Neptune: Supporting Semantics-Based Runtime Software Refactoring to Achieve Assured System Autonomy

Philip L. Miseldine[♭*] and Azzelarabe Taleb-Bendiab[♭†]

[♭]*Department of Computing and Mathematical Sciences,*
Liverpool John Moores University, Byrom Street, Liverpool. L3 3AF

Abstract

We consider language support for the domain of runtime software adaptation from the perspective of codebase analysis and self-adaptation, whilst guaranteeing predictable autonomic software behaviours. The paper reviews the current state-of-the-art mechanisms employed to actuate runtime analysis and adaptation, followed by the definition of a set of language requirements to support the development of assured autonomic systems. A supporting framework and language Neptune is defined to meet these goals, which is here illustrated using a set of examples taken from a case study that highlights the behavioural and architectural autonomy produced by Neptune. The discussion concludes with further examples that highlight the manner in which Neptune can ensure constraints and requirements are met during adaptation processes.

1 Introduction

With more complex tasks being represented by software, the difficulty associated with the isolation and co-ordination of adaptation in software is increasing [BNG06]. Similarly, the heavy reliance on software in the operation of core infrastructure of commerce and government and national concerns [AJKP04], is driving the need for continuous online systems that reflect current knowledge and process to be available. As an example, knowledge in healthcare best-practice is often-changing, and software used to aid decision processes are required to remain up to date to produce best-practice care [RMN+03]. Delays in updating software mean patients are diagnosed using out-of-date medical knowledge and guidelines. Coupled with distribution, where different parts of the application may be executed and stored on different machines, the complexity of deployment and its operating environment drives the need for computational techniques to aid in the analysis and actuation of changes in requirement whilst the software remains operational.

Research into Autonomic Software has yielded many potential solutions to adaptation with respect to complexity [BMK+05], often from the viewpoint of providing self-aware behaviour to predict and react to changes in requirements for preservation [KC03]. As identified in [ST05] however, a systematic approach to the adaptation of software at runtime to new requirement is still an outstanding issue for research. In addition, providing adaptation based on assured, bounded governance requires knowledge of not only what should be adapted, but also its impact 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 system is vital for bounded autonomy through adaptation to be accomplished [AE88].

Issues arise when this knowledge for autonomy is required to be inferred at runtime, and adaptation is required to be instigated. Often times, applications are deployed in a compiled denotation unsuited for reification and co-ordinated adaptation. Similarly, the compilation and language used removes much of the semantic direction of the code: code is commonly written as instructions to achieve a goal, without implicitly specifying why those specific instructions were chosen for the task. In modern, distributed

---

[*]E-mail address: p.l.miseldine@2004.ljmu.ac.uk
[†]E-mail address: a.talebbendiab.ac.uk

systems, it is often the case that source code is not available for adaptation, with the use of *software as a service* model hiding the implementation from consumers. As a result, much research focuses on behaviour driven adaptation and tuning of an application, rather than focusing on the construction of its deployed codebase.

It is the contention of this paper that the functionality and knowledge required for autonomic adaptation of a system can be achieved via semantic analysis of the core componentry and codebase of the software itself. This analysis can in turn inform traditional techniques of critique used in Autonomic Software, such as emergence identification and signal-grounding. The paper highlights the limitations in current research into computational runtime adaptation and moves to identify properties of a new language NEPTUNE that attempts to provide the requisite features to meet the goal of runtime computational adaptation of a system. Several proof of concept scenarios are given to further illustrate the approach advocated by NEPTUNE and to justify a semantics-based approach. The paper concludes with discussion of limitations introduced by the approach, with work relating to possible future solutions given.

2 The Runtime Requirement

A classically described problem when analysing software-based behaviour computationally is its inherent complexity. To begin to assess and reify a system based on its codebase therefore, appreciation for code complexity is needed. In language-based research, the need for software to use computational techniques to improve the design and structure of applications and their codebase is well documented for both operational and behavioural concerns with respect to software adaptation and evolution [Leh96]. When the requirements of an application change, having a codebase that is both well organised and structured can decrease the complexity of the task of introducing the change, as well as increasing the integrity of the change [Ber94].

Software refactoring has traditionally involved techniques that change the structure of source code without changing executed behaviour to improve readability and the usefulness of code in respect to its re-use and integrity [vD02]. Other approaches have attempted to improve the construction of applications via separation of concerns, a form of modular organisation exemplified by object-oriented programming, and to a greater extent, Aspect-Oriented Programming (AOP). These techniques, however, are principally aimed at design-time code analysis. AOP frameworks, such as ASPECTJ [KHH+01], help isolate change, however require the developer to specify the join points, define aspects, and indicate how they are to be woven. In effect, these techniques require human intervention to trigger their use, or human knowledge to define their form at some point. As such, as classical techniques they are not well-suited to runtime analysis and adaptation as modification often requires recompilation.

2.1 Runtime Adaptation

Dynamic AOP, as implemented by many AOP frameworks such as ASPECTWERKZ, allow aspects to be deployed and undeployed at runtime using specific features of the host language or operating environment. In the case of ASPECTWERKZ, extensions to JAVA such as HOTSWAP or JVMTI are used. These allow classes to be reloaded into the virtual machine after changes are made to their construction to provide a level of runtime adaptation. It should be noted that these changes are made via human interpretation, though frameworks exist that monitor and can assess in deliberating on the semantics of the aspects and concerns, such as PROSE which provides metadata representations of aspect weaving [PGA02], EOAP [DMS01] that modifies source code to include monitoring features, and AXON which uses a form of axiomatic semantic, ECA, to control monitoring [AH03]. Whilst providing monitoring, they still rely on human input: the ECA rules in AXON for example, still require elicitation before being included.

Such reliance on features of the host language ensure that dynamic AOP can only be realised on a specific implementation: a non-JAVA implemented module in a distributed application based on a JAVA AOP framework cannot benefit from the same runtime adaptation as native JAVA components. This reliance often stems from the fact that commonly used object oriented languages such as JAVA are statically compiled and the semantics and relationships implied in the source code cannot be modified at runtime. Solutions to these inherent limitations often involve producing specialist execution environments, such as the JVM used in JAVA, as shown by IGUANA/J [RC02].

2.2 Runtime Analysis

In tandem with allowing code changes at runtime, to allow computational processes to assess where and in what form change is required, a level of reflection and introspection is required by the host language. Structural reflection abilities in modern OO languages exist that allow methods to be analysed and fields to be identified, however it is not possible ordinarily to add a method nor add a field without recompiling the application. As with dynamic AOP, these limitations are often circumvented via extensions to the execution environment. For example, behavioural reflection – a process of reflecting on the behaviour produced by a system, rather than its construction – is often implemented as an extension to a language using a framework or toolkit such as KAVA [WS01], or via a dynamic proxy approach as used in JAVA that approximates method invocations to provide analysis.

Other approaches to runtime analysis include exposing the Abstract Syntax Tree – the structured analysis of a source code by a parser – to allow full inspection of the source code to be made. These are both specialised, such as JAVAML [Bad00] for JAVA, and generic representations [CMM02]. Coupled with XML, portable forms of code can be achieved [ZK01]. Such a technique has been used to aid dynamic AOP in an *operator approach* [SPS02], though as with existing methods, the operator approach relies on the existing representation of advice and weaving via a compiler for it to be enacted. Unlike classical AOP methods however, the operator approach, by using XML based ASTs, has shown that code can be analysed effectively via tools such as XPATH to embed concerns, such as monitoring. These results indicate the feasibility of using an XML based AST for analysis, yet do not approach the problem of interpreting why the AST is represented as it is: in effect, the semantics behind the decisions to organise and write the codebase are not represented.

The form of compilation for general-purpose languages, as typified by GCC and Lex [LS75], do not cater for explicit semantic representations to allow a machine to elicit why the codebase is written and organised as it is [Gun92] as ordinarily, this is the role of the developer. Techniques such as Design By Contract (DBC) [Mey91] use a combination of axiomatic and denotational semantic forms to imply constraints for a class or behaviour. Extensions of existing languages have been made to further refine and support such descriptions, such as NANOJAVA [ON02] though are limited to specific semantic representations and design-time inclusion. New holistic paradigms such as the Semantic Web have emerged to help co-ordinate semantic descriptions of web-based software components such as OWL-S [PV04] though imply web-based architectures to be used.

2.3 Autonomic Computing

Work conducted under the Autonomic Computing initiative has attempted to predict and react to changes in software requirements for self preservation. Using the naturally-derived autonomous behaviour exhibited by living organisms, research into Autonomic Computing is often linked with runtime adaptation and analysis to provide solutions to imbue self-aware behaviour to machines [PH05]. A wide variety of solutions exist, often targeting different aspects of behaviour in terms of implementation and deployment. Autonomic middleware frameworks such as [SJK+05, ABL+03] target Grid-based computing to aid in self-management, REMMOC [GBS05] targets mobile environments, whilst the author's own Cloud Architecture can be applied to disparate, localised networks [MTB06b] for self-organisation.

As stated in the introduction, Autonomic Computing has traditionally viewed systems from a behavioural perspective via monitoring. There are techniques however, that use codebase analysis. TOSKANA [EF05] and similar approaches use AOP as a basis for adaptable change, however these changes are enabled by specific modifications to one or more OSI layer, 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 been made using agent theory [Cas95] that treat a complete system as a series of agents, each with their own behaviour models to collectively perform set tasks and behaviours. Effectively co-ordinating and providing communication between agents, essentially providing unified, directed behaviour, are cited as research challenges of such research [KL03].

Other problems arise from the lack of availability of source code. Often in the systems discussed in Autonomic Computing research, such as service-oriented architectures, only consumption of behaviour is available with the actual implementation and its source code hidden. In these cases, adaptive middleware components that marshal and re-route calls are popular solutions [Cas05], however often work at an abstraction layer above the codebase, often via policy documentation, itself a form of metamodel of behaviour

that relies on observation for tuning.

3 Proposed Solution

The preceding section highlighted several research challenges that face runtime adaptation for applications, whilst situating the problem area within those covered by Autonomic Computing. Accordingly, qualities required in a proposed solution, named NEPTUNE, that can enable full runtime adaptation of a distributed, component-based application can be sought based on this critique.

As discussed, extensions to specific languages, such as JAVA, are a popular solution to circumvent the static nature of modern OOP languages. By basing an adaptive platform upon such specialist editions of languages however, components written in languages or executed in an environment other than those specially designed are isolated from adaptation in the complete system. Goal 1 addresses this limitation:

Goal 1 NEPTUNE *must not rely on an extension to, or specialism of, a specific language, operating system, or other system component to function. In effect, it must be* deployment-platform agnostic.

Related to this goal, any new solution written to perform adaptation must therefore transcend individual languages and development environments, and instead be made available to languages via API or other standardised forms, leading to Goal 2.

Goal 2 NEPTUNE *must be accessible such that it has the ability to be dispatched by, and respond via, standard methods of invocation and communication. In effect, it must be* development-platform agnostic.

Goal 1 and 2 imply that the solution can allow code written in perceivably any language to be made accessible for analysis and implementation in NEPTUNE. These qualities may be tested by applying the solution to two different development and deployment environments, and adapting their combined behaviour.

As is evident from the previous discussion, it is often the case that actual source code is not available for analysis, and that analysing fully a variety of languages and maintaining their behaviour is unfeasible given with breadth and depth of programming languages in use today. In addition, complete analysis of all programs produced by a Turing-complete language is computationally impossible. As such, a semantic reification of code in terms of its purpose and meaning is needed. In this way, code can be described in terms of what properties that must hold for it to safely achieve its goals (axiomatic semantics), the goals and tasks it is to complete (operational semantics), and how this is to be achieved (denotational semantics). This information is known by the developer of the code in any case, it is simply obfuscated by the denotation and compilation process.

Once defined and exposed, these semantics of individual components can be matched against the semantic description of an application and the processes they represent to provide a form of behaviour binding. This leads to Goal 3.

Goal 3 NEPTUNE *must be expressive enough to* represent the semantics of base language components *– which themselves may be implemented in any programming language that is supported in Goal 2 – such that the effect, requirement, and mechanism of actuation of the component can be ascertained at runtime.*

As components are exposed in a semantic form, the application itself can be expressed in terms of tasks or processes that it needs to accomplish. As shown by the popularity of modelling methodologies such as UML and BON, and in particular research relating these approaches to semantic forms [KC01], specifying what an application should do is a task eminently suited to human interpretation and current, well-accepted computation methods. Specifying *how* these requirements are to be achieved is a task better suited to computational process due to the complexity and fluctuation of the possible ways in which the task can complete. For this reason, in keeping with classical classifications [Rol98], an activity-based process model is required, forming a strategic model of a process from the semantics defined in each of the activities contained or referenced in the model. This leads to Goal 4.

Goal 4 NEPTUNE *must* express an application in terms of the processes *it is to represent using an activity-based classification, using the same semantic forms defined in Goal 3.*

Thus, rather than being built or woven from components at compile-time, <mark>the application becomes a container for processes described using</mark> NEPTUNE, <mark>that are linked at runtime to available base language components</mark> – themselves described in Goal 3 – that represent solutions that need to be achieved though without specifying the actual source code invocations needed. In effect, the axiomatic and operational semantics expressed within the process model are used to infer the denotational semantics of an actuation of the task given. The process of semantic manipulation and inference are well understood in literature [Sch86], particularly in linguistics, and such techniques are used to inform other frameworks that attempt to view intentions in a semantic form, as exemplified by Action Semantics [ES90].

As components operate within tasks and processes, for completeness Goal 5 enforces that the effects of a component must be modelled operationally by changes in the state of the application or process used so that the process reflects the behaviors actuated in the components.

Goal 5 *The intended purpose of a component as given in Goal 3 must be defined by its side effects to the state of the process it is used within.*

To adapt an application at runtime in terms of both its behaviour and therefore its construction, all semantic information expressed in NEPTUNE must be changeable and interpretable at runtime. In this way, if a base language component changes or is introduced, its semantic representation can also change. As NEPTUNE by way of Goal 3 exposes such information, it is critical that this information be adaptable at runtime, as the semantic matching employed in the solution will be able to interpret these changes or additions, and adapt the linked component model accordingly. Such analysis can take place by exposing the AST of parsed semantics and structures into an object model at runtime which may be deliberated upon, as discussed in subsection 2.2. This is detailed in Goal 6.

Goal 6 NEPTUNE *must reify in an open manner the complete composition of all information described through it, and enable the resultant model to be adapted fully at runtime.*

Whilst the qualities so far have dealt with information pertinent to base language components, whose access to source cannot be guaranteed, it may be eminently feasible and preferable for new components to be written exclusively for use within NEPTUNE, rather than in a traditional base language. In addition, Goal 4 implies the need for an adaptable, semantically defined form of logic to be expressed in the process models, a limitation discussed previously in BPEL. This can be achieved by defining the logic using the NEPTUNE based instruction set. This leads to Goal 7.

Goal 7 NEPTUNE *must express instructions itself that can be fully described semantically, and that must be fully adaptable at runtime under Goal 6.*

To ensure NEPTUNE can be applied to Autonomic Computing, a notion of distribution is required. For an assured autonomous system to be represented, each component must be described in such a way as its purpose and operation can be ascertained, to help predict how best they should be employed and organised within the system. In particular, sympathetic and parasympathetic states of the system based on the behaviour of the individual components can be calculated from behaviour profiles [MTB05] – given by Goal 3 – to enable recuperative actions to be identified. Portability of service and code is implicit in such solutions, and the wider field of Autonomic Computer and adaptive middleware, and therefore must be supported by NEPTUNE. This leads to Goal 8.

Goal 8 NEPTUNE *must support runtime distribution of all information expressed within it via computation methods and communication standards.*

4 Implementation Details

As a conceptual basis, the qualities specified in section 3 have been formed to solve many of the outstanding issues highlighted in current research into runtime analysis and adaptation. It follows therefore, that discussion of an implementation of the conception given should take place, so that the assertions made can be qualified, and refined accordingly.

4.1 Base Language Architecture

Goal 1 and 2 both indicate that 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 so NEPTUNE can dispatch a call that is required to be handled by the base language.

It is also clear from the discussion (in particular Qualities 3, 5, and 7) 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 [Cho56], NEPTUNE will present information using a context-free language NEPTUNESCRIPT.
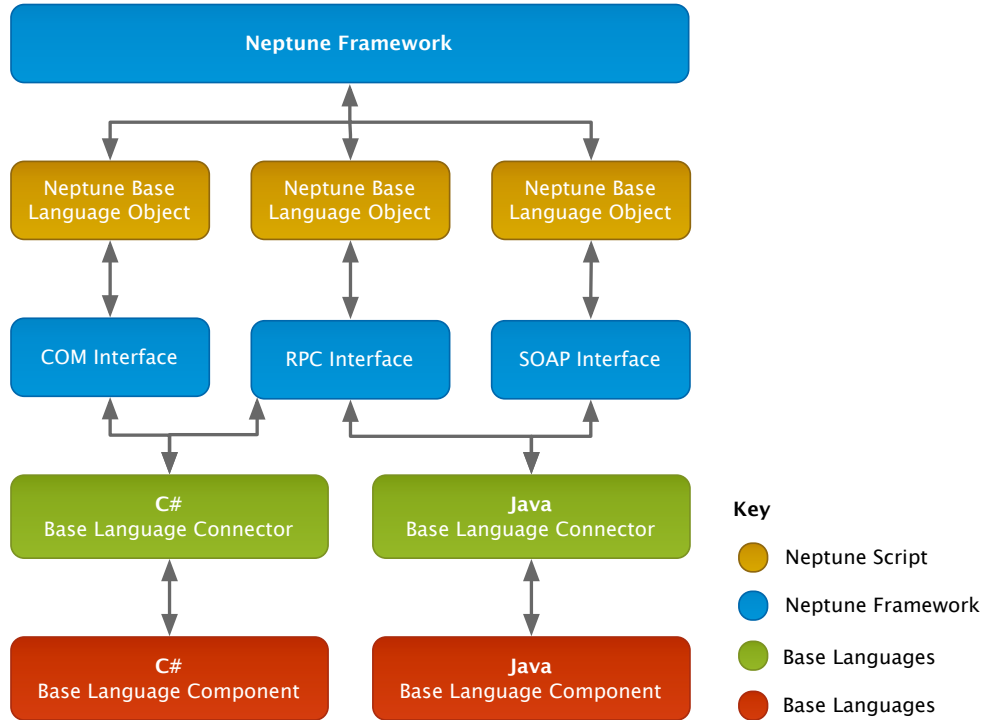

Figure 1: Base Language Interface

Figure 1 shows a schematic view of the method in which NEPTUNE communicates with base language components. The *Base Language Connector*'s are themselves written within a 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, as governed by Goal 2. 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. The NBLO itself is written within NEPTUNESCRIPT, as governed by Goal 3. 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 can be used by the NEPTUNE Framework, to invoke the behaviours described in the NBLO.

A semantic linker module within the framework encapsulates the functionality required to match and relate the semantics defined within process models, to those contained within the NBLOs to provide so-

lutions for activities or tasks given in the process. Once a match is found, the calls to the base language contained within the NBLO are actuated. In effect, the axiomatic and operational semantics of a task are related to the denotational semantics given as an actuation within an NBLO's definition. As discussed in Goal 3, the process of semantic manipulation and inference are well understood in literature and similar techniques are used to inform the linking module.

4.2   Design Basis Example

As an example to highlight the design basis of Neptune, and the form that NeptuneScript requires, an example is offered with detailed explanation. The example produces a simple task that can be used by an application to format a string as a title, using different styles which are implemented in different languages, and form a process that be updated autonomously at runtime without re-compilation. As a basis, a method `bold` written in C# is of the following form:

Listing 1: format.cs

```csharp
public class Formatting {
  public string bold(string s)    {
    return "<b>"+s+"</b>"; } }
```

A similar method `italic` can be written in Java, and is of the following form:

Listing 2: format.java

```java
public class Format {
  public String italic(string s)    {
    return "<i>"+s+"</i>"; }}
```

Both are described using operational and axiomatic semantics within an NBLO definition to show that the purpose is to provide a feature to a type defined in Neptune, `NString` to transcend the types used in base languages. In effect, the NBLO is defined by its effect on the state of any application it is part of – in keeping with goal 5 – such that the type `NString` represents the state of a string through the process being executed. Tasks that require particular features of the type, requirements that are axiomatically defined, can thus be related to the relevant NBLO that can operationally ensure conformance to the requirement.

Listing 3: NBLO definitions

```
define nblo.CSharpBold with NString s
{
  purpose  {
    feature bold to s {
      evaluate as NBool {
        return s.startsWith("<b>") && s.endsWith("</b>");
      } } }
  actuation {
    call BaseLanguage.Csharp("bold.dll",
                             "Formatting:bold", s)
  } }
```

Evaluations, if possible, of the operational semantics defined can be included to determine if the purpose has been successfully actuated. Such evaluations can be elaborate, such that Neptune can write instructions to manipulate and control data itself as ruled by Goal 7. As is in keeping with traditional language design, the evaluation should be side-effect free. Due to the `evaluate` elements' implicit relationship to a single purpose in an NBLO, it's meaning can be inferred as an evaluator of the purpose for the NBLO, to meet the stipulation of a semantic description given in Goal 7. Similarly, details of the base language component are provided in the form of an `actuation` and actual `call` needed by the base language connector, to provide the denotational semantics that the semantic linker module should produce.

Each base language connector defines the meaning of a set of base types provided by Neptune which are related to equivalent types in the base language, such that `NString` represents a string as understood

by Neptune, which is related to `System.String` in C# and the `String` type in Java via their respective base language connectors, to maintain interoperability.

With this behavioural basis, a task can be written that requires a string to have two features, bold and italic. These are named `a` and `b` respectively, so that the requirements can be referenced within the body of the task for adaptation purposes. This helps satisfy the requirement of Goal 6, that states that all information contained within Neptune should be adaptable.

Listing 4: Task definition

```
1  task FormatStringAsTitle with NString s {
2    requirements   {
3      a: require s.Bold;
4      b: require s.Italic;
5    }
6  }
```

Neptune can rewrite the task to provide an `action` element that can satisfy the requirements specified in the task by inferring suitable denotational semantics by linking them to the purposes defined in the NBLOs such that the requirements hold. In this regard, Neptune relates the semantic description given in a task to the denotational semantics provided in the NBLOs given as actual source code invocations, without human intervention. To further aid readability and assurance, as each requirement is uniquely named, the keyword `via` is used to denote which NBLOs are used to meet which requirement. This provides the maximum level of semantic-based assurance to the solution of the task: each operation to take place via the base language components can be traced and asserted to individual requirements.

Listing 5: Task definition with annotated actions

```
1  task FormatStringAsTitle with NString s {
2    requirements   {
3      a: require s.Bold;
4      b: require s.Italic;    }
5    //actions determined and outputted by Neptune
6    actions {  a via nblo.CsharpBold(s);
7              b via nblo.JavaItalic(s); } }
```

As the requirements expressed in the task are met, Neptune can then marshal the appropriate calls to the base language connectors to instigate the actual functionality. To highlight the adaptive qualities of Neptune, the task can be rewritten at runtime to only require the following:

Listing 6: Adapted task definition

```
1  task FormatStringAsTitle with NString s {
2    requirements    {
3      a: require s.Bold;
4    }
5  }
```

As the actions produced before no longer satisfy the single requirement, such that `nblo.Java.Italic` provides a feature not required by the task, this can be treated as a trigger for adaptation. This is important as the requirements defined in the task can thus act as a governance mechanism ensuring the behaviours required are actually actuated. In this way, a developer can specify actions which are asserted against the requirements and refined accordingly. The process of semantic linking can repeat given this trigger, to refine the action element of the task, giving:

Listing 7: Adapted task definition with new inferred actions

```
1  task FormatStringAsTitle with NString s {
2    requirements   { a: require s.Bold; }
3    //actions determined and outputted by Neptune
4    actions    { a via nblo.CsharpBold(s); }
5  }
```

Furthermore, a requirement can be specified to only be applied if the NBLO evaluates appropriately using the logic outlined in the `evaluate` element of the NBLO definition. As such, the task can be written to determine requirements logically.

Listing 8: Adapted task definition with new inferred actions

```
1   task FormatStringAsTitle with NString s {
2     requirements   {
3       when (!s.Bold)      {
4          a: require s.Bold;
5       }
6     }
7     //actions determined and outputted by Neptune
8     actions    {
9       when (!s.Bold) {
10         a via nblo.CsharpBold(s);
11      }
12    }
13  }
```

In this way, the feature `Bold` is defined to be evaluated to true if a string starts with "" and ends with "" as given in Listing 10. If the string does not contain this feature – it equates to `false` – then the requirement is actuated. This control structure is repeated within the `action` element of the task to optimise the process; without the control structure, the action element would require adaptation upon each execution of the task with a given string.

To show how a NEPTUNE process can be adapted to reflect new behaviours, a new assembly can be added to the runtime environment of the application, and described via a new NBLO that is registered with the controller at runtime, giving the C# class:

Listing 9: underline.cs

```
1   public class Underline {
2     public string underline(string s) {
3       return "<u>"+s+"</u>";
4     }
5   }
```

and the NBLO:

Listing 10: NBLO definitions for Underline.cs

```
1   define nblo.CSharpUnderline with NString s {
2     purpose  {
3       feature underline to s {
4         evaluate as NBool {
5           return s.startsWith("<u>") && s.endsWith("</u>");
6         }
7       }
8     }
9
10    actuation {
11      call BaseLanguage.Csharp("underline.dll",
12                               "Underline:underline",
13                               s)
14    }
15  }
```

The task `FormatStringAsTitle` can be rewritten at runtime due to its exposition, to include the requirement of the functionality provided for by the new behaviour:

Listing 11: Adapted task with reference to Underline behaviour

```
1   task FormatStringAsTitle with NString s {
2     requirements {
```

```
3        a: require s.Bold;
4        b: require s.Underline;
5    }
6  }
```

which is semantically linked and refactored by the respective modules to produce:

Listing 12: Adapted task definition with new inferred actions for underline

```
1  task FormatStringAsTitle with NString s {
2    requirements   { a: require s.Bold;
3                     b: require s.Underline; }
4   //actions determined and outputted by Neptune
5    actions   { a via nblo.CsharpBold(s);
6              b via nblo.CsharpUnderline(s); }
7  }
```

Thus, the semantic definition of `NString.Underline` implies that of `NString.Bold` to be present, allowing the task to be refactored to:

Listing 13: Inferred Bolded Behaviour

```
1  task FormatStringAsTitle with NString s {
2    requirements   { a: require s.Underline; }
3
4   //actions determined and outputted by Neptune
5    actions { a via {
6              nblo.CsharpBold(s);
7              nblo.CsharpUnderline(s);
8    }}
9  }
```

5   Case Study

Neptune has been used to develop a number of autonomic applications to test and evaluate its design, performance and further analyse its approach. In particular, NEPTUNE has been used to implement an autonomic service-oriented decision-support system, which enables via situation and role-awareness, the reconfiguration of medical decision agents to adapt to the individual preferences of expert clinicians and hospitals local clinical governance rules [MTBER07]. Below we outline NEPTUNE's suitability to produce self-adaptive autonomic systems bounded within organisational guidelines via examples of architectural and behaviour adaptation required to be implemented at runtime. In this way, NEPTUNE's suitability can be better assessed.

5.1   Architectural Autonomy

In partnership with several NHS dental trusts, the triage process of dental access services were assessed and implemented in NEPTUNE, using an activity-based process model. The model defined individual tasks that often represented data input, such that the process required knowledge of a patient. The data could feasibly be read from a database, if it was available, or from the operator directly asking the patient. As such, two NBLOs were defined to describe these methods of data input for the patient's data, given their NHS ID:

Listing 14: NBLO definition for patient lookup

```
1  define nblo.getPatientFromDatabase with NString NHSID {
2    purpose { feature Patient to NHSID {
3            evaluate as NBool {
4              return (Patient != null); }}}
5    actuation {
6      call BaseLanguage.Csharp("db.dll",
```

```
7                              "DAL:retrieveFromDatabase",
8                              NHSID) giving NHSID.Patient
9    }}
10
11   define nblo.getPatientFromOperator with NString NHSID {
12     purpose  { when !nblo.getPatientFromDatabase {
13             feature Patient to NHSID {
14               evaluate as NBool {  return (Patient != null); }}}}
15     actuation {
16       call BaseLanguage.Csharp("ui.dll",
17                              "UI:preparePatientEntry",
18                              NHSID) giving NHSID.Patient
19   }}
```

As such, both NBLOs have the same purpose, however define different base language methods of obtaining the data. In addition, the `getPatientFromOperator` purpose specifies that it should only be enacted if the data was not available in the database. A task was setup to obtain this data, however, how the data was to be found was left to NEPTUNE to deliberate upon. In this way, the actual behaviour within the system of locating a patient was left to autonomous control, it is not specified in any source code other than via intentions and semantics of its desired effect.

Listing 15: Task Definition for Patient Data Lookup

```
1    task obtainPatient with NString NHSID {
2      requirements   { a: require NHSID.Patient; }
3    }
```

The semantic linking module in NEPTUNE relates this requirement to the following actions:

Listing 16: Task definition with new inferred actions

```
1    actions {
2      a via { nblo.getPatientFromDatabase(s);
3            when (!s.Patient)
4               nblo.getPatientFromOperator(s);
5    }}
```

During the lifetime of the application, a new Electronic Patient Register was produced. The existing mechanism for retrieving data now had to not use the local database, but rather the new EPR via a web service interface, written in JAVA. Accordingly, the existing process model still held, however any reference to `nblo.getPatientFromDatabase` was required to be refactored to point towards a new NBLO `getPatientFromEPR`, which semantically defined the new mechanism for retrieving data. For brevity, its source is not fully included, however due to its web service basis, it set a requirement that a SOAP endpoint was made available to it before it could execute, giving a different semantic to that of the database.

Listing 17: Refactor commands in an NBLO

```
1    define nblo.getPatientFromDatabase with NString NHSID {
2      purpose  { refactor.redirect(nblo.getPatientFromEPR) }
3      actuation {
4        refactor.rewrite(nblo.getPatientFromDatabase,nblo.getPatientFromEPR);
5                  refactor.remove(nblo.getPatientFromDatabase);
6    }}
```

The `refactor` types are pre-defined blocks of utility, executed by NEPTUNE via its reified sources. In this way, when called, the NBLO will rewrite any reference to the existing NBLO in the tasks and processes to the new method required. The original NBLO is then removed from NEPTUNE, giving a redirection of behaviour at runtime. These mechanisms are the same as employed by IDE's such as ECLIPSE and VISUAL STUDIO, and as such follow classical research into refactoring. In NEPTUNE however, such refactoring is machine-led and performed at runtime to the existing source base without a recompilation, as the code is simply re-interpreted.

Rather than being at a pre-compile stage however, the interpreter handles refactoring tasks given, such that when called, it modifies the source code directly to instigate the change. Thus, rather than applying the policy in a meta-form, either via additional policy or monitoring, the changes are applied at the source code level, leading to cleaner and more manageable code base, in keeping with the benefits expressed in [Leh96].

To conclude, the actions are re-interpreted after the `refactor` commands were interpreted. As the `getPatientFromEPR` NBLO required a SOAP service endpoint to be discovered, the actions, after re-interpretation, contain the new requirement set forth by the NBLO, to yield line 7:

Listing 18: Task with new refactored actions

```
1  task obtainPatient with NString NHSID {
2    requirements   {
3      a: require NHSID.Patient;
4    }
5    actions {
6      a via { nblo.locateServiceStructure giving se;
7              nblo.getPatientFromEPR(s) using se;
8              when (!s.Patient) { nblo.getPatientFromOperator(s); }
9  }}}
```

Thus, any point in which the application required a feature from the database was automatically refactored at runtime to provide the same feature from a different approach that itself required modifications to the source based on its own requirements, to maintain integrity. As all inferences are semantic based, changes to the requirements and properties of the new system could easily be instigated, and via the checks made by NEPTUNE, full adaptation of the denotational semantics used to represent the task could be made. In addition, the project saw how NEPTUNE could be used to describe components described in different languages, and how these could be combined to produce a single solution to a set process. This was achieved without requiring the execution sequence of components to be specified. It was, instead, inferred autonomically from the boundaries described semantically via process models.

5.2   Behavioural Autonomy

Using NEPTUNE, the process that occurred after pain levels were ascertained could be modified directly, producing a unified decision model for each clinician. Axioms set forth in the base decision model were set, allowing behaviours such as instructing a patient to go to hospital to be ratified against institutional procedures. In this way, as long as these were met, the requirements of the model could be instigated. Thus, the task, `afterPain` could be rewritten, at runtime, thus:

Listing 19: Updated task behaviours

```
1  task afterPain with NInt painLevel, Patient p {
2    requirements   {
3      when (painLevel < 3) {
4        p.ToHospital = true;
5      } else {
6        p.Appointment.Emergency != null;
7      }
8    }
9  }
```

Other tasks and NBLOs could be written to extend the definition of `p.ToHospital` such that this can only occur if there is space in the hospital, or waiting times are lower than a specified value.

As each aspect of information is defined independently, the cause and effect of such policy does not need to be determined: NEPTUNE relates the semantics to an end-goal, rather than requiring human intervention to achieve runtime autonomic refactoring of behaviour. Where a behaviour is deemed unsuitable, a boundary is formed, such that if no hospitals were available to send a patient, the preference could not be safely realised, and as such, control defaults back to the original model which is rigourously defined. In this way, behaviourally, concerns can be autonomously introduced to the system with assured behaviour through maintaining the purposes and requirements of the system.

5.3 Bounded Behaviours

In terms of self-governance, the manner in which NEPTUNE allows computational encoding using semantically well-defined language primitives enables a monitoring service to annotate new requirements and purposes based on observed behaviours. For example, a task can be assigned priority that itself is defined semantically as a machine-level requirement. Via observation, the performance effect of each NBLO can be assessed and annotated as a purpose of the NBLO. Thus the semantic linking procedure in NEPTUNE will automatically assess and take account of the observations that are analysed and updated in real-time within the system to ensure that the inferred operations are not too costly, and thus will not take the system out of control of the boundaries defined. Given a choice of NBLO actuation, this strengthens the selection process by improving the criteria used for assessment to produce trustworthy behaviour.

Listing 20: Observation-annotated NBLO requirement

```
task
{
  requirements { a: machine.idlecpu >~ 50;
                 b: Data.DecryptResult != null; }

  actions { a,b via {
                      nblo.DecryptDataOnCluster } }
}

define nblo DecryptDataOnLocal with Data
{
    purpose { feature DecryptResult to Data;
              //from observation service
              machine.idlecpu = 30 }}
define nblo DecryptDataOnCluster with Data
{
  purpose { feature DecryptResult to Data;
            //from observation service
            machine.idlecpu = 90 }}
```

Here, the requirement `a` and `b` is fulfilled by `DecryptDataOnCluster` due to the observations of behaviour given to the NBLOs, namely that `DecryptDataOnCluster` operates with 90% idle CPU (line 19), whereas `DecryptDataOnLocal` has been observed to operate with only 30% idle CPU (line 14). If later the cluster becomes overloaded and the NBLO is observed to operate with only 20% idle CPU, the requirements will only be fulfilled by the local service, such that it is closer to the requirement set in line 3. Thus the actions will be re-assessed and re-written at runtime, and behaviour provided for by `DecryptDataOnLocal`. These requirements themselves can be inferred by the process itself, such that criticality can propagate down to individual tasks and thus, trustworthy behaviour can be assured.

Similar mechanisms can be used for assessing risk, in that critical tasks and processes can be defined, and their operation monitored accordingly via techniques such as signal grounding that in turn create and refine the semantics of the operation. An example of this technique is given in [RTBM06]. In effect, observation services can be built that provide further semantic definition for individual NBLOs, tasks, and processes dependent on the granularity needed. This information is then used to produce behaviours based on this additional information, allowing the system to introspect its execution environment to refactor its behaviour autonomously ensuring trustworthiness of the solution inferred.

Work has been undertaken previously and is on-going to provide such observation and co-ordination of the semantic concerns expressed within a system using CA-SPA [MTB06a]. CA-SPA provides a zero-configuration network layer that facilitates communication and introspection between agents operating within the system. By way of CA-SPA therefore, the observation services are separated from NEPTUNE itself, allowing disparate observation services to be written. NEPTUNE processes the annotations made via these mechanisms, to ensure they are represented in the software's behaviour.

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 behaviours. 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 being undertaken to provide adaptable behaviours via NEPTUNE to enterprise level templates devised by both SUN MICROSYSTEMS and MICROSOFT to better inform judgement over how best to refine the methodology presented. 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.

References

[ABL⁺03]   M. Agarwal, V. Bhat, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, L. Zhen, M. Parashar, B. Khargharia, and S. Hariri, *AutoMate: Enabling autonomic applications on the grid*, Proceedings of the Autonomic Computing Workshop, 2003, 2003, pp. 48–57.

[AE88]   Sharon Anderson and Deborah Estrin, *On supporting autonomy and interdependence in distributed systems*, EW 3: Proceedings of the 3rd workshop on ACM SIGOPS European workshop (New York, NY, USA), ACM Press, 1988, pp. 1–4.

[AH03]   S. Aussmann and M. Haupt, *Axon – dynamic aop through runtime inspection and monitoring*, First workshop on advancing the state-of- the-art in Runtime inspection (ASARTI'03), vol. 1, 2003.

[AJKP04]   P. Adeshara, R. Juric, J. Kuljis, and R. Paul, *A survey of acceptance of e-government services in the uk*, 26th International Conference on Information Technology Interfaces (2004), 415–420 Vol.1.

[Bad00]   Greg J. Badros, *JavaML: a markup language for Java source code*, Computer Networks (Amsterdam, Netherlands: 1999) **33** (2000), no. 1–6, 159–177.

[Ber94]   Paul L. Bergstein, *Managing the evolution of object-oriented systems*, Ph.D. thesis, College of Computer Science, Northeastern University, Boston, MA, June 1994.

[BMK⁺05]   Dharini Balasubramaniam, Ron Morrison, Graham Kirby, Kath Mickan, Brian Warboys, Ian Robertson, Bob Snowdon, R. Mark Greenwood, and Wykeen Seet, *A software architecture approach for structuring autonomic systems*, DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software (New York, NY, USA), ACM Press, 2005, pp. 1–7.

[BNG06]   Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi, *Toward open-world software: Issue and challenges*, Computer **39** (2006), no. 10, 36–43.

[Cas95]   Cristiano Castelfranchi, *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., 1995, pp. 56–70.

[Cas05]   Yves Caseau, *Self-adaptive middleware: Supporting business process priorities and service level agreements*, Advanced Engineering Informatics **19** (2005), no. 3, 199–211.

[Cho56]   N. Chomsky, *Three models for the description of language*, Information Theory, IEEE Transactions on **2** (1956), no. 3, 113–124.

[CMM02]   Michael L. Collard, Jonathan I. Maletic, and Andrian Marcus, *Supporting document and data views of source code*, DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering (New York, NY, USA), ACM Press, 2002, pp. 34–41.

[DMS01]   Rémi Douence, Olivier Motelet, and Mario Südholt, *A formal definition of crosscuts*, REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (London, UK), Springer-Verlag, 2001, pp. 170–186.

[EF05]   Michael Engel and Bernd Freisleben, *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, 2005, pp. 51–62.

[ES90]   Susan Even and David A. Schmidt, *Type inference for action semantics*, ESOP'90, Proc. European Symposium on Programming, Copenhagen, Lecture Notes in Computer Science, vol. 432, Springer Verlag, 1990, pp. 118–133.

[GBS05]   Paul Grace, Gordon S. Blair, and Sam Samuel, *A reflective framework for discovery and interaction in heterogeneous mobile environments*, SIGMOBILE Mob. Comput. Commun. Rev. **9** (2005), no. 1, 2–14.

[Gun92]   C. Gunter, *Semantics of programming languages: Structures and techniques*, Foundations of Computing Series, MIT Press, Cambridge, Massachusetts, 1992.

[KC01]   Soon-Kyeong Kim and David A. Carrington, *A formal denotational semantics of uml in object-z.*, L'OBJET **7** (2001), no. 1, 31–37.

[KC03]   Jeffrey O. Kephart and David M. Chess, *The vision of autonomic computing*, Computer **36** (2003), no. 1, 41–50.

[KHH+01]   Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold, *An overview of aspectj*, ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming (London, UK), Springer-Verlag, 2001, pp. 327–353.

[KL03]   Karl Kurbel and Iouri Loutchko, *Towards multi-agent electronic marketplaces: what is there and what is missing?*, Knowl. Eng. Rev. **18** (2003), no. 1, 33–46.

[Leh96]   M. M. Lehman, *Laws of software evolution revisited*, EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology (London, UK), Springer-Verlag, 1996, pp. 108–124.

[LS75]   M.E. Lesk and E. Schmidt, *Lex - a lexical analyzer generator*, Technical Report 39, Bell Laboratories, New Jersey, NJ, USA, October 1975.

[Mey91]   Bertrand Meyer, *Design by contract*, Advances in Object-Oriented Software Engineering (D. Mandrioli and B. Meyer, eds.), Prentice Hall, 1991, pp. 1–50.

[MTB05]   Philip Miseldine and Azzelarabe Taleb-Bendiab, *A Programmatic Approach to Applying Sympathetic and Parasympathetic Autonomic Systems to Software Design*, Frontiers in Artificial Intelligence and Applications (Hans Czap, Rainer Unland, Cherif Branki, and Huaglory Tianfield, eds.), Self-Organization and Autonomic Informatics, vol. 135, IOS Press, December 2005, pp. 293–303.

[MTB06a]   _____, *CA-SPA: Balancing the Crosscutting Concerns of Governance and Autonomy in Trusted Software*, AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 2 (AINA'06) (Washington, DC, USA), IEEE Computer Society, 2006, pp. 471–475.

[MTB06b]   _____, *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, 2006, pp. 93–97.

[MTBER07]   Philip Miseldine, Azzelarabe Taleb-Bendiab, David England, and Martin Randles, *Addressing the Need for Adaptable Decision Processes in Healthcare*, Medical Informatics and the Internet in Medicine, Taylor and Francis **37** (2007), 1–7.

[ON02]   David von Oheimb and Tobias Nipkow, *Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited*, Formal Methods – Getting IT Right (FME'02) (Lars-Henrik Eriksson and Peter Alexander Lindsay, eds.), LNCS, vol. 2391, Springer, 2002, pp. 89–105.

[PGA02]   Andrei Popovici, Thomas Gross, and Gustavo Alonso, *Dynamic weaving for aspect-oriented programming*, AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development (New York, NY, USA), ACM Press, 2002, pp. 141–147.

[PH05]   Manish Parashar and Salim Hariri, *Autonomic computing: An overview*, Springer Verlag, 2005.

[PV04]   Joachim Peer and Maja Vukovic, *Web services*, Lecture Notes in Computer Science, vol. 1, ch. A Proposal for a Semantic Web Service Description Format, pp. 285–299, Springer Verlag, 2004.

[RC02]   Barry Redmond and Vinny Cahill, *Supporting unanticipated dynamic adaptation of application behaviour*, ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming (London, UK), Springer-Verlag, 2002, pp. 205–230.

[RMN+03]   N. Rousseau, E. McColl, J. Newton, J. Grimshaw, and M. Eccles, *Practice based, longitudinal, qualitative interview study of computerised evidence based guidelines in primary care*, British Medical Journal **326** (2003), no. 7384, 1–8.

[Rol98]   Colette Rolland, *A comprehensive view of process engineering*, Lecture Notes in Computer Science **1413** (1998), 1–24.

[RTBM06]   Martin Randles, Azzelarabe Taleb-Bendiab, and Philip Miseldine, *Addressing the signal grounding problem for autonomic systems*, Proceedings of International Conference on Autonomic and Autonomous Systems, 2006 (ICAS06), July 2006, pp. 21–27.

[Sch86]   David A. Schmidt, *Denotational semantics: a methodology for language development*, William C. Brown Publishers, Dubuque, IA, USA, 1986.

[SJK+05]   Ali Sajjad, Hassan Jameel, Umar Kalim, Young-Koo Lee, and Sungyoung Lee, *A component-based architecture for an autonomic middleware enabling mobile access to grid infrastructure.*, EUC Workshops, 2005, pp. 1225–1234.

[SPS02]   Stefan Schonger, Elke Pulvermüller, and Stefan Sarstedt, *Aspect-oriented programming and component weaving: Using XML representations of abstract syntax trees*, February 2002, Second German AOSD Workshop, Bonn, Germany. To appear.

[ST05]   Mazeiar Salehie and Ladan Tahvildari, *Autonomic computing: emerging trends and open problems*, DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software (New York, NY, USA), ACM Press, 2005, pp. 1–7.

[vD02]   Arie van Deursen, *Software architecture recovery and modelling: [wcre 2001 discussion forum report]*, SIGAPP Appl. Comput. Rev. **10** (2002), no. 1, 4–7.

[WS01]   I. Welch and R. Stroud, *Kava - using byte code rewriting to add behavioural reflection to java*, 2001.

[ZK01]   Y. Zou and K. Kontogiannis, *Towards a portable XML-based source code representation*, Proceedings of (XSE2001), 2001.

15