The Knowledge Engineering Review, Vol. 21:3, 249–259. © 2006, Cambridge University Press
doi:10.1017/S0269888906000919 Printed in the United Kingdom

Composing simulation architectures for autonomic systems

ALUN R. BUTLER, MOH IBRAHIM, KEITH RENNOLLS and LIZ BACON

University of Greenwich, Old Royal Naval College, Greenwich, London SE10 9LS, UK;
e-mail: a.r.butler@gre.ac.uk, m.t.ibrahim@gre.ac.uk, k.rennolls@gre.ac.uk, e.bacon@gre.ac.uk

Abstract

Simulation has long played a part in testing new configurations and new functionality in a diverse range of software. Through such simulations, the boundaries of the system state are explored and the relationship of that state to other applications tested — sometimes to destruction. A critical differentiator between a simulation and a *live*, deployed application is that simulations are allowed to fail. As truly autonomous applications evolve, this capacity for simulation must be built in from the ground up or the benefits of experience — including the ability to tolerate failure — will be lost. This must be achieved without undermining the global correctness of visible application behaviour. We suggest an engineering approach to enable the introduction of such simulation with minimal or no recoding and we propose a composition architecture to allow for safe dynamic deployment in substantial autonomic systems. We have identified our approach as application *Dreaming*.

1 Introduction

The full realization of the benefits of adaptive approaches will only be achieved when autonomic algorithms can take configuration and deployment decisions which match or exceed those of human administrators and engineers. Humans typically base their decisions on a foundation of knowledge and experience which has been hardened by brutal failure, heartened by courageous success and (occasionally) rewarded by the outrageousness of fortune. A cautious developer will carry out simulations of particularly risky approaches — and perhaps nibble fingernails when the system goes *live*. In autonomic systems, there is no development platform on which to test ideas, only a remorseless workload; there is no pool of experience when new compositions present themselves, only raw performance data; there is no possibility of wild experimentation, only clients with a service level agreement. Perhaps mercifully neither are there fingernails to provide false comfort.

In this paper we describe an application framework which will allow the incorporation of similarly risky, error prone and even downright dangerous software artefacts into live systems — without undermining the certainty of correctness at the application level. We have called our approach *Dreaming*, largely because it reflects a deployment regime which — like human dreaming — need have no real world consequences. It is a simulation architecture for live, working systems.

Although presented here as a specific solution to an intractable problem of autonomic computing, the moment of live deployment following the addition of new functionality, new performance enhancements or even new robustness features for an existing application, presents a generic risk to all software development. And while first deployment is the obvious instant of highest risk, the hazards remain during the processes of system roll-out as the full control flow space is explored by actual users interacting with actual data over time.

Our approach incorporates key features required by the most dynamic and adaptive systems undergoing changes in configuration, functionality or physical deployment, specifically:

- a defined process for safe application composition;
- a re-use driven approach for establishing system correctness;
- a light-weight engineering method for introducing simulation into systems where simulation is not the first engineering concern.

To achieve this goal we introduce a number of simulation-specific behaviours into the system. We reduce the engineering overhead associated with this by treating these behaviours as an *aspect* of the system (Kiczales *et al.*, 1997). The imposition of a simulation is viewed as a crosscutting concern — separated from core functionality.

This means new configurations both separately or in composition form can be considered and explored by the system while limiting their effects on existing deployed components. In particular, we can isolate the behaviour of new sections of code and run them concurrently in an active system.

We define a system-wide state in which these algorithms can be tested in live systems and we call this state *Dreaming*. While this is a somewhat poetic term for a simulation, there are a number of features which map the metaphor to the implementation in a manner which is beyond lyricism. The system (or rather, part of the system) enters the dream state and different rules apply.

The key facet of the dream state is that operations carried out while dreaming are idempotent with respect to the live system. In other words, while in the dream state, the application can guarantee there will be no real-world consequence of any operation.

This allows another facet of human dreaming to be utilized. In the dream state it is safe for applications to free associate unrelated optimization and or robustness techniques. Such associations may be weird. Weird behaviour is defined as counter-intuitive or unanticipated couplings of performance/stability enhancing and/or functional algorithms. Such coupling may be positive or negative. Our approach makes no assumptions or impositions on the nature of any particular new algorithms or system configuration. This allows the algorithm to work at any level of granularity and can set its own rules for monitoring, analysis and planning. Indeed the suggested configuration/algorithm need not even be autonomic (adaptive) in nature. In the core example we use in this paper, the algorithms considered are merely strategic and autonomic behaviour is emergent. Thus dreaming represents an attempt to engineer emergence to beneficial effect.

Our approach does make certain assumptions about application design, but these assumptions reflect current development best practice (see Fowler (2003), for example):

- that the application has coarse-grained service level interfaces;
- that the classes and methods of the service interfaces conform to a reasonable naming convention;
- that unit tests exist for all service-level method calls and at other mission-critical levels;
- that the system has sufficient spare capacity to dream.

We have been exploring these ideas by developing a functional prototype based on the Aspect Orient Programming framework provided by the open source application server JBoss (JBoss Web site `http://www.jboss.org/index.html`). The framework is intended to work within the application server product, but can run independently as a desktop application. Our design approach is generally language-neutral however, and successful experiments have been carried out using an alternative Java framework AspectJ (Kiczales *et al.*, 2001).

In the next section, we will briefly examine related work which has attempted to use dreaming as a motivating metaphor; we consider alternative approaches to treating autonomic behaviour as an aspect and the relationship of dreams and dreaming to deployment, composition and emergence with a particular focus on autonomic systems. In the third section, the power of the dream metaphor will be expanded and the basic architecture of the system will be explained. In Section 4,

we give an outline of a worked example and in the penultimate section, we discuss some real-world engineering concerns arising from the imposition of dreams.

2 Related work

Like dreaming, the term autonomic — when applied to computers — is itself metaphorical (Kephart & Chess, 2003) — the aim being to see if the application of the autonomic metaphor to the intractable problems of information systems will yield fruitful approaches to recalcitrant problems. Other metaphors — such as market economics of self interest (Eymann *et al.*, 2003; Cheliotis & Kenyon, 2003), the micro-economic price mechanism (Waldspurger *et al.*, 1992) and even emotions (Lee & Ibrahim, 2003; Chandarana & Skillicorn, 2004) have been applied to potentially adaptive systems with varying degrees of success. It is thus natural that the introduction of yet another layer of metaphor should be treated with caution. However, effective metaphor is now recognized as the basis for communication and common understanding in designing systems (Beck, 2000) and a sufficiently powerful metaphor provides continuous support as the system is designed and redesigned.

It may be supposed that the notion of dreams as a basis for system design would be popular, but although the term crops up regularly in various products, it is usually as a mere product label with little connection to the real-world concept — or rather as an aspirational marker (Sommer & Potter, 1996; Kim *et al.*, 1996).

The framework described by Arenas *et al.* (2002) which allows the dynamic addition of evolutionary algorithms to a distributed computational system — and is called DREAM, certainly sounds like it meets the criteria for a dream-based autonomic approach. However, Arenas *et al.*'s DREAM really only uses the dream term because the distributed nature of the system allows underutilized CPUs to join in processing. This mirrors the sleeping (i.e. not busy) facet of dreaming, but misses out the other key attributes — that human dreams are idempotent with respect to the real world and that human dreams free associate. Despite the obvious link with simulation, dreaming does not seem to have been specifically used as an engineering building block in a previous architecture.

We are increasingly familiar with dynamic deployment regimes on the desktop, with applications such as Windows Update or Java WebStart Sun Microsystems (available from http://java.sun.com/) but it is an article of faith that such deployment arrives on the desktop having been fully tested in non-production environments. Hohmann (2003) pointed out that exhaustive testing against unknown deployment regimes is intractable — and while shrink-wrapped software may have some chance of working on most platforms on most systems — the phase space of the autonomic systems is inevitably greater. Indeed, we define a fully autonomic system to be a system with no non-moving parts — as re-composition and re-configuration are endemic.

Coupaye & Estublier (2000) argue that while the discipline of software engineering has, until recently, focused on the modelling and evolution of applications, the elephant in the room that no one likes to talk about in application development has always been deployment. In traditionally developed (i.e. non-autonomic) systems, significant testing can be carried out prior to deployment on development servers — although many risks remain at final release as an explosion of composite variables meet each other for the first time (file paths and database URLs, IP and name dependencies, Operating Systems and OS versions, dependent software and dependent software versions). By definition, an autonomic system lives with these problems at every step.

Hall *et al.* (1999) enumerate the discrete phases of the software deployment lifecycle as release, install, activate, reconfigure, update, adapt, deactivate and retire. Other frameworks, such as the Open Software Description Format (van Hoff *et al.*, 1997), share with Hall *et al.* the implicit assumption that software ready to be deployed is fully working software. This naïve, assumption means they miss out two crucial stages necessary for real-world deployment — consideration and exploration.

To be useful, autonomic approaches need to be deployed on live systems. To be effective, autonomic approaches need to be able to make brave decisions to guide the control flow and state of applications; decisions that could — in principle — be disastrously wrong. Some compositions lead to emergent failure (see the power grid failure described in Strogatz (2003)). Less often some compositions lead to emergent success (see, for example, the self organization of loosely coupled peers in Babaoglu (2005)). Either way, autonomic approaches must make brave decisions which are never wrong.

Attempts are being made to constrain and even engineer the emergent complexity of dynamic and autonomic systems (see Dowling *et al.* (2005), who identify the importance of announcement within autonomic systems; Camorlinga & Barker (2004), who emphasis the importance of local interaction; and Fisher & Lipson (1999), who draw attention to the inevitable unbounded nature of future systems). In Anthony *et al.* (2005), the inherent emergent stability of an election algorithm was suggested as a means to provide global stability across many application layers, but the authors were not able to guarantee such stability except by inspection and testing. Part of the purpose of this paper is to provide a stronger level of certainty of the correctness of behaviour.

The application of an Aspect-Oriented approach to adaptive methods is a recognized role of aspect technologies and formed part of the description in the seminal work (Kiczales *et al.*, 1997). Later adaptive approaches have focused on the use of particular technologies (such as AspectJ (Kiczales *et al.*, 2001) or the DJ library (Lieberherr *et al.*, 2001)). In contrast, the authors discovered, when attempting to integrate the dream scenario into existing applications, that no particular technology was sufficient unto itself, but that a range of approaches could be utilized to achieve the desired goal (for example, we made extensive use of language-specific cloning facilities to prevent contamination of non-immutable objects). However, to the extent that approaches such as replacing functionality, testing configurations, monitoring core activity and deploying acceptable code were associated with cross-cutting concerns, we are justified in identifying them as aspect-oriented.

3 Dreaming

'[Human] dreaming permits the testing and practicing of genetically programmed (i.e. instinctual) behaviours without the consequences of overt motor responses', LaBerge (1985). Computer Dreaming is a transactional state that is the superset of all of the transactional states of the software system. A single dream is ACID, but the D is for Dreamlike rather than Durable — which is simply another way of saying the state of the dream is ephemeral and local to the dream alone.

Dreams are Atomic because a dream inhabits the boundaries of a single system operation (we prefer the term *service*). The granularity of this service is defined by a (ideally pre-existing) unit test (or composition of unit tests, invoked as a root test in a hierarchy). Dreams are Isolated because they exist with complete independence from the system as a whole — as an intercepting aspect of an application service. Dreams are idempotent with respect to the non-dreaming system and guaranteed to have no long lasting real-world side effects. For this reason, Dreams are always Consistent. Although inconsistent states can temporarily emerge within the dream — these will always lead either to failure or success entirely within the dream. Dreams are ethereal with no impact.

Like human dreams, computer dreams are simulations. What makes a dream simulation more valuable and distinct from just any old simulation? Unlike normal simulations, but again like human dreams, computer dreams work on live systems and may take as input live data (usually simply by cloning service level arguments at the entry point). Dreams thus act as a way of validating and verifying your software against live data in the live system on a live machine(s). When Dreaming (running Dreams), the system will evaluate success against the Dreams' own success criteria and against system-wide prerequisites. A Dreaming system can evaluate and compose unusual and undetermined combinations of states, functionality and variables. Finally dreams that

have been assessed to be successful, either individually or in composition, are installed on the live system dynamically — with the release, install, activation, etc. cycle — as previously described. This may, in the general case, call for human intervention. In autonomic systems, however, the criteria for installation must be arrived at independently, the algorithm must be verified independently and the final install must be dynamic. These facets are available in our proof of concept implementation.

Composition can lead to indeterminate effects — specifically to unanticipated re-entrance (where a call causes a call back which causes the original call again — and so on); thrashing (where no equilibrium can be reached between alternate approaches and the system constantly switches either one in and out of memory); and resonance effects (where the interaction of algorithms forces the system out of stability).

To illustrate why composition is a possible source of error in autonomic systems, consider a composition exemplar. A (perhaps autonomic) generalized caching strategy may well improve local latency and perhaps system-wide scalability — but local caches may well lead to inconsistency at the application level. Whether such inconsistency is catastrophic is primarily a business decision — but infusing this knowledge within any general algorithm is problematic — particularly as the caching strategy and the distribution of live data change over time. Encoding system policies in defined tests is possible — but exploring the realistic activity space is more challenging. If it is possible to simulate — to dream — compositions which run in precise live conditions and based on live data — but with no real world impact — our dreaming agent can make installation and configuration recommendations based on significantly sounder knowledge.

4 Dreaming implemented

We describe an application which sorts data. In the real world, this sorted data has consequences. In a potentially optimized dream sort — there are no real-world consequences, but in all other respects the dream sort and its system-wide effects are the same as the non-dream world sort. During prototype implementation, it was realised that the dream itself could be isolated from the rest of the system by mandating that dreams ran on a controlled thread group with no interaction with the live system threads (except passively during the consideration phase). At system start-up, the main thread of operation begins, but a low priority background thread monitors system busyness. When conditions become suitable (the live system is relatively idle), dreaming commences. A DreamController class dynamically loads classes supporting the dream interface. Most dreams will subclass the default DreamInterceptor class:

```
public interface Dream
{
  public void consider();

  public void explore();

  public void install();

  public void uninstall();
}
```

By calling *consider* the dream will create a series of joinpoints described by pointcuts. Pointcuts are an Aspect Oriented Programming (AOP) term — a sort of regular expression that can pick out method calls, constructors, exceptions and even assignments across one or many classes (and — as of Java 5 — annotations). The pointcuts will have been defined by the developer of the new configuration/functionality and will identify the necessary entry points where new functionality is added. In the sort example, the pointcut will identify constructor invocation of the particular sort instance and the service level method call (the method sort(List list)).

The pointcuts are passed to the Aspect engine along with an interceptor class to handle the various phases of the dream. In the *consider* phase, the target method/constructor/exception etc, call is silently intercepted and copies of the incoming arguments and the resulting return value, along with calling metrics information (for example, how long it took to run the method) are passed asynchronously to the dream state holder — the DreamingContext. This stores and hides dream state from the main thread by using the Java ThreadLocal utility class — which isolates state information on a per thread basis. The gathered and copied parameters and return values will be used as source data for predefined unit tests. They will act as Mock Object (Beck, 2003) input for unit tests. When sufficient input and output data have been gathered, the explore phase begins.

In the *exploration* phase — the live application carries on as normal — the harvesting of passed-in parameters and return values ceases. The dream controller will start the dream on a new thread in which the DreamingContext is mandated to return true to the method DreamingContext.isDreaming(). This will only be true on specifically designated dream threads and their child threads — this method will never return true on the live system (i.e. the system which has real-world consequences).

```
public class DreamingContext
{
  static ThreadLocal local = new ThreadLocal();
  DreamingContext NOT_DREAMING = new DreamingContext(false);
  final boolean isDreaming;

  DreamingContext(boolean isDreaming)
  { this.isDreaming = isDreaming;}

  private boolean getDreaming()
  { return this.isDreaming;}

  public static void startDreaming()
  { local.set(new DreamingContext(true)); }
//etc..
```

Consider a class Sort with a sort(List list) method. During consideration, this method would have had its input variables and return values harvested. During dreaming — and only on the dreaming thread — creation of instances of the Sort class will be intercepted and a new BetterSort subclass of sort substituted. This class will carry the improved sort(List list) method. The entry point to the dream is one or more unit test classes. For this we used the JUnit framework (available from the JUnit Web site `http://www.junit.org/index.htm`). Developing software with Unit test alongside or even prior to use is now an established successful software engineering technique (Beck, 2003). Many tests should exist for any new functionality or improvement and they can be incorporated into the operation of a dream without amendment — or with very little amendment. This is because the dream intercepts any persistent state changes using the AOP framework.

Dreaming depends on test cases establishing that the system is in a consistent state within the dream context. The test cases reflect the underlying business rules and policies governing the application. Using these test cases, the dream calls the test methods based on the real input and return values. These test methods in their turn create Sort classes in the normal way but get back BetterSort classes as they are running on the Dreaming thread. The test classes will run the sort method and validate any pre and post conditions mandated for that method (for example, that the post sorted list is indeed sorted).

Dreams behave similarly to traditional transaction handlers implemented as interceptors, i.e. withholding persistence on given code paths until the general commit is agreed. Any persistent changes or changes visible to the live system are intercepted using AOP techniques and statefully

represented in the Dreaming Context so as to be available later on in a particular dream. Calls requiring interception include:

- memory resident shared objects;
- static (global class level) variables;
- file and network IO operations;
- database writes or other persistent store updates.

The performance overhead of this transactional state is much lower than a traditional transaction because, in live system terms, commitment of the transaction is guaranteed to fail, so many expensive protective locking techniques are unnecessary.

In the simple case, the dream self-evaluates its performance against the initial metrics gathered from the live system. If it finds the results satisfactory the exploration phase is ended. The dream signals the DreamController and is installed. Once installed a new interceptor is dynamically added to the live system and any calls to the new Sort() will silently return a BetterSort object. An installed dream-verified component is stateful so installation survives server shut down. Without explicit pre-existing factory methods it would be impossible to implement this fine grained handling without using AOP techniques.

In the complex case the DreamController composes all available dreams in unusual combinations. The pseudo code for our current composition algorithm is given below. The present implementation does reflect an attempt to try weird compositions, e.g. trying failing dreams with successful dreams, or failing dreams with failing dreams, trying operations in partnership or apart. Although it does not exhaustively explore the complete dream space, to see if any or all weird combinations result in systematic improvements, it allows for some such combinations in a random fashion, while remaining tractable overall.

Pseudo code for composition algorithm

```
//note enhancements purge duplicates in the composition
Compose(ListOfEnhancements)
  LoEClone = ListOfEnhancements
  Do
    ForEach enhancement in ListOfEnhancements
      IF(!Test(enhancement)) //test will also update
                             //the enhancements DreamMetrics
        ListOfConfigurations.Remove(enhancement)
        continue
      END IF
      IF(enhancement.DreamMetrics == Worse)
        ListOfConfigurations.Remove(enhancement)
        continue
      END IF
    END ForEach
    ListOfEnhancements.Sort(ByDreamMetrics, ByLargestComposition)
    ListOfEnhancements.RemoveCutOff(Cutoff)
    CutOff = ListOfEnhancements.DreamMetrics
    Declare NewList
    FOR i = 0 TO Length
      //A new enhancement composed of the two arguments
      //Top list item will be ListOfEnhancements
      NewList.Add(ListOfEnhancements +
                  ListOfEnhancements)
```

```
    END FOR
    ListOfConfigurations = NewList
    IF(LENGTH 1)
      //try weird things - most will be removed
      ListOfConfigurations.Add(ListOfConfigurations+LoEClone)
    END IF
  while LENGTH 1
  INSTALL ListOfEnhancements; END Compose
```

Essentially, the Compose algorithm assesses each enhancement, running tests and gathering metric data. Failing enhancements (causing system failure) and/or enhancements which provide failing metrics are removed. Enhancements are then sorted and the best is defined as the cut-off (implying any further composition must be at least as good as or better than the best algorithm). Now the best enhancement is composed together with each successively good enhancement and the process is run again. Any composition that is worse than the previous run is removed. The process is continued until a single composed enhancement is delivered and this is installed. During the process of exploration, the compose method throws in some composition wild cards (randomly).

This algorithm does not explore the entire phase space of composition, but it does follow a most likely survivor first approach, while allowing some investigation into unlikely couplings. Thus it is tractable without being excessively inflexible. This algorithm (and competitors) are susceptible to composition analysis and we are exploring and assessing alternatives — this is future work.

From an implementation point of view, the Dream developer needs to provide:

(a) the new functionality/performance enhancing configuration;
(b) some pointcut specifications;
(c) a subclass of the DreamInterceptor class — overriding any special handling;
(d) an appropriate unit test; given that unit tests should be written anyway, the development footprint of a dream is relatively light;
(e) a DreamMetrics implementation (usually simply picking from a library implementation).

We anticipate tooling this up as an add-on component for one or more visual development environments to significantly minimize the overhead of this work.

5 Problems and solutions

Early work on the prototype system has provided insights on how to limit the development overhead of our approach. As stated in the introduction, we assume the target system has capacity to dream at least some of the time. However, AOP presents an overhead to the system — and within that overhead are two levels — pre-process pre-compiled interceptions (slower) and dynamic interceptions (slower still).

In Figure 1 we anticipate a highly replicated deployment architecture with perhaps one node designated as the dreaming system. This node provides full services while the system is busy, but carries the AOP dynamic overhead. When installing and deploying, it may force a full reboot, but swap roles with another replicated server. Composition information is transported so work is not duplicated. Although the diagram shows these systems as server applications, there is no particular reason beyond raw processing power why each node could not be any type of computing node for instance wireless mobile devices sharing processing.

The engineering overhead of providing for simulation is also non-trivial. Specifically arriving at ways of meaningfully defining comparable metrics is a continuing problem. We are currently investigating micro-economic equilibrium models (a price mechanism) to establish comparability. However, experience did lead to one interesting engineering insight. Our first instinct was generally to intercept at method level and build AOP interceptors accordingly (see the top part of Figure 2).

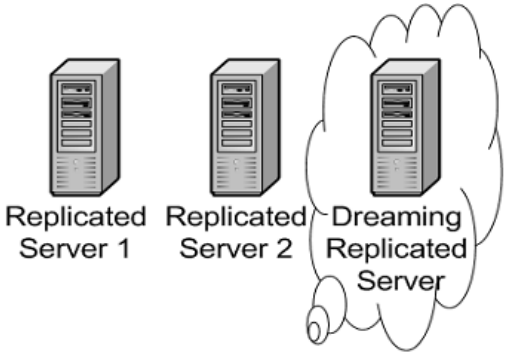

Figure 1 A deployment model in resource-hungry environments. The dreaming server is able to provide full services when the system is busy, but will be slower than the raw servers

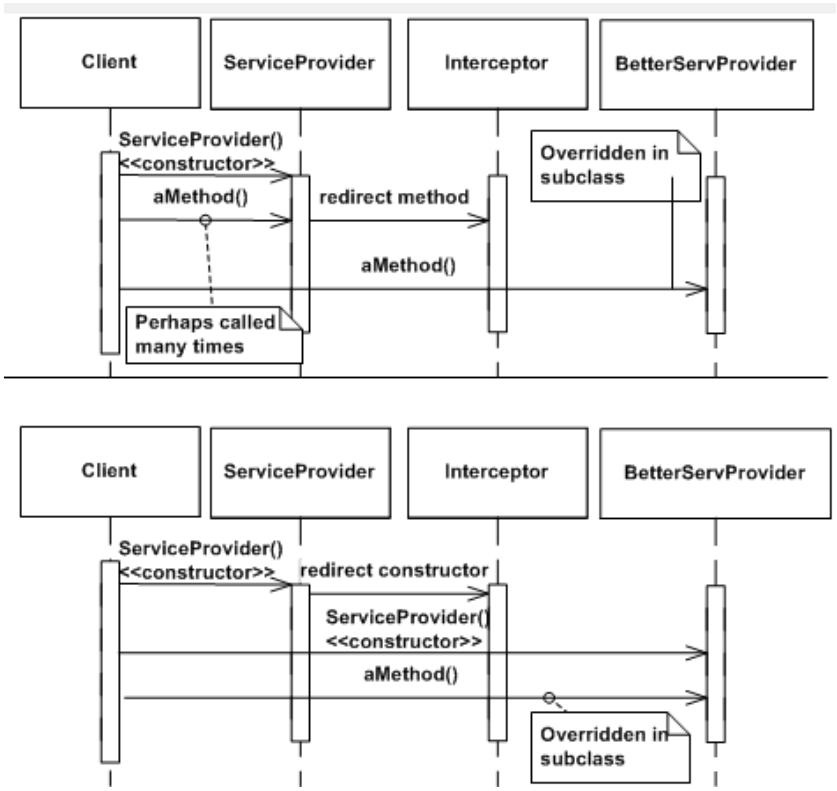

Figure 2 In the top part the AOP framework intercepts at method level. In the bottom, interception is at instance construction and the invocation overhead is reduced for each method call

Generally, this approach was flawed, leading to tangled copying of parameters and uncertain control flow paths. The alternative approach — construction interception, as shown in the lower part of Figure 2 — provided a cleaner approach which was easier to optimize. In essence, to add new functionality or new configuration, the developer need only sub-class the key classes and the AOP framework will intercept construction of the old type and replace it with the new type on the dreaming thread.

As stated earlier, dreams are a transactional super-state of the entire system with guaranteed rollback. Our early work indicates that transactional complexity can be significantly simplified when applications run under an existing transactional context such as that provided by Enterprise JavaBeans (EJB) (DeMichiel, 2003) running inside an EBJ compliant application server like JBoss. However, because this transactional behaviour often relies on externally provided agents (such as

from a database product), performance in the dream may compromise actual system behaviour as unnecessary lock are held. It would help if all transaction managers were Dream aware!

A more significant problem is that the nature of the system means that the dreaming application is not exactly the same as the non-dreaming application. The process of monitoring changes in the system and may compromise metrics. The fact that dreams must be carried out when the system is relatively lightly loaded compromises part of the purpose of the test. This criticism is fundamental not simply to dreaming but to all simulation approaches. The advantage of dreaming is that the simulation is as close to the metal of genuine experience as it is possible to get. Other more heavyweight approaches may allow exacting inputs and outputs to be specified, but this implies that only the problems caused by that particular I/O can be solved — not any I/O as in the dreaming system. Autonomic systems cannot survive on fixed I/O.

6 Conclusion

This paper identifies two new mutually dependent system states necessary for application deployment in adaptive/autonomic applications — Consideration and Exploration. We have recognized that these states must be idempotent — which is to say that, while consideration and exploration are taking place, their operation will have no effect on the remainder of the live system. The conjunction of consideration and exploration we have called Dreaming. We have proposed that Dreaming will be a powerful metaphor for deploying updates and changes to any significant system.

We have suggested that Dreaming can be viewed as a transactional state which is a superset of all other transactional states in the system. However, unlike other transactional states — no record locking or visibility concerns arise except with respect to dreaming participants. This is because the final transactional state of the dream will never be visible to the system as a whole and need not be resilient to system failure. We have identified a means whereby transactional state can be generally introduced to existing systems using aspect-oriented techniques. We are developing a proof of concept architectural framework in support of this hypothesis based on the JBoss AOP framework. We have provided a composition algorithm for accepting new functionality/new configurations into the system after exploration.

The initial impetus for this work came from a desire to consider and explore self-learning approaches and the art of composing such methods together. Future work will continue to explore how we can seamlessly weave these approaches into existing systems. We are particularly interested in how weird we can make our compositions during this 'free association' phase without affecting overall system performance. There is significant work to be completed integrating with existing transaction managers and exploring the generality of data handling in the dreaming scenario, without compromising the core benefit of the co-deployment on the live system.

References

Anthony, R., Butler, A. & Ibrahim, M. T. 2005a Layered autonomic systems. In *Proceedings 2nd International Conference on Autonomic Computing (ICAC)*. Piscataway, NJ: IEEE, pp. 383–384.

Anthony, R., Butler, A. & Ibrahim, M. T. 2005b Composing emergent behaviour in a fault intolerant system. In *Workshop on Self-Adaptive and Autonomic Computing Systems (SAACS), DEXA 2005, Copenhagen, Denmark*. Piscataway, NJ: IEEE, pp. 145–149.

Arenas, M. G., Collet, P., Eiben, A. E., Jelasity, M., Merelo, J. J., Paechter, B., Preu, M. & Schoenauer, M. A. 2002 A framework for distributed evolutionary algorithms. In *Proceedings of PPSN VII, Granada*.

Babaoglu, O. 2005 Grassroots approach to autonomic computing. *Keynote Address at ICAC 2005*, available from `http://www.caip.rutgers.edu/parashar/icac2005/presentations/ICAC05`.

Beck, K. 2000 *Extreme Programming Explained*. Reading, MA: Addison-Wesley.

Beck, K. 2003 *Test Driven Development*. Reading, MA: Addison-Wesley.

Camorlinga, S. & Barker, K. 2004 The emergent thinker. In *Proceedings International Workshop on Self-* Properties in Complex Information Systems, Bertinoro, Italy*.

Chandarana, R. & Skillicorn, D. B. 2004 Emotions as a metaphor for altering operational behavior in autonomic computing. External Technical Report ISSN-0836-0227-2004-491, available from `http://www.cs.queensu.ca/TechReports/Reports/2004-491.pdf`.

Cheliotis, G. & Kenyon, C. 2003 Autonomic economics. In *IEEE International Conference on E-Commerce Technology*, p. 120.

Coupaye, T. & Estublier, J. 2000 Foundations of enterprise software deployment. In *CSMR, Germany*, pp. 65–74.

DeMichiel, L. G. (Specification Lead) 2003 *Enterprise JavaBeansTM Specification, Version 2.1*. Sun Microsystems, available from `http://java.sun.com/`.

Dowling, J., Cunningham, R., Harrington, A., Curran, E. & Cahill, V. 2005 Emergent consensus in decentralised systems using collaborative reinforcement learning. In *Self-star Properties in Complex Information Systems*, pp. 63–80.

Eymann, T., Reinicke, M., Ardiaz, O., Artigas, P., Fritag, F. & Navarro, L. 2003 Self-organizing resource allocation for autonomic networks. In *Proceedings of the 14th International Workshop on Database and Expert Systems Application (DEXA 2003)*. Piscataway, NJ: IEEE, pp. 656–660.

Fisher, D. A. & Lipson, H. F. 1999 Emergent algorithms — a new method for enhancing survivability in unbounded systems. In *Proceedings of 32nd Annual Hawaii International Conference on System Sciences (HICSS-32), Maui, HI, 5–8 January*. Los Alamitos, CA: IEEE Computer Society Press.

Fowler, M. 2003 *Patterns of Enterprise Application Architecture*. Reading, MA: Addison-Wesley.

Hall, R. S., Heimbigner, D. & Wolf, A. L. 1999 A cooperative approach to support software deployment using the software dock. In *Proceedings of the 21st International Conference on Software Engineering*. Los Alamitos, CA: IEEE Computer Society Press, pp. 174–183.

Hohmann, L. 2003 *Beyond Software Architecture: Creating and Sustaining Winning Solutions*. Reading, MA: Addison-Wesley.

Kephart, J. O. & Chess, D. M. 2003 The vision of autonomic computing. *IEEE Computer*, pp. 41–50.

Kiczales, G., Lampling, J., Meneka, A., Maeda, C., Lopes, C., Loingtier, J. & Irwin, J. 1997 Aspect oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming(ECOOP) (Lecture Notes in Computer Science, 1241)*. New York: Springer, pp. 220–242.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W.G. 2001 *An Overview of AspectJ (Lecture Notes in Computer Science, 2072)*. New York: Springer, p. 327.

Kim, K. H., Subbaraman, C. & Kim, Y. 1996 The DREAM Library Support for PCD and RTO.k programming in C++. In *Proceedings of WORDS '96 (IEEE Computer Society 2nd Workshop on Object-oriented Real-Time Dependable Systems), Laguna Beach*, pp. 59–68.

LaBerge, S. 1985 *Lucid Dreaming*. New York: Ballantine.

Lee, A. & Ibrahim, M. 2003 Emotional attributes in autonomic computing systems. In *Proceedings of the 14th International Workshop on Database and Expert Systems Application (DEXA 2003)*. Piscataway, NJ: IEEE, pp. 681–681.

Lieberherr, K., Orleans, K. & Ovlinger, J. 2001 Aspect-oriented programming with adaptive methods. *Communications of the ACM* **44**(10), 39–41.

Sommer, S. & Potter, J. 1996 An overview of the real-time dreams extensions. In *Proceedings of the 3rd Australasian Conference on Parallel and Real-Time Systems (PART'96)*, pp. 144–147.

Strogatz, S. 2003 *Sync — The Emerging Science of Spontaneous Order*. Penguin Books.

van Hoff, A., Partovi, H. & Thai, T. 1997 The open software description format (ODS). Microsoft Corp. and Marimba Inc. Available at `http://www.w3.org/TR/NOTE-OSD.html`.

Waldspurger, C., Hogg, T., Huberman, B., Kephart, J. O. & Stornetta, W. 1992 Spawn: a distributed computational economy. *Software Engineering* **18**(2), 103–117.