary file type or database solution.

Perhaps it is due to the proprietary nature of these BPM suites, but no 3rd party dependency management solutions are available for these development environments. With no support for popular tools such as Gradle or Maven in these environments, developers are limited to the tools and functionality that come built into the BPM suite.

These features are often very basic, limited in scope and quickly become unsuitable to use as the size and complexity of BPM projects increase.

One such commonly found feature is a warning that appears beside a dependency when a new version has been released. This feature is of limited use for several reasons. There is no ability to tag a specific snapshot in order to disable this message which means that every new version of a toolkit will result in a warning seen by all developers that leverage it. Seeing warnings on every dependency quickly becomes the norm and they are no longer informative to developers. There is no analysis which compares the changes in the new version with the services being used by dependent projects. This means that presently developers must deduce the impact of upgrading to a new version of a toolkit by manually inspecting the incremental changes between the two.

BPM suites also do not offer very good tools for assessing the scope of projects that can be impacted by changes to a toolkit. Taking, IBM BPM 8.5 as an example, the only tool provided for assessing impacted projects is the “where used” panel. This panel can be found in each toolkit’s information screen in the BPM suite. The contents of this panel are a list of all the other toolkits or process apps that use this particular toolkit as a dependency. This feature only allows the developer to see projects that are directly dependent on the toolkit they are inspecting. In order to ascertain the impact that a toolkit change can have in a complex, multi-leveled dependency chain, the developer must aggregate all the projects listed as dependent in the “where used” panel. For each of these projects, they must also look at the “where used” panel and aggregate those names as well. This process must be repeated as long as there are toolkits or process apps appearing in the “where used” panels of the toolkit names that they are aggregating. This

is obviously a highly manual process and one that quickly becomes infeasible for developers as the complexity of BPM projects increases.

Both dependency management features detailed in this section are very limited in their usefulness to BPM developers. These features quickly become ineffective when dealing with project structures that feature numerous or complex dependency relationships. This is highly at odds with the BPM development paradigm which encourages the reuse of processes wherever possible as the best means of saving time and effort in development. The impact assessment and dependency management tools offered by these suites need to be improved in such a way that they better match the potential and scope of what can be developed in them. Developers need the ability to see what projects they will be impacting when they make changes to a toolkit. They must have access to information that lets them understand the dependency structure surrounding their toolkit in order to plan more efficient and more reliable dependency upgrade activities.

Although quantitative measures for business process quality have been proposed by Jan Mendling in his work *Metrics for Process Models*, these are targeted towards measuring the quality of business processes and they are not well suited to measure time or effort involved in conducting dependency upgrade operations (Mendling, 2008). Also, the ability to obtain other quantitative measures such as user interaction time and run time were precluded by the lack of a framework implementation. Instead this thesis opts to use eight qualitative measures. Two of these, Visual and Low Learning Curve, are inspired by key characteristics of BPM cited ubiquitously in literature (Strobl, Mullner, & Rausch, 2009). The next two, Accurate Dependency Analysis and Recursive Dependency discovery are based on Maven and the literature about it that was reviewed (Hernandez,

2016). The remaining four criteria are based on the author's nearly six years of industry experience in BPM development including participation in multiple projects that received industry recognition in the fields of healthcare and telecommunications.

The criteria used to evaluate the framework proposed in this thesis will be the following:

3.1.1 **Visual**

Despite the graphical nature of the process diagrams present in most BPM suites, dependency relationship information is usually conveyed by means of lists and version identifiers. Introducing visual elements to the dependency management aspects of BPM is a natural extension of the graphical coding style used in development. A visual representation of the dependency relationships between projects presents several opportunities for BPM developers such as increased comprehension and more visible violations of design best practices. A visual interface for the proposed framework would also provide a more intuitive experience for controlling its operation with regard to manually selecting/deselecting dependencies for upgrade.

Since traditional coding and BPM both tend to represent project dependencies as lists, this criterion will be considered successfully met if the BPM dependency management experience can be enhanced with a graphical interface or diagram.

3.1.2 **Low Learning Curve**

As with any development assistance tool, a low learning curve is preferable as it allows developers to start leveraging the tool more quickly. In order to do this, a tool

must avoid introducing new concepts, unfamiliar GUI controls or undue complexity in its functionality. The framework must allow developers to navigate through the core functionality in an easy, logical manner. The controls and GUI elements used should make it clear what inputs are required of the user at each step.

Success on this criterion requires that a developer be able to operate the dependency framework with minimal knowledge beyond what is needed to work with the BPM suite itself. The vocabulary and the symbols used in the prompts and menus should be consistent with BPM suites and development standards. Any new functionality provided by the framework should provide ample instruction and feel like a natural extension of the existing functionality that can be accessed in the BPM suite. Current developers must be able to use the tool proficiently with minimal practice time.

3.1.3 **Accurate Dependency Analysis**

The framework will need to produce accurate analyses. This means that it must be able to accurately capture the structure of dependent projects based on an identified snapshot containing changes and BPM project details gathered through either direct database access or through API calls to the BPM server. Using this information, the framework must be able to follow dependency relationships from project to project up the hierarchy. The analysis must show all of the projects potentially impacted by the change and only those projects. In other words, projects not related to the change must not appear in the analysis results.

The data gathered from the BPM environment must be sufficient to plan upcoming upgrade activities with respect to the order of projects to upgrade, upgrades

that can be parallelized for greater efficiency and whether developer intervention is required between upgrading and snapshotting activities.

3.1.4 Recursive Dependency Discovery

First of all, the framework must be able to recursively discover projects impacted by a change. During the course of a single execution, the framework must identify all impacted projects regardless of how many degrees of separation (other projects) there are in the chain of dependencies that links them to the project containing the change. Given that visibility over only directly dependent projects is standard for IBM 8.5 and other BPM suites, this criterion will be considered improved upon if the framework is able to produce an output that describes two or more subsequent relationships. To be considered a success, the framework must be able to traverse and report on a dependency chain of indefinite length or long enough that it surpasses the length of any dependency chain likely to be encountered in a BPM development environment.

3.1.5 Robust Upgrade Algorithm

In order for the framework to produce some practical benefit, it will need to generate insights from the data gathered during the dependency analysis. This will be accomplished by evaluating the data with a series of logical operations collectively named the upgrade algorithm. From the data, the algorithm will organize the necessary upgrades into the optimal order and grouping while also providing some level of automation where possible.

To consider this criterion met, the framework will need to provide developers with a structured, semi-automated upgrade process. This upgrade process must be robust and

should successfully handle all the dependency relationship patterns that a human developer can handle. The algorithm must also identify the appropriate times to pause when developer intervention is required to accommodate the changes in the upgrade.

3.1.6 Development Effort

Ultimately, for such a system to be useful to BPM development teams, it should reduce the effort involved in maintaining dependency relationships and speed up the time it takes to complete upgrade activities. The framework proposes to accomplish this by providing superior guidance to developers during the upgrade process in order to avoid mistakes which can incur large amounts of effort to correct. In addition, developer effort can further be reduced by replacing manual tasks with automation of the dependency upgrade and snapshot activities, where possible.

To meet this criterion, the total effort involved in completing upgrade operations using the proposed framework should be less than using the native functionality available in the BPM suite. The expectation is that, using this framework, one developer would able to monitor and complete a large upgrade cycle that may otherwise have taken the efforts of a team of individuals.

3.1.7 Setup and Maintenance Costs

Although a BPM dependency tool might provide benefits such as reducing developer effort and improving the quality of business processes, no organization will choose to implement it if the setup and maintenance costs of the solution outweigh the savings it can provide. Necessary hardware, software installation and configuration all contribute to the setup cost of a dependency management solution. Continued activities

such a framework updates and data cleanup constitute the ongoing maintenance cost of employing such a solution. Purchase cost will not be considered when evaluating this criterion as this amount is determined in function of the value provided by a given solution and does not represent mandatory cost for running the software.

Success on this criterion requires that the setup and maintenance costs of the framework be lower than the value of the benefits it provides. A promising solution will have negligible setup costs in relation to the cost of the BPM suite and the servers on which it runs. With regard to ongoing costs, any additional hours spent on operating and maintaining the tool must be offset by a similar or greater savings in BPM development hours.

3.1.8 **Standardization**

The final criterion for evaluating this thesis will be standardization across BPM suites. Considering that BPM suites often employ the same development paradigms despite being disparate products, any proposed improvement to the functionality of one BPM suite can likely be applied to other BPM suites. The proposed framework should display consideration for the variety of BPM suites currently in use in development shops. The objects, data and logic used in the core functionality should be abstracted to a sufficient extent to allow the framework to support a variety of BPM suites. The framework architecture should identify what platform specific components are necessary and compartmentalize them into a layer that operates between the abstracted framework logic and the BPM suite in question.

The components of the framework must feature design elements that would allow it to be easily adapted for use with new and different BPM suites. The functionality of the tool should remain consistent from the user's perspective while implementing the necessary back end changes to maintain compatibility.

3.2. Framework Overview

The Dependency Management and Impact Assessment framework is depicted in Figure 3-1. This diagram shows the human and technological actors responsible for providing the input data, processing it into useful information and using this information to manage the process of propagating changes across entire BPM projects.

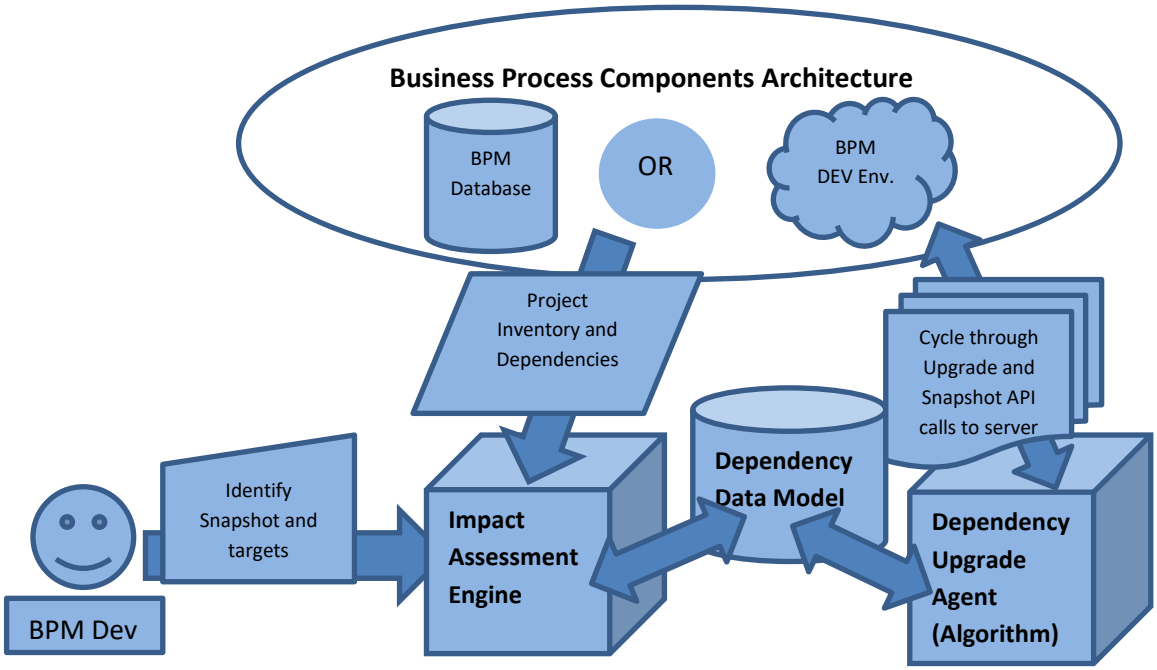

Figure 3-1: Framework for Dependency Management and Impact Assessment

BPM Dev represents the human element of the framework. BPM Devs are responsible for identifying the snapshot containing the change that needs to be propagated using the **Impact Assessment Engine's** initial interface. They are also responsible for reviewing the **Impact Assessment Engine**'s output and specifying the scope of the new snapshot's propagation within the impacted projects. Finally, they are required to monitor

the **Dependency Upgrade Agent**'s progress and respond to any exceptions that may arise during the agent's execution of the upgrade algorithm.

The **BPM Database** is the database that stores the code and models of the projects developed within the BPM platform. The tables that store snapshots and project dependencies will provide the data used by the **Impact Assessment Engine** to assess the impact of the upgrade as well as to fill the **Dependency Data Model** required by the **Dependency Upgrade Agent**. Directly querying the database may not be necessary if the **BPM Dev Env** provides an API that can efficiently supply us with the same information.

The **BPM Dev Env** is the application or web-based interface through which BPM projects are designed and programed. In order for this framework to be possible, the **BPM Dev Env** must provide an API that allows for various development functions to be performed programmatically. The necessary functions are to retrieve an inventory of projects, to retrieve the list of dependencies of a project, to display the process model contents of a project, to upgrade a dependency within a project and to create a snapshot of a project. With these functions present, it is not necessary for the **Impact Assessment Engine** to query the **BPM Database** directly. However, if the **Impact Assessment Engine** can access the database directly, then only the snapshot and dependency upgrade functions need to be programmatically accessible.

The **Impact Assessment Engine** queries the **BPM Dev Env** or the **BPM Database** and first obtains an inventory of BPM projects and their snapshots. Then, based on the snapshot(s) identified by the developer, the **Impact Assessment Engine** queries the data source to obtain a hierarchical tree of projects impacted by the snapshot(s). The developer

can then indicate which portion of the potentially impacted projects need to make the upgrade. Finally, the **Impact Assessment Engine** updates the dependency data model with the scope of the upgrade and passes it along to the **Dependency Upgrade Agent**.

The **Dependency Upgrade Agent** is responsible for managing and reporting on the automated process of upgrading dependencies down through the tree of impacted projects. The **Dependency Upgrade Agent** upgrades projects with the new snapshot and then solidifies this change by snapshotting the project and upgrading the next project down the hierarchical tree of dependencies stored in the data model. The **Dependency Upgrade Agent** listens for the responses to its API calls in order to maintain the sequence of events necessary to successfully complete all of the upgrades as well as to notify the user if an exception or error is returned in the response. The **Dependency Upgrade Agent** could conceivably benefit from features like automated changes packaged along with upgrades and a rollback feature if critical errors are encountered.

3.2.1 User Interaction

This framework is designed to be used as a development aid and time saving measure for any impact assessment and dependency upgrade activities. As such, the functionality of this framework is designed to be user initiated and, for the purposes of this discussion, no self start functionality or automatically triggered update process are anticipated. BPM developers can use this framework to evaluate the impact of their changes to projects before deployment time as well as to enhance the speed and reliability of their upgrade and snapshot activities while preparing the deployment.

When a BPM developer starts the framework, they will be presented with a simple list interface containing project snapshots from the **Business Process Component Architecture** present on the BPM server. The **Impact Assessment Engine** module will make a series of queries to the **BPM database** or calls to the **BPM Dev Env** API to populate this list. The user must then choose a snapshot to assess at which point the **Impact Assessment Engine** will send an iterative series of queries to the BPM system to determine the extent of the impacted projects and the dependency structure between them. This information is filled into the **Dependency Data Model**. The structural and error related information from the Dependency Data Model are then displayed for the user to review.

After correcting any errors in the project dependencies, the user can choose to adjust the scope of the automated dependency upgrade operation. The user's selections are translated into a series of tags in the **Dependency Data Model** which encode information about which projects must be updated, but also the order in which the upgrade operations must occur to be successful. This structured information is then sent along by the **Impact Assessment Engine** to the **Dependency Upgrade Agent**.

Once received by the **Dependency Upgrade Agent**, the populated dependency data model tags are analyzed to produce an ordered series of upgrade and snapshot commands which correspond to the structure and annotations in the model. The **Dependency Upgrade Agent** will attempt, where possible, to include parallelization of these operations when it can speed up completion of the task and is within the capabilities of the BPM server. This ordered list of API calls to the BPM server is performed by the **Dependency Upgrade Agent** which uses the responses from these calls to ensure that the order of sequential operations is strictly enforced when this is necessary. The **Dependency Upgrade Agent**

must also be able to handle scenarios when it receives errors or exceptions to some of its calls. It will accomplish this by requesting user intervention although it may be possible to offer some level of automated response ability to handle and report less critical "warning" messages when encountered. When the **Dependency Upgrade Agent** has completed its operations, a status report and a summary of the actions taken is presented to the user for their records.

3.2.2 Abstract Communication Module

The **Impact Assessment Engine** and the **Dependency Upgrade Agent** are two distinct pieces of functionality operating on a shared **Dependency Data Model**. Both of these modules require communication with the **BPM database** and/or **BPM Dev Env**. To facilitate the adaptation of the framework to a variety of BPM suites, the framework will need a communications module (as shown in Figure 3-2) that abstracts the logical dependency management operations from the specific API calls provided by a particular BPM server. This module will be capable of interfacing with the **BPM Dev Env**'s API in order to collect data about the projects residing on the server as well as issuing commands to upgrade and snapshot them. Splitting this part of the code off into its own module is very practical from a technical perspective as it allows the **Impact Assessment Engine** and **Dependency Upgrade Agent** to remain unchanged regardless of the BPM suite being used. However, it would still be necessary to create a version of the communications module for each different BPM server API specification.

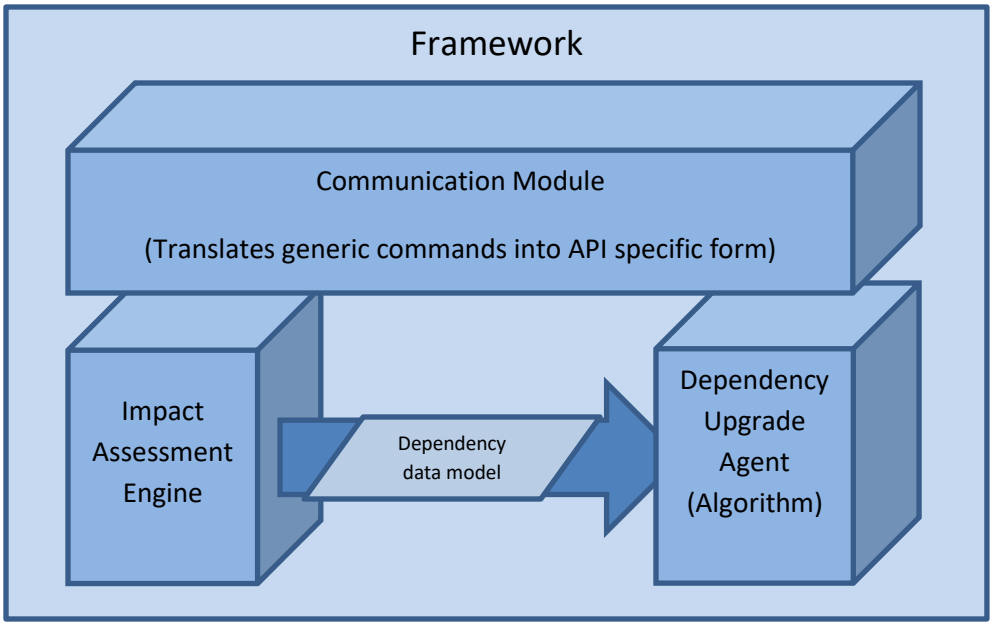

Figure 3-2: Communication Model for Abstracting Specific BPM APIs

The **Communication Module** acts as a translation layer for all commands to and from components of the BPM system in question. As such, its presence and function will be assumed during much of the following discussion about the framework architecture. When there is mention of any interaction between the framework and any outside systems, the communication layer is assumed to have handled the translation of those messages to the format used by the BPM suite.

3.3. Business Process Component Architecture

BPM platforms are used to develop executable process diagrams that are used to guide employees in completing their tasks and collaborative work. These processes help ensure that work is completed in a standardized and coordinated fashion by employees. To understand how processes use and are affected by dependencies, it is essential to discuss the components that constitute these processes and how they can be assembled.

Processes in BPM suites are most often visually represented as flowcharts. Generally speaking, process diagrams are composed of executable elements which appear as blocks in the flow chart and directional elements which take the form of arrows and diamond-shaped decision gateways. These three elements are present from the most low-level diagram to the largest, most over-arching process diagrams. However, only one of these three elements are subject to the fundamental BPM concept of component reuse and those are the executable "blocks" within the diagrams. These components can contain scripts, web service calls, database calls, decision tables and other business logic, but they can also contain references to existing process diagrams. The variables defined within the referenced process become the inputs and outputs of the block and the process elements contained within determine how this data is manipulated. In this way, the logic of an entire process can be reused within another process to minimize the duplication of effort that would result from having to redevelop multiple processes to accomplish similar things.

Figure 3-3 provides a visual example of the concept of component reuse. It also shows how these components can reside in separate projects stored as distinct entities in the

BPM database.

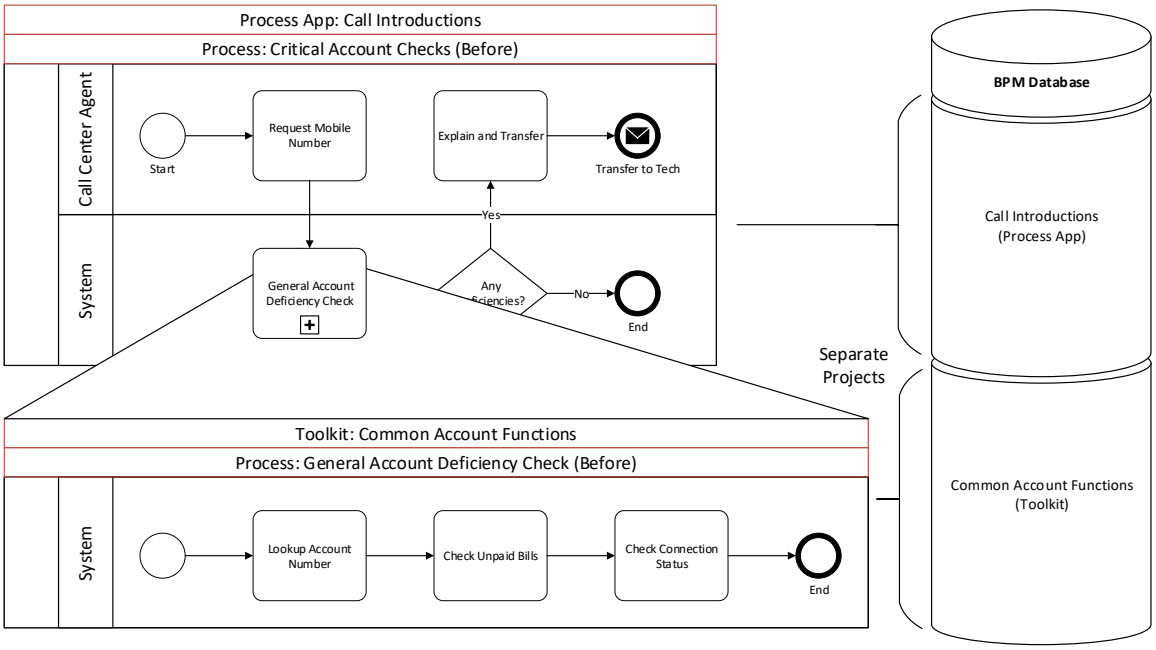

Figure 3-3 Example Business Components

This core concept of reusable components can be applied to processes within the same project where they reside in a common space and can be updated simultaneously in order to ensure compatibility. However, this is not always the case. Sometimes, processes are so generally useful that it is desirable to share them among many projects in order to allow access to anyone who may need them during development. These processes are placed within "toolkits". The only distinction between standard process apps and toolkits is that toolkits can be referenced by other projects in order for them to use the processes and data structures contained within.

In order to leverage the content of a toolkit, it must be added as a dependency to the process app/toolkit where it is intended to be used. As changes are made to a toolkit, its dependent projects will need to upgrade their dependency in order to benefit from the

new content. When doing so, changes within toolkit processes may require adjustments around the blocks where these processes are referenced. In this way, a change to a toolkit can affect many other toolkits and process apps that have registered it as a dependency. To compound the problem, each of the affected projects can then, in turn, affect all of the projects dependent on it. This list of affected projects can quickly balloon to a size that makes it difficult to organize deployments and maintain clean project dependencies.

Figure 3-4 is a dependency tree diagram that represents the dependency relationship structure between a collection of projects. The dashed lines denote the dependency tree starting from the process app in question and moving through the toolkits on which it is dependent. The impact of the change to the Technical Services toolkits is identified by the solid arrows ("Inherits change") traveling in the opposite direction as the dependency relationship arrows.

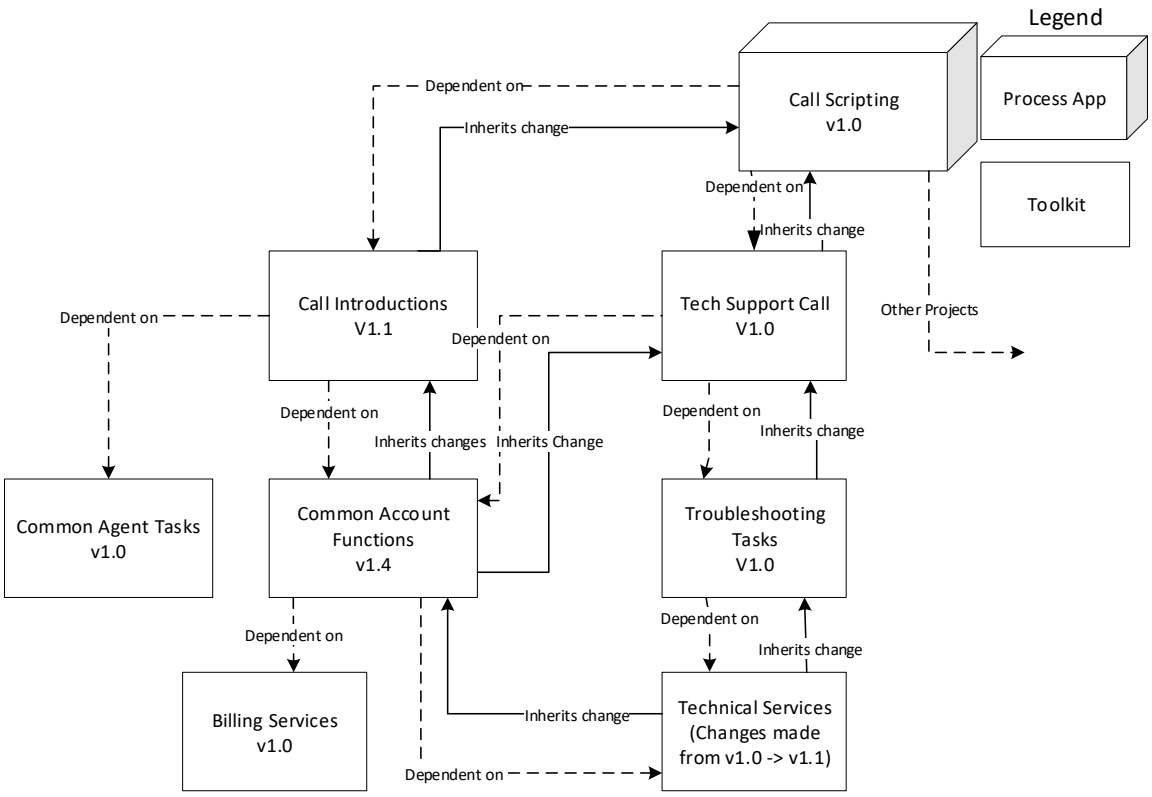

Figure 3-4: Example of project hierarchy impacted by change

For this example, the top level of this diagram is occupied by the Call Scripting process app. From what is depicted in the diagram, we can see that Call Scripting has two 1st degree dependencies: The Call Introductions and Tech Support Call toolkits. In turn, Call Introductions is dependent on both Common Agent Tasks and Common Account Functions toolkits while Tech Support Call is dependent on Common Account Functions and Troubleshooting Tasks toolkits. These 3 distinct toolkits form the 2nd degree dependencies of the Call Scripting process app. Going down to the final level, we can see that the Common Agent Tasks toolkit has no dependencies, Common Account Functions is dependent on Billing Services and Technical services toolkits, and Troubleshooting Tasks is dependent only on the Technical Services toolkit. These 2 unique toolkits constitute the 3rd degree dependencies of the Call Scripting process app.

The adjustments that must be made to accommodate any particular sub-process change are dependent on the type of the change. For example, if a new input variable is added to a process in a toolkit, then any other projects leveraging that process will need to specify the value of that input in the relevant process diagrams. Because the type of fix required is often closely dictated by the type of change in the sub-process, it is possible to provide developers a list of recommended general fixes based of what has changed within the sub-process.

In order to provide such a recommendation functionality to developers, the framework needs the ability to compare the previous and new version of the toolkit against one another and produce a manifest detailing the change differential between them. Beyond this list of changes, the framework will also need a way of translating these changes into recommendations that can help guide a developer. This will require some type of change dictionary that maps categories of changes to their respective recommendations. This dictionary can be implemented through a database lookup table and would be matched to all of the changes brought forth by the comparison.

#	Change	Recommendation
1	Added endpoint to diagram	Add exit line from sub-process block
2	Added input variable to process	Set input value in sub-process block.
3	Added output variable to process	Map output value in sub-process block.
…	…	…

Figure 3-5: Sample of Change Dictionary

3.4. Dependency Data Model

The dependency data model is a data structure definition that facilitates the description of the dependency relationships between BPM projects. At a high level, this model is comprised of two main elements. The first is a list of fields necessary to accurately describe a BPM project and to store the related meta data that is generated by the assessment portion of this framework. The second part is a structure to link multiple sets of this information together in a way that is representative of the dependency relationships between projects and that will facilitate the traversal of the data for the purposes of both impact assessment as well as the automated upgrade activity.

Let us start by defining the list of data points that will be stored for a BPM project within the dependency data model table shown in Figure 3-6.

Fields	Sample Value	Notes
Upgrades[n]	n/a	Repeating data structure to store all upgrade snapshots.
Upgrades[n].ProjectName	Common Account Functions	1:1 ratio per upgrade. User friendly identifier.
Upgrades[n].ProjectID	CACCFUN	1:1 ratio per upgrade. Format may vary by BPM platform.
Upgrades[n].Name	Common Account Func v1.4	1:1 ratio per upgrade. User friendly identifier.
Upgrades[n].ID	a88fb9cc-06a5-4e0f-bc25-871bd1e8d5d0	1:1 ratio per upgrade. Format may vary by BPM platform.
Upgrades[n].Tier	2	Matches the Tier of projects impacted by change.
Upgrades[n].Path	TECSERV-CACCFUN	Displays path of token through dependency discovery.
Project[n]	n/a	Repeating data structure to store all BPM projects.
Project[n].Name	Bill Information Web Services	1:1 ratio per project. User friendly identifier.

Project[n].ID	BILLWS	1:1 ratio per project. Format may vary by BPM platform.
Project[n].Tier	3	Only for projects directly or indirectly dependent on new version being assessed.
Project[n].Build[n]	n/a	Repeating data structure to store versions/snapshots
[…].Build[n].Name	Bill Information WS v2.28	1:1 ratio per snapshot. User friendly identifier.
[…].Build[n].ID	35ef9955-7caf-47d8-81ed-ddb8367cab5f	1:1 ratio per snapshot. Format may vary by BPM platform.
[…].Build[n].Services[n]	n/a	Repeating data structure to store build services
[…].Services[n].Name	Bill Lookup	1:1 ratio per Service. Service name.
[…].Services[n].Content	*Service Contents	1:1 ratio per Service. Definitions of service contents. Code, nested services, etc…
[…].Build[n].Dep[n]	n/a	Repeating data structure to store dependencies
[…].Dep[n].ProjectName	Customer DB Integration	1:1 ratio per dependency. Readable project identifier.
[…].Dep[n].ProjectID	CSDBINT	1:1 ratio per dependency. Unique project identifier.
[…].Dep[n].Name	Customer DB v1.44	1:1 ratio per dependency. Readable version identifier.
[…].Dep[n].ID	0e6d5d01-5971-4e63-9f33-7dae3a3b4fe8	1:1 ratio per dependency. Unique version identifier.
[…].Dep[n].Changes[n]	n/a	Repeating data structure to store change log.
[…].Changes[n].Process	General Account Deficiency Check	Process in which the change is situated.
[…].Changes[n].Change	Added end point “To Billing” to diagram.	Description of the change.
[…].Changes[n].Rec	Add “To Billing” exit line from sub-process block.	Recommendation to accommodate change if any are needed.
[…].Changes[n].Places	Critical Account Checks	All of the processes that must be adjusted for the change.

Figure 3-6: Dependency Data Model Table

In order to completely traverse the dependency tree, the completed data structures for every project will be required. If any BPM projects are missing from this list, it may result in process applications or toolkits being mistakenly excluded from the list of impacted projects. All of the project data structures will be stored in a simple array as denoted by the "[n]" following the "Project" label in figure 3-6. The data points captured about each project constitute only a part of the information needed for assessing the impact of introducing a new version of a BPM project. The structure of the dependencies between projects is another set of information that is key to our assessment and automatic upgrade activities.

This structure is partly contained within the information captured about each project. The "Project[n].Build[n].Dep[n]" section of the Project data structure provides a list of all of the projects which the current project is dependent on. By comparing the "Project[n].Build[n].Dep[n].ID" field of a project to the "Project[n].Build[n].ID" values in other projects, we are able to identify the projects associated to all of our dependencies. This process of identifying the dependencies can be repeated until the bottom most level of dependencies is reached. However, the proposed framework requires the opposite search paradigm. Rather than travelling down the chain or tree of dependencies towards lower level, shared projects, it becomes necessary to start at the project version containing new changes and work upwards through the dependency tree by identifying the first set of projects which are directly dependent on our new version. The framework does this by comparing the "Project[n].ID" of our new version to the "Project[n].Build[n].Dep[n].ProjectID" values in other projects and storing the relevant Project[n] of matches into a new Upgrades[n] list entry. It should be noted that ProjectID

is used for searching purposes instead of the ID of the snapshot. This is necessary as we are trying to identify projects that must be upgraded to a new snapshot version. These projects would not yet be dependent on the new snapshot, so searching for that snapshot ID would yield no results.

Once this process has been completed for the directly dependent projects, the **Impact Assessment Engine** will repeat the search process for each dependent toolkit that it has found. This will allow it to retrieve a list of all projects impacted by the first round of changes to the directly dependent projects. These will then form the 2nd tier of upgrade activities. This process can be repeated add infinitum until we reach a level where only process apps are being impacted by changes. Since process apps cannot be leveraged as dependencies by other projects, changes made to them will not impact any projects further down the line. The results from each cycle of searching are then organized into upgrade tiers corresponding both to their order of discovery and to their sequence during the upgrade process (direct, 2nd tier, 3rd tier, etc…)

The resulting data will describe an inverse dependency tree with our new version as the lone starting point at the bottom and the projects dependent on it organized into higher levels and ordered by degrees of separation from our new version. This structure represents the projects impacted by our new version. Furthermore, the levels in the tree describe the process that must be followed to fully propagate the new version of the BPM project. Directly dependent projects must be considered first as they, in turn, may impact other toolkits. The lowest level must be done before the second and the second before the third, and so on. However, within each level, the projects can be upgraded in any order and can even be done in a parallel fashion to decrease the total time spent upgrading. Every

time a particular process application or toolkit is encountered the “Project[n].Tier” value is overwritten with the current iteration of dependency discovery cycle and a new “Upgrades[n]” entry is created storing that same value. Once the assessment has been completed, the “Project[n].Tier” field represents the highest dependency level or tier on which a particular project is situated. This value is copied over whatever values are currently present in the “Upgrades[n].Tier” fields where the value in “Upgrades[n].ProjectID” matches “Project[n].ID”. This ensures all dependencies are available when the **Dependency Upgrade Agent** reads the values in Upgrades[n] list to determine the sequence in which the upgrades are performed.

The **Dependency Data Model** can also assist in early detection of unresolvable dependency relationship structures. The Upgrades[n].Path data is designed to store a record of discovered toolkits and can be used to detect circular dependencies. Starting from the snapshot(s) containing the change(s), Upgrades[n].Path will be appended with the ID of the projects encountered along that upgrade path. Every entry in Upgrades[n] will therefore be tagged with all of the project IDs that connect the new changes with the latest project dependencies to be discovered in the chain. With this information, it then becomes a simple matter to check for circular dependencies when discovering projects by searching the Upgrades[n].Path string for matches to project ID.

The final set of information required to complete the upgrade activities is the comparative analysis between the old and new versions of processes in the dependencies being upgraded. Any of the dependency’s processes that are used by dependent projects, must be compared against the previously used version in order to produce a comprehensive list of changes. Using a change dictionary, these changes are translated into development

recommendations and stored in the "Project[n].Build[n].Dep[n].Changes[n]" list. Each list item details the process that changed, the nature of the change, a recommendation for accommodating the change and a list of processes where these accommodations need to be made. When a toolkit service is not used by a project that is leveraging that toolkit, then its changes will be ignored at this stage. The results of the **Impact Assessment Engine** will list every service that is utilized by any dependent project AND has changes since the previous dependency version.

The **Impact Assessment Engine** only runs this comparative algorithm for the directly dependent projects during impact assessment. It is not possible to perform this comparative analysis beyond the direct dependencies as these have not yet received the changes from their upgrade activities. For this reason, the **Impact Assessment Engine** must perform the comparison for the next tier of upgrades only after the development team has finished the previous tier. This implies a back and forth flow of information between the **Impact Assessment Engine** and the **Dependency Upgrade Agent** after the first round of upgrades has been performed.

The information in these "Changes[n]" lists is aggregated for each project and displayed for developers during the **Dependency Upgrade Agent**'s activities. This information is used by the **Dependency Upgrade Agent** to determine whether a particular dependency upgrade is so low impact that it can completely automated or if the changes within the new version are such that the **Dependency Upgrade Agent** must pause before snapshotting to allow developers time to make the necessary adjustments to business processes in order to accommodate the changes introduced by the new versions of the dependencies.

3.5. Upgrade Algorithm

Very little information has been given so far about the process that the **Dependency Upgrade Agent** uses to take the model's information and translate it into a series of upgrade activities. This translation from data model to a concrete list of upgrade activities is handled by the upgrade algorithm. The list of upgrade activities generated by the **Dependency Upgrade Agent** will be generic and are not specific to any particular BPM platform. The generic activities executed by the **Dependency Upgrade Agent** pass through the communication module which was previously mentioned in 3.2.2 Abstract Communication Module and in Figure 3-2. The communication module takes the generic commands understood by the **Dependency Upgrade Agent** and converts them to the form used by a specific, supported BPM platform.

At the individual project level, the upgrade algorithm is simply a short set of commands necessary to upgrade a dependency within a project and create a new version of this project. The upgrade algorithm looks at the "Project[n].Tier" and "Upgrades[n].Tier" fields in the dependency data model and identifies the project with a Tier value of 0. The project with this value is at the top of the inverse dependency tree and will equate to the BPM project version initially chosen by the user in the **Impact Assessment Engine**. As this version contains the changes to be propagated throughout our BPM environment, it is not necessary to cut a new version of this project. Instead, the algorithm will match the "Project[n].Build[tip].Dep[n].ProjectID" of Upgrades[1] entries to the value in "Upgrades[0].ProjectID". It then compares the corresponding "Project[n].Build[tip].Dep[n].ID" from "Upgrades[1]" to the value of "Upgrades[0].ID". If this project contains a version of the dependency other than the one that we initially

identified, then the algorithm instructs the **Dependency Upgrade Agent** to perform an upgrade action on this project dependency.

Once the identified upgrades have been completed, the next step in the algorithm is to compare the old and new versions of the dependencies and produce a list of differences between them. This list of differences represents the changes that have been made to the project since the earlier version. These changes are filtered to only display those that are being used in the current project. The changes are then compared to a dictionary of change types in order to produce a list of development recommendations. These recommendations can help guide developers in making the necessary changes to the project they are upgrading in order to accommodate the changes in the upgraded dependency. Once all of the recommendations have been implemented, the developers signal to the framework that it is time to proceed. The framework then snapshots a new version of the BPM project that was just upgraded.

For projects with larger dependency trees, it is entirely possible for BPM projects that were dependent on our new version to, themselves, be dependencies for other BPM projects. The value of "Project[n].Tier" can be interpreted as the number of dependency relationships through which two projects are linked. The upgrade algorithm instructs the **Dependency Upgrade Agent** to upgrade all of the projects with "Project[n].Tier" values of 1, then, after saving new versions for all of these projects, it will begin to look inside projects with a Tier value of 2 in order to identify any dependencies that match with Tier 1 or 0. The algorithm is designed to be applied in a cyclical fashion until it reaches projects that are not included as dependencies to other projects. These will be referred to as final tier projects.

In the cases where there are many tiers of projects identified for upgrade, the change analysis will have to be performed at every tier. This analysis cannot all be done up front as the upgrade activities in the previous tier will affect the next tier and must be completed before being analysed. For this reason, there will often be a manual step between the automated steps of upgrading the dependencies and taking the snapshot.

In the rare cases where a fatal error, such as a circular dependency, has been detected by the **Impact Assessment Engine**, the results of the assessment will still be displayed for the BPM developer, the projects involved in the circular dependency will be identified and the **Dependency Upgrade Agent** portion of the algorithm will be blocked from proceeding with any dependency upgrade or snapshot activities. This is a necessary precautionary measure that must occur before upgrades are started in order to avoid unnecessary rework or version rollback as it is impossible to successfully complete upgrade operations that involve projects in a circular dependency. If for any other reason it becomes necessary to rollback project versions to their previous state before the dependency management framework was used to upgrade them, this responsibility falls to the developers as this is outside the scope of what was considered during the design of the framework. However, this should not be much of an issue as the snapshot versioning system present in IBM BPM 8.5 allows the new snapshot to be disabled and the old one to be easily reactivated in its place using the BPM suite's deployment functions.

Figure 3-7 displays the dependency upgrade algorithm to propagate change through the dependency structure that links projects. The activities falling under the **Impact Assessment Engine** and the **Dependency Upgrade Agent** are identified by the brackets on the right. The algorithm is expressed using BPMN style notation. The rectangle blocks

indicate activities performed either by the framework or by the developers and the diamond shaped blocks represent decision logic that guides the execution of the algorithm. The exiting lines are labeled with the results of the decision logic contained within each diamond shaped decision block. Rectangle activity blocks annotated with a small person icon identify activities performed by developers. In the top portion of the algorithm labeled **IAE**, the algorithm describes the logic used to search for projects impacted by a new version of a dependency. The framework queries the **BPM Dev Env** to obtain a complete view of the projects currently residing on the BPM server. **BPM Devs** then indicate which snapshots are to be upgraded within their dependent projects. The framework then searches for dependencies matching the identified snapshots and stores their information. Decision logic then examines the results and decides whether to continue searching for further tiers of dependencies or to move on to the dependency upgrade portion of the algorithm labeled **DUA**. **BPM Devs** then review the complete list of dependent projects and deselect any nodes of the dependency hierarchy tree to specify projects that are not to be upgraded. The diagram then describes iterating through and performing the list of upgrades. Changes from the cumulative upgrades are assessed and the framework is paused to give **BPM Devs** an opportunity to implement the recommended changes. The framework then snapshots all upgraded projects and decides whether any further upgrade activities are required based on the contents of the Upgrades[n] data structure. If no further projects require dependency upgrades, the framework concludes execution of the upgrade algorithm.

Fields	Sample Value	Notes
Upgrades[n]	n/a	Repeating data structure to store all upgrade snapshots.
Upgrades[n].ProjectName	Common Account Functions	1:1 ratio per upgrade. User friendly identifier.
Upgrades[n].ProjectID	CACCFUN	1:1 ratio per upgrade. Format may vary by BPM platform.
Upgrades[n].Name	Common Account Functions v1.4	1:1 ratio per upgrade. User friendly identifier.
Upgrades[n].ID	a88fb9cc-06a5-4e0f-bc25-871bd1e8d5d0	1:1 ratio per upgrade. Format may vary by BPM platform.
Upgrades[n].Tier	2	Matches the Tier of projects impacted by change.
Upgrades[n].Path	TECSERV-CACCFUN	Displays path of token through dependency discovery.
Project[n]	n/a	Repeating data structure to store all BPM projects.
Project[n].Name	Bill Information Web Services	1:1 ratio per project. User friendly identifier.
Project[n].ID	BILLWS	1:1 ratio per project. Format may vary by BPM platform.
Project[n].Tier	3	Only for projects directly or indirectly dependent on new version being assessed.
Project[n].Build[n]	n/a	Repeating data structure to store versions/snapshots
[…].Build[n].Name	Bill Information WS v2.28	1:1 ratio per snapshot. User friendly identifier.
[…].Build[n].ID	35ef9955-7caf-47d8-81ed-ddb8367cab5f	1:1 ratio per snapshot. Format may vary by BPM platform.
[…].Build[n].Services[n]	n/a	Repeating data structure to store build services
[…].Services[n].Name	Bill Lookup	1:1 ratio per service. Service name.
[…].Services[n].Contents	*Service Contents	1:1 ratio per service. Definitions of service contents. Code, nested services, etc…
[…].Build[n].Dep[n]	n/a	Repeating data structure to store dependencies
[…].Dep[n].ProjectName	Customer DB Integration	1:1 ratio per dependency. Readable project identifier.
[…].Dep[n].ProjectID	CSDBINT	1:1 ratio per dependency. Unique project identifier.
[…].Dep[n].Name	Customer DB v1.44	1:1 ratio per dependency. Readable version identifier.
[…].Dep[n].ID	0e6d5d01-5971-4e63-9f33-7dae3a3b4fe8	1:1 ratio per dependency. Unique version identifier.
[…]Dep[n].Changes[n]	n/a	Repeating data structure to store change log.
[…].Changes[n].Process	General Account Deficiency Check	Process in which the change is situated.
[…].Changes[n].Change	Added end point "To Billing" to diagram.	Description of the change.
[…].Changes[n].Rec	Add "To Billing" exit line from sub-process block.	Recommendation to accommodate change if any are needed.
[…].Changes[n].Places	Critical Account Checks	All of the processes that must be adjusted for the change.

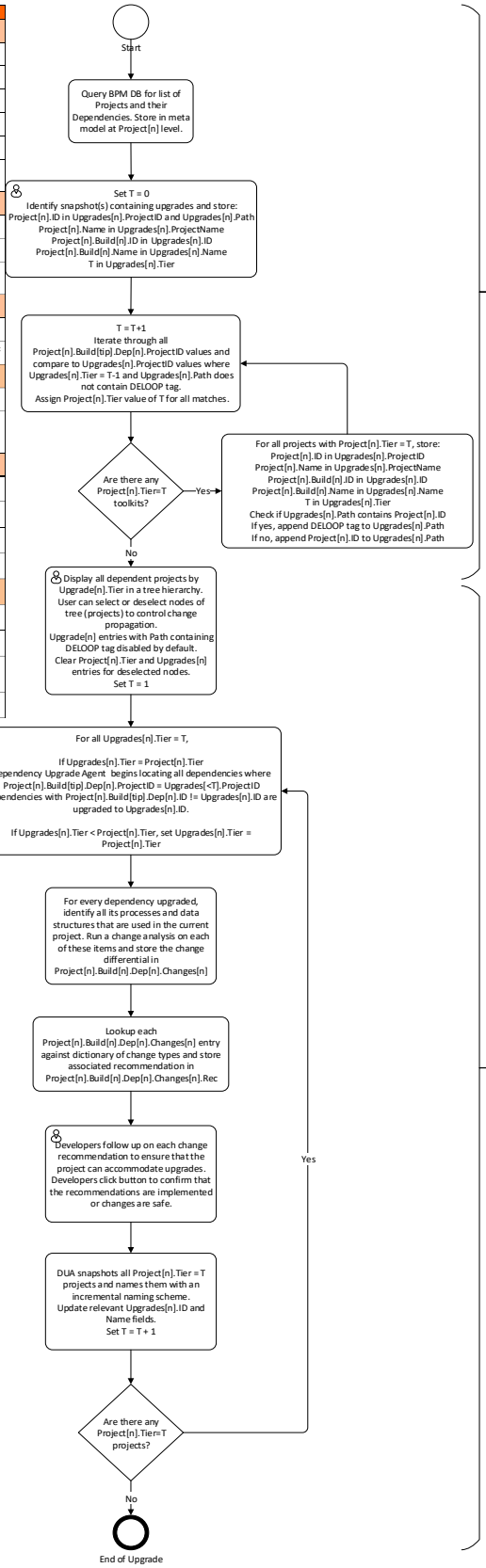

Figure 3-7 Upgrade Algorithm

3.6. Chapter Summary

In this chapter, we have provided a detailed explanation of all the components required by our proposed framework for impact assessment and dependency management in BPM. We have also highlighted how these components interact in order to achieve our goal of more effective and efficient management of dependencies in BPM projects. In the next chapter, we will attempt to demonstrate the impacts of this framework through the use of four case studies.

Chapter 4. Case Studies

This chapter provides a detailed analysis of four examples to demonstrate how our proposed framework can enhance the functionality typically found in BPM suites. Starting with chapter 4.1, these examples exemplify typical BPM upgrade scenarios and with them the limitations commonly experienced by developers conducting upgrade operations in BPM development suites. The case studies are based on our experiences working on BPM projects at Bell Canada but have been deliberately been altered so as not to reveal any aspect of Bell Canada operations. The examples will describe how the framework, which consists of a Business Process Component Architecture, a Dependency Data Model, and an Upgrade Algorithm contributes to improving dependency management operations for typical BPM developers.

4.1. Simple toolkit example

This example demonstrates the simplest possible case of a toolkit being leveraged as a dependency and the minimum steps required to upgrade a project after a change to the toolkit is introduced. In order to isolate the steps required to upgrade a single project, this case will consist of only one updated toolkit and one affected process app.

4.1.1 **Overview**

In this scenario, an organization has made changes to a Common Account Functions toolkit, which provides some BPDs and other artifacts that can be used in different process applications for managing customer accounts. The development team wants to push the changes to this toolkit to any process apps or other toolkits which may

be using the Common Accounts Functions toolkit. They will do so using our Impact Assessment and Dependency Management Framework.

4.1.2 **Business Process Component Architecture**

The dependency tree diagram in Figure 4-1 shows all toolkits on which the Call Introductions process app is dependent. This includes the Common Accounts Functions toolkit which has recently been changed. This diagram indicates that the only changes made by the development team are located in the Common Account Functions toolkit. Starting from this point of change, only one dependency link is necessary to propagate the change all the way back to the top level, the Call Introductions process app. Due to this being the sole change, only one cycle of dependency upgrade is required. The type of change will affect what must be done within that cycle to successfully upgrade the process app with the changes contained in the new version of the toolkit.

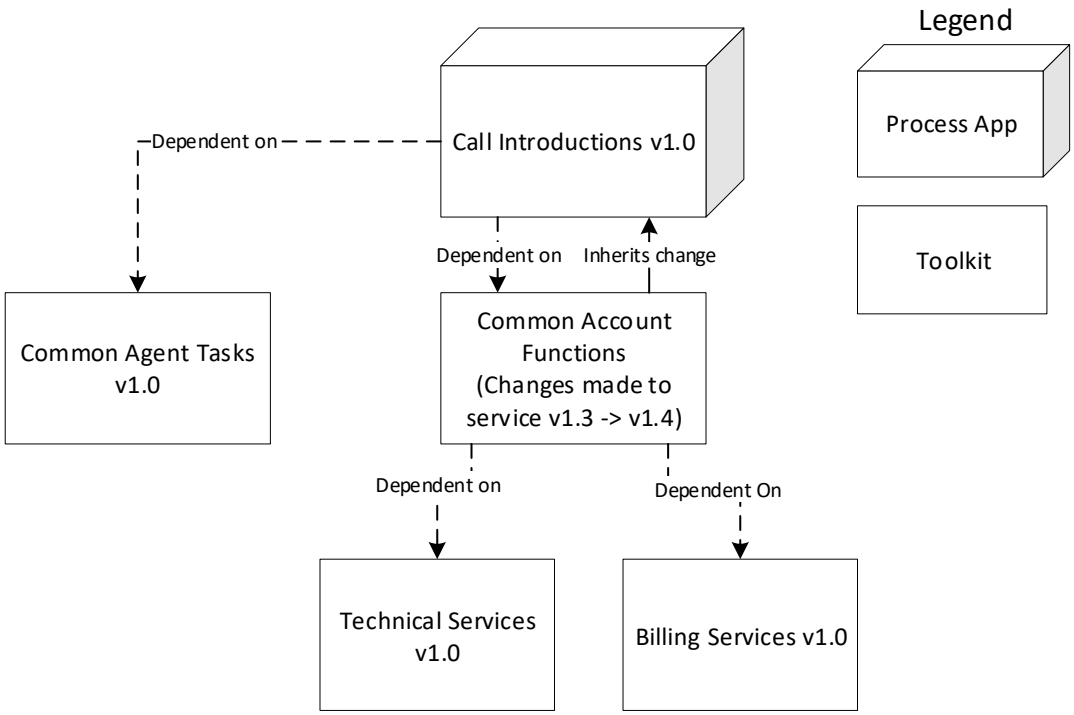

Figure 4-1: Call introductions Process App dependency tree

4.1.3 **Dependency Data Model**

The development team starts the framework which queries the BPM database to obtain a full roster of the projects on the BPM server. Then, the development team selects v1.4 of the Common Account Functions toolkit as shown in figure 4-2, to indicate which toolkit upgrade they are working on.

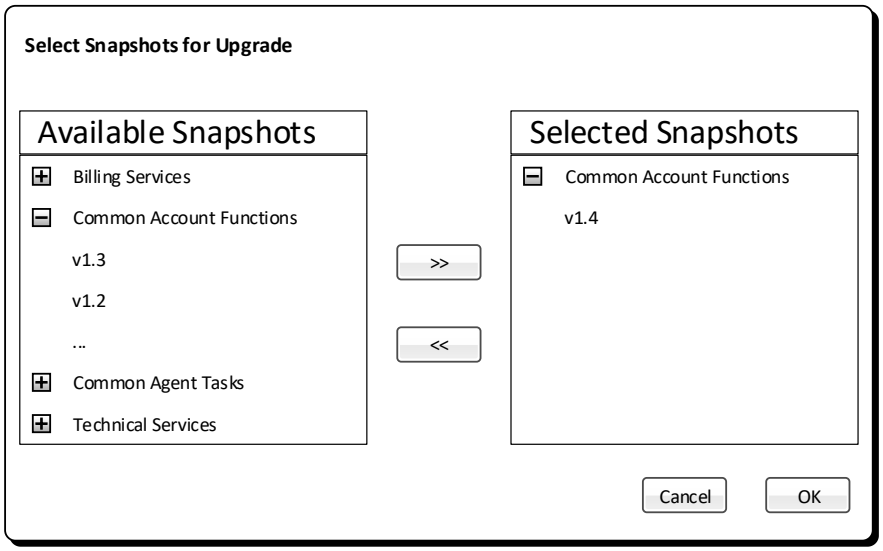

Figure 4-2: Change Selection Screen

The Impact Assessment Engine (IAE) then processes the Dependency Data Model (shown in Figure 4-1) to compile a complete list of all process apps and toolkits that are utilizing the Common Account Functions V1.4 toolkit as a direct dependency. For the sake of brevity, Figure 4-1 omits certain list entries from the dependency data model where these are not strictly necessary for the completion of Case 1.

Fields	Value
Upgrades[0]	*Upgrades list entry
Upgrades[0].ProjectName	Common Account Functions
Upgrades[0].ProjectID	CACCFUN
Upgrades[0].Name	Common Account Functions v1.4
Upgrades[0].ID	a88fb9cc-06a5-4e0f-bc25-871bd1e8d5d0
Upgrades[0].Tier	0
Upgrades[0].Path	CACCFUN
Upgrades[1]	*Upgrades list entry
Upgrades[1].ProjectName	Call Introductions
Upgrades[1].ProjectID	CALLINT
Upgrades[1].Name	Call Introductions v1.1 (Filled in after Tier 1 upgrades)
Upgrades[1].ID	aec2c6fb-d824-46ca-89d9-676e55691211 (Filled in after Tier 1 upgrades)

Upgrades[1].Tier	1
Upgrades[1].Path	CACCFUN-CALLINT
Project[0]	*Project list entry
Project[0].Name	Call Introductions
Project[0].ID	CALLINT
Project[0].Tier	1
Project[0].Build[0]	*Project Build list entry
[…].Build[0].Name	Call Introductions v1.0
[…].Build[0].ID	3aa089ed-5a93-4894-9e11-6e563696bcac
[…].Build[0].Services[0]	*Build Services list entry
[…].Services[0].Name	Critical Account Checks
[…].Services[0].Content	*Service Contents
[…].Build[0].Dep[0]	*Build Dependency list entry
[…].Dep[0].ProjectName	Common Account Functions
[…].Dep[0].ProjectID	CACCFUN
[…].Dep[0].Name	Common Account Functions v1.3
[…].Dep[0].ID	68c9e78b-e89b-443c-aefb-c336da268cf4
[…].Dep[0].Changes[0]	*Dependency Changes list entry
[…].Changes[0].Process	General Account Deficiency Check
[…].Changes[0].Change	Added end point “To Billing” to diagram.
[…].Changes[0].Rec	Add “To Billing” exit line from sub-process block.
[…].Changes[0].Places	Critical Account Checks
[…].Build[0].Dep[1]	*Build Dependency list entry
[…].Dep[1].ProjectName	Common Agent Tasks
[…].Dep[1].ProjectID	COMAGTA
[…].Dep[1].Name	Common Agent Tasks v1.0
[…].Dep[1].ID	a3f84d4f-e8c0-439c-b120-feec9855a988
Project[1]	*Project list entry
Project[1].Name	Common Account Functions
Project[1].ID	CACCFUN
Project[1].Tier	0
Project[1].Build[0]	*Project Build list entry
[…].Build[1].Name	Common Account Functions v1.4
[…].Build[1].ID	a88fb9cc-06a5-4e0f-bc25-871bd1e8d5d0
[…].Build[0].Services[0]	*Build Services list entry

[…].Services[0].Name	General Account Deficiency Check
[…].Services[0].Content	*Service Contents
[…].Build[0].Dep[0]	*Build Dependency list entry
[…].Dep[0].ProjectName	Technical Services
[…].Dep[0].ProjectID	TECSERV
[…].Dep[0].Name	Technical Services v1.0
[…].Dep[0].ID	cd0aff6d-72ef-49d6-90e7-7076da13e435
[…].Build[0].Dep[1]	*Build Dependency list entry
[…].Dep[1].ProjectName	Billing Services
[…].Dep[1].ProjectID	BILSERV
[…].Dep[1].Name	Billing Services v1.0
[…].Dep[1].ID	1d3043c7-a705-44e5-a893-3a0bf6cf37b2
Project[1].Build[1]	*Project Build list entry
[…].Build[1].Name	Common Account Functions v1.3
[…].Build[1].ID	68c9e78b-e89b-443c-aefb-c336da268cf4
[…].Build[1].Services[0]	*Build Services list entry
[…].Services[0].Name	General Account Deficiency Check
[…].Services[0].Content	*Service Contents
[…].Build[1].Dep[0]	*Build Dependency list entry
[…].Dep[0].ProjectName	Technical Services
[…].Dep[0].ProjectID	TECSERV
[…].Dep[0].Name	Technical Services v1.0
[…].Dep[0].ID	cd0aff6d-72ef-49d6-90e7-7076da13e435
[…].Build[1].Dep[1]	*Build Dependency list entry
[…].Dep[1].ProjectName	Billing Services
[…].Dep[1].ProjectID	BILSERV
[…].Dep[1].Name	Billing Services v1.0
[…].Dep[1].ID	1d3043c7-a705-44e5-a893-3a0bf6cf37b2

Figure 4-3: Example 1 Dependency Data Model Table

This will result in a very simple impact assessment. The results for this case will indicate that only one service in the Call Introductions process app is affected by upgrading the Common Account Functions toolkit. This can be verified in Figure 4-3 where Project[0].Build[0].Dep[0].ProjectID, a dependency of the Call introductions

process app, is equal to the value stored in Upgrades[0].ProjectID, the upgrade list entry corresponding to our desired toolkit upgrade. This identifies Call Introductions as a project impacted by the upgrade to Common Account Functions v1.4 and results in the creation of the Upgrades[1] list entry. The version of Common Account Functions referenced as a dependency in Project[0].Build[0].Dep[0] is v1.3 and the version we are trying to propagate is v1.4. The **Impact Assessment Engine** then searches Project[0].Build[0].Services[n].Contents for references to Project[1].Build[1].Services[n].Name entries which indicate Common Account Functions v1.3 services being used by Call Introductions V1.0. In this manner, the **Critical Account Checks** function in Call Introductions v1.0 is found to be leveraging the **General Account Deficiency Check** from Common Account Functions v1.3. The **Impact Assessment Engine** then compares the **General Account Deficiency Check** service from v1.3 to the new version from v1.4. Figures 4-4 and 4-5 show the service contents for each of these versions of General Account Deficiency Check.

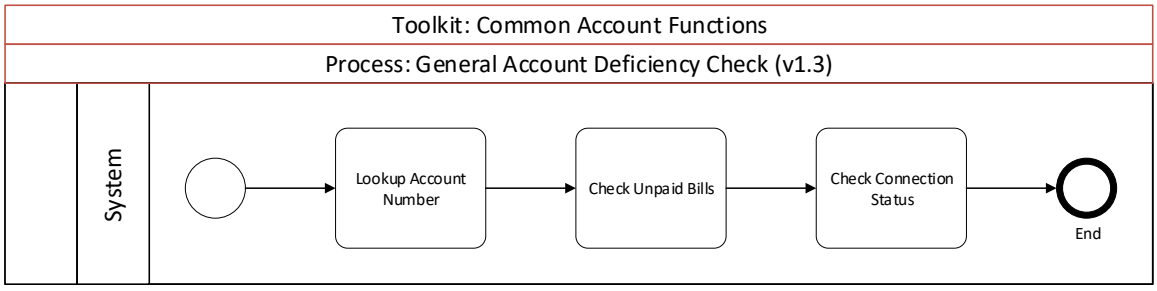

Figure 4-4: General Account Deficiency Check v1.3

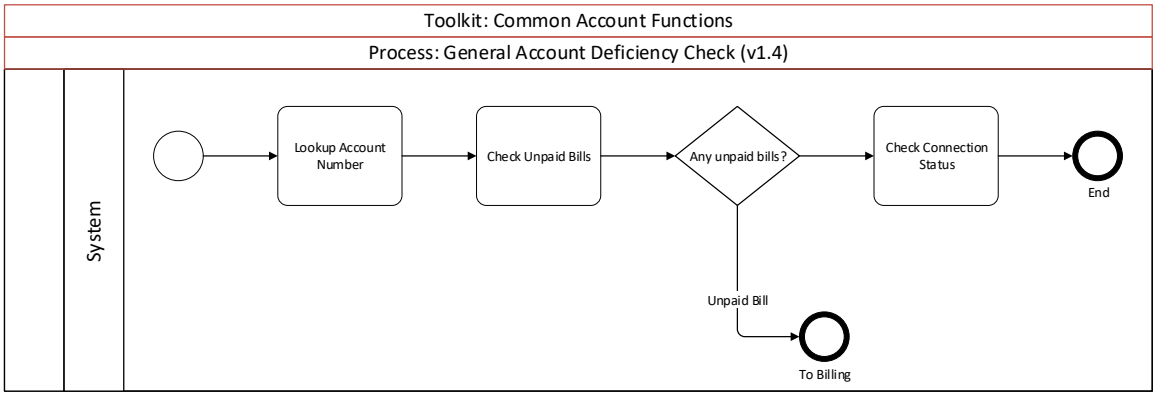

Figure 4-5: General Account Deficiency Check v1.4

The changes here consist of the insertion of a decision gateway into the flow, a new endpoint for the service and a flow line connecting the two. Of these three changes made to the General Account Deficiency Check service, the algorithm would flag the new endpoint as being the only change that would affect the Critical Account Checks process in the Call Introductions process app. As shown in Figure 4-6 and Figure 4-7, The suggested change will be to add a flow path for the "To Billing" end point out of the "General Account Deficiency Check" block. Failing to do so will result in an "undefined next step" error if our service execution takes us through that endpoint.

4.1.4 **Upgrade Algorithm**

At this point, the **IAE** has filled the Upgrades[n] section in the dependency data model table. The information in Upgrades[n], will serve as a map of the upgrades to be performed in the **BPM Dev Env** for our change to the Common Account Functions toolkit. It has also produced a list of differences between our new version, 1.4, and our previous version, 1.3. The **IAE** now transfers this information over to the **Dependency Upgrade Agent (DUA)** in order to begin the actual upgrade activities. The **DUA** starts by

automatically making API calls back to the BPM server. These calls are meant to exactly replicate the content and format of the native Dependency Upgrade command usually sent by the BPM suite when manually upgrading dependencies during development. In this case, only one API call is made to the server to upgrade the dependency of the "Common Account Functions" toolkit from version 1.3 to version 1.4. Upon receiving a message of success to the upgrade call, the **DUA** then notifies the development team that it is time to inspect the "Call Introductions" process app for unforeseen errors and to make the changes suggested by the **IAE**. The dev team decides to connect the new "To Billing" endpoint in the "General Account Deficiency Check" service to a newly created message endpoint called "Transfer to Billing".

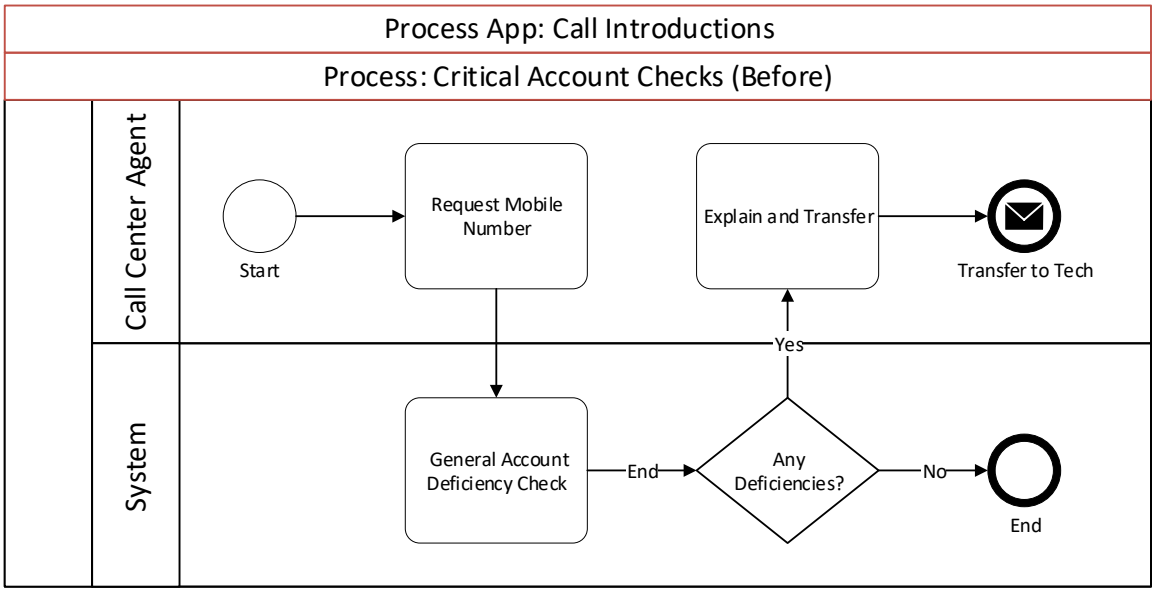

Figure 4-6: Critical Account Checks before upgrade

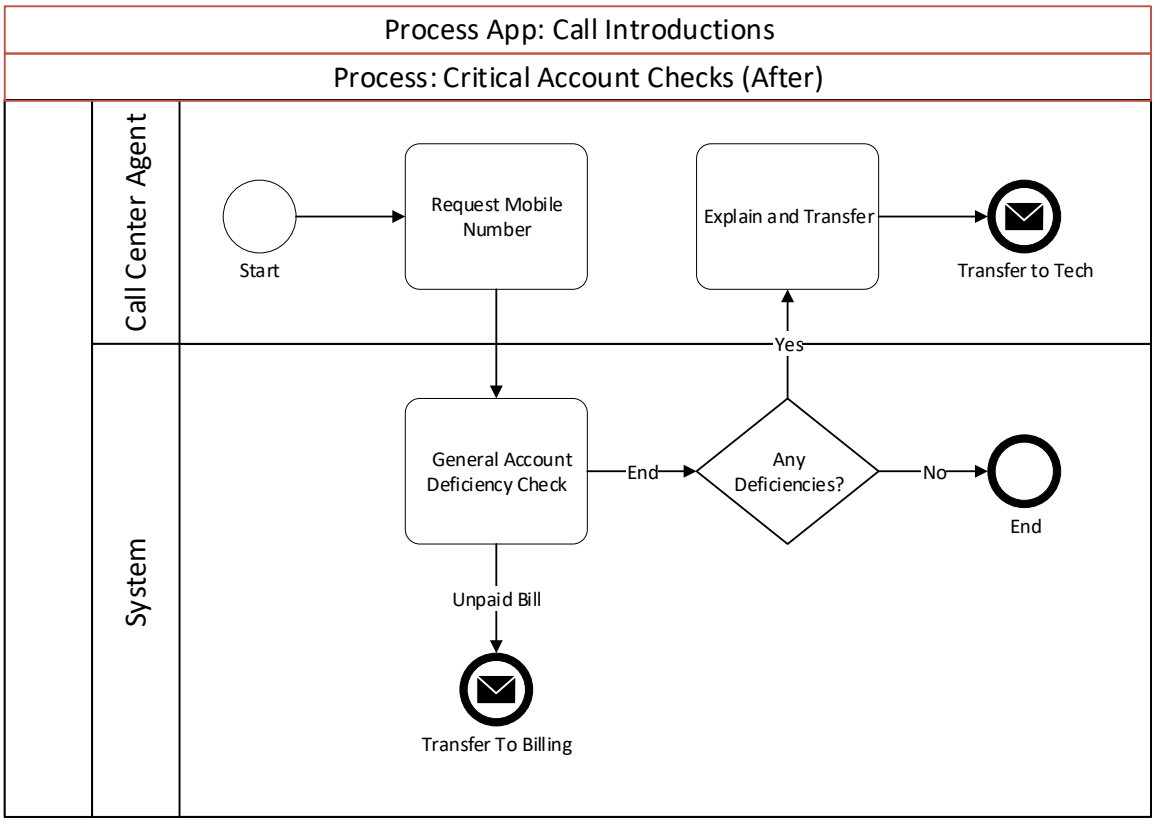

Figure 4-7: Critical Account Checks after upgrade

When reached, this new message endpoint will send out a message before it ends the "Critical Account Checks" process. This message starts another process with a billing agent where the call will be transferred due to an unpaid balance on the account.

After the dev team has made the necessary changes to the "Call Introductions" process app, they will then confirm in the **DUA** interface that the first tier of upgrades is complete. The **DUA** will then send out an API call instructing the BPM server to take a snapshot of the process app and name it with an incremental version number. Upon receiving a response of success from the server, the **DUA** will then move on to the next tier of upgrades. Since this case only has one tier of upgrades, the **DUA** will then

conclude its upgrade activities and will generate the framework summary report of activities performed.

4.1.5 **Results**

This first case is designed to demonstrate the most basic abilities of the impact assessment and dependency management framework. Applying the framework to this simple case shows that the **Impact Assessment Engine** is able to identify projects that are dependent on a change made to a toolkit and the **Dependency Upgrade Agent** is able to initiate the upgrade process and snapshot the updated versions of dependent projects. Given that this example is the simplest form of dependency upgrade that can be performed in the BPM Dev Env, the assistance of the framework is not needed to address issues of large upgrade volumes or complex dependency relationship structures. At such a small scale, using the framework provides no added value over manually upgrading the dependency with the default IBM BPM 8.5 toolset.

4.2. Common toolkit upgrade example

This case looks at how the framework fairs when confronted with an increased number of toolkits and process apps accompanied by additional complexity in the dependency relationships between these projects. Having already demonstrated the minimum requirements for upgrading a single, simple dependency, this example illustrates how these activities can be applied iteratively to accommodate BPM environments of any size and varying degrees of complexity.

4.2.1 Overview

In this scenario, changes have been made to the Technical Services toolkit. Compared to Case 1, this toolkit is lower down on the dependency chain and an additional level of process applications has been added to the dependency tree. This will result in the need for several coordinated upgrade activities to fully propagate the changes throughout the dependency tree. A case such as this is where the framework is intended to demonstrate its usefulness over having to manually execute the series of upgrade activities to successfully manage all the necessary dependencies.

4.2.2 Business Process Component Architecture

Again, we can see a dependency diagram, but this time it shows the structure of projects on which the Call Scripting process app is dependent. The diagram indicates that the only changes planned by the development team are in the Technical Services toolkit. From the diagram, it is clear that the impact of this change is much larger than what was presented in Case 1. The resulting upgrade activities will not only include more toolkits but will also need to be divided into several tiers in order the accommodate the several levels of toolkits and process apps impacted by the change.

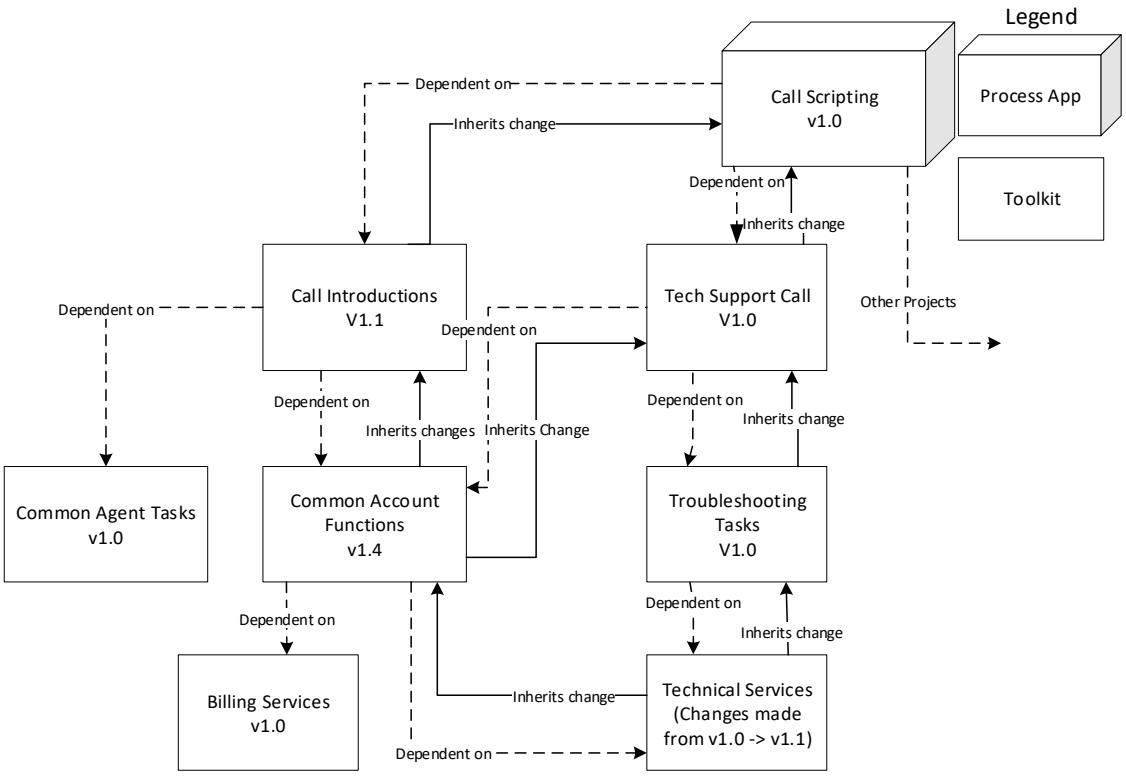

Figure 4-8: Call Scripting Process App dependency tree

Starting the impact assessment from Technical Services toolkit, we can see that the tier 1 toolkits affected by the change are the Common Account Functions and Troubleshooting Tasks toolkits. Moving on, we can see that the changes to the tier 1 toolkits will impact Call Introductions and Tech Support Call. These toolkits are tier 2 of the upgrade activities. Finally, the changes to tier 2 projects will impact the process app at the top of the dependency tree. The Call Scripting process app becomes tier 3 of the upgrade activities. Having these 3 tiers means that 3 cycles of upgrades must be done before the changes can be propagated to all the necessary toolkits and process apps.

4.2.3 **Dependency Data Model**

Once again, the **Impact Assessment Engine** queries the BPM database and obtains the full list of server contents and project metadata. The development team must

now indicate the snapshot(s) that correspond to the change that they are trying to propagate. In this case, v1.1 of the Technical Services project is identified as the one containing the changes for this upgrade. The Project[Technical Services].Tier value is set to 0 to identify this toolkit as the project containing our new upgrades. This project's information is also stored in the Upgrades[n].ProjectName, Upgrades[n].ProjectID, Upgrades[n].Name and Upgrades[n].ID fields along with the value of 0 in the Upgrades[n].Tier field. This allows us to identify the specific snapshot and serves a validation function when it comes to projects appearing in multiple tiers.

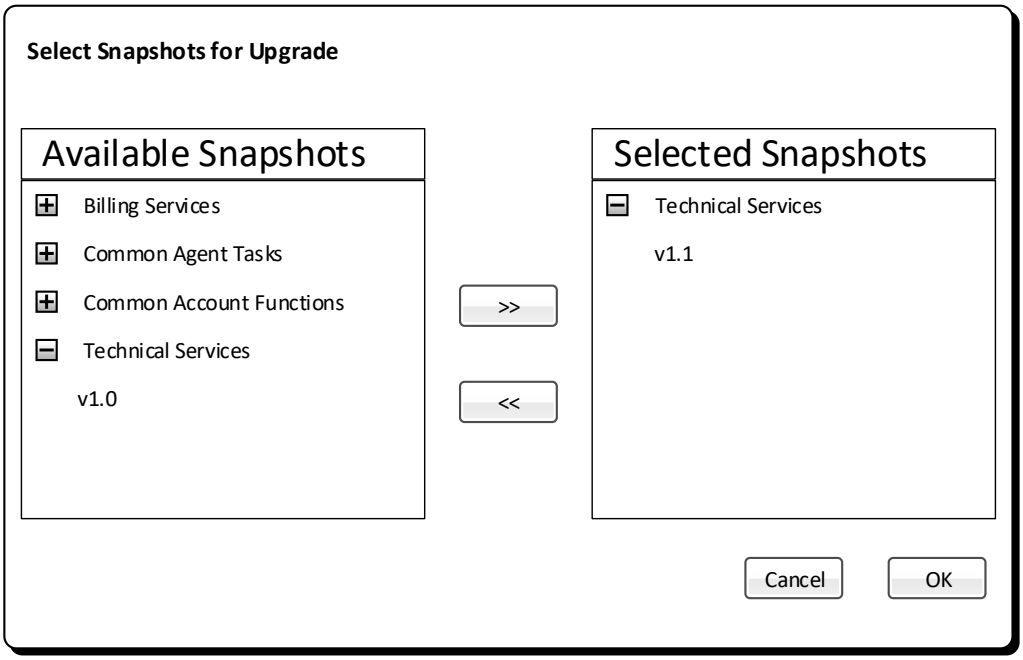

Figure 4-9: Change Selection Screen

The **IAE** then compares the values stored in Upgrades[n].ProjectID against the project ID values of other projects stored in Project[n].Build[tip].Dep[n].ProjectID. Matching IDs have their project marked with the current update tier in Project[n].Tier and their information stored in the Upgrades[n].ProjectName, Upgrades[n].ProjectID and Upgrades[n].Tier fields. The Upgrades[n].ID and Upgrades[n].Name can only be filled

later when the dependencies in that project have been upgraded and a snapshot has been taken. After doing this once to obtain a list of direct dependencies, this process is repeated for each successive tier of upgrades until only process apps are returned and no further dependent projects are possible. Given the current dependency structure between projects and the upgrades made to Technical Services toolkit, the resulting upgrade tiers will be as follows:

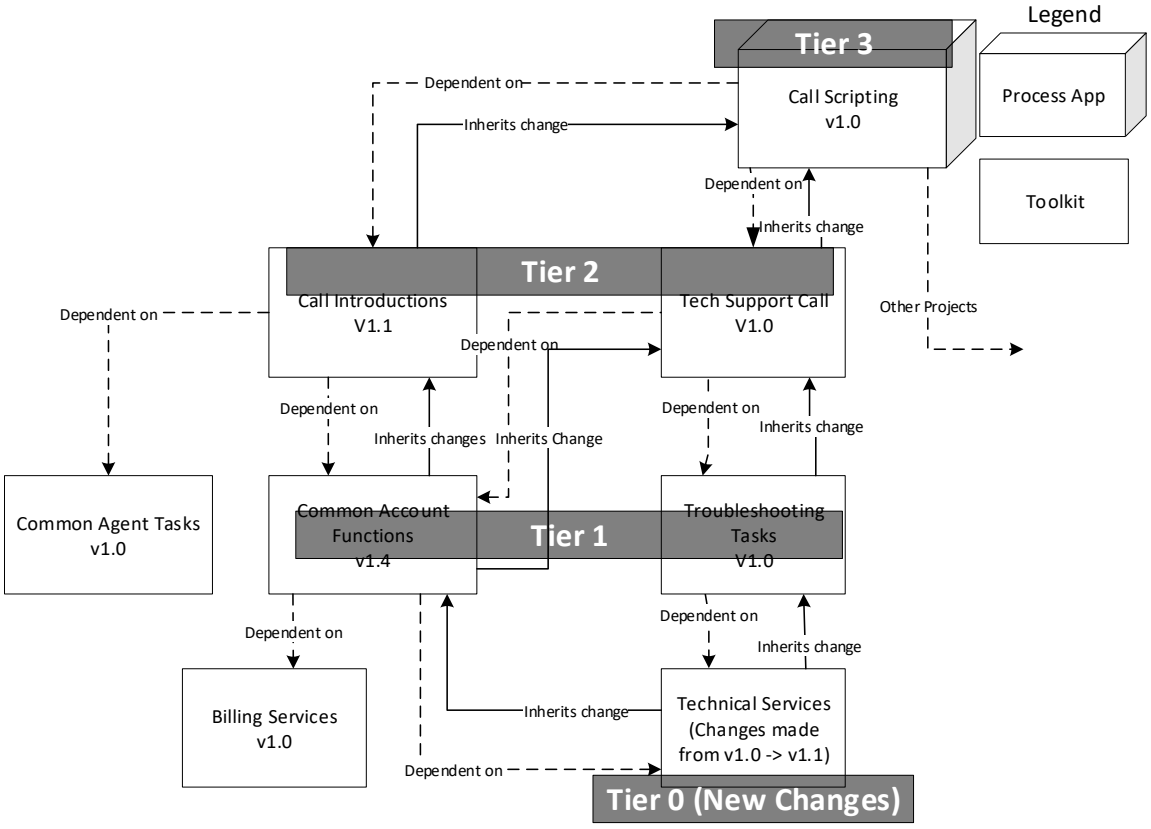

Figure 4-10: Upgrade Tiers

Having identified the full hierarchy tree for the current changes to the Technical Service toolkit as pictured above, the **Impact Assessment Engine** will have populated the Dependency Data Model with all of the relevant information currently available (similar to what was done in section 4.1 and is shown in Figure 4-3). Of particular

importance when reaching this stage is the contents of the Upgrades[n] section of the Dependency Data model. The contents of the Upgrades[n] section for this particular scenario are displayed below in Figure 4-11 which shows only that portion of the Dependency Data Model:

Fields	Value
Upgrades[0]	*Upgrades list entry
Upgrades[0].ProjectName	Technical Services
Upgrades[0].ProjectID	TECSERV
Upgrades[0].Name	Technical Services v1.1
Upgrades[0].ID	7526c879-bfed-4e52-b963-dddd3cce0635
Upgrades[0].Tier	0
Upgrades[0].Path	TECSERV
Upgrades[1]	*Upgrades list entry
Upgrades[1].ProjectName	Common Account Functions
Upgrades[1].ProjectID	CACCFUN
Upgrades[1].Name	Filled in after Tier 1 upgrades
Upgrades[1].ID	Filled in after Tier 1 upgrades
Upgrades[1].Tier	1
Upgrades[1].Path	TECSERV-CACCFUN
Upgrades[2]	*Upgrades list entry
Upgrades[2].ProjectName	Troubleshooting Tasks
Upgrades[2].ProjectID	TROTASK
Upgrades[2].Name	Filled in after Tier 1 upgrades
Upgrades[2].ID	Filled in after Tier 1 upgrades
Upgrades[2].Tier	1
Upgrades[2].Path	TECSERV-TROTASK
Upgrades[3]	*Upgrades list entry
Upgrades[3].ProjectName	Call Introductions
Upgrades[3].ProjectID	CALLINT
Upgrades[3].Name	Filled in after Tier 2 upgrades
Upgrades[3].ID	Filled in after Tier 2 upgrades
Upgrades[3].Tier	2
Upgrades[3].Path	TECSERV-CACCFUN-CALLINT
Upgrades[4]	*Upgrades list entry

Upgrades[4].ProjectName	Tech Support Call
Upgrades[4].ProjectID	TECCALL
Upgrades[4].Name	Filled in after Tier 2 upgrades
Upgrades[4].ID	Filled in after Tier 2 upgrades
Upgrades[4].Tier	2
Upgrades[4].Path	TECSERV-CACCFUN-TECCALL
Upgrades[5]	*Upgrades list entry
Upgrades[5].ProjectName	Tech Support Call
Upgrades[5].ProjectID	TECCALL
Upgrades[5].Name	Filled in after Tier 2 upgrades
Upgrades[5].ID	Filled in after Tier 2 upgrades
Upgrades[5].Tier	2
Upgrades[5].Path	TECSERV-TROTASK-TECCALL
Upgrades[6]	*Upgrades list entry
Upgrades[6].ProjectName	Call Scripting
Upgrades[6].ProjectID	CSCRIPT
Upgrades[6].Name	Filled in after Tier 3 upgrades
Upgrades[6].ID	Filled in after Tier 3 upgrades
Upgrades[6].Tier	3
Upgrades[6].Path	TECSERV-CACCFUN-CALLINT-CSCRIPT
Upgrades[7]	*Upgrades list entry
Upgrades[7].ProjectName	Call Scripting
Upgrades[7].ProjectID	CSCRIPT
Upgrades[7].Name	Filled in after Tier 3 upgrades
Upgrades[7].ID	Filled in after Tier 3 upgrades
Upgrades[7].Tier	3
Upgrades[7].Path	TECSERV-CACCFUN-TECCALL-CSCRIPT
Upgrades[8]	*Upgrades list entry
Upgrades[8].ProjectName	Call Scripting
Upgrades[8].ProjectID	CSCRIPT
Upgrades[8].Name	Filled in after Tier 3 upgrades
Upgrades[8].ID	Filled in after Tier 3 upgrades
Upgrades[8].Tier	3
Upgrades[8].Path	TECSERV-TROTASK-TECCALL-CSCRIPT

Figure 4-11: Example 2 Upgrades Section of Dependency Data Model

As previously explained, it is not possible to know the snapshot name or snapshot ID of a project until that project has been upgraded. Once a new snapshot containing the

necessary dependency upgrades has been created, its name and ID can be inserted into the Upgrades[n].Name and Upgrades[n].ID fields of the corresponding project.

4.2.4 **Upgrade Algorithm**

The completed dependency model is then displayed visually for the development team to see. All projects impacted by the new change through direct or indirect dependency will be visible as nodes in a tree diagram. These nodes are all selected for upgrade by default, but the development team are given the ability to manually deselect any nodes in the tree where they may deem the upgrade to be unnecessary. In this case, they leave all of the nodes selected in order to upgrade all dependent projects with the new changes.

With all projects selected, the data model is now transferred to the **Dependency Upgrade Agent** (**DUA**). Looking at all projects with Project[n].ID values equal to Upgrades[n].ProjectID where Upgrades[n].Tier = 1, the **DUA** performs a comparative analysis between the snapshot already present in the project dependencies and the new snapshots identified by the value in Upgrades[n].ID where Upgrades[n].Tier = 0. This effectively compares the Project[n].Build[Old].Services[n] against Project[n].Build[New].Services[n] and allows the **Dependency Upgrade Agent** to generate a list of changes between versions. This list of changes is stored in the Project[dependent].Build[tip].Dep[upgrade].Change[n] section of all dependencies to be upgraded within the directly dependent projects. In case 2, the changes to the Technical Services toolkit are limited to one service.

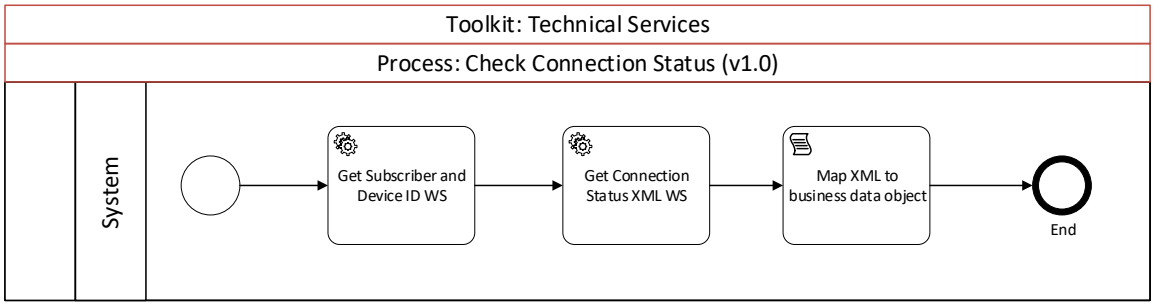

Figure 4-12: Check Connection Status v1.0

The Check Connection Status service contains the only changes between Technical Services v1.0 and v1.1. Most obvious is the addition of a new script block at the end of the service. This script block is responsible for setting some new output variables that have been added to this service. When this service is upgraded in dependent projects, the developers will need to indicate where these new output values are mapped to in the higher-level process. Another change made is to a data object referenced by another script in the service. Dependent projects using this service will need to ensure that their data definitions align with the new format of the business data object.

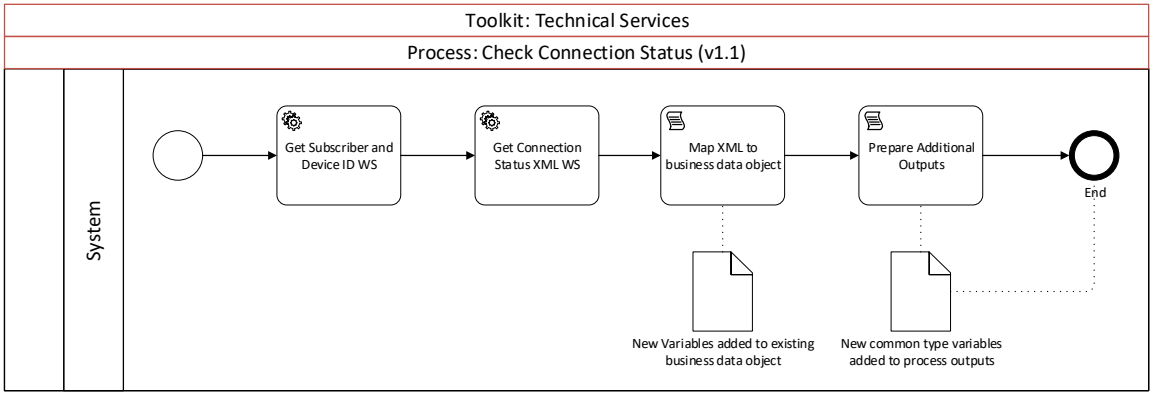

Figure 4-13: Check Connection Status v1.1

Based on figure 4-10, the tier 1 upgrades are Common Account Functions and Troubleshooting Tasks. The **DUA** initiates the dependency upgrade command to upgrade

the Technical Services toolkit from v1.0 to v1.1 simultaneously in both of these projects. Once upgraded, the developers must now go and verify any locations where this service is present in our Tier 1 upgrade projects. In the Common Account Functions toolkit, the General Account Deficiency Check service is affected by the changes to Check Connection Status from the Technical Services toolkit.

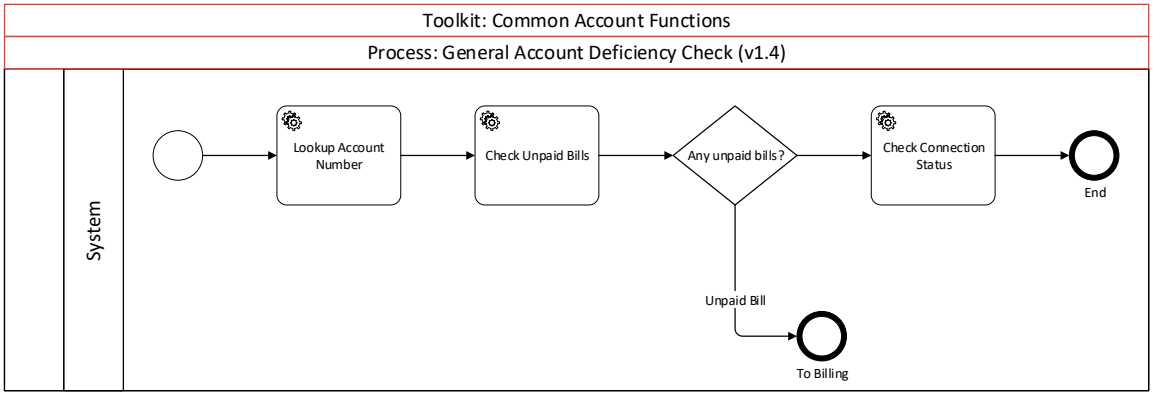

Figure 4-14: General Account Deficiency Check v1.4

From the perspective of the General Account Deficiency Check service, new data outputs have been added to Check Connection Status and existing outputs have been expanded. The BPM developers must indicate what the General Account Deficiency Service does with these new output values. In this case, they leave the new outputs unassigned as they are part of future release preparation. As for the existing output, the business data object used to store that information has been expanded. No action is required by the developers here as the simple act of upgrading the dependency has updated the definition of the business data object in the dependent project.

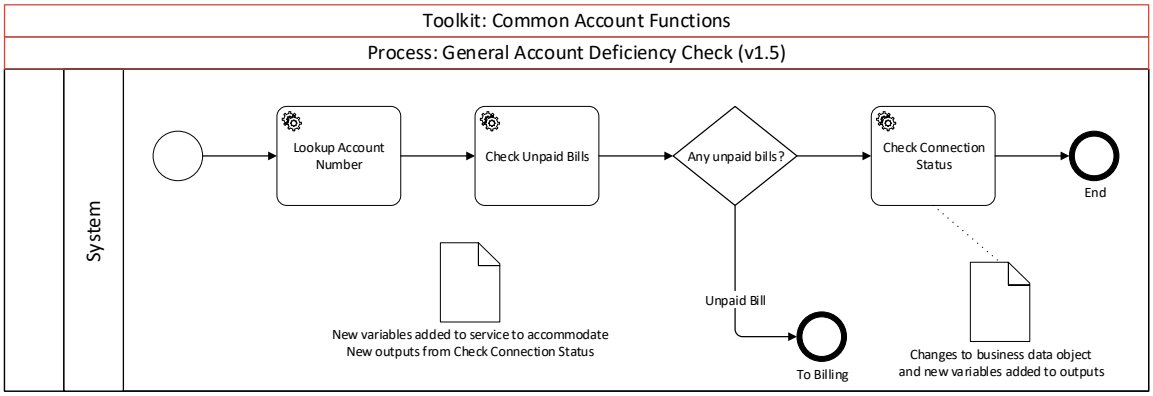

Figure 4-15: General Account Deficiency Check v1.5

In the Troubleshooting Tasks toolkit, another service is affected by the changes to Check Connection Status. The service called “Display Connection Details” makes use of the Check Connection Status service and then displays some of the data returned by this service for the agent to see.

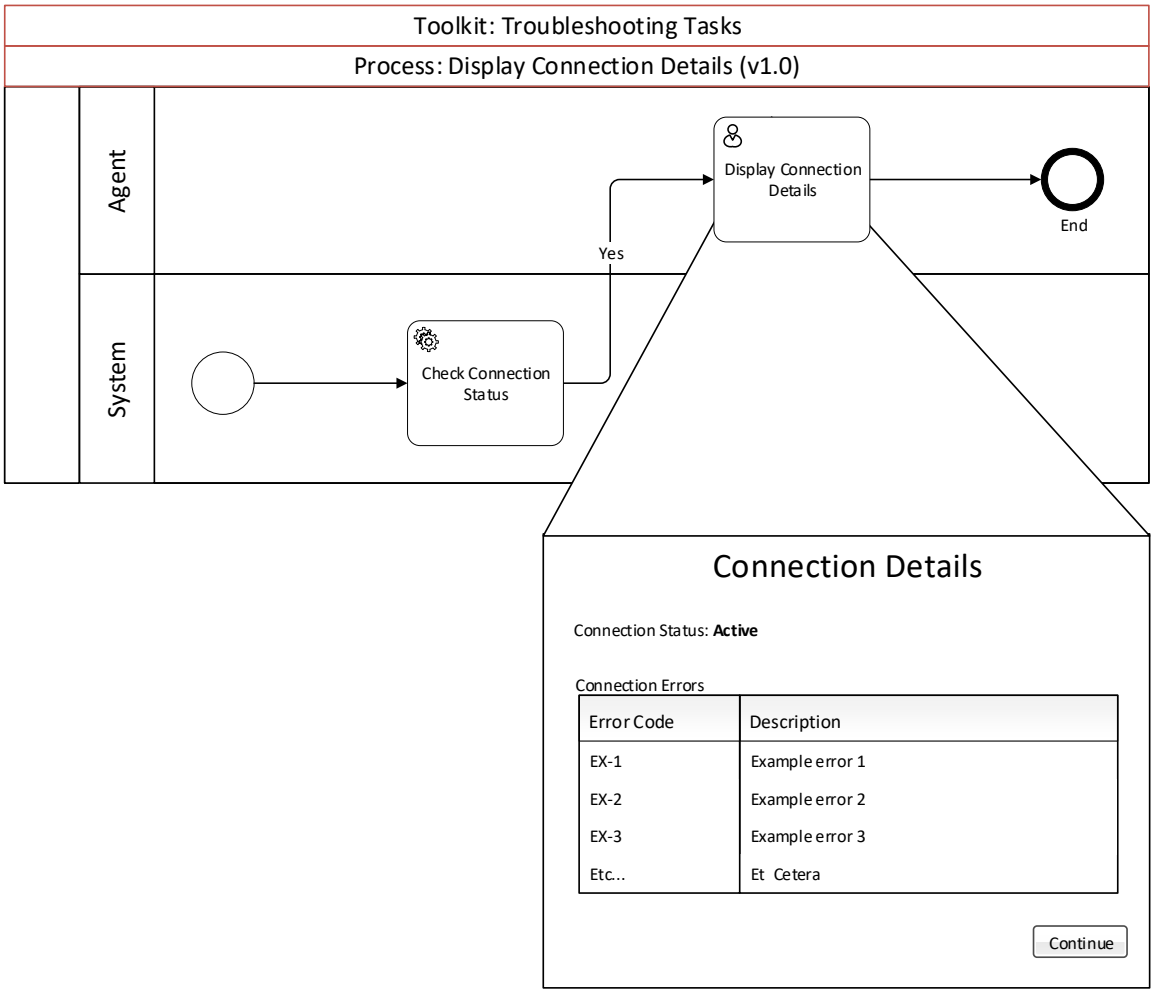

Figure 4-16: Display Connection Details v1.0

Once again, the changes to the business data object become effective as soon as the toolkit dependency is updated within the Troubleshooting Tasks project. With regard to the newly created outputs in the Check Connection Status service, the developers must again decide what changes, if any, to make to in order to accommodate these changes. In this case, the developers output the information to some newly created private variables within the Display Connection Details service. This data is then displayed in the subsequent user interface for the call center agent. The use of private variables here indicates that this new information will not be sent to any services further up the chain of

dependencies. This will minimize the impact of this change on the remaining projects and simplify the process of upgrading them.

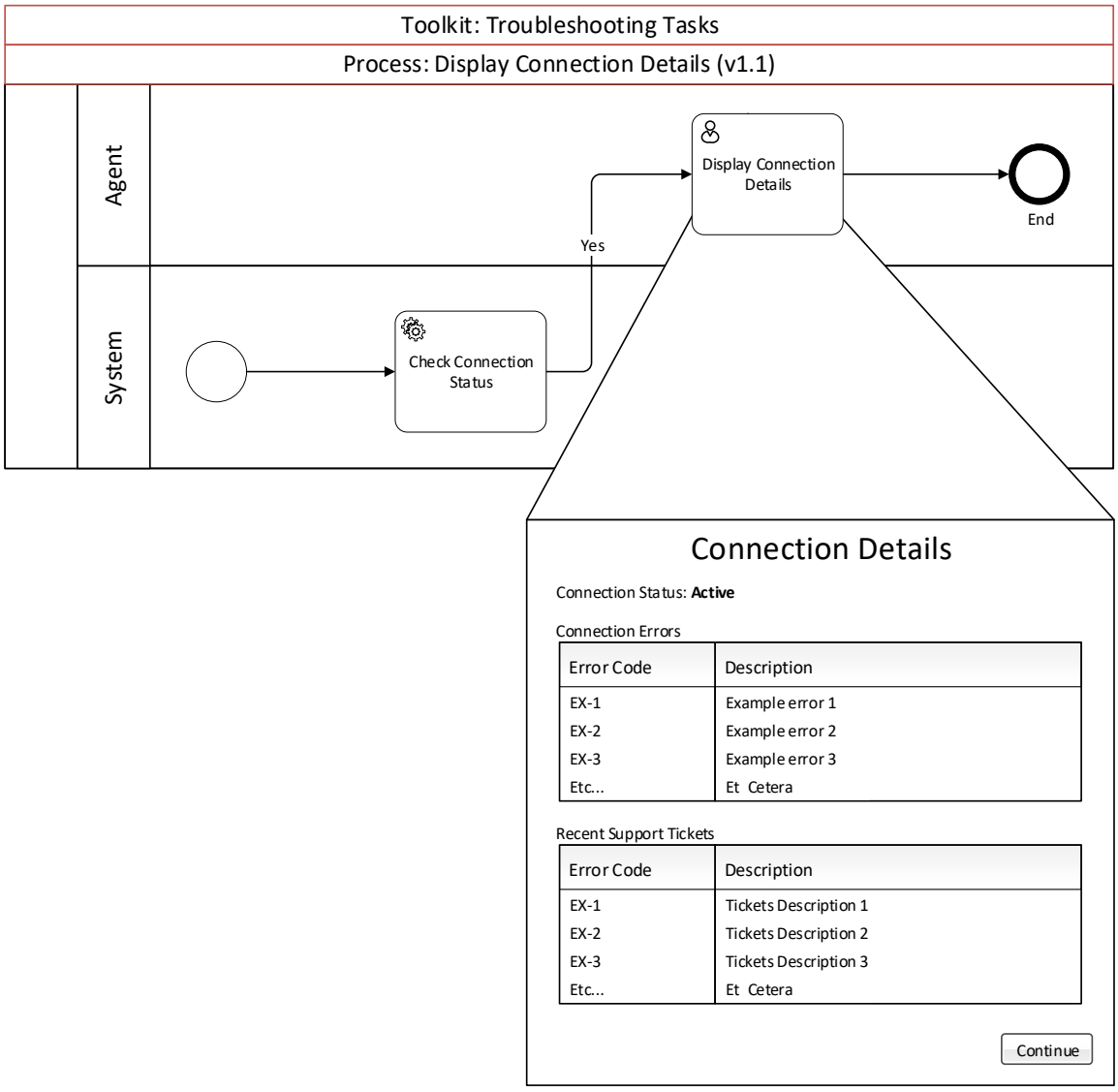

Figure 4-17: Display Connection Details v1.1

With the appropriate changes made after our Tier 1 upgrades, the **DUA** snapshots both of these projects and incrementally labels them Common Account Functions v1.5 and Troubleshooting Tasks v1.1 respectively. The **DUA** then moves on to the Tier 2 upgrades. It simultaneously upgrades the dependency to the Common Account Functions

toolkit in both the Call Introductions and Tech Support Call toolkits while also updating the dependency to the Troubleshooting Tasks toolkit in the Tech Support Call toolkit.

With the Tier 2 dependency upgrades complete, it is time for the developers to make the necessary changes to the Call Introductions and Tech Support Call toolkits. Given that the modifications made to the General Account Deficiency Check and Display Connection Details services do not introduce any changes that impact dependent services, the development team is not required to do anything within the Tier 2 toolkits to accommodate these changes. The **DUA** is free to immediately snapshot the two upgraded Tier 2 toolkits and incrementally names them Call Introductions v1.2 and Tech Support Call v1.1.

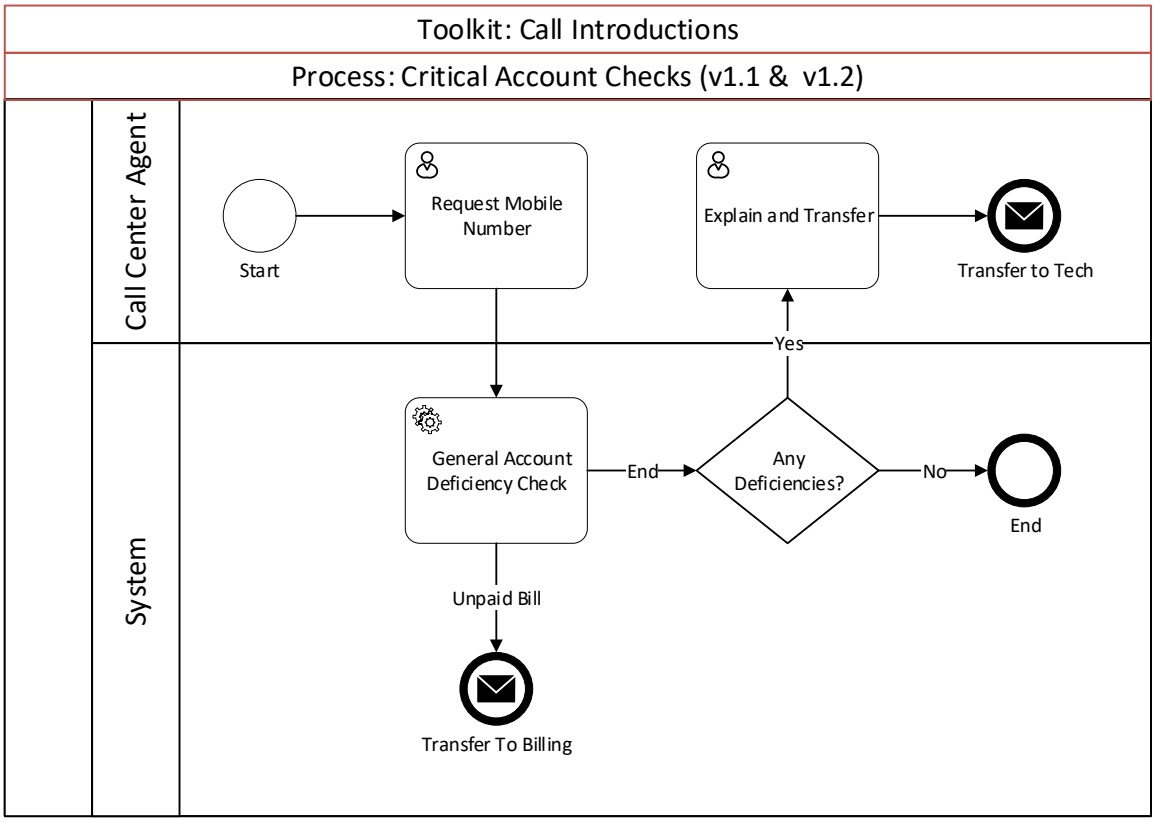

Figure 4-18: Critical Accounts Checks

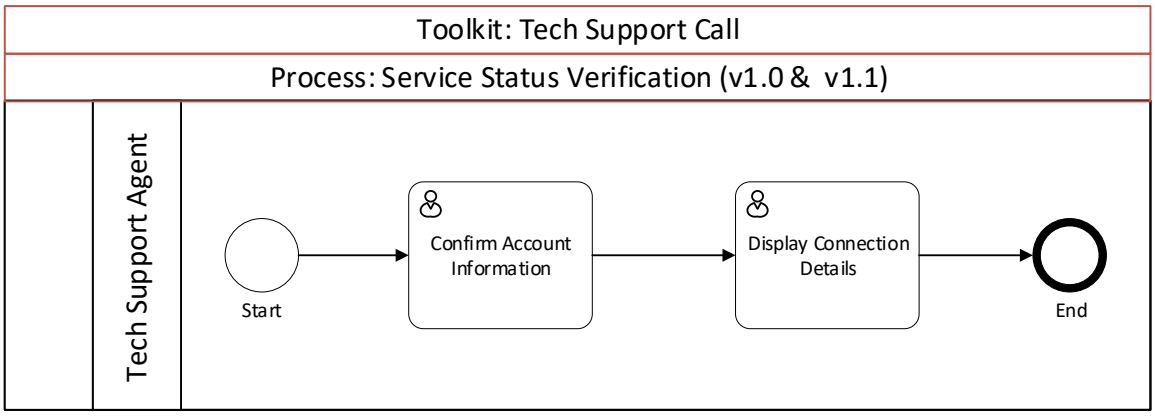

Figure 4-19: Status Verification

With the Call Introductions and Tech Support Call toolkits now containing the updated business data object definitions, it is time for the **DUA** to move on to the Tier 3 upgrades which will be the final upgrades in this Case 2 scenario. Tier 3 contains only the Call Scripting process app. The **DUA** proceeds to upgrade the Call Introductions and Tech Support Call dependencies within the Call Scripting process app. With the upgrades complete, the developers have the opportunity to make changes within the process app to accommodate any changes inherited from the dependencies. However, since no changes were made during the Tier 2 upgrades, then no changes are required by the developers for the Tier 3 upgrades. The **DUA** is then free to snapshot the Call Scripting process app and incrementally name the snapshot Call Scripting v1.1.

4.2.5 **Results**

Case 2 demonstrates that the impact assessment and upgrade activities of our framework can be successfully scaled up to a multi-tier scenario while maintaining the integrity of our projects and the dependency structure between them. Despite the larger scope of the change and necessary upgrades in this case, it was assumed that the developers maintained a compartmentalized development structure that lent itself well to

the upgrade activities required. Even at this limited scale, the framework begins to show utility by expediently identifying projects impacted by the newly introduced changes. The framework increases the organization of project upgrades by ensuring the correct order of upgrade operations. The framework also provides increased efficiency by allowing the last three projects of Case 2 to be upgraded automatically in the absence of any development changes requiring accommodation.

4.3. Flawed dependency structure upgrade example

Similar to case 2, case 3 looks at the same collection of toolkits and process apps arranged in nearly the same structure, but with the addition of several dependency relationships to complexify the impact assessment task of the framework. These additional dependencies are meant to represent the result of poor development practices and the lack of proper segmentation of toolkit functions in the dependency structure. For this reason, case 3 will contain less detail in areas that are largely similar to case 2. In depth analysis will focus more on the differences caused by the added complexity of new dependency relationships between the same BPM projects.

4.3.1 **Overview**

In this scenario, the same changes have been made to the Technical Services toolkit. However, let us assume that a different approach was taken during the development of the tech-specific toolkits (Technical Services, Troubleshooting Tasks, Tech Support Call). During the design of their business data objects, the tech developers added the lower level data structures, residing in the Technical Services toolkit, directly to all of the later tech toolkits. This approach was favoured by the developers as it allowed them to flatten the overarching data objects found in the Tech Support Call toolkit, but it also introduced an additional dependency which is of a non-standard nature.

There is a second change to be noted in the dependency relationships of case 3. We can see that there is a new dependency introduced between the Common Account Functions and Common Agent Tasks toolkits. This dependency also displays an abnormal pattern with respect to the general hierarchy of the pre-existing projects from case 2.

4.3.2 **Business Process Component Architecture**

The dependency diagram below is nearly identical to that of case 2, but includes the addition of two new dependency relationships between existing projects. These additions have been circled in red in order to highlight the changes. With the only development changes for this scenario being, once again, confined to the Technical Services toolkit, we can trace the impact of the modifications by following the solid lines labeled “Inherits Change” from our starting point to all the other projects in the dependency diagram. The difference now being that following these lines will lead us to count certain toolkits twice. How should this new scenario be handled and how will it impact the tiers of upgrades that result from the impact assessment?

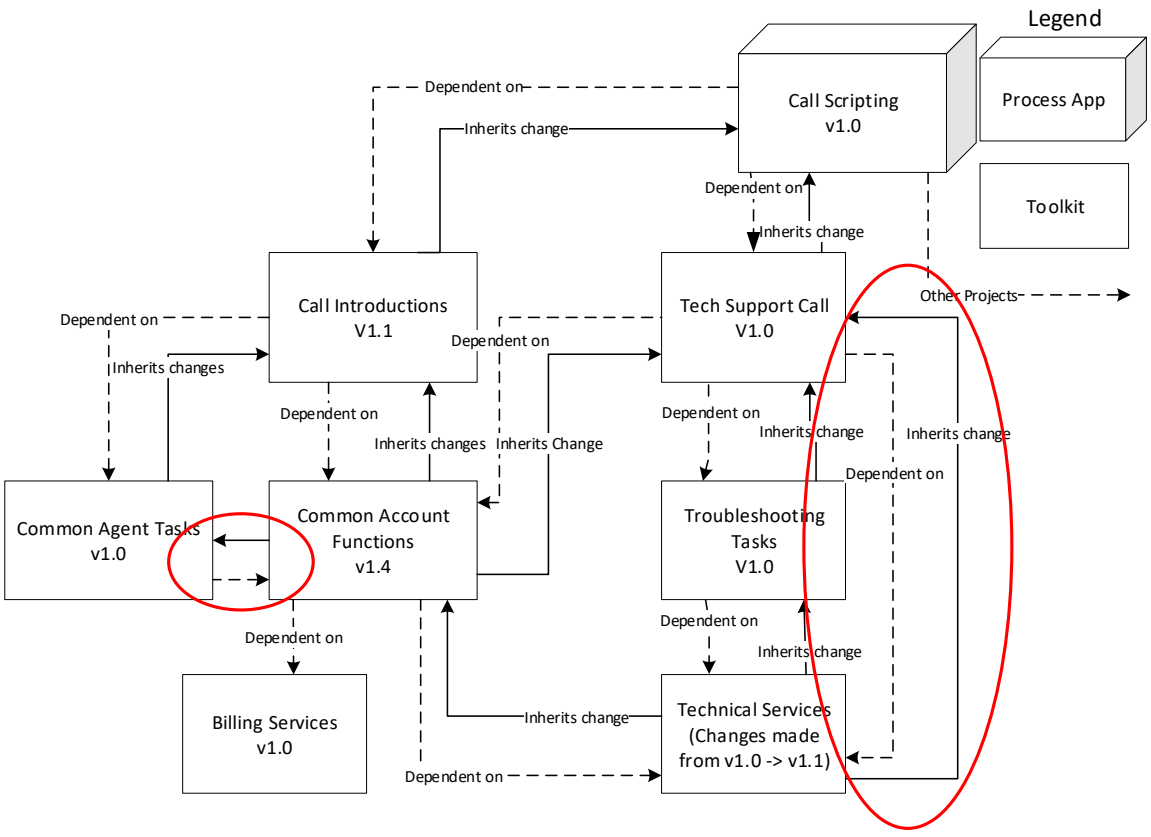

Figure 4-20: Call Scripting Process App dependency tree

Starting the impact assessment from the point of change, we can see that the tier 1 toolkits affected by the change are the Common Account Functions, Troubleshooting Tasks and Tech Support Call toolkits. This last one differs from the tier 1 obtained in case 2. Moving on, we can see that the changes to the tier 1 toolkits will impact Common Agent Tasks, Call Introductions, Tech Support Call and Call Scripting. These toolkits are tier 2 of the upgrade activities. It is important to note that Tech Support Call has appeared in both tier 1 and tier 2 toolkits thus far.

The changes to tier 2 projects will, in turn, impact Call Introductions and Call Scripting which will become tier 3. Again, it should be noted that Call Introductions and Call Scripting have both already been identified in tier 2.

Finally, the Call Scripting process app becomes tier 4 of the upgrade activities resulting from the changes to Call Introductions in tier 3. How can the same toolkits and process apps result in 4 tiers of upgrades when previously there were only 3? How can the appearances of toolkits and process apps in multiple tiers be handled safely and efficiently?

4.3.3 **Dependency Data Model**

As previously discussed, the impacts of the changes introduced in case 3 will be limited to the information in the populated dependency data model and how this information is leveraged by the **Dependency Upgrade Agent** algorithm.

The **Impact Assessment Engine** queries the BPM database and the development team must indicates the snapshot(s) that correspond to the change that they are trying to propagate. Again, v1.1 of the Technical Services project is identified as the one

containing the changes for this upgrade and the Project[Technical Services].Tier value is set to 0 to identify this toolkit as the original upgrade. Upgrades[n] values are also filled including 0 in the Upgrades[n].Tier field.

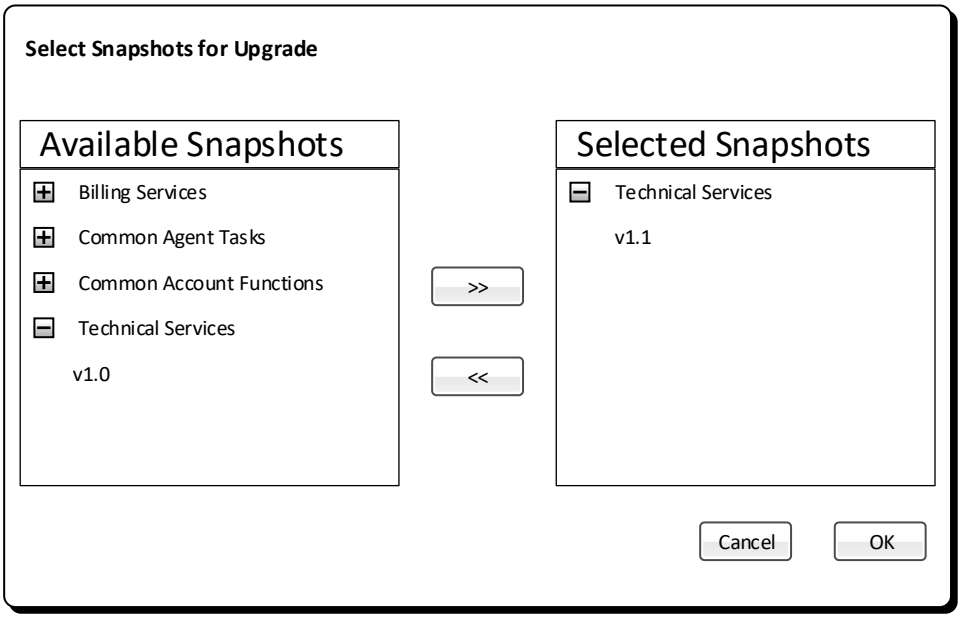

Figure 4-21: Change Selection Screen

After the **IAE** compares the values stored in Upgrades[n].ProjectID against the project ID values in Project[n].Build[tip].Dep[n].ProjectID fields, matching IDs have the current update tier stored in Project[n].Tier and their information stored in a new entry of Upgrades[n]. Dependency matching continues on each new tier until only process apps are returned and no further dependent projects are possible. The changes to Technical Services toolkit in case 3 will result in the following upgrade tier groupings:

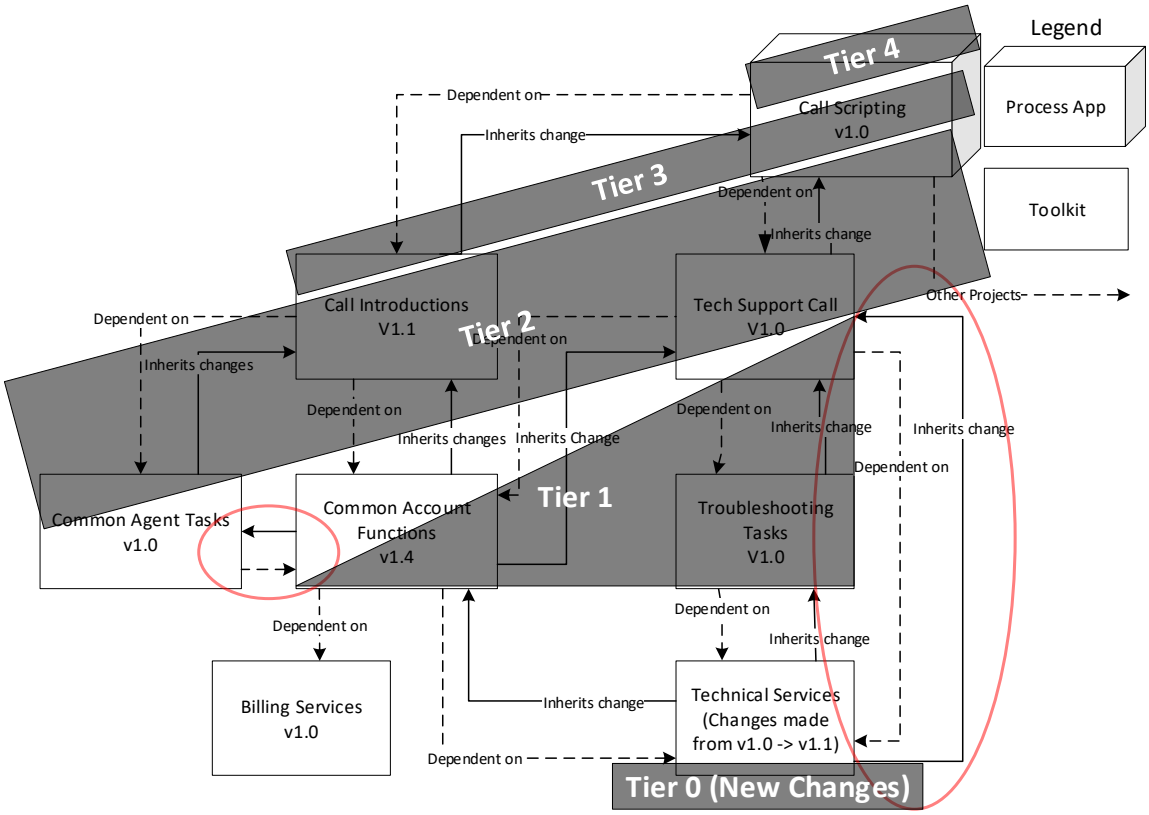

Figure 4-22: Upgrade Tiers

We can see here that the upgrade tiers in case 3 seem far less organized. There is an additional tier and the sequence of the tiers is irregular due to the repetition of several projects amid multiple tiers. Each tier in which a project appears represents a certain set of dependency upgrades that must be done within that toolkit or process app. By combining all of the upgrades indicated in each tier of a multi-tier project, we obtain a full list of the toolkits that could be impacting our multi-tier project. Only when all of these upgrades have been completed, will it be possible for a multi-tier project to be finalized and snapshotted. This will impact the timing of snapshots during the upcoming upgrade activities. The information necessary to properly coordinate these activities can be found in the Upgrades[n] and Projects[n] sections of the Dependency Data Model.

Fields	Value
Upgrades[0]	*Upgrades list entry
Upgrades[0].ProjectName	Technical Services
Upgrades[0].ProjectID	TECSERV
Upgrades[0].Name	Technical Services v1.1
Upgrades[0].ID	7526c879-bfed-4e52-b963-dddd3cce0635
Upgrades[0].Tier	0
Upgrades[0].Path	TECSERV
Upgrades[1]	*Upgrades list entry
Upgrades[1].ProjectName	Common Account Functions
Upgrades[1].ProjectID	CACCFUN
Upgrades[1].Name	Filled in after Tier 1 upgrades
Upgrades[1].ID	Filled in after Tier 1 upgrades
Upgrades[1].Tier	1
Upgrades[1].Path	TECSERV-CACCFUN
Upgrades[2]	*Upgrades list entry
Upgrades[2].ProjectName	Troubleshooting Tasks
Upgrades[2].ProjectID	TROTASK
Upgrades[2].Name	Filled in after Tier 1 upgrades
Upgrades[2].ID	Filled in after Tier 1 upgrades
Upgrades[2].Tier	1
Upgrades[2].Path	TECSERV-TROTASK
Upgrades[3]	*Upgrades list entry
Upgrades[3].ProjectName	Tech Support Call
Upgrades[3].ProjectID	TECCALL
Upgrades[3].Name	Filled in after Tier 1 upgrades
Upgrades[3].ID	Filled in after Tier 1 upgrades
Upgrades[3].Tier	1
Upgrades[3].Path	TECSERV- TECCALL
Upgrades[4]	*Upgrades list entry
Upgrades[4].ProjectName	Common Agent Tasks
Upgrades[4].ProjectID	COMAGTA
Upgrades[4].Name	Filled in after Tier 2 upgrades
Upgrades[4].ID	Filled in after Tier 2 upgrades
Upgrades[4].Tier	2
Upgrades[4].Path	TECSERV-CACCFUN- COMAGTA
Upgrades[5]	*Upgrades list entry

Upgrades[5].ProjectName	Call Introductions
Upgrades[5].ProjectID	CALLINT
Upgrades[5].Name	Filled in after Tier 2 upgrades
Upgrades[5].ID	Filled in after Tier 2 upgrades
Upgrades[5].Tier	2
Upgrades[5].Path	TECSERV-CACCFUN-CALLINT
Upgrades[6]	*Upgrades list entry
Upgrades[6].ProjectName	Tech Support Call
Upgrades[6].ProjectID	TECCALL
Upgrades[6].Name	Filled in after Tier 2 upgrades
Upgrades[6].ID	Filled in after Tier 2 upgrades
Upgrades[6].Tier	2
Upgrades[6].Path	TECSERV-CACCFUN-TECCALL
Upgrades[7]	*Upgrades list entry
Upgrades[7].ProjectName	Tech Support Call
Upgrades[7].ProjectID	TECCALL
Upgrades[7].Name	Filled in after Tier 2 upgrades
Upgrades[7].ID	Filled in after Tier 2 upgrades
Upgrades[7].Tier	2
Upgrades[7].Path	TECSERV-TROTASK-TECCALL
Upgrades[8]	*Upgrades list entry
Upgrades[8].ProjectName	Call Scripting
Upgrades[8].ProjectID	CSCRIPT
Upgrades[8].Name	Filled in after Tier 2 upgrades
Upgrades[8].ID	Filled in after Tier 2 upgrades
Upgrades[8].Tier	2
Upgrades[8].Path	TECSERV-TECCALL-CSCRIPT
Upgrades[9]	*Upgrades list entry
Upgrades[9].ProjectName	Call Introductions
Upgrades[9].ProjectID	CALLINT
Upgrades[9].Name	Filled in after Tier 2 upgrades
Upgrades[9].ID	Filled in after Tier 2 upgrades
Upgrades[9].Tier	3
Upgrades[9].Path	TECSERV-CACCFUN-COMAGTA-CALLINT
Upgrades[10]	*Upgrades list entry
Upgrades[10].ProjectName	Call Scripting
Upgrades[10].ProjectID	CSCRIPT

Upgrades[10].Name	Filled in after Tier 3 upgrades
Upgrades[10].ID	Filled in after Tier 3 upgrades
Upgrades[10].Tier	3
Upgrades[10].Path	TECSERV-CACCFUN-CALLINT-CSCRIPT
Upgrades[11]	*Upgrades list entry
Upgrades[11].ProjectName	Call Scripting
Upgrades[11].ProjectID	CSCRIPT
Upgrades[11].Name	Filled in after Tier 3 upgrades
Upgrades[11].ID	Filled in after Tier 3 upgrades
Upgrades[11].Tier	3
Upgrades[11].Path	TECSERV-CACCFUN-TECCALL-CSCRIPT
Upgrades[13]	*Upgrades list entry
Upgrades[8].ProjectName	Call Scripting
Upgrades[8].ProjectID	CSCRIPT
Upgrades[8].Name	Filled in after Tier 3 upgrades
Upgrades[8].ID	Filled in after Tier 3 upgrades
Upgrades[8].Tier	3
Upgrades[8].Path	TECSERV-TROTASK-TECCALL-CSCRIPT
Upgrades[14]	*Upgrades list entry
Upgrades[14].ProjectName	Call Scripting
Upgrades[14].ProjectID	CSCRIPT
Upgrades[14].Name	Filled in after Tier 4 upgrades
Upgrades[14].ID	Filled in after Tier 4 upgrades
Upgrades[14].Tier	4
Upgrades[14].Path	TECSERV-CACCFUN-COMAGTA-CALLINT-CSCRIPT
Project[0]	*Project list entry
Project[0].Name	Technical Services
Project[0].ID	TECSERV
Project[0].Tier	0
…	Remainder of data
Project[1]	*Project list entry
Project[1].Name	Common Account Functions
Project[1].ID	CACCFUN
Project[1].Tier	1
…	Remainder of data
Project[2]	*Project list entry

Project[2].Name	Troubleshooting Tasks
Project[2].ID	TROTASK
Project[2].Tier	1
…	Remainder of data
Project[3]	*Project list entry
Project[3].Name	Common Agent Tasks
Project[3].ID	COMAGTA
Project[3].Tier	2
…	Remainder of data
Project[4]	*Project list entry
Project[4].Name	Tech Support Call
Project[4].ID	TECCALL
Project[4].Tier	2
…	Remainder of data
Project[5]	*Project list entry
Project[5].Name	Call Introductions
Project[5].ID	CALLINT
Project[5].Tier	3
…	Remainder of data
Project[6]	*Project list entry
Project[6].Name	Call Scripting
Project[6].ID	CSCRIPT
Project[6].Tier	4
…	Remainder of data

Figure 4-23: Upgrades and Project Sections of Dependency Data Model

For each project, the ratio of relevant Project[n] entries to Upgrades[n] entries is 1:n. This is because every time a project is matched to a change in its dependency, a new entry is created in the Upgrades[n] section of the data model, each having its own Tier value. In contrast, the Projects[n] section only has one entry per project and the Tier value within it gets updated whenever a new Upgrades[n] entry for that project is created. This results in Projects[n].Tier being equal to the largest Upgrades[n].Tier value relating to the

same project. This value helps us identify, for any given project, the last tier of upgrades and when the project will need to be snapshotted in order to continue on through any upgrades that may yet remain.

4.3.4 **Upgrade Algorithm**

Once again, with the completed dependency data model, the development team is presented with a tree diagram of all the impacted projects and they opt to leave all of them selected for upgrade with the new changes. With the only differences from case 2 being the less structured use of business data objects in the Tech toolkits as well as a unimpacted function from the Common Account Functions toolkit being leveraged by the Common Agent Tasks toolkit, the development activities during the upgrade process remain unimpacted. The only salient difference here is the number of tiers and the projects present within each.

There are two distinct options with regards to handling Tech Support Call, Call Introductions and Call Scripting, our three multi-tiered projects. They can either be upgraded in sections corresponding to each tier to which they belong, or their upgrade operations can be deferred to their last tier. Both of these options result in the multi-tier projects being fully upgraded, but the latter is a far more efficient solution. Upgrading each project in a piecemeal fashion is not an ideal solution as, in more complicated dependency structures, this can lead to redundant snapshotting activities and unnecessary work that will be overwritten by the end of the upgrade process. The only advantage to considering this approach is that it would require less logic in the upgrade algorithm.

The other option, deferring the upgrades to the last tier, would minimize the time in which the project is in flux (read partially upgraded). Doing so requires a bit of processing to be performed on the Upgrades[n] section of the dependency data model before beginning the upgrade process.

Deferring updates to the last tier in which they appear will be the method employed by the upgrade algorithm to address these multi-tier projects. This method requires only slight modification of the upgrade algorithm and is definitely more efficient in that it reduces the number of activities performed by the **Dependency Upgrade Agent** and the number of snapshots created within the BPM development environment. In this case, when a multi-tier project is first upgraded, the upgrade algorithm will compare the Upgrades[n].Tier value of an upgrade with the Project[n].Tier value of the corresponding project. The Project[n].Tier value should always be higher than or equal to the Upgrades[n].Tier value. When these two values do not match, the **DUA** will not perform the upgrade activity described in that Upgrades[n] entry. Instead, that entry of Upgrade[n].Tier will be updated to the latest value stored in Project[n].Tier. This will ensure that the upgrade is deferred until later in the process when all the required snapshots will be in place to process the upgrades.

When Project[n].Tier and Upgrade[n].Tier do match, the **DUA** then resumes standard behaviour for this project and provides the development team with the opportunity to address the feedback from Project[n].Build[n].Dep[n].Changes[n]. After completing their planned modifications, the development team signals to the **DUA** to proceed and the multi-tier project is finally snapshotted and ready to potentially become the upgrade to another toolkit or process app in the next tier.

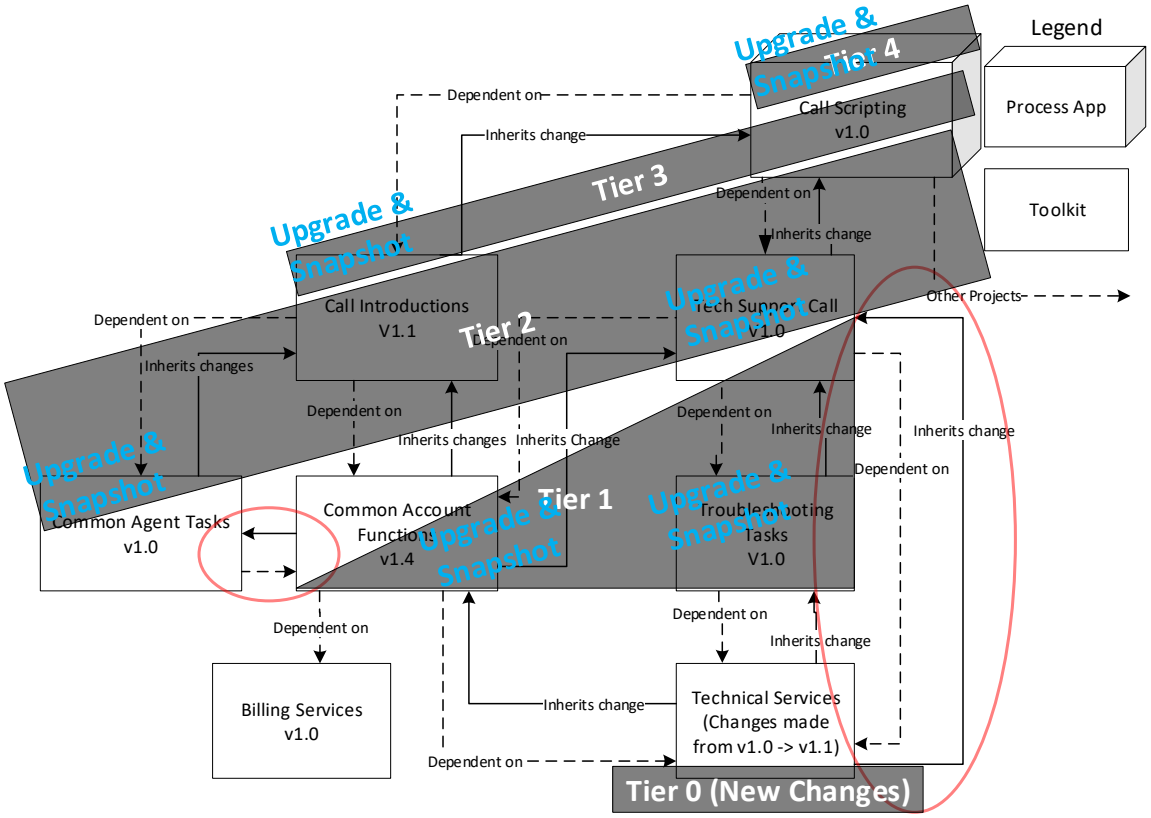

Figure 4-24: Upgrade Tiers with snapshot markers

We can see from the diagram that, despite the new configuration of tiers, the **DUA** is able to complete all of the desired upgrade operations while respecting a sequence that ensures their successful alignment to the new changes. The only additional complexity arises from the appearance of multi-tier projects which the **DUA** accommodates by suspending upgrade operations until the last tier relevant to that project is reached. These are denoted by the word "Upgrade & Snapshot" overlaid on our existing dependency diagram for case 3. This visually demonstrates the tier in which a particular multi-tiered project must be finalized and snapshotted to successfully continue operations.

4.3.5 Results

Case 3 serves to demonstrate the resiliency of the **Impact Assessment Engine** and the **Dependency Upgrade Agent**, particularly the upgrade algorithm within the latter. Development practices that commonly increase the complexity of maintenance and upgrade activities further down the line can be effectively analyzed by the **Impact Assessment Engine** in such a way as to gather all the information needed to successfully upgrade entire project ecosystems without accomplishing needless work or redundant activities. The counterproductive development practices shown here in case 3 represent just a small subset of what can be encountered, but applying the same assessment and algorithm should result in successful outcomes in all but the most poorly implemented projects. Case 4, the next and final case, will explore a scenario in which poor development practices have been used to such an extent that project versions become irreconcilable and cannot be properly upgraded.

4.4. Irredeemable dependency structure example

This case will explore a scenario where the dependency links established between projects are untenable. This is meant to demonstrate the limits of what this automated Impact Assessment and Dependency Management framework can handle in terms of compensating for bad development practices. It should be noted that it is not just the automated framework that cannot successfully upgrade the dependency structure in case 4. Human agents, such as developers would also be unable to successfully upgrade the projects in the scenario presented in case 4.

4.4.1 Overview

Once again, the same collection of toolkits and process apps used as in cases 2 and 3. The same changes have been made to the Check Connection Status service in the Technical Services toolkit. Also, the dependency relationships from case 3 are the starting point, but some new dependency relationships are added in a configuration that is impossible to upgrade while maintaining version matches throughout all the toolkit dependencies in the project ecosystem. The goal here is to demonstrate the framework's ability to recognize this situation before any upgrade activities have started and to provide a warning to the developers to help them pinpoint and manually resolve the issue.

4.4.2 Business Process Component Architecture

In this instance, the dependency diagram we are presented with is very similar to that of case 3, including the additional, "messy" dependencies that were added for complexity. However, the added complexity of those relationships was already addressed and they are not related to the impossibility of conducting the upgrades in this scenario.

Rather, the change at fault is circled in red at the center of the diagram. The assumption this time is that the development teams involved with implementing the Call Introductions and Tech Support Call toolkits are being too broad in the design of their toolkits. They are incorporating certain flow elements from each toolkit in the other in order to account for more call scenarios with their processes. Though this appears to reduce dependencies needed by future process apps and avoid some development work when leveraging these dependencies, it introduces a circular dependency which can never again be brought into version number alignment with the other project(s) in the dependency cycle. Additionally, one of these two opposing dependencies has also superseded an existing dependency of the Tech Support Call toolkit on the Common Account Functions toolkit. Those dependency lines are no longer present on the diagram.

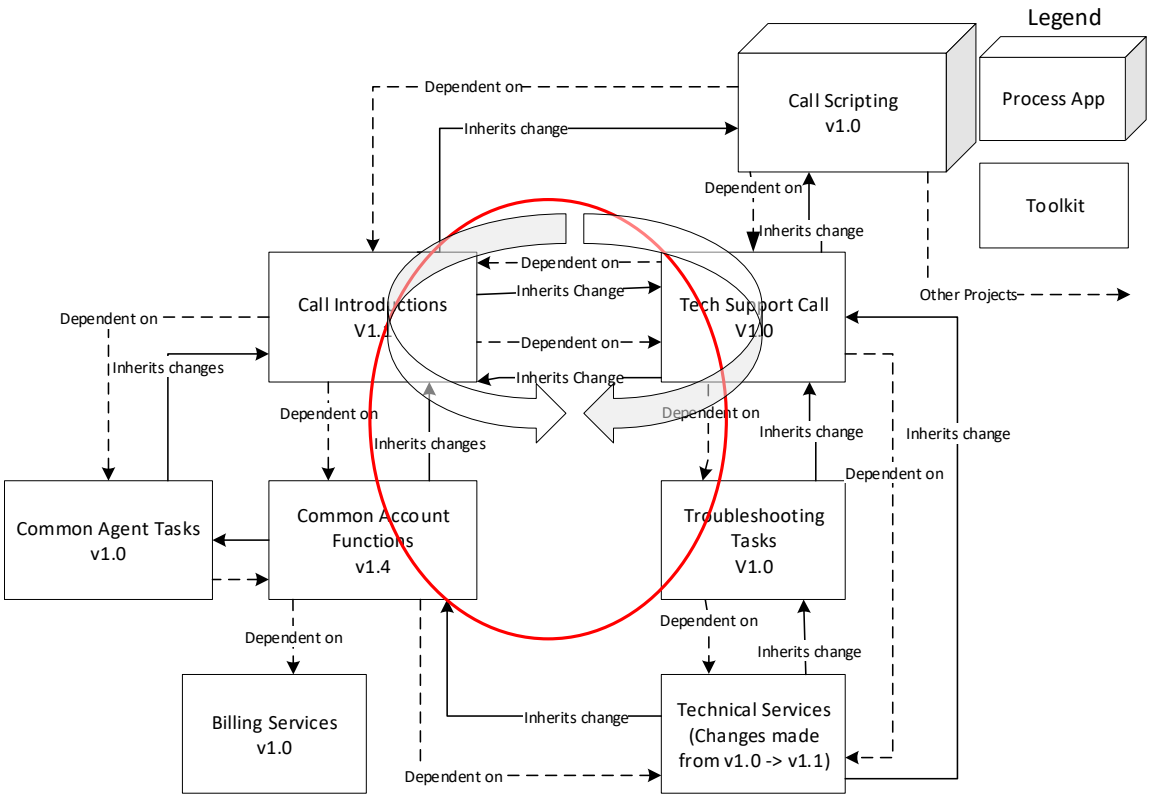

Figure 4-25: Call Scripting Process App dependency tree with circular dependency

Again, with Technical Services as the source of change and the upgrade starting point, following the “Inherits Change” lines reveals the impacted projects and the configuration of the upgrade tiers to which they belong. The tier 1 toolkits are the Common Account Functions, Troubleshooting Tasks and Tech Support Call toolkits. Another step along these lines yields a tier 2 composed of the Common Agent Tasks, Call Introductions and Tech Support Call toolkits as well as the Call Scripting process app.

The logical problem starts to emerge when the lines are followed a third time in order to establish tier 3. As both Call Introductions and Tech Support Call toolkits are present in tier 2 upgrades, then it follows that, being dependent on one another, tier 3 will include both Call Introductions and Tech Support Call toolkits, yet again. Since the presence of these toolkits begets the reappearance of these toolkits, an endless loop of dependency has been created.

Although it is possible to create consecutive snapshots of either of these toolkits in order to manually force an upgrade to a version that is functionally compatible with the rest of the projects, this will result in a permanent version mismatch warning and should not be tolerated in any BPM development shop. The remainder of the case will assume that any situation resulting in such a permanent version mismatch should, before any other action is taken, be redesigned in such a way as to remove the chain of dependencies causing the circular relationship pattern.

4.4.3 **Dependency Data Model**

Once again, the **Impact Assessment Engine** retrieves the projects contained in the BPM database. The development team indicates Technical Services v1.1 as the snapshot containing the changes for the upgrade.

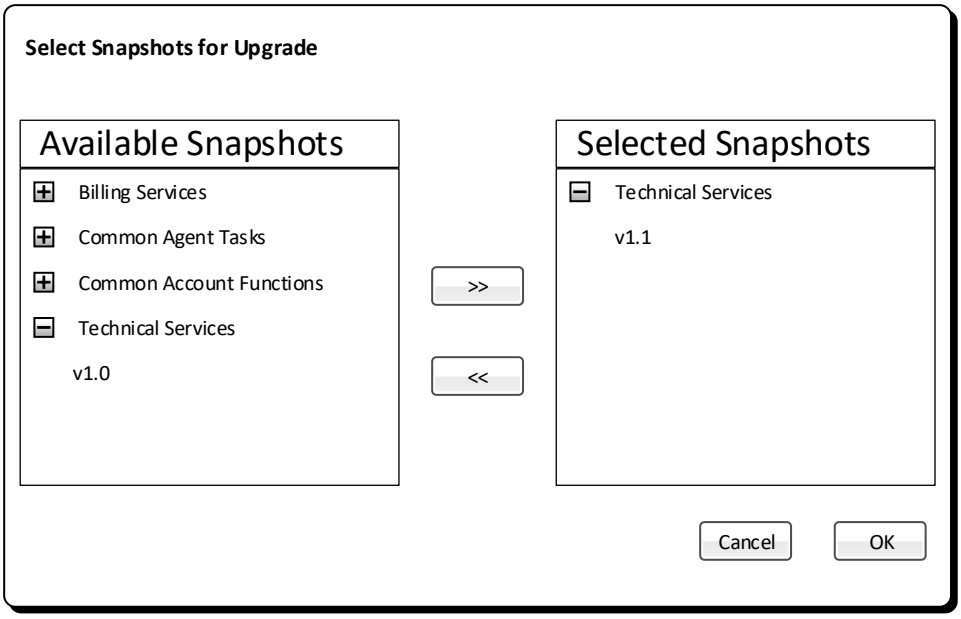

Figure 4-26: Change Selection Screen

As the dependency matching occurs, it becomes clear by the third tier of upgrades that something is amiss. In fact, without a mechanism designed to detect circular dependencies, the dependency matching process of the **Impact Assessment Engine** will be unable to reach its end criteria. In the following diagram representing the upgrade tiers for case 4, it is apparent that there are infinite, identical tiers from tier 3 onwards.

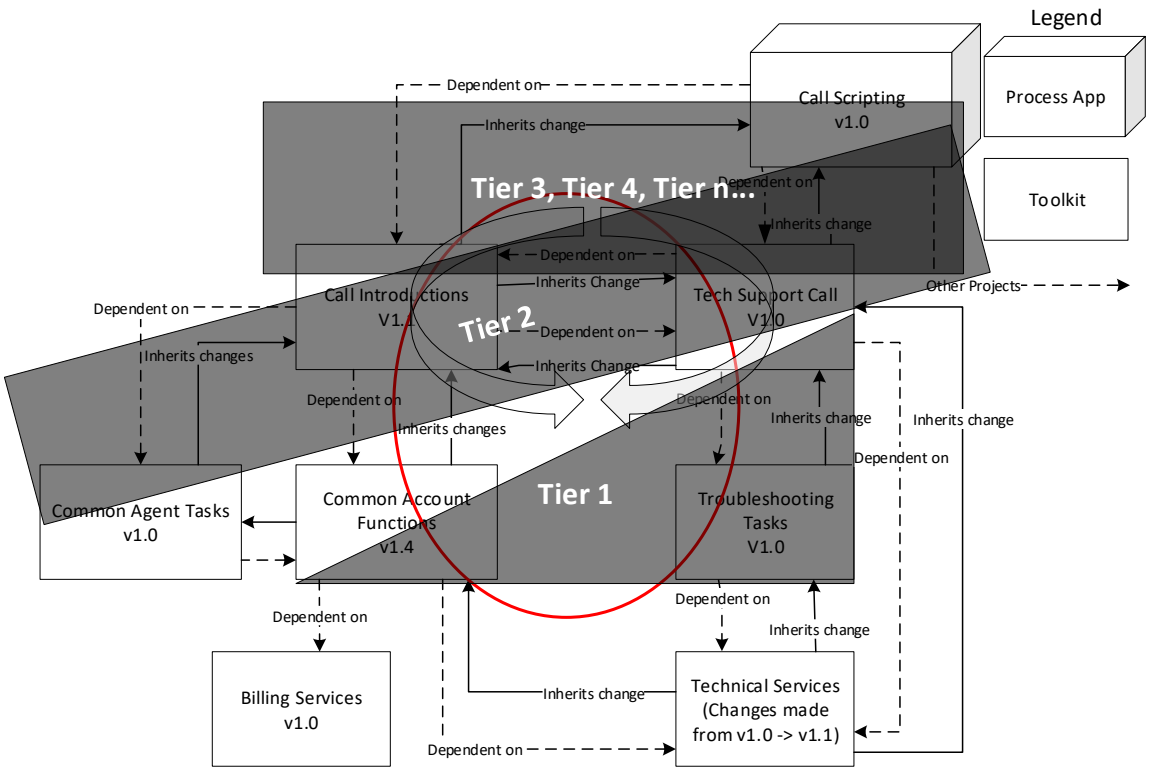

Figure 4-27: Upgrade Tiers

In order to stop the impact assessment process from continuing indefinitely, a checking mechanism needs to be implemented to identify circular dependencies quickly. If any such impossible dependencies are detected, the **IAE** must either prune the offending toolkits from the Upgrades[n] list or must abort the impact assessment process entirely. The first step is determining how the framework will identify a circular dependency situation.

When looking at the dependency discovery algorithm in the **Impact Assessment Engine**, case 3 demonstrated that it is indeed possible to have the same toolkit or process app appear several times throughout different upgrade tiers. However, if each "path" through the dependency structure is inspected separately, it then becomes very simple to define the minimum requirements for a circular dependency situation. When looking at

any individual path of dependencies through the projects, a circular dependency can be defined as a dependency on any toolkit that has previously been encountered along that path. This definition is particularly suitable as it provides the ability to identify circular dependencies from the most basic situations involving two toolkits to those involving long chains of toolkits spanning multiple tiers.

In order to achieve this analysis while the dependency discovery is performed, the progression of our upgrade paths needs to be stored within our Upgrades[n] data structure. Upgrades[n].Path is the field used for this purpose. The **IAE** uses the entries previously made in the Upgrades[n] data, in order to continue discovering the next tier of dependencies. When searching for dependencies corresponding to these entries, the **IAE** will compare any matching project IDs with the Upgrades[n].Path string. If the project ID string can be found within Upgrades[n].Path then we know that this dependency is the offending link that is creating a circular dependency within our BPM project ecosystem. This would also indicate that it is impossible to move forward with a proper upgrade process within the current configuration of projects. Although it will not be possible to move on to the upgrading portion of the framework. It is still possible to provide some value to the development team with the results of the impact assessment.

If, when encountering these circular issues, the corresponding Upgrades[n] entries are tagged as such, then the framework can avoid continuing the discovery process down these specific paths that result in infinite loops. With the rest of the discovery proceeding unimpeded, the dependency model will still get filled with the remainder of the dependency relationship data. It will also be possible for the framework to present the developers with a readout of the impact assessment results with the problematic, circular

dependencies highlighted for the developers to focus on correcting. This tagging of Upgrades[n] entries can be as simple as adding string “DEPLOOP” to the aforementioned Upgrades[n].Path along with the project ID of the dependency creating the circular link. Checking for the presence of this string is an additional step that is performed when searching the project IDs in the Path field for cycles. If “DEPLOOP” is encountered, then the DUA discontinues the dependency search activities for that entry of Upgrades[n].

Fields	Value
Upgrades[0]	*Upgrades list entry
Upgrades[0].ProjectName	Technical Services
Upgrades[0].ProjectID	TECSERV
Upgrades[0].Name	Technical Services v1.1
Upgrades[0].ID	7526c879-bfed-4e52-b963-dddd3cce0635
Upgrades[0].Tier	0
Upgrades[0].Path	TECSERV
Upgrades[1]	*Upgrades list entry
Upgrades[1].ProjectName	Common Account Functions
Upgrades[1].ProjectID	CACCFUN
Upgrades[1].Name	Filled in after Tier 1 upgrades
Upgrades[1].ID	Filled in after Tier 1 upgrades
Upgrades[1].Tier	1
Upgrades[1].Path	TECSERV-CACCFUN
Upgrades[2]	*Upgrades list entry
Upgrades[2].ProjectName	Troubleshooting Tasks
Upgrades[2].ProjectID	TROTASK
Upgrades[2].Name	Filled in after Tier 1 upgrades
Upgrades[2].ID	Filled in after Tier 1 upgrades
Upgrades[2].Tier	1
Upgrades[2].Path	TECSERV-TROTASK
Upgrades[3]	*Upgrades list entry
Upgrades[3].ProjectName	Tech Support Call
Upgrades[3].ProjectID	TECCALL
Upgrades[3].Name	Filled in after Tier 1 upgrades
Upgrades[3].ID	Filled in after Tier 1 upgrades

Upgrades[3].Tier	1
Upgrades[3].Path	TECSERV- TECCALL
Upgrades[4]	*Upgrades list entry
Upgrades[4].ProjectName	Common Agent Tasks
Upgrades[4].ProjectID	COMAGTA
Upgrades[4].Name	Filled in after Tier 2 upgrades
Upgrades[4].ID	Filled in after Tier 2 upgrades
Upgrades[4].Tier	2
Upgrades[4].Path	TECSERV-CACCFUN- COMAGTA
Upgrades[5]	*Upgrades list entry
Upgrades[5].ProjectName	Call Introductions
Upgrades[5].ProjectID	CALLINT
Upgrades[5].Name	Filled in after Tier 2 upgrades
Upgrades[5].ID	Filled in after Tier 2 upgrades
Upgrades[5].Tier	2
Upgrades[5].Path	TECSERV-CACCFUN-CALLINT
Upgrades[6]	*Upgrades list entry
Upgrades[6].ProjectName	Call Introductions
Upgrades[6].ProjectID	CALLINT
Upgrades[6].Name	Filled in after Tier 2 upgrades
Upgrades[6].ID	Filled in after Tier 2 upgrades
Upgrades[6].Tier	2
Upgrades[6].Path	TECSERV-TECCALL-CALLINT
Upgrades[7]	*Upgrades list entry
Upgrades[7].ProjectName	Call Scripting
Upgrades[7].ProjectID	CSCRIPT
Upgrades[7].Name	Filled in after Tier 2 upgrades
Upgrades[7].ID	Filled in after Tier 2 upgrades
Upgrades[7].Tier	2
Upgrades[7].Path	TECSERV-TECCALL-CSCRIPT
Upgrades[8]	*Upgrades list entry
Upgrades[8].ProjectName	Tech Support Call
Upgrades[8].ProjectID	TECCALL
Upgrades[8].Name	Filled in after Tier 2 upgrades
Upgrades[8].ID	Filled in after Tier 2 upgrades
Upgrades[8].Tier	2
Upgrades[8].Path	TECSERV-TROTASK-TECCALL

Upgrades[9]	*Upgrades list entry
Upgrades[9].ProjectName	Call Introductions
Upgrades[9].ProjectID	CALLINT
Upgrades[9].Name	Filled in after Tier 3 upgrades
Upgrades[9].ID	Filled in after Tier 3 upgrades
Upgrades[9].Tier	3
Upgrades[9].Path	TECSERV-CACCFUN-COMAGTA-CALLINT
Upgrades[10]	*Upgrades list entry
Upgrades[10].ProjectName	Call Scripting
Upgrades[10].ProjectID	CSCRIPT
Upgrades[10].Name	Filled in after Tier 3 upgrades
Upgrades[10].ID	Filled in after Tier 3 upgrades
Upgrades[10].Tier	3
Upgrades[10].Path	TECSERV-CACCFUN-CALLINT-CSCRIPT
Upgrades[11]	*Upgrades list entry
Upgrades[11].ProjectName	Tech Support Call
Upgrades[11].ProjectID	TECCALL
Upgrades[11].Name	Filled in after Tier 3 upgrades
Upgrades[11].ID	Filled in after Tier 3 upgrades
Upgrades[11].Tier	3
Upgrades[11].Path	TECSERV-CACCFUN-CALLINT-TECCALL
Upgrades[12]	*Upgrades list entry
Upgrades[12].ProjectName	Call Introductions
Upgrades[12].ProjectID	CALLINT
Upgrades[12].Name	Filled in after Tier 3 upgrades
Upgrades[12].ID	Filled in after Tier 3 upgrades
Upgrades[12].Tier	3
Upgrades[12].Path	TECSERV-TECCALL-CALLINT-TECCALL-DELOOP
Upgrades[13]	*Upgrades list entry
Upgrades[13].ProjectName	Call Scripting
Upgrades[13].ProjectID	CSCRIPT
Upgrades[13].Name	Filled in after Tier 3 upgrades
Upgrades[13].ID	Filled in after Tier 3 upgrades
Upgrades[13].Tier	3
Upgrades[13].Path	TECSERV-TECCALL-CALLINT-CSCRIPT
Upgrades[14]	*Upgrades list entry
Upgrades[14].ProjectName	Call Introductions

Upgrades[14].ProjectID	CALLINT
Upgrades[14].Name	Filled in after Tier 3 upgrades
Upgrades[14].ID	Filled in after Tier 3 upgrades
Upgrades[14].Tier	3
Upgrades[14].Path	TECSERV-TROTASK-TECCALL-CALLINT
Upgrades[15]	*Upgrades list entry
Upgrades[15].ProjectName	Call Scripting
Upgrades[15].ProjectID	CSCRIPT
Upgrades[15].Name	Filled in after Tier 3 upgrades
Upgrades[15].ID	Filled in after Tier 3 upgrades
Upgrades[15].Tier	3
Upgrades[15].Path	TECSERV-TROTASK-TECCALL-CSCRIPT
Upgrades[16]	*Upgrades list entry
Upgrades[16].ProjectName	Call Introductions
Upgrades[16].ProjectID	CALLINT
Upgrades[16].Name	Filled in after Tier 4 upgrades
Upgrades[16].ID	Filled in after Tier 4 upgrades
Upgrades[16].Tier	4
Upgrades[16].Path	TECSERV-CACCFUN-COMAGTA-CALLINT-CSCRIPT
Upgrades[17]	*Upgrades list entry
Upgrades[17].ProjectName	Tech Support Call
Upgrades[17].ProjectID	TECCALL
Upgrades[17].Name	Filled in after Tier 4 upgrades
Upgrades[17].ID	Filled in after Tier 4 upgrades
Upgrades[17].Tier	4
Upgrades[17].Path	TECSERV-CACCFUN-COMAGTA-CALLINT-TECCALL
Upgrades[10]	*Upgrades list entry
Upgrades[18].ProjectName	Call Introductions
Upgrades[18].ProjectID	CALLINT
Upgrades[18].Name	Filled in after Tier 4 upgrades
Upgrades[18].ID	Filled in after Tier 4 upgrades
Upgrades[18].Tier	4
Upgrades[18].Path	TECSERV-CACCFUN-CALLINT-TECCALL-CALLINT-DELOOP
Upgrades[19]	*Upgrades list entry
Upgrades[19].ProjectName	Call Scripting

Upgrades[19].ProjectID	CSCRIPT
Upgrades[19].Name	Filled in after Tier 4 upgrades
Upgrades[19].ID	Filled in after Tier 4 upgrades
Upgrades[19].Tier	4
Upgrades[19].Path	TECSERV-CACCFUN-CALLINT-TECCALL-CSCRIPT
Upgrades[20]	*Upgrades list entry
Upgrades[20].ProjectName	Call Scripting
Upgrades[20].ProjectID	CSCRIPT
Upgrades[20].Name	Filled in after Tier 4 upgrades
Upgrades[20].ID	Filled in after Tier 4 upgrades
Upgrades[20].Tier	4
Upgrades[20].Path	TECSERV-TROTASK-TECCALL-CALLINT-CSCRIPT
Upgrades[21]	*Upgrades list entry
Upgrades[21].ProjectName	Tech Support Call
Upgrades[21].ProjectID	TECCALL
Upgrades[21].Name	Filled in after Tier 4 upgrades
Upgrades[21].ID	Filled in after Tier 4 upgrades
Upgrades[21].Tier	4
Upgrades[21].Path	TECSERV-TROTASK-TECCALL-CALLINT-TECALL-DELOOP
Upgrades[22]	*Upgrades list entry
Upgrades[22].ProjectName	Call Introductions
Upgrades[22].ProjectID	CALLINT
Upgrades[22].Name	Filled in after Tier 5 upgrades
Upgrades[22].ID	Filled in after Tier 5 upgrades
Upgrades[22].Tier	5
Upgrades[22].Path	TECSERV-CACCFUN-COMAGTA-CALLINT-TECCALL-CALLINT-DELOOP
Upgrades[23]	*Upgrades list entry
Upgrades[23].ProjectName	Call Scripting
Upgrades[23].ProjectID	CSCRIPT
Upgrades[23].Name	Filled in after Tier 5 upgrades
Upgrades[23].ID	Filled in after Tier 5 upgrades
Upgrades[23].Tier	5
Upgrades[23].Path	TECSERV-CACCFUN-COMAGTA-CALLINT-TECCALL-CSCRIPT
Project[0]	*Project list entry

Project[0].Name	Technical Services
Project[0].ID	TECSERV
Project[0].Tier	0
…	Remainder of data
Project[1]	*Project list entry
Project[1].Name	Common Account Functions
Project[1].ID	CACCFUN
Project[1].Tier	1
…	Remainder of data
Project[2]	*Project list entry
Project[2].Name	Troubleshooting Tasks
Project[2].ID	TROTASK
Project[2].Tier	1
…	Remainder of data
Project[3]	*Project list entry
Project[3].Name	Common Agent Tasks
Project[3].ID	COMAGTA
Project[3].Tier	2
…	Remainder of data
Project[4]	*Project list entry
Project[4].Name	Tech Support Call
Project[4].ID	TECCALL
Project[4].Tier	4
…	Remainder of data
Project[5]	*Project list entry
Project[5].Name	Call Introductions
Project[5].ID	CALLINT
Project[5].Tier	3
…	Remainder of data
Project[6]	*Project list entry
Project[6].Name	Call Scripting
Project[6].ID	CSCRIPT
Project[6].Tier	5
…	Remainder of data

Figure 4-28: Upgrades and Project Sections of Dependency Data Model

4.4.4 Upgrade Algorithm

Given that the **Impact Assessment Engine** has determined that it is impossible to complete the upgrades on this project in a safe manner, there is no need to describe the upgrades portion of case 4. Upon discovering a circular dependency, indicated by the "DELOOP" entries that appear in Upgrades[n].Path, the framework will display the information regarding the toolkits involved in this bad dependency pattern and will then block developers from proceeding with upgrade operations.

4.4.5 Results

Case 4 is an extreme example in which to test this framework. Despite being unable to proceed on to the upgrading portion of the framework, this case can still be considered a success due to the handling of a circular dependency situation. These situations are impossible to resolve when trying to upgrade whether the operations are performed manually by developers or programmatically by a framework such as this. The ability to recognize this situation and to exit out gracefully is all that can be expected of the Impact Assessment and Dependency Management Framework. Developers who find themselves at this point will need to put their efforts towards reworking their dependency relationships should they want to be able to proceed.

4.5. Chapter Summary

This chapter has provided several realistic BPM dependency management examples with varying degrees of difficulty. The examples are based on the BPM 8.5 development suite and were specifically chosen to progressively test and validate the proposed framework starting from basic functions to core functionality and, finally, to advanced features such as intelligent upgrade planning and recognition of fatal errors.

Example 1 demonstrated one iteration of dependency discovery and the upgrade/snapshot activities necessary to upgrade a single dependency in a process application. Example 2 combined these elements into an iterative process to demonstrate the framework's potential to scale up these activities in order to assess and upgrade larger groups of projects with multiple levels of dependency relationships. Example 3 increased the complexity in order to validate the robustness of the analysis and dependency upgrade logic. Finally, example 4 acknowledged the impossibility of successfully resolving every dependency upgrade scenario and tests the framework's ability to recognize impossible scenarios and exit out as gracefully as possible.

Chapter 5. Evaluation

This chapter will evaluate the performance of the proposed impact assessment and dependency management framework. The framework will be evaluated against other solutions providing similar functionality for general software and BPM development activities. The framework will also be assessed in terms of how well it addressed each of the four cases presented in Chapter 4.

5.1. Related Works/Dependency Management Tools

This section presents a comparison between the proposed framework and the related works discussed in chapter 2.4. The table below compares our proposed framework with the two related works (IBM BPM, Maven) based on the 8 evaluation criteria we identified and discussed in Chapter 3.1. The evaluation is our subjective opinion, having worked extensively with all 3. Green indicates that a criterion is mostly addressed and is satisfactory. Yellow indicates it is partially or somewhat addressed but is barely satisfactory. Red indicates that the criteria is ignored or barely addressed and is unsatisfactory.

Criteria	IBM BPM	Maven	Proposed Framework
Visual	Provides consistent visual elements, but of low informational value. Mostly lists for viewing snapshots and dependencies. Lacking visuals to describe information beyond a single project level.	Mostly code-based interaction with textual output making visuals inapplicable. Visual elements not necessarily present and limited to those found in the Java IDE being used. Maven itself contains no visuals.	Framework specifies graphical interface elements for capturing upgrade information. Results of inter-project dependency analysis are displayed as hierarchy tree for developers' comprehension and upgrade process monitoring.

Low Learning Curve	Dependency upgrade functionality packaged with BPM suite. No learning curve to existing BPM devs.	Non-Comparable to BPM suites, being a more generally applicable, code-based tool. Estimation is that Maven is more difficult to learn for a Java coder than framework would be for BPM Dev.	Supplementary upgrade functionality built using BPM suite resources. No new controls or concepts to learn. New features use config wizard to ease any learning challenges.
Accurate Dependency Analysis	Provides direct dependencies for each project. Developers responsible for tracking dependencies and sequencing upgrades. Room for error.	Identifies project dependencies and analyses the relationships between them including sequencing. Applies to Java development.	Identifies dependency structure and sequencing of dependencies. Pre-emptive identification of upgrade errors.
Recursive Dependency Discovery	No iterative or recursive element to dependency discovery. Only direct dependents can be viewed for each project within editing view.	Allows for dependency identification across multi-leveled relationships through the use of transitive dependency concept.	Allows for dependency identification across multi-leveled relationships through indefinitely iterative dependency discovery process.
Robust Upgrade Algorithm	All necessary information is present, but visible in different locations. Developers must consult snapshot screen and designer screens of each project involved. Manual tracking of dependencies and upgrades creates chance of errors.	Not geared towards upgrade planning, but rather build dependency organization. Maven outputs could be used to assist in the planning of upgrade operations, but there is no explicit impact assessment and nor guided upgrade process.	Produces viable upgrade plan from information in the dependency meta-model. Plan includes considerations of upgrade parallelization and sequencing. Greatly reduced navigation time. Early discovery of errors and cancelation of upgrades when necessary.
Development Effort	No batch snapshot or upgrade functionality. Can only match automation with perfect knowledge of projects, no mistakes and enough developers to maximize parallelization. Much greater effort in medium/high complexity	Initial effort required for setup, configuration and identification of dependencies. Further development incurs a small overhead for keeping .pom files updated. Reduces effort and errors when creating new builds.	One developer required for majority of upgrade procedure. Errors actively detected and flagged by framework. Maximum parallelization implemented. Development adjustments flagged, but effort to resolve is same.

	dependency relationships.		
Setup and Maintenance Costs	Baked-in functionality, no additional development or administration costs. Upgrades affecting multiple projects are tedious and prone to error, increasing cost of project upgrade operations.	Cost is man-hours for setup, configuration. Further development incurs a small overhead for keeping .pom files updated. Reduction in build errors should save sufficient dev time to justify costs.	All necessary operations performed when running the framework from any PC on the network, no maintenance costs. Only additional cost would be initial installation and configuration. Lifetime development hour savings more than compensates.
Standardization	Standardized for BPMN 2.0 notation and for process reuse paradigm. However, these standard elements remain a feature within a proprietary development suite. Upgrade operations cannot interact with other BPM suites.	Helps standardize dependency relationships for Java project builds. Can be used in various IDEs, but not with any other programming languages. Not quite analogous to framework being used across BPM suites.	Ability to be tailored to other BPM suites based on BPMN 2.0 and process re-use. Same algorithm can be applied with substitute API call or other invocation method through the communication module described in section 3.2.

Figure 5-1: Framework and Related works evaluation grid

IBM BPM 8.5 is representative of the status quo for BPM development environments. Functionality surrounding impact assessment and dependency management is limited to the most basic features which puts the onus on BPM developers to manually maintain dependencies and coordinate upgrade activities. This makes IBM BPM 8.5 of limited usefulness with respect to dependency analysis, recursive discovery and upgrade planning. This ultimately results in increased costs when development teams expend more effort on dependency maintenance as BPM environments scale up. Despite the heavily graphical nature of BPM development, BPM 8.5 does not leverage any visual elements in its dependency management tools. There is no appreciable learning curve for BPM developers as this is default functionality of the BPM environment. Being a proprietary

development platform, BPM 8.5 adheres to standard BPM development paradigms but is not designed to interface with any other BPM development tools.

Maven serves as an example of how dependency management can be enhanced in software development. It does not introduce any visual elements and presents a moderate learning curve to developers. It is also not applicable to most BPM environments since it is text-based and BPM environments are graphical interface or DB-based. However, it is excellent for traditional text-based development environments, and it serves well to highlight the lack of dependency management and impact analysis support in current tools like BPM 8.5. Through the use of the transitive dependency concept, Maven allows dependency analysis to be performed across all levels of large collections of dependent projects. This allows Maven to perform recursive discovery and results in accurate analysis of dependencies. The analysis produced by Maven can be used to assist developers in conducting dependency upgrade activities and reduce the effort required to complete them, but falls short of offering automation or a guided upgrade process. There is little cost to using Maven, only effort for initial implementation and maintenance of .pom files are required by developers. Maven is also standardized in the sense that it will work with any project written in Java regardless of the development environment used.

Our proposed framework is an attempt to integrate Maven-like capabilities into BPM development environments and enhance them, where possible. Similar to Maven, the framework is able to recursively discover dependent projects in order to obtain a complete list impacted projects across any number of dependency tiers. This allows it to surpass the performance of IBM 8.5 with respect to accurate dependency analysis and recursive dependency discovery. The framework takes things a step further by introducing a guided

and automated upgrade process to reduce errors during upgrade operations and to speed them up considerably. This allows the framework to surpass the assistance offered by Maven with respect to having a robust upgrade algorithm and reducing development effort with respect to the BPM 8.5 status quo. The presence of visual elements in the framework maintains familiarity and enhances developer comprehension of the dependency structures with which they are working. This ease of comprehension combined with the guided approach to upgrade operations result in the framework presenting a very low learning curve to existing BPM 8.5 developers. The framework is also advantageous from a setup and maintenance cost standpoint. Aside from installing the framework on a machine with access to the BPM server, there are no significant setup costs. The framework reads the current state of the BPM environment from the server every time it is run, eliminating costs of maintaining the framework itself. With respect to standardization across BPM suites, the framework attempts to address this through the communication module discussed in section 3.2. This layer of abstraction between the internal logic of the framework and the BPM upgrade operations being performed allows the framework to be tailored for use with other BPM suites regardless of differences in APIs and coding language provided that the targeted BPM suites employ similar principles of process reuse and dependency linking between projects.

5.2. Case Evaluation

In this section we evaluate our framework across each of the examples from chapter 4 based on the criteria identified in chapter 3.1.

Criteria	Case 1	Case 2	Case 3	Case 4
Visual	Graphical user interface controls. Visually displays the impact of changes and the extent of upgrade operations.	Graphical user interface controls. Visually displays the impact of changes and the extent of upgrade operations.	Graphical user interface controls. Visually displays the impact of changes and the extent of upgrade operations.	Graphical user interface controls. Visually displays the impact of changes and the location of circular dependencies.
Low Learning Curve	Framework easily mastered by existing BPM developers. New steps presented graphically and within guided "wizard".	Framework easily mastered by existing BPM developers. New steps presented graphically and within guided "wizard".	Framework easily mastered by existing BPM developers. New steps presented graphically and within guided "wizard".	Framework easily mastered by existing BPM developers. New steps presented graphically and within guided "wizard".
Accurate Dependency Analysis	Can complete dependency analysis without difficulty.	Can complete dependency analysis without difficulty.	Can complete dependency analysis despite convoluted dependency structure. Produced output requiring update to upgrade algorithm	Cannot complete dependency analysis for circularly dependent projects. Framework can match performance of regular work performed by developer.
Recursive Dependency Discovery	N/A Recursive Dependency Discovery not necessary for single tier	Recursive discovery is successful within multi-tiered collection of BPM projects.	Recursive discovery is successful within multi-tiered collection of BPM projects. Despite	Circular dependencies led to infinite loop of discovery. Modified framework halts analysis when circular

	dependency relationship.		projects appearing in multiple tiers.	dependencies are detected in Upgrades[n].Path
Robust Upgrade Algorithm	Identifies the necessary upgrade. Also provides development recommendations to accommodate new changes.	Identifies all necessary upgrades to fully propagate change across multiple tiers of upgrades with development recommendations.	Initially, algorithm produced redundant upgrade and snapshot activities. Algorithm modified to optimize resulting operations.	Framework is unable to upgrade all impacted projects. Projects directly/indirectly dependent on circular dependencies cannot be upgraded. This matches performance of human developer.
Development Effort	Using framework introduces additional config wizard steps when only a single change is needed. Framework provides additional certainty.	Automation and organization of framework more than offset effort of additional config wizard steps. Framework enhances reliability and avoids errors.	Framework provides automation, organization and enhanced reliability. Helps avoid errors and unnecessary upgrade operations.	Despite inability to perform a complete upgrade, framework can provide guidance for resolution of circular dependencies.
Setup and Maintenance Costs	Man-hours using framework are similar to manually upgrading in BPM suite. No opportunity to offset setup and maintenance costs.	Automation produces more benefits over larger set of upgrade operations. Time savings sufficient to break even with framework costs.	Automation and improved organization of framework can provide substantial time savings by avoiding errors and rework. Framework costs quickly recuperated.	Automation and organization are still potential benefits. The early detection of circular dependencies avoids costly rework and quickly offsets framework costs.
Standardization	Case based on IBM BPM 8.5 Standardization assumed through the use of communication module discussed in 3.2	Case based on IBM BPM 8.5 Standardization assumed through the use of communication module discussed in 3.2	Case based on IBM BPM 8.5 Standardization assumed through the use of communication	Case based on IBM BPM 8.5 Standardization assumed through the use of communication module discussed in 3.2

			module discussed in 3.2	

Figure 5-2: Case evaluation grid

Some aspects of the framework remain constant across the four cases considered in chapter 4. In all cases, the framework is able to visually represent the results of impact assessment and use these visuals to assist developers with upgrade planning. In case 4, the framework can also display the location of circular dependencies that are discovered during impact assessment. The framework also presents a low learning curve for BPM developers regardless of the scenario in which it is used. The inputs required from developers are minimal and the "wizard" style of the framework execution provides developers with step by step guidance based on the context of the upgrades that they are currently performing. The same difficulties can occur for developers whether they are using the framework or not with the only difference being that the framework provides analysis and guidance on how to resolve any issues being encountered. Standardization is the last element to be common across all cases, but did not feature explicitly as a part of these examples. It is assumed that the framework can be adapted for use with other BPM suites through the implementation of a communications module to translate the operations performed by the framework into the API calls specific to the BPM suite being considered.

Moving on to the individual cases, case 1 tested the basic units of functionality that are needed for the impact assessment and dependency upgrade operations. The framework was able to accurately identify the lone, dependent project requiring upgrade. The simple nature of this case did not require the use of recursive dependency discovery. Therefore, this capability could not be verified using case 1. The upgrade algorithm was able to complete upgrade operations based off of the information produced during the assessment.

This includes producing a list of recommended developments activities based on the new changes introduced in the upgrade. This specific case is so simple that there is no opportunity for the framework to provide much benefit with respect to reducing effort for developers. Running the framework when only a single upgrade operation is required represents slightly more work than manually performing the upgrade operations. With little room to provide improved functionality in this case, the minimal setup costs of the framework make it unfavourable to use from a cost perspective.

Case 2 increases the difficulty by introducing a larger number of projects which, in turn, increases the number of dependency relationships that can be managed. Faced with the larger dependency structure and more dependency levels, the framework maintains its ability to accurately assess and identify all of the projects impacted by the new dependency upgrade. The presence of projects that are indirectly dependent on the initial dependency upgrade is currently handled by the framework using recursive dependency discovery and results in five dependencies being identified. The upgrade algorithm is able to successfully interpret the analysis results and performs the upgrade activities with the correct sequencing while pausing and providing development recommendations at all appropriate points. Given the higher volume of upgrade operations required, the framework significantly reduces the development effort required by automating upgrade operations and by avoiding errors and rework related to forgotten or out-of-sequence upgrades. From a cost perspective, using the framework no longer presents a disadvantage as the cost of setup is now offset by reduced developer effort which translates to reduced developer hours.

Case 3 maintains a larger set of dependent projects, but adds complication by introducing additional dependencies that are irregular and do not respect the strict hierarchy present in case 2. The framework was able to assess the dependency structures present in case 3, though it was observed that certain projects appeared multiple times in different tiers of the assessment output. The framework used recursive dependency discovery to successfully traverse the entire dependency structure and accurately identify all projects requiring upgrade activities. Case 3 presented a challenge for the upgrade algorithm as projects appearing in multiple tiers of upgrades was not an anticipated outcome. When first working through case 3, the upgrade algorithm would directly translate the results of the assessment into the upgrade operations to be performed. Though this did not break anything, it did lead the framework to perform many redundant upgrade and snapshot operations that had to be discarded as they produced project snapshots that were only partially upgraded. This challenge was addressed by adding a "Tier" field to the Upgrades[n] data structure and adding logic to the upgrade algorithm that checks this field against Project[n].Tier. This change allows the upgrade algorithm to identify when the final upgrades on a project have been performed and it is time to take the snapshot. In such a case, where irregular dependencies make developers prone to errors, the framework goes a long way to reducing development effort. Advanced analysis ensures that developers are planning only the necessary upgrade operations and in the correct order while automation reduces the effort involved in performing these upgrade activities. The increased utility of the framework in this case also makes the cost situation much more favourable with the setup costs now being easily surpassed by the value of the assistance that the framework offers developers.

Case 4, the final case, intentionally ups the difficulty to the point of challenging the capability the framework's core functionality. It introduces a circular dependency among the two projects involved in the case. Immediately, serious issues with the **Impact Assessment Engine** became apparent. With no mechanism to detect the presence of a circular dependency, the framework continues assessing the offending projects in an endless loop. This serious error led to the inclusion of an Upgrades[n].Path variable in the dependency data model. This new field allows newly discovered dependents to be compared to previous projects on the upgrade path. The upgrade algorithm was modified to leverage this path data in order to identify circular dependencies and cease operations on them. Following this, the framework was then able to accurately assess the projects in case 4 without falling into an endless loop during recursive discovery. The upgrade algorithm was robust enough to handle the increased numbers of relatively disordered dependency relationships between projects, but there is no way to programmatically resolve circular dependencies when they are discovered. The framework could still be used to upgrade the projects not directly/indirectly dependent on the circularly dependent projects, but it cannot upgrade all projects while maintaining acceptable development standards. Still, the framework provides extra clarity and guidance to developers when they previously had no such assistance, thus reducing the effort of detecting circular dependencies and the upgrade rework required when they are discovered mid-work. Less development hours spent reactively addressing dependency issues results in continued cost savings for BPM development teams that easily offset any framework setup costs.

5.3. Assumptions, Limitations and Threats to Validity

It is not possible to propose new solutions to research problems without making certain assumptions about the manner and context in which it will be used. It is equally true that no proposed solution is without its own flaws and limitations. In the spirit of the design and research methodology, the following section will discuss the assumptions, limitations and threats to validity surrounding this thesis and the framework it proposes.

5.3.1 **Assumptions**

Several assumptions needed to be made in order to simplify the topic of impact assessment and dependency management in BPM sufficiently to make designing a framework a manageable endeavor.

1. It is assumed for the purposes of this framework that an organization would have all of its business process development centralized on a single BPM suite. Dependency to other internal or external systems through web services or event messages is broad topic and enough out of scope to warrant its own discussion.

2. BPM 8.5 was the only BPM suite used as the basis for the proposed framework, despite standardization across BPM suites being a stated objective. This thesis assumes that other BPM suites possess the API methods required for the framework to gather dependency data and perform the necessary upgrade operations. It also assumes that there is a similar mechanism to processes and toolkits as the base components for tracking dependencies.

3. It is assumed that developers who have sufficient knowledge to work in BPM are also generally skilled enough with computers to navigate a guided, “wizard”-type,

interface. It is also assumed that they are familiar enough with dependency management in BPM, that they will understand the vocabulary and information required in the framework prompts.

5.3.2 Limitations

There are several limitations that impact this thesis and the ideas present within.

1. The scope was narrowed to exclude certain topics like dependencies to outside systems, specific implementation details around logic and API methods or automated version rollbacks in response to failed upgrades. This also limited the ability to address the specifics surrounding the framework's suitability for use with other BPM suites. As was also mentioned in 5.2, the currently proposed framework is not able to handle circular dependencies.

2. Given the countless factors that can differentiate one organization from another, it is unfeasible to provide exact values when evaluating factors like effort and cost in relation to framework performance. For this reason, qualitative measures were used to describe the performance of the framework relative to the related works and the cases studied.

5.3.3 Threats to Validity

Although the utmost care was taken when making design considerations and performing case studies, it is always possible that some element of the work presents a threat to the validity of the thesis. This section lists the most likely threats to validity from the perspective of the author.

1. Again, in relation to the first limitation listed in section 5.3.2, the difficult decision was made to not implement a working version of the design for testing and evaluation purposes. Despite the initial intent to do this and a fair amount of groundwork done based on plans of implementation, tight time considerations and early difficulties encountered surrounding upgrade call serialization forced the decision to stop short of implementation.

2. The examples were all developed and worked on by the author. Although, the author has extensive experience in BPM projects and the challenges of dependency management and impact assessment, the four examples are not sufficient to provide a complete test case coverage for validating the framework. Similarly, it would be good to have other BPM developers try to follow the framework to provide validation that the framework would be valid for all (or at least most) BPM developers and not just the author.

Chapter 6. Conclusions and Future Work

6.1. Conclusions

This thesis proposes an impact assessment and dependency management framework to help remedy the shortcomings of the tools currently included with BPM suites. The desired outcome was to offer developers the ability to conveniently view all dependency relationships affected by a given change and to offer automation and guidance during the upgrade process. The thesis puts forward three contributions to support this goal: a business process component architecture, a dependency data model and an upgrade algorithm.

This framework was designed based on extensive experience with BPM 8.5 in multiple development teams and is validated by manually working through four example cases of increasing difficulty. The results of these exercises show that there is great potential for such a dependency management framework to provide significant benefit to BPM development teams. Tools with similar capabilities, such as Maven already exist in the traditional coding space with proven results. This thesis starts with that premise and tweaks it to the BPM development paradigm, incorporating visual interface elements and additional automation to make full use of the dependency data being assessed.

There are also some obvious limits to the issues with which such a framework can assist. Even within the limited scenarios tested, the circular dependency in case 4 presented a challenge for which the framework's logic was not able to compensate. The framework

is designed to recognize these scenarios and alert developers that manual intervention is needed. Though it may be subject to some of the same limitations as human developers, the framework exhibits acceptable to excellent performance across the seven evaluation criteria identified. In layman's terms, using the framework is not likely disadvantageous for developers in any way. It should be noted that, in the simplest cases with few dependency relationships, the framework may not provide sufficiently worthwhile benefits to justify its use. However, the setup cost of such a framework would be nominal and any extra work would be limited to a few supplementary clicks. As the complexity of the dependency structure between projects increases and the number of projects grows, the organization and automation benefits also increase in value.

6.2. Future Work

Based on the threats to validity in section 5.3, the next step of this research would be to implement a functioning version of the framework design laid out in this thesis. Many design considerations remain to be established such as what language the framework would be written in? Can it be built as a business process within the BPM suite itself? Testing a working version of the framework may provide further insights about the format of the tool and how users can best interact with it.

One potential approach would be to leverage Maven itself as much as possible by developing a type of middleware that would allow Maven to communicate with the BPM suite. This option foregoes the creation of a BPM centric dependency management framework in favour of using Maven's existing interface and functionality and focusing implementation efforts on getting these two systems to speak to one another.

Once the framework gets implemented, it would be possible to move away from strictly qualitative evaluation criteria and begin using quantitative data to better describe the performance gains that can be expected by developers when using the framework. When such quantitative measures become possible, developers can be recruited to perform a sample set of upgrades using strictly the functionality built into the BPM suite as well as perform the identical upgrades with the assistance of the framework. Both of these data sets can then be compared to obtain numerical values of the time saving benefits of the framework.

Beyond work surrounding implementation, there is also much that can be done to strengthen the design of the framework and to bridge some of the gaps that were left

unresolved in this thesis. It would be very interesting for further research to be done on dependencies to systems outside the BPM suite. How can dependencies to web services and message feeds be detected and included in the impact assessment and dependency management activities? Viable suggestions on how outside changes can be fed into the framework and managed alongside business process changes would help make the framework a more compelling value proposition for existing BPM developers.

Another promising avenue of future research surrounding the proposed framework would be to investigate the possibility of introducing a learning component to its operation. The framework could potentially keep a history of the project structures analyzed and the resulting upgrade and development activities. This information could simply be kept as historical information in the application for developers to refer back to or it could be combined with machine learning in order to test the feasibility of offering development recommendations or other forms of predictive suggestions to developers.

Finally, a deeper investigation of BPM suites would allow the framework to be assessed with regards to its suitability for use with BPM suites other than IBM 8.5.

References

Aguilar-Savén, R. S. (2004). Business process modelling: Review and framework. *International Journal of Production Economics*, *90*(2), 129–149. https://doi.org/10.1016/S0925-5273(03)00102-6

Alam, K. A., Ahmad, R. B., & Akhtar, M. (2014). Change Impact analysis and propagation in service based business process management systems preliminary results from a systematic review. *Software Engineering Conference (MySEC), 2014 8th Malaysian*, 7–12. https://doi.org/10.1109/MySec.2014.6985981

Hernandez, F. (2016). How to use Maven for Dependency Management. Retrieved March 15, 2019, from https://examples.javacodegeeks.com/enterprise-java/maven/use-maven-dependency-management/

Hevner, A. R., & Gregor, S. (2013). Positioning and Preenting Design Science: Types of Knowledge in Design Science Research. *MIS Quarterly*, *37*(2), 337–355. https://doi.org/10.2753/MIS0742-1222240302

Hevner, March, Park, & Ram. (2004). Design Science in Information Systems Research. *MIS Quarterly*, *28*(1), 75. https://doi.org/10.2307/25148625

Jezek, K., & Dietrich, J. (2014). On the use of static analysis to safeguard recursive dependency resolution. *Proceedings - 40th Euromicro Conference Series on Software Engineering and Advanced Applications, SEAA 2014*, 166–173. https://doi.org/10.1109/SEAA.2014.35

Kolban, N. (2014). *Kolban's Book on IBM Business Process Management* (December,2014).

Levina, O., Holschke, O., & Rake-Revelant, J. (2010). Extracting business logic from business process models. *ICIME 2010 - 2010 2nd IEEE International Conference on Information Management and Engineering*, *2*, 289–293. https://doi.org/10.1109/ICIME.2010.5477554

Leymann, F., Roller, D., & Schmidt, M.-T. (2010). Web services and business process management. *IBM Systems Journal*, *41*(2), 198–211. https://doi.org/10.1147/sj.412.0198

Mallur, K., & Peyton, L. C. N.-E. B. E. uOttawa I. N. L. (2015). A Quality Assurance Framework for Business Process Management, 1 online resource. Retrieved from internal-pdf://178.143.229.108/Mallur_Kavya_2015_thesis.pdf%5Cnhttp://hdl.handle.net/10393/32273

Mendling, J. (2008). *Metrics for Process Models*. *An Overview of BPMN 2.0 and Its Potential Use* (Vol. 6). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-89224-3

Owen, B. M., Raj, J., & Software, P. (2014). Handbook on Business Process Management 2. *Handbook on Business Process Management 2, 2678*, 27. https://doi.org/10.1007/978-3-642-45103-4

Oyetoyan, T. D., Falleri, J. R., Dietrich, J., & Jezek, K. (2015). Circular dependencies and change-proneness: An empirical study. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*, 241–250. https://doi.org/10.1109/SANER.2015.7081834

Paier, A. (2013). "Where Used" API IBM BPM 8. Retrieved April 5, 2019, from https://www.ibm.com/developerworks/community/forums/html/topic?id=6033c16e-c554-4a1c-b809-b1bf9d1e4864

Paier, A. (2014). BP Labs: Unraveling the Toolkit Dependency Problem. Retrieved April 14, 2019, from https://www.bp-3.com/blog/bp-labs-unraveling-the-toolkit-dependency-problem/

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2008). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, *24*(3), 45–77. https://doi.org/10.2753/mis0742-1222240302

Radovan, A. (2012). No Title. Retrieved October 24, 2017, from http://www.croz.net/eng/business-process-management-ibms-way/

Retesh. (2015). IBM BPM - History & Versions. Retrieved March 14, 2019, from http://queryibmbpm.blogspot.com/2015/04/ibm-bpm-history-versions.html

Reynolds, J., Collins, M., Ducos, E., Frost, D., Knapp, D., Kornienko, I., Pfau, G. (2014). Leveraging the IBM BPM Coach Framework in Your Organization, 378. Retrieved from https://play.google.com/store/books/details?id=-FdhAwAAQBAJ&rdid=book--FdhAwAAQBAJ&rdot=1&source=gbs_atb&pcampaignid=books_booksearch_atb

Sánchez-González, L., Ruiz, F., García, F., & Piattini, M. (2013). Improving quality of business process models. In L. A. Maciaszek & K. Zhang (Eds.), *Communications in Computer and Information Science* (Vol. 275, pp. 130–144). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-32341-6_9

Seun, M. (2017). What are Dependency Managers? Retrieved April 13, 2019, from https://medium.com/prodsters/what-are-dependency-managers-26d7d907deb8

Strob, R., Müllner, T., & Rausch, T. (2009). Web-based process portals: Powering business process management within large organisations. *2009 IEEE Conference on Commerce and Enterprise Computing, CEC 2009*, 312–316. https://doi.org/10.1109/CEC.2009.88

van der Aalst, W. M. P. (2010). Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management. In J. Desel, W. Reisig, & G. Rozenberg (Eds.), *Lectures on Concurrency and Petri Nets: Advances in Petri Nets* (pp. 1–65). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-27755-2_1

van der Aalst, W. M. P. (2013). Business Process Management: A Comprehensive Survey. *ISRN Software Engineering*, *2013*, 1–37. https://doi.org/10.1155/2013/507984

Weidlich, M., Weske, M., & Mendling, J. (2009). Change propagation in process models using behavioural profiles. *SCC 2009 - 2009 IEEE International Conference on Services Computing*, 33–40. https://doi.org/10.1109/SCC.2009.58

Wetzstein, B., Ma, Z., Filipowska, A., & Kaczmarek, M. (2007). Semantic Business Process Management: A Lifecycle Based Requirements Analysis 2 Business Process Management Lifecycle. In *SBPM* (Vol. 251, pp. 1–11). Innsbruck, Austria.

Wohed, P., van der Aalst, W. M. P., Dumas, M., ter Hofstede, A. H. M., & Russell, N. (2006). Pattern-based Analysis of BPMN. *Lecture Notes in Computer Science*, *4102/2006*, 161–176. https://doi.org/10.1197/jamia.M2389

Workflow Patterns Initiative, van der Aalst, W. M. P., & ter Hofstede, A. H. M. (2017). Workflow Patterns. Retrieved June 5, 2019, from http://www.workflowpatterns.com/