IN SITU: A PROCESS-ORIENTED SUPPORT FOR SOFTWARE AUTONOMY

T. BAKER, A. TALEB-BENDIAB, P. MISELDINE
Department of Computing and Mathematical Sciences,
Liverpool John Moores University, Byrom Street, Liverpool, L3 3AF, UK
t.shamsa@2007.ljmu.ac.uk , a.talebbendiab@livjm.ac.uk, p.l.miseldine@2004.ljmu.ac.uk

Keywords: Autonomic Computing and Business Process Oriented Language.

Abstract: The recent years has seen a flurry of research inspired by social and biological models to achieving software autonomy. This has been prompted by the need to automate laborious administration tasks, recovery from unanticipated systems failure, and provide self-protection from security vulnerabilities, whilst guaranteeing predictable autonomic software behaviour. However, runtime assured adaptation of software to new requirement is still an outstanding issue for research. This paper presents a language support for process-oriented programming of autonomic software. The paper starts by a review of the state-of-the-art into runtime software adaptation. This is followed by a developed Neptune framework and language support, which is here described via an illustrative example pet shop benchmark. The paper ends with a discussion and some concluding remarks leading to suggested further works.

1 AUTONOMIC COMPUTING

Mimicking the autonomous behaviour exhibited by living organisms, research into autonomic computing is exploring ways to imbue software systems with self-managing capabilities (Parashar and Hariri, 2005). Much research development already exist including autonomic middleware frameworks such as (Sajjad et al., 2005) and (Agarwal et al., 2003) target grid-based computing to aid in similar self-management, ReMMoC (Grace et al, 2005) targets mobile environments, whilst the author's own Cloud Architecture can be applied to disparate, localised networks (Miseldine and Taleb-Bendiab, 2006) for self-organisation. Other works focused on code base analysis adaptation such as Toskana (Engel and Freisleben, 2005) and similar approaches use AOP as a basis for adaptable change. However, these changes are enabled by specific modifications to one or more OSI layers, such that with Toskana an operating system kernel is made adaptable. Thus, for complete autonomy, components must be built to support the kernel.

Further holistic paradigms for autonomic software have emerged based on software agent theory (Castelfranchi, 1995) that treat a complete system as a series of agents, each of which processes their own behaviour models to collectively perform set tasks and behaviours. Effectively co-ordinating and providing communication between agents, essentially providing unified, directed behaviour (Kurbel and Loutchko, 2003) remains a challenge. Other approaches to software autonomy are based on service-oriented architecture and service hot-swapping, in that, for instance adaptive middleware marshals and re-routes calls to alternative services (Caseau, 2005). Recent development in Business Process Management (BPM) and Business Process Execution languages, such as WS-BPEL 2.0 has widened the scope of process modelling to encompass cross-enterprise and inter-enterprise processes with most often a widespread of heterogonous business processes including their associated enactment, governance and assurance requirements to name but a few non-functional requirements. Hence, the task of provisioning and managing such systems far outstrips the capabilities of

human operatives, with most adaptations to operational circumstances requiring the system to be taken offline reprogrammed, recompiled and redeployed.

However, a systematic approach to the adaptation of software at runtime for new or emergent requirements is still an outstanding issue for research. In addition, providing adaptation of a business process based on assured, bounded governance requires knowledge of not only what should be adapted, but also its impact on other processes and the safety of producing such adaptation within its defined boundaries. In effect, knowledge of the construction and the meanings behind the formation of a business process system is vital for bounded autonomy through adaptation to be accomplished.

However, runtime assured adaptation of software to new requirements is still an outstanding issue for research. This paper presents a language support for process-oriented programming of autonomic software. The paper starts with a review of the state-of-the-art into runtime software adaptation. This is followed by a developed Neptune framework and language support, which is here described via an illustrative example pet shop benchmark. The paper concludes with a discussion and concluding remarks leading to suggested further works.

In short this paper presents a developed Neptune method and language to enable full runtime adaptation of a distributed, component-based application. The full description of the language is outside the scope of this paper but can be found in (Miseldine and Taleb-Bendiab, 2006).

2 PROPOSED SOLUTION

Current approaches to process-oriented architecture Enterprise Application Integration (EAI) based on web services use the WS-BPEL 2.0, in which explicit definitions of the mapping between task and their associated web service is provided (Khoumbati et al, 2006). This amongst the many other deficiencies inherent in WS-BPEL 2.0 create a tight and explicit coupling between business processes and their associated services and resource layer, which often works against the requirements for agile adaptation of information systems to changing business processes and/or users requirements.

As shown in Figure 1, the proposed autonomic process-oriented service component architecture takes a process (intention) model, and then interprets it via a semantic linking model to map each process to an appropriate service. The emerging annotated model is then executed via the service execution engine. The full description of the proposed and developed domain specific language and framework is outlined below and described in (Miseldine and Taleb-Bendiab, 2006).

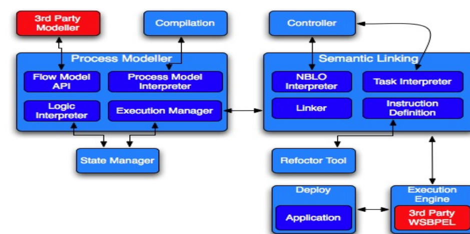

Figure 1: Process Modeller and Semantic Linking Schematic.

2.1 Implementation

The solution should sit above and be separate from any single language used to produce language components. In effect, the components are hidden from Neptune by way of a semantic interface: Neptune need only be able to dispatch calls defined within the semantic interface, rather than needing to directly host and manage the component itself. In this way, a bridge between Neptune and a base language is required both to facilitate communication between the base language and Neptune so that the language can instruct Neptune and interpret results from it, but also Neptune can dispatch a call that is required to be handled by the base language.

It is also clear from the discussion that Neptune requires a form of expression to be defined such that it is capable of representing the ontological model desired to fulfil the remaining qualities. In keeping with established, classical works, Neptune will present information using a context-free language NeptuneScript. Figure 2 shows a schematic view of the method in which Neptune communicates with base language components. The Base Language Connector's are themselves written within the base language rather than with NeptuneScript, and encapsulate the structure and communication methods needed to interact fully with Neptune and to be able to

interpret and write NeptuneScript. Neptune itself provides numerous interfaces using standardised protocols such as COM, SOAP, and RPC forms such as Java RPC or .NET Remoting to enable a base language connector to dispatch a call to Neptune, and receive responses from it. These calls are marshalled either directly to the higher Neptune Framework for processing, or situated as Neptune Base Language Objects (NBLOs), a reified semantic form that describes the intentions and conditions needed by a base language component (Miseldine, 2007). The NBLO itself is written within NeptuneScript. In addition, the NBLOs contain the requisite information to determine how to interoperate with the appropriate base language connector to marshal a call to the base language component. This information is used by the Neptune framework to invoke the behaviour described in the NBLO.

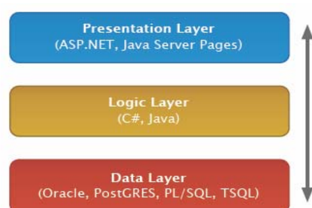

Figure 2: Base Language Interface.

3 AN ADAPTIVE SCENARIO

PetStore is an architectural blueprint provided by Sun Microsystems to highlight many of the capabilities of Java as well as the design guidelines issued for the language. The application produced by the PetStore blueprint builds a web-based solution for browsing and purchasing animals within a store front interface. Facilitates exist such that administrators of the system can add, remove, and modify the animals available for sale, with their data stored on a database. In this way, the PetStore is publicly available, best-practice system design architecture for e-commerce and enterprise systems in general. The PetStore is now becoming a benchmark application, with many other companies such as Microsoft have produced their own implementations (PetShop). Hence, we use the PetShop (PetStore) models to evaluate Neptune support for self-management.

3.1 Introducing Neptune to PetShop

The design of the PetStore model was one that advocated the use of 3-tier architecture (Fig. 3). In this approach, a logic layer is produced that processes the data contained in the data layer to be shown to the user in the presentation layer. As such, it is the logic layer that will be primarily of interest and the focus of the adaptation of process. Dependencies exist such that the presentation layer must present any new data or obtain any new inputs required by the adaptation, and the data layer must be able to store any introduced data. In keeping with the 3-tier design the presentation layer logic, and data layer logic should be isolated from the underlying processes represented in the logic layer. In the PetShop model, the use of thin classes, that represent directly the data types used within the software, act as isolators of dependency between the layers. The logic layer consists of classes that each encapsulate an entity used within the shop model; a customer class represents all the features and data model required from a customer, whereas the order class represents the processes that pertain to order transactions within the system. Methods are used to encapsulate processes, such that `Product.getProductFromID()` encapsulates the process needed to obtain a product from the database given an ID. To adapt a process, its associated class and method contained within the logic layer needs to be determined.

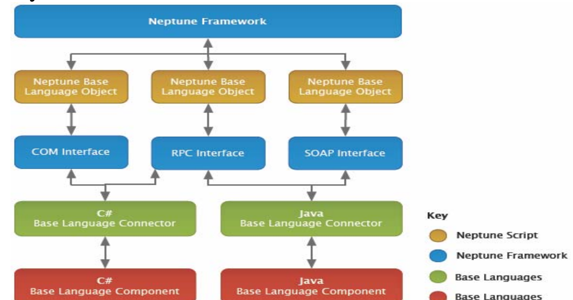

Figure 3: Layer Structure of PetShop.

Self-management capabilities provided by Neptune can be considered as a concern, and thus cross-cut against the existing classes representing the processes requiring adaptation. As the implementation of Neptune is written in C# and compiled as a .NET assembly, usage of the .NET implementation of the

architecture will ensure any performance results produced by the case study will not be affected by the protocol chosen to interface with Neptune as this can be performed via a native interface for .NET. Thus the overhead of calling and receiving messages from Neptune will be negligible in relation to a protocol such as SOAP.

3.2 Scenario Implementation

In the comparison between PetShop and PetStore, several criteria were considered to compare the design. For instance, the benefits of the PetShop were measured in terms of number of lines of code produced, performance and scalability judged by response time per user, and CPU resources required by both approach. It should be noted that whilst these criteria may have been chosen for commercial gain – such that the PetShop was produced with significantly fewer lines of code than the PetStore and this led to its choosing as a criteria – adopting the same criteria allows for a direct comparison between each system (PetShop, PetStore, and PetMart) to be made.

As the main design goal of Neptune is its ability to produce runtime adaptability. As such, ascertaining whether an adaptation took place at runtime or at design time is inherently simple; if no re-compilation and re-deployment of the application occurred in a traditional design sense, then the adaptation can be considered to be made at runtime. However the associated cost of providing runtime adaptation must be made, as essentially runtime adaptation is advocated for its use to produce reliable, self-aware systems. Thus the evaluation scenario outlined here should be chosen from the PetShop example to be modelled within Neptune, and analysis made using the criteria set by Microsoft and Sun upon the relative impact of introducing Neptune to an existing system. The model should then be adapted to introduce a new behaviour, with a comparison made between the method required in Neptune to achieve this adaptation, and the method used in the traditional system design, with benefits and limitations discussed accordingly. The process of performing an order within the PetShop is split between the logic layer and the presentation layer. Data is retrieved to inform the process via discrete web pages, such that `OrderShipping.aspx` contains the trigger code that moves the process towards another task, when the user has completed the page and clicked a relevant button. The execution of the process then passes to the logic layer to interpret the data, and store this within the database. Once stored, the logic moves the execution to produce another page for the user. As such, the classes within the logic layer can be said to change the state of the process by signifying that a task is complete, and that data is added to the data layer structures, as these are the side-effects of its actuation. Similarly, pages within the presentation layer can be said to introduce data to the state of process, and signify that the task of producing data is complete. In this way, exposing the logic layer and presentation layer classes with NBLOs to Neptune will allow the operational semantics of the classes to be imparted.

3.2.1 Writing Neptune Support

Listing 1 shows the description of shipping components within the logic layer and the presentation layer. In this way, the `nbloOrderShipping` describes the method `getShipping` within the presentation layer as adding the features `ShippingAddress`, `ShippingPostcode` and `ShippingBand` to the `orderID` element within the state of the process. Similarly, the `nbloSaveOrderShipping` adds the feature `Shipping` to an element Database to describe the fact that operationally, the method `ProcessShipping` in the logic layer of the PetShop saves shipping data to the database. Both actuation calls are deemed synchronous as the methods `getShipping` and `ProcessShipping` should complete before continuation of the process flow. Values are set for the features by way of the return value of the underlying component. As such, `ShippingAddress` will take the value of the return the property `ShippingAddress` of the object returned by the method `getShipping`. Other values can be specified by the as operator in Neptune, such that the values of the return type can be reflected.

```
1 define nbloOrderShipping with NString
orderID
2 {purpose
4   {
5 feature ShippingAddress to orderID ;
6 feature ShippingPostcode to orderID ;
7 feature ShippingBand to orderID ;
8   }
9 actuation
```

```
10 {
11 // cal l the presentation layer
12 call
BaseLanguage.Csharp(”orderPL.dll”,
13 ”getShipping”,
14 orderID,sync);
15  }}
17 // logic layer component
18 define nbloSaveOrderShipping with
NString orderID
19 {purpose
21      {
22 feature orderID.Shipping to Database ;
23      }
24 actuation
25      {
26 // call the logic layer
27call  BaseLanguage.Csharp(”orderLL.dll”,
28 ”processShipping”,
29 orderID,sync);
30     }}
```

Listing 1: NBLO definition of logic and presentation components.

The process itself can be said to act from discrete tasks that each require particular information to be available within the state of the process. The process model used within the PetShop to perform an order is given in figure 4.

As each stage in the process has a related class, it can be said, for example, that before entering the task designated for taking shipping information from the customer, data of the products ordered must first be available. In this way, axiomatic semantics for the tasks can be set based on information required by the task. Similarly, the task itself requires information to be added to the state after its completion, such that for the class `OrderShipping` the data relating to the shipping details of the customer are the task performed by that part of the process.

Figure 4: Original order process

The next listing shows the process `OrderShipping`, that requires that shipping data should be available in the state element `orderID`, and that this should be saved to the database, or rather `Database.Shipping` should be available given the ordered element.

```
1 task OrderShipping with NString
orderID
2 {requirements
5 //need data
6 needAddress:require
orderID.ShippingAddress;
7  needPostcode:require
orderID.ShippingPostcode;
8 needBand:require orderID.ShippingBand ;
9//need to store this data
10 saved:require
Database.Shipping(orderID ) ;
12       }}
```

Listing 2: Task Definition for Shipping.

The semantic linking module, when presented with the task description, can then locate the NBLO that can provide the data required to be completed by the task, providing a level of autonomy between process description and the logic and presentation layers of the application. As both actions are synchronous, the execution of the actions will take place in sequence, allowing the data to be located, and added to the database.

```
1 task OrderShipping with NString
orderID
2 {requirements
4  {
5 //need data
```

```
6 needAddress:require
orderID.ShippingAddress;
7  needPostcode:require orderID.
ShippingPostcode;
8 needBand:require  orderID.ShippingBand  ;
9 //need to store this data
10 saved:require
Database.Shipping(orderID) ;
11       }
13//as processed by the semantic linker
14 actions
15 {needAddress,needPostcode,needBand
via nbloOrderShipping;
17 saved via nbloSaveOrderShipping ;
18     }}
```

Listing 3: Task Definition for Shipping.

The process model, for completeness, is given in Listing 4.

```
1 process processOrder
2  {tasks
4 {if (validateAccount == false)
6                     end;
7 orderBilling giving orderID ;
8 orderShipping (orderID) ;
9 orderProcess (orderID) ;
10    } }
```

Listing 4: Task Definition for Shipping.

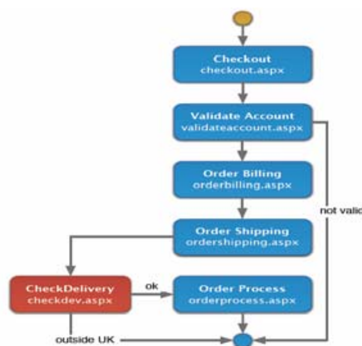

Figure 5: Adapted order process.

3.2.2 Adapting the Process Model

New behaviour can be described by way of creating a new NBLO relating to the new component, and the process model adapted to introduce the behaviour as and when required. The semantic linking process reassesses the completion of the process in light of the new task, and relates it to the new functionality. Thus, runtime adaptation occurs. A refined process model shows that a new task `CheckDelivery` should be completed. This check is introduced to ascertain whether the customer is shipping to a UK based address. If the customer is not UK-based it should be cancelled.

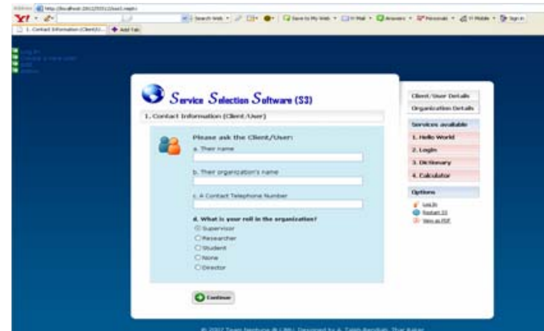

Figure 6: Screenshot for S3 web application.

As shown in figure 6, to illustrate Neptune support for runtime self-adaptation a Wiki-based tool is developed, which enables the edit of the process (intention) model (Fig. 7). This triggers Neptune via the controller and the semantic linking component to map processes to the required web services and activate them ready to be used via the intended web application. As such, the complete operational semantics encapsulated within the process `processOrder` can be changed without impact to the original application, providing a high level of runtime adaptation. In contrast, for the same behaviour implemented in pure C#, for the new PetShop application required the class `OrderShipping` to be changed. The class `AddressCheck` was added to the logic layer assembly, and a conditional statement added to the `ProcessOrder` method of the `OrderShipping` class. The application was then re-compiled, and re-deployed on the server.

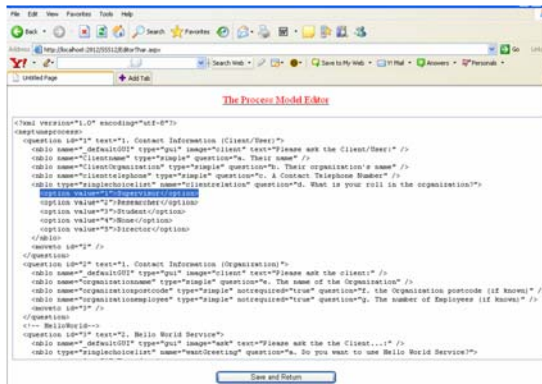

Figure 7: Screenshot for the process model Editor.

4 RESULTS

Using the criteria set forth in the original disclosure of PetShop, the following results were obtained for the execution of a complete order process. In this way, these results will show the impact of using Neptune in terms of performance. To this end, Microsoft ACT was used to provide dummy information to the pages so that the process could complete via machine instruction. Averages of 10,000 process executions were made with 100 concurrent visitors. Whilst the original PetShop did not include any need for reflection, it can be considered vital, due to the relationship between runtime analysis and adaptation, that for a system to be available for adaptation, reflection is required. As such, the same test was performed using a modified PetShop implementation that called its methods via reflection. In this way, a benchmark for adaptation of any form, could be ascertained.

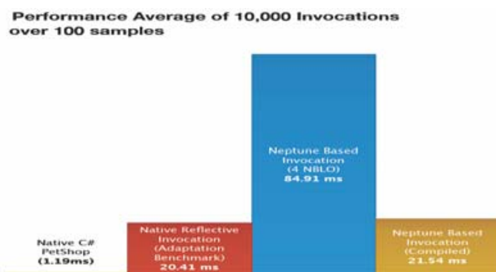

Figure 9: Performance relationship.

The results indicate that the overhead of using Neptune is significant upon the first execution of the process (Fig. 9). This is due to the semantic linker having to process the available NBLO descriptions and refactor the description to perform the task. Upon further execution however, as the tasks have already been provided the NBLOs that satisfy their requirements, execution time improves dramatically, introducing a slight overhead in terms of computational performance over the adaptation benchmark. Such overhead is introduced by the way Neptune dispatches the call to perform the executions, and for Neptune itself to locate the actions required. In terms of CPU usage, the processes of Neptune produced more in the way of computational complexity than the traditionally compiled application. This is expected due to the interpretation that occurs at runtime via use of Neptune. The result of using reflection produced a slight increase in CPU consumption (Fig. 10). However, the compiled Neptune system – produced after the first semantic linking – performed relatively worse.

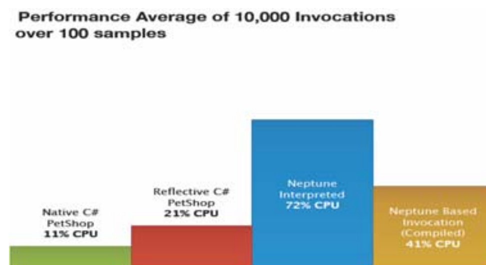

Figure 10: CPU consumption relationship

It should be noted that the implementation of the Neptune framework is such that it was designed as a prototype, rather than being optimized for performance. As such, it could be expected theoretically, that any overhead can be reduced further with subsequent revisions of the implementation of the framework.

5 Critical Analysis of the Approach

Significant limitations were identified during architecting the case study. As the application was web-based, the model for page execution is stateless. In this way, the operating environment provided for

by .NET acts to manage the session variables for the user, such that a cookie is formed to save data with the user. As the Neptune process executed data away from the operating environment offered by .NET, and instead instigated the calls directly itself, session data was lost by the process as it executed. Storing session data with the server circumvented this, however this increases complexity. It is also evident that the process descriptions and NBLO definitions must match specifically in terms of ontological models. If an NBLO was created such that it described its effect as adding a variable *AddressForShipping* to the process state representing the shipping address, and a task required *ShippingAddress*, the machine would not be able to relate the two variables by name alone, and as such would not be able to link the NBLO to the task. Though further extensions to the semantic linking components are underway to address this deficiency including the runtime generation of NBLOs.

6 CONCLUSIONS

Neptune is still in its infancy for testing and refinement of its methodology. Issues of complexity in type safety, and scalability of the semantic linking processes employed in Neptune are particularly pressing in terms of fulfilling Neptune's promised behaviour. It is foreseeable for example, that as the number of NBLO and semantic concerns increase, the difficulty associated with maintaining optimal behaviour will increase. As is evident in this paper however, the method has shown promise in both theoretical and practical application for providing autonomous behaviour bounded by behavioural and architectural concerns.

As such, work is underway to provide adaptable behaviour via Neptune to enterprise level templates devised by both Sun Microsystems and Microsoft to better inform how best to refine the methodology presented here. This is currently tested by comparing the implementation of the PetShop template in Neptune via a set of adaptation tasks, to assess the impact and difficulty in completing the tasks in each implementation. As with the performance issues of Neptune, the ability of Neptune to provide exposition of the actions needed to produce a process and satisfy its requirements, mean that issues of scalability can be largely circumvented after the semantic linking process has completed.

REFERENCES

Agarwal, M, Bhat, V, Liu, H, Matossian, V, Putty, V, Schmidt, C, Zhang, G, Zhen, L, Parashar, M, Khargharia, B & Hariri, S 2003, 'AutoMate: Enabling autonomic applications on the grid' , Proceedings of the Autonomic Computing Workshop, pp. 48–57.

Caseau, Y 2005,'Self-adaptive middleware: Supporting business process priorities and service level agreements', Advanced Engineering Informatics 19, no. 3, 199–211.

Castelfranchi, C 1995, 'Guarantees for autonomy in cognitive agent architecture', ECAI-94: Proceedings of the workshop on agent theories, architectures, and languages on Intelligent agents (New York, NY, USA), Springer-Verlag New York, Inc., pp. 56–70.

Engel, M & Freisleben, B 2005 , 'Supporting autonomic computing functionality via dynamic operating system kernel aspects' , AOSD '05: *Proceedings of the 4th international conference on Aspect-oriented software development (New York, NY, USA),* ACM Press, pp. 51–62.

Grace, P, Blair, G & Samuel, S 2005, 'A reflective framework for discovery and interaction in heterogeneous mobile environments, SIGMOBILE Mob', Comput. Commun. Rev. No. 1, 2–14.

Khoumbati, K, Themistocleous, M & Irani, Z 2006, 'Evaluating the Adoption of Enterprise Application Integration in health-Care Organizations', Journal of Management Information System, volume 22 Issue 4,pp 69-108.

Kurbel K & Loutchko, I 2003, 'Towards multi-agent electronic marketplaces: what is there and what is missing?', Knowl. Eng. Rev. 18, no. 1, 33–46

Miseldine. P & Taleb-Bendiab, A 2006, 'Retrofitting Zeroconf to Type-Safe Self-Organising Systems', Proceedings of the 17th IEEE International *Conference on Database and Expert Systems Applications (DEXA'06) (Los Alamitos, CA, USA),* IEEE Computer Society, pp. 93–97.

Miseldine. P & Taleb-Bendiab, A 2007, 'Neptune: Supporting Semantics-Based Runtime Software Refactoring to Achieved Assured System Autonomy', Technical report, DASEL Technical Report 2007/07/PM01, Liverpool John Moores University, 2007.

http://www.cms.livjm.ac.uk/taleb/Publications/07/TR-2007.07.PM01.pdf.

Miseldine. P 2007, 'Language Support for Process-Oriented Programming of Autonomic Software Systems', PhD thesis, Liverpool John Moores University.

Sajjad,A, Jameel, H, Kalim, U, Lee, Y & Lee, S 2005, 'A component-based architecture for an autonomic middleware enabling mobile access to grid infrastructure., EUC Workshops, pp. 1225–1234.