Visualising Architectural Dependencies

John Brondum
NICTA and UNSW
Sydney, Australia
john.brondum@nicta.com.au
johnb@cse.unsw.edu.au

Liming Zhu
NICTA and UNSW
Sydney, Australia
liming.zhu@nicta.com.au
limingz@cse.unsw.edu.au

***Abstract*—Visibility of technical debt is critical. A lack thereof can lead to significant problems without adequate visibility as part of the system level decision-making processes [2]. Current approaches for analysing and monitoring architecture related debt are based on dependency analysis to detect code level violations of the software architecture [2,3,6]. However, heterogeneous environments with several systems constructed using COTS, and/or several programming languages may not offer sufficient code visibility. Other limiting factors include legal contracts, Intellectual Property Rights, and just very large systems. Secondly, the complexity of a software dependency is often greater than simple structural dependencies, including; multi-dimensional properties (as argued by [10]); behavioural dependencies [5,9]; and 'implicit' dependencies (i.e., dependency inter-relatedness [11]). This paper proposes a simple modelling approach for visualising *dependency relationships* as an extension of the current approaches, while supporting complex dependencies. The model can be built using existing dependency analysis and general architectural knowledge; thus is better suited for heterogeneous environments. We demonstrate the proposed modelling using an exemplar, and two field case studies.**

Keywords: **Software Architecture, Architectural Dependency Analysis, Architectural Debt.**

I. INTRODUCTION

'Technical debt' has been used as a software design metaphor in situations where compromises are made for short term gains, at the expense of the longer-term viability of a software intensive system [4]. The purpose of the metaphor is to highlight scenarios, where a short term gain equates to taking on 'debt' with an on-going payable 'interest'; as well as a 'principle' to be re-paid at some point in order to retain the health of the system or project. Thus the size of a debt can be characterised as the gap between the current, and hypothesized 'ideal', state of a system [2].

Technical debt at the architectural level is considered to be a debt that cannot be repaid through local code refactoring due to a lack in a coherent system-wide architecture. Significant challenges with architectural debt can arise from a lack of visibility of the debt, especially as those who incurred the debt may not be the same as those who will have to re-pay later [7]. Current approaches to visualising architectural debt are based on dependency analysis of the structural code, thus the debt calculations are based on the number of architecture violations (e.g., [3] and [6]).

However, as demonstrated by [1] and [5], architectural debt can manifest itself due to behavioural (or dynamic interaction) dependencies [9], while (conceptually) structural and behavioural architectural dependencies have the potential to be both multi-dimensional (the different sources of dependencies as described by [10]), as well as inter-related entities (described as 'implicit dependencies' by [11]). Thus, visualising architectural debt must be based on an ability to visualise the true complexity of architectural dependencies, rather than just a binary relationship between two elements.

Section II describes the modelling concepts and notations. Section III demonstrates its usage through an exemplar and two case studies. Section IV summarises our conclusions; while Section V further discusses future research.

II. VISUALISING ARCHITECTURAL DEPENDENCIES

Parnas [12] defined a module as an element that one is able to write with little knowledge of the code in another module; thus allowing it to be reassembled and replaced without reassembly of the whole system. A coherent system is, by extension, a system consisting of elements that are only determined or conditioned by another to a limited degree (as we are allowed to have a 'little knowledge').By 'limited', we refer to the concept of loose coupling, where the dependency between two elements is mitigated by something. E.g., the use of an element interface can mitigate the effect of dependencies, while the use of Web Services can mitigate the technology dependencies between two interfaces.

We conclude that understanding architectural dependencies is about architectural knowledge, not just a question of element decomposition. So the question is then: what architectural knowledge do we need to capture? Our proposal is to model architectural dependencies along three design dimensions: 1) Type of dependency, 2) Source of Dependency; and 3) Effect Degree.

A. Architectural Dependency Relationship Types

As identified by [9], we have two types of architectural dependencies: structural, and behavioural. In principle, a dependency can exist between elements' visible (external) parts, and/or their internal parts. The combination provides a simple classification of the possible dependency types (as

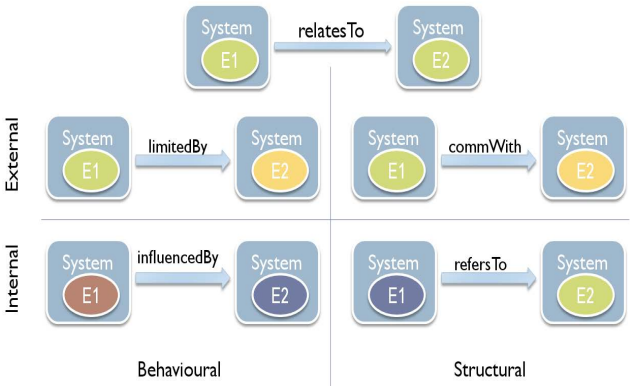

Figure 1: Dependency Relationship Types

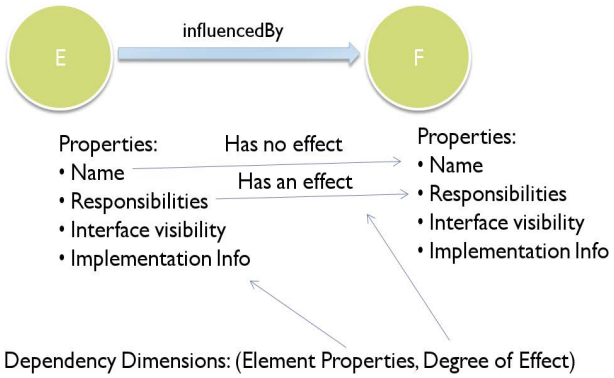

Figure 2: Effect Dimensions

illustrated in Figure 1). Our approach relies on modelling a relationship as a first class architectural element to support: 1) module and component independence (Section *C. Cross Views & Cross Models*), 2) relationship attributes to support modelling of dependency dimensions and degrees (next section), and 3) relationship references to support the modelling of implicit dependencies (Section *D. Relatedness of Dependencies*).

As illustrated in Figure 1, the five types are:

1. *communicatesWith (External – Structural)*: These cover the typical interface or contract based interactions between elements realised through a connector [13]. But as we scale the system(s) size up, these may also include scenarios where data are transferred between two systems through (semi-) manual processes; thus not fully connector automated.
2. *limitedBy (External – Behavioural)*: These are dependencies where quality attributes of one element constrain other elements, e.g., license, technology choice, or usage patterns [11].
3. *influencedBy (Internal – Behavioural)*: The behaviour of one element influences the behaviour of another element, e.g., the responsibility of one element changes, resulting in other elements needing to change their architectural role.
4. *refersTo (External – Structural)*: An element utilises another element 'owned' or managed by another system. E.g., Section III refers to an element, 'Student', shared across six different systems in its conceptual form, while each system has its own 'Student' implementation.
5. *relatesTo*: This serves as the *super* type for the other four types to support the ability to: a) "sub-type" the dependency relationships, and; b) aggregate two or more dependencies.

B. Dimensions & Degrees of Architectural Dependency Relationships

As illustrated by [10], understanding the source of a dependency is an important factor in maintaining a coherent architecture (i.e., minimising architectural debt). As shown, an architectural element has a set of properties relevant to its architectural view (e.g., as described in [14] and [15]).

As illustrated in Figure 2, element F influences element E, and the source of that dependency is found as part of the element responsibilities; thus we note 'responsibilities' as a dependency dimension of the 'influencedBy' relationship, with an effect degree of 'direct'. Also as illustrated, the 'name' property is noted as having 'no' effect, while the interface visibility is a 'mitigated' dependency. I.e., we have three effect degrees in total: No, Mitigated, and Direct.

We do not require a one-to-one relationship between the element properties, as it is feasible for one property to have an effect across several properties of the dependent element. The complete set of possible dependency dimensions include: the view specific element properties, the design parameters for the system (of which the element belongs to), and quality attributes—although only the architectural significant dimensions should be considered.

C. Cross Views & Cross Models

A key aspect of technical debt is that a compromise can be made as part of one concern (e.g., modularity) to meet an urgent demand for another concern (e.g., deadline). Similarly, architectural debt can be incurred due to compromises made as part of the Allocation Views (Chapter 5 in [14]) due to project license cost constraints (e.g., a 'License Cost' View). Thus to effectively visualise architectural debt, we need the ability to capture these cross view dependencies.

Our approach is based on retaining the dependency relationships as architectural element type independent. We achieve this by modelling them as first class elements, similar to 'Staff' modelled as a normalised data object representing a relationship between a 'Person' and an 'Organisation'.

D. Relatedness of Dependencies

Woods and Rozanski [11] illustrated the importance of awareness of implicit dependencies, i.e., inter-dependencies between external elements. E.g., system A produces data X, and then transfers X to system B for external consumption by system C. For C, it can be important to understand A's data production frequency while interacting with B. Our approach suggests a model of the following dependency relationships:

- C refersTo(X) A
- C limitedBy(X.timeOfProduction) A
- C communicatesWith(X) B

The refersTo relationship is then viewed as an aggregation of the limitedBy and communicatesWith relationships, while explicitly highlighting the 'implicit dependency' between C and A.

III. Illustrative Cases

A. Online Product System (Examplar)

A company offers a product web portal where their customers can purchase games and download them. The games are sold based on a per unit price, and managed by the 'Unit' System. Customer accounts are managed by the Accounts System, while the Product Portal utilises a number of portlets to display the products and implement a shopping cart-like functionality. Figure 3 illustrates the architecture. The Portal invokes functions of the 'Unit' system to obtain product information and purchases. For each purchase, 'Unit' system invokes the 'debtUPrice' function in the Accounts System, and subsequent invocations of the 'getBalance' function will return the account balance for the customer.

1) Adding a New Service: The company decides to launch a new service allowing their customers to play the game online rather than download it. This time they decide that the best business model is to charge 'per time unit' played, e.g., 1 dollar per 1 minute booked in 5-minute intervals (using the 'reserve' function). Once the customer stops playing (or is disconnected due to a lack of funds), the 'Time' system issues a final 'debtTPrice', which commits the final charge and releases any unused reserved funds (e.g., playing 3 mins would release 2 dollars).

The software architecture utilises Web Services for all interfaces to improve the overall loose coupling of the architectural elements. A separate system is added to avoid interfering with the existing 'Unit' system, and the existing interfaces for the accounts system remain untouched. The portal functionality is extended using additional portlets, leaving the existing user interactions intact.

However, the semantics of the 'getBalance' did alter despite no code changes. It can either display 'balance – reserve' or 'balance only'. At first hand, this does not appear to be of significance, but consider the following scenario.

A customer begins to play a game online, and 5 mins are reserved against his account with an original balance of $10. Two minutes into game play, he decides to purchase another game costing him $6. If the 'getBalance' function returns $5, he'll be denied the purchase even if he stops playing after 4 mins (and thus have $6 left in his account). Alternatively, if the 'getBalance' returns $10 (i.e., the reserve is not counted) then he will be able to make the purchase, but the company runs the risk of having customers overdrawing their accounts (e.g., if he plays for the full 5 mins).

2) Architectural Debt: Regardless of the approach, the portal user interaction will need updating to ensure the customer is aware of the altered behaviour. If the former 'balance – reserve' approach is chosen, it equates to a first come, first serve product priority strategy, but the company may prefer to prioritise unit price products. If the later

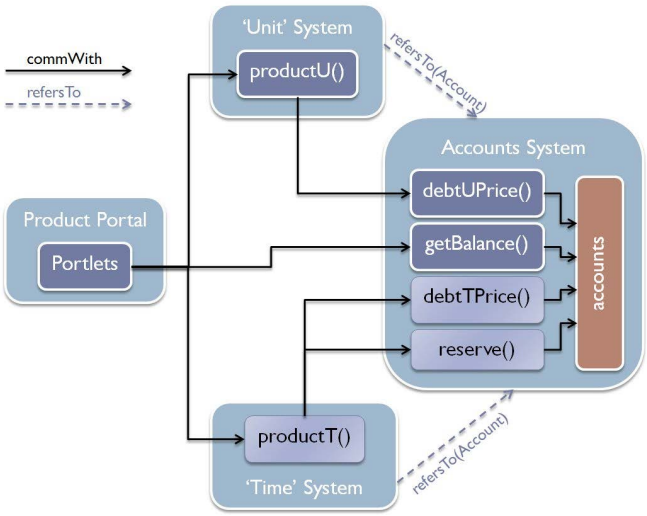

Figure 3: Online Product System

'balance only' approach is chosen, then the company either needs to accept the overdraft potential, or the team will need to search for an alternative to the 'reserve' function.

If the company chooses to launch the new product without any changes to existing products –due to a lack of time, or necessary funds to make the required changes in the other systems, or perceiving it as too risky (i.e., prefer a new project to deal with the impacts to 'unit' system) – they'd incur a technical debt where the interest is the described effects; and the principal is the required changes in the portal and 'unit' system.

The source of the above debt would not be visible as part of a structural code analysis. The 'getBalance' function would not visualise how the 'reserve' function is implemented. The internal account object would similarly not highlight the semantics without some commentary updates by the 'reserve' function implementer. And if the architecture consists of COTS software components, then the detection would be virtually impossible.

Alternatively, the architect could (or rather, should) document this during the architectural evaluation and within the architectural model (using modelling language such as UML). But without explicit representation within the model, the 'Time' system architect may not realise the impact to the Unit system. They are, after all, not changing any code that is shared with the Unit system (all four Account System functions use the same, protected 'credit', 'debit', and 'get' functions of the private class 'account' – unchanged).

3) Visualising the Debt Potential: For our proposed approach, the dependency relationship view of the architecture is illustrated in Figure 3. The 'Unit' software architect would note a 'refersTo' (internal-structural) relationship between the two systems.

A subsequent dependency dimension and effect degree analysis of 'Account' properties (as a data element [14]) –

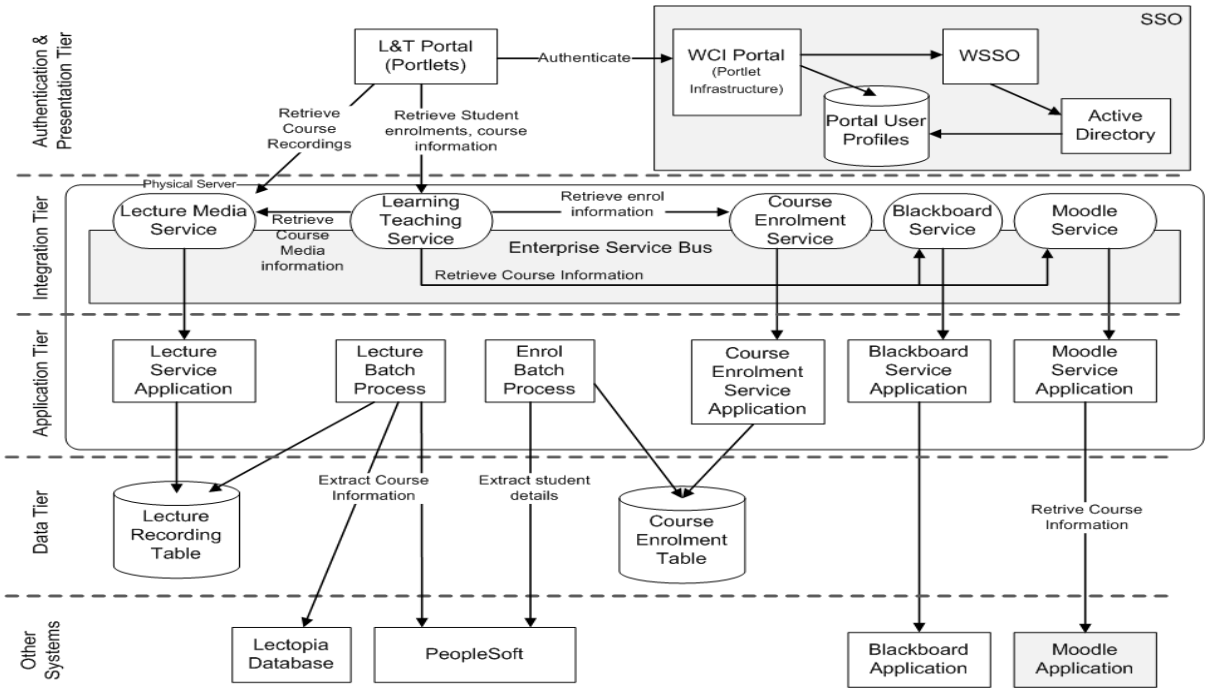

Figure 4: Learning & Teaching Portal

can be performed without knowledge of the Accounts System's element decomposition:

1. *Name*: No effect, as the implementation of Account is hidden.
2. *Visibility of Interface*: No effect, as the implementation of Account is hidden.
3. *Responsibilities*: A statement describing in adequate detail what it is that an Account does (semantics). This property has a *direct effect*, thus noted as an attribute of the 'refersTo' dependency relationship.
4. *Implementation Information*: No effect, as the implementation of Account is hidden.

4) Conclusion: Once the 'Time' system project kicks off, the dependency relationship is embedded in the architectural model. The advantage is that DSM based debt analysis approaches (such as [3]) have an explicit representation.

B. Learning Teaching Portal (LTP) System

The case study was performed in collaboration with a large, tertiary educational organisation. The organisation offers a broad range of degrees and utilises a number of software applications to support the diverse set of functional requirements across the offered courses. The main drawback for their students is the multitude of application interfaces to learn, and account details to remember, in order for them to maintain a concise view of course related statuses and communications. Thus the organisation embarked on a strategic project to provide a student portal, which would deliver an aggregation of course content, and communication within a single sign-on environment.

The above diagram illustrates the overall Portal architecture split into five tiers: presentation, integration, application, data, and external systems. The user entry point is via the Portal landing page. The page forwards the user to an authentication page, where the student's credentials are validated against those stored within the Active Directory. Once authenticated, the student's portal profile is loaded from the Portal database, and the personalised web page is rendered based on content gathered by the composite `Learning Teaching' Service (integration tier). The service collates the student specific course information from four other web services: `Lecture Media Service', `Course Enrolment', `Moodle', and `Blackboard'.

The four services aggregate student and course information via a set of `service applications' (application tier) encapsulating the actual systems responsible for the course information (found in the external systems tier). The data tier consists of two databases for lecture recordings and course enrolments synchronised with Lectopia and PeopleSoft, primarily to protect them from performance impact associated with Portal usage.

1) Managing Project Scope and Complexity: The project was the first in a number of anticipated subsequent projects aimed at delivering additional portal functionality through integration of additional functionality and systems. With a limited budget, the central strategy of the project team was to contain complexity through standardisation: 1) Strict layering of functionality and responsibility, 2) all elements communicate across the layers must use Web Services standards, and; 3) applications to be integrated had to meet a basic set of functional and technical capabilities. Secondly, portal functionality was focused on the aggregation and display of information. The aim was not to duplicate information capture, or other existing functionality, but clearly focused on providing the students

with a single reference point for their course work and communications. Although significant compromise had to be made, the strategy was central to the team's ability to deliver a successful project.

2) Architectural Debt: Student and course information was core to the project. The PeopleSoft system was the source for information about students, including enrolments, whereas the course information systems managed the course related information, with a reference to the student. Yet, PeopleSoft also had an overall function as the administrative system, and thus was viewed as the 'master' system for student and course information. So, although a significant challenge was piecing the wide range of data formats and semantics together in order to present a coherent view for the students, it was aided by the centralised administration function.

However, each system is operated as individual systems managed by a mix of internal and external service providers. This covers both the technical and operational sides of the systems; thus, the portal is indirectly dependent on a set of operational processes tokeep the data synchronised across the integrated systems. For example, students must be enrolled in a course (PeopleSoft) prior to gaining access to the course delivery material (i.e., account creation within Moodle, Blackboard, etc.). A similar scenario exists for the course administration, as course information within PeopleSoft must be synchronised with the course delivery systems such as Lectopia, Moodle, and Blackboard.

While neither the LT portal nor the project team created those dependencies – they existed well before – the portal does act as an amplifier, where synchronisation errors or gaps may become increasingly visible to the students, simply due to the intended function of the portal; the interest payable by the Portal system, while the principal is the lack of integrated information management across the systems.

3) Visualising the Debt Potential: The estimation of interest and principal will require a detailed analysis of the complete system (of systems). However, our proposed approach would enable the portal architect to visualise the potential dependencies behind the debt. Figure 5 illustrates a simplified dependency relationship view of the architecture in Figure 4. The diagram seeks to capture the underlying synchronisation dependencies of the portal. E.g., the 'student' and 'course' elements of the LT Portal 'refersTo' the PeopleSoft 'student' and 'course' elements.

An important part of the modelling is the utilisation of architectural abstraction, i.e., the six major systems will each have independently developed separate models of *their* student and course representations. For example, Lectopia does not have a student concept, while SSO does not have a Course concept. And the relevant information from a portal perspective is spread across many interfaces requiring several method invocations. The aim of diagram (Figure 5) is to capture:

1. All systems are dependent on the PeopleSoft system for correct life-cycle management of the abstract 'Student' and 'Course' entities. The dependencies are modelled using the 'limitedBy' relationship with the attributes Student/Course. The dependency dimensions include CRUD (Create, Retrieve, Update, and Delete) – all of which have a 'direct' effect degree. I.e., SSO is limited by PeopleSoft, as it cannot create its student entity until after PeopleSoft has created its equivalent entity. Similar for the other CRUD functions. Note: From the Portal point of view, these are all examples of Woods and Rozanski's 'implicit dependencies'.
2. All systems refer to PeopleSoft for the Student and Course entities. However, as they are all separate systems, the dependency relationship arose from the architectural decision to dedicate the administrative responsibility of the entities to PeopleSoft; thus there is no effect degree for the dependency dimension such as entity name, structure, and technology. But the semantic will have a 'direct' effect degree, as the data recorded within PeopleSoft will have a direct effect on how data is maintained within the other systems.

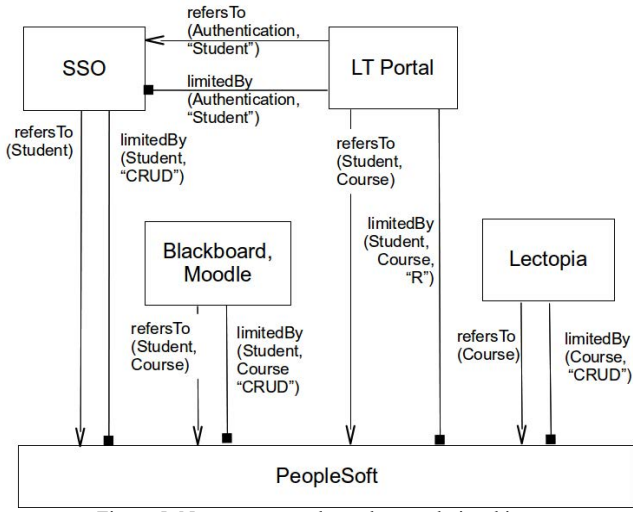

Figure 5: Non-connector dependency relationships

In summary, the aim of the Portal project was to automate the aggregation of the Student/Course entities as far as 'R' in the limitedBy relationship. The 'CUD' part was outside the project scope, and thus made the Portal dependent on a number of external operational and technical processes in order to work correctly.

4) Conclusion: A simple dependency relationship representation (as the one illustrated in Figure 5) provides valuable information about the operational impact from the chosen functional scope, and the semantic dependencies across all systems. The approach offers a simple visual representation, as well as the potential to be incorporated into technical debt estimations tools (such as [3]), while offering a richer notion than the approach proposed by Hinsman et al. [6].

C. Lending Valuation System

The third case study is based on a lending property valuation case. A Distributed Property Valuation (DPV) system was developed for a lending organisation, utilises the Australian Lending Industry XML Initiative (LIXI) standard, and is predominantly web services based. Version 1 of the system was launched in 2008, employing a mix of Tomcat server and Microsoft SQL Server technologies. Users would interact with the application via a web browser interface, or

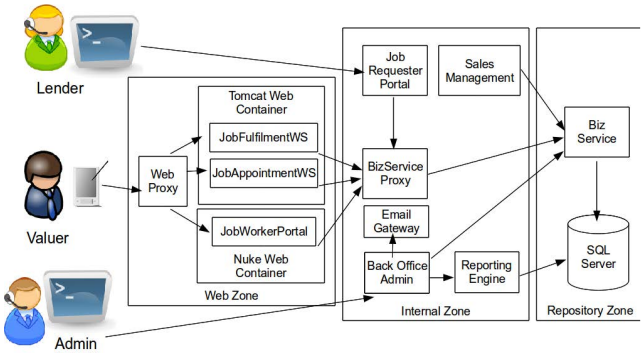

Figure 6: DPV System

an application running on a mobile device. The system supported the property valuation process and utilised the LIXI defined property valuation schemas. The system supported the complete process of property valuations, including the request, acceptance, scheduling, fulfilment, and invoicing.

1) System Description: A subsequent upgrade to version 2 required a number of technology substitutions, including replacement of the Microsoft SQL Server database with a MySQL database, as well as code upgrades for all of the deployed components. In addition, the upgrade needed to be able to cope with in-complete transactions e.g., property valuation request had been accepted, but not completed by an evaluator. Such transaction could remain incomplete up to several weeks.

The system is split into four zones: 1) Internet zone: Users can access the system via a web interface or mobile device application (MDA) to download valuation requests and upload property valuations, 2) Web zone: This zone hosts the web proxy providing MDA access to three web services, which deliver the functionality for acceptance and completing of property valuations, 3) Internal zone: This zone contains five applications for the request of valuations, invoice management, general administration (user accounts, etc.), email gateway, and a reporting application, and; 4) Repository zone: The central database for the DPV, accessed via a web services abstraction layer, hosted in a separate application. The complete system consists of twelve independently deployed applications, communicating via WSDL over SOAP/HTTP. The system was developed by a single company utilising Agile development methods, while the users of the system work for a separate lending company. The system was developed and deployed by the same team, all co-located together.

2) The Architectural Debt: The company developing the application experienced a number of unforeseen challenges when they upgraded the system to version 2 – challenges that were inadequately addressed through their in-house agile development methodology. A separate case study was performed to gain experience in dealing with dynamic deployments, to be able to better manage future releases.

The issues uncovered were primarily related to the need to handle long-running transactions, as valuations could spread across weeks. The second issue was related to changes in underlying technologies. For example, the reporting engine was unable to handle long reports due to a database driver incompatibility (CrystalReport to MySQL), and dependencies were not explicitly identified, because the drivers were third party.

3) Visualising the Debt Potential: Remodelling the DPV architecture using the dependency relationships, we found the following relationships: **1)** All connectors were marked as communicatesWith relationships (including a 'connector' between the job components and their container), **2)** The BizService was identified as the master element for the utilised data objects: Valuations, Jobs, Appointments, and Invoices (refersTo relationships), **3)** the joint hosting of two elements within the Tomcat web container was marked as limitedBy relationships between the JobFulfilmentWS (JF) and JobAppointmentWS (JA) components, and; **4)** the overall process consists of a) sell the job (Sales Management – SM), b) request a job (Job Requester Portal – JRP), c) appoint and fulfil a job (JA, JF and JobWorkerPortal – JWP), and; d) generate invoice (Back Office Admin). The order is marked using an influencedBy relationship.

Figure 7 contains a Dependence Structure Matrix (DSM), where identified relationships have been marked accordingly: 'C' for 'communicatesWith', 'L' for 'limitedBy', 'I' for 'influencedBy', and; 'R' for 'refersTo'. It should be noted that the analysis did not include a dependency dimension and degree analysis.

The root cause analysis based on stakeholder interviews recommended a phased upgrade approach, covering four interim releases, with 34 individual cases of application level changes. In 20 of those, two or more changes had to be performed simultaneously in order to retain consistency. Figure 8 illustrates another DSM representing the connectors (labelled 'X'), and the subsequent change cases (labelled 'S').

4) Conclusion: Figure 8 illustrates that the change cases are clustered around the migration activities of the application containers. A connector based dependency analysis would have located the majority of those cases. However, it would have missed the cases where changes to JA, JF, and JWP affected the BizService (BS); as well as the case where Sales Management affects the SQL Server Datastore (SSD). Figure 7, illustrating the alternative dependency model, highlights that the functional

	MJA	WP	PW1	eW1	JF	JA	JWP	JRP	BSP	SM	BOA	RE	EG	BS	SSD
Mobile Job App (MJA)	-	C												R	
Web Proxy (WP)		-	C	C	C	C	C								
pdaWeb1 (PW1)			-		C	C									
endUserWeb1 (eW1)				-			C								
JobFulfilmentWS (JF)			C		-	L		I	C					R	
JobAppointment (JA)			C		L	-		I	C					R	
JobWorkerPortal (JWP)				C			-	I	C					R	
JobRequesterPortal (JRP)								-	C	I				R	
BizServiceProxy (BSP)									-					C	
SalesManagement (SM)										-				C,R	
BackOfficeAdmin (BOA)					I						-	C	C	C,R	
ReportingEngine (RE)												-			C,R
EmailGateway (EG)													-		
BizService (BS)														-	C,R
SqlServerDatastore (SSD)															-

Figure7: Dependency Relationship Model (DSM)

decomposition of the DPV architecture, combined with the Web Container strategy, appears to have mitigated the potential inter-dependencies. This was expected, as the upgrade from v1 to v2 did not alter the responsibilities of the architectural elements, and deployment separation was based on well tested web based containers.

However, the migration from SQL server (DPV v1) to MySQL (DPV v2), JF, JA, and JWP were impacted, as the upgrade plan sought to avoid migrating transactional state. V1 and V2 had to exist for a concurrent period, until all active transactional state had been initiated within the DPV v2 database (reflected by the cases involving BS and JF, JA, JWP, and SM). The transactional state problem was not visible from the Mobile Job App, as jobs were started and completed using the same version (therefore no change case despite highlighted as a potential dependency by our model).

Our model highlighted internal structural dependencies ('refersTo') as a potential dependency; a potential that is dependent on the change case, e.g., if the team had decided to retain SQL Server, then those change cases may not have been necessary. But the change in technology forced alterations to the internal data structure.

In summary, our model presented explicit notations of potential dependencies over a connector only model, while retaining support for DSM based debt calculations. Some of those dependencies did not present an impact for the upgrade scenario (due to a good functional and allocation decoupling within the architecture), but it did highlight that a Web Services architecture based on functional decomposition does not necessarily encapsulate all data dependency aspects – something a structural only DSM would not have been able to capture.

IV. CONCLUSION

Current approaches for analysing and monitoring architecture related debt are based on the ability to detect code level violations of the specified software architecture [2,3,6]. However, as we have illustrated, heterogeneous environments with several systems constructed using COTS, and/or several programming languages, may not offer sufficient code visibility. Other limiting factors include legal contracts, Intellectual Property Rights, or just very large systems. Secondly, the complexity of a software dependency is often greater than simple structural dependencies, including; multi-dimensional properties (as argued by [10]), behavioural dependencies [5,9]; and 'implicit' dependencies (i.e., dependency inter-relatedness [11]).

	MJA	WP	PW1	eW1	JF	JA	JWP	JRP	BSP	SM	BOA	RE	EG	BS	SSD
Mobile Job App (MJA)	-	X													
Web Proxy (WP)		-	S,X	S,X	S,X	X	X								
pdaWeb1 (PW1)			-		S,X										
endUserWeb1 (eW1)				-											
JobFulfilmentWS (JF)			S,X		-				X					S	
JobAppointment (JA)			S,X			-			X					S	
JobWorkerPortal (JWP)				S,X			-		X					S	
JobRequesterPortal (JRP)								-	X					S	
BizServiceProxy (BSP)									-					S,X	
SalesManagement (SM)										-				S,X	
BackOfficeAdmin (BOA)											-	X	X	S,X	
ReportingEngine (RE)												-			S,X
EmailGateway (EG)													-		
BizService (BS)														-	S,X
SqlServerDatastore (SSD)															-

Figure 8: Detailed Change Scenario Analysis

Our aim in this paper was to illustrate a simple modelling approach for visualising these 'complex' dependency relationships as an extension of the current approaches, and support a general architectural knowledge capture; thus better suited for heterogeneous environments.

We based our approach on a dependency relationship model, also allowed for the representation of dependencies across different architectural views; as well as enabling the architect to capture otherwise implicit architectural knowledge about the system dependencies. We believe that an improved visualisation of architectural dependencies can support: 1) a better strategic use of technical debt; 2) more accurate estimation models; and 3) help identify sources of debt – especially for heterogeneous technology environments with limited access to source code.

We believe the approach enables better visibility of behavioural dependencies; as well as dependencies that either have not or cannot be shielded by standard component encapsulation (e.g., interfaces or contracts), and software connectors, such as operational and transactional state.

V. Future Research

We plan to perform further validation of the modelling approach – in particular regarding large enterprise wide environments. Our future work includes enhancing our dependency relationship identification and analysis method to appropriately aid the architect, as the quality of our approach over a structural only DSM based approach rests with the abilities of the software architect and their understanding of the architecture.

Similarly, as illustrated in our third case, further research is required to adequately aid the software architect in identifying the situations where a dependency relationship will manifest itself.

We believe this will be based on the dependency dimension and degree concepts introduced. And lastly, although we have illustrated the use of our model as a DSM, further work will be required to enable the use of our annotated DSM for the purpose of calculating technical debt such as illustrated by [3].

Acknowledgment

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[1] M. Bass, V. Mikulovic, L. Bass, H. James, and C. Marcelo. Architectural misalignment: An experience report. In WICSA '07, 6th Working IEEE/IFIP Conf. on Soft. Arch., pages 17–17, 2007.

[2] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim,A. MacCormack, R. Nord, I. Ozkaya, et al. Managing technical debt in software-reliant systems. In Proceedings of the FSE/SDP workshop on Future of software engineering research, pages 47–52. ACM, 2010.

[3] N. Brown, R. L. Nord, I. Ozkaya, and P. Kruchten. Quantifying the value of architecting within agile software development via technical debt analysis, 2012.

[4] W. Cunningham. The wycash portfolio management system. In ACM SIGPLAN OOPS Messenger, volume 4, pages 29–30. ACM, 1992.

[5] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In Proceedings of the International Conference on Software Maintenance, ICSM '98, pages 190–, Washington, DC, USA, 1998. IEEE Computer Society.

[6] C. Hinsman, N. Sangal, and J. Stafford. Achieving agility through architecture visibility. Architectures for Adaptive Software Systems, pages 116–129, 2009.

[7] T. Klinger, P. Tarr, P. Wagstrom, and C. Williams. An enterprise perspective on technical debt. In Proceeding of the 2nd working on Managing technical debt, pages 35–38. ACM, 2011.

[8] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. Computer, 40(11):38 –45, nov. 2007.

[9] J. Stafford, A. Wolf, and M. Caporuscio. The application of dependence analysis to software architecture descriptions. Formal methods for software architectures, pages 52–62, 2003.

[10] R. High Jr, G. Krishnan, and M. Sanchez. Creating and maintaining coherency in loosely coupled systems. IBM Systems Journal, 47(3):358, 2008.

[11] E. Woods and N. Rozanski. The system context architectural viewpoint. In WICSA/ECSA '09, Joint Working IEEE/IFIP Conf. on Soft. Arch. & European Conf. on Soft. Arch., pages 333–336, 2009.

[12] D. L. Parnas. On the criteria to be used in decomposing systems into modules. ACM Communications, 15(12):1053–1058, 1972.

[13] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In ICSE '00, 22nd Int. Conf. on Soft. Engineering, pages 178–187, 2000.

[14] P. Clements, F. Bachmann, L. Bass, D. Garlan, P. Merson, J. Ivers, R. Little, R. Nord, and J. Stafford. Documenting software architectures: views and beyond. Addison-Wesley Professional, 2010.

[15] P. B. Kruchten. The 4+1 view model of architecture. IEEE Software, 12(6):42–50, Nov 1995.