DRASTIC: A Run-Time Architecture for Evolving, Distributed, Persistent Systems

Huw Evans and Peter Dickman
{huw,pd}@dcs.gla.ac.uk
http://www.dcs.gla.ac.uk/~drastic

Department of Computing Science
University of Glasgow
Glasgow, G12 8RZ, UK

Abstract. Modern systems must be adaptable — to changing requirements, bug-fixes, new technologies and reconfiguration. For critical applications this must be possible at run-time; for complex applications it should be limitable to major sub-divisions of the system. The DRASTIC architecture addresses these goals by exploiting object persistence and distributed systems implementation techniques. It enables run-time changes of types, implementations, and the system configuration. This is based on a novel architectural abstraction of locality for evolution, called the 'zone'. Contracts between zones automatically limit the visibility of such changes between zones. We present work in progress on DRASTIC's computational model and run-time system, illustrating support for software evolution and highlighting key features of our current implementation.

1 Introduction

Existing platforms offer little or no support for the management of change in, and evolution of, enterprise-critical applications. Furthermore, the facilities for the management (garbage collection, recovery, reorganisation and so on) of large distributed datasets rarely meet the needs of modern information systems. For example, CORBA [Obj95a] provides the software engineer with little support for managing the complexities involved when changing object interface definitions. CORBA also does not provide abstractions to constrain the effects of a change in a software component, making such a change visible throughout the entire system.

Software starts to evolve almost as soon as it is written: users require new features and ask for amendments. To compound the problem, within large systems, initially identical software components can evolve in different directions. In addition, large systems cannot always be shut down in their entirety because they may be too complex or mission critical, for example airline seat reservation systems and on-line banking systems.

As systems are becoming more complex and ever larger, maintaining them will become more of a problem. The DRASTIC project recognises this difficulty

and proposes tackling it by providing the programmer with a run-time system where the focus is placed on describing and controlling the effects of software evolution. Four key points underpin our approach.

Firstly, a DRASTIC system is subdivided into smaller, more easily managed sub-domains called zones. The organization of zones can follow the structure of the business using them, although this is not enforced. Zones encapsulate change; if a software component is changed in one zone, that change is not visible outside it. Zones are introduced at design time, subsequently becoming explicit components in the run-time system.

Secondly, the software in each zone is evolved as a coherent whole, largely autonomously from software in other zones, even though the source code may originally have been shared by components in many zones.

Thirdly, the evolution of the entire system is made explicit at the level of a zone by defining 'zone contracts' between them. These contracts specify which program types can be exchanged between zones and the transformations that are required should an object move from one zone to another or if a method invocation is made that crosses a zone boundary.

Lastly, zone autonomy is supported by inserting code supplied by the software engineer at the zone boundary. These software fragments, called change absorbers, handle the transformations indicated above and enforce the zone contract.

The DRASTIC run-time platform provides a distributed systems programming environment with support for orthogonally persistent processes. The run-time environment contains a number of repositories that hold information regarding the current configuration of the system. This information can be queried by processes and the software engineer to dynamically reconfigure the system at run-time. In addition, at run-time, DRASTIC intercepts all inter-process application-level object references. The ability to do this is the key to our approach to supporting run-time software evolution.

1.1 Project Goals

The DRASTIC project aims to provide support for large, evolving and integrated information systems. This support is being demonstrated by building a series of prototype systems. These systems explore different ways that distributed reference management, zones and zone contracts can be implemented and presented to the software engineer. Software evolution is supported by providing a platform that allows program types to be changed at run-time.

1.2 Overview of Paper

The first half of this paper presents DRASTIC's design: our model is presented in §2, an extended example is elaborated in §3 and DRASTIC's architecture is discussed in §4. With reference to the example, the second part of the paper details how software evolution is supported: §5 shows how the architecture performs computation within an evolving system and §6 provides some preliminary

performance measurements. Section 7 describes related work, with §8 and §9 detailing the project's current status and areas of further work.

2 DRASTIC's Model

This section describes the key models and critical decisions that underpin the DRASTIC approach. The zone model and contract model, described in sections 2.2 and 2.3, are concerned with how to sub-divide and describe a complex system so that it can be effectively evolved and maintained. Section 2.4 describes the model of evolution assumed. In an evolving system the potential for changing types, and corresponding modification of objects, means that questions of identity must be addressed; sections 2.5, 2.6 and 2.7 present the approach taken to naming types, objects and processes respectively. This section is concluded with a description of the distributed object model (§2.8) and persistence model (§2.9) which are concerned with access to objects over space and time respectively.

Since evolution of a complex system requires deep understanding of the application semantics, evolution cannot be fully automated. Therefore, our goal is to minimise the effort required by the software architect and we present their rôle in using the DRASTIC system in §3.1.

2.1 Motivating Example

Organisations tend to be divided into smaller groups or subdomains, for example, departments such as finance and payroll. Each subdomain should be as autonomous as possible, so they can effectively manage their rôle in the organisation.

The design, installation and maintenance of software in an organisation should also reflect this subdivision for reasons of autonomy. Updating all of the software in an organisation in one go will often be prohibitively expensive, but may be affordable on a per-department basis. Therefore, it is much more realistic for a particular sub-domain to be able to manage its own software independently of others. Changes made in one sub-domain may not make sense in another and should a change be made that does affect another, ideally we would like to limit the effect of that change to only the two sub-domains concerned. DRASTIC allows a system to be decomposed into such sub-domains, which are called zones.

One such organisation that can be clearly divided into zones is a hospital. Hospitals are usually decomposed along the lines of medical speciality — such as pathology, X-Ray, gynaecology and geriatrics — and administrative boundaries, finance, hospital catering, management information systems and so on. Imagine a simplified hospital consisting of just two zones, the X-Ray zone and the morgue zone, which has a computerised patient management system. When the system was first installed, the software in both zones agreed on the data making up a patient record. However, over the life of the hospital, the software will need to be

updated many times. This can create disparities between the system's software components and such changes may affect the definition of the patient record.

If the X-Ray department chooses to amend its patient record type, to capture a patient's X-Ray history, this change should not necessarily affect any other department in the hospital. The change could instead be encapsulated within the zone changing the type. But, departments may wish to exchange information, which may require a patient record object to move from the X-Ray zone to the morgue zone. The morgue department's software, however, has no concept of a patient record with an X-Ray history. The object needs to be transformed in some way when it crosses the boundary between zones and simply discarding the additional data may not be an acceptable solution. DRASTIC supports such transformations by inserting software components, provided by the software engineer, between zones. These transformers are called change absorbers and are more fully described in section 2.4.

Whenever a change is made to the software in one zone that will affect the software in another, both zones need to be aware of the kind of change being made. In a complex system, such as the hospital, some way of agreeing what can pass from one zone to another needs to be defined. This is to ensure that zones can be used to encapsulate change and so that change management becomes an explicit part of the maintenance of the system. DRASTIC supports this by defining a pair-wise agreement between zones which specifies the types that two zones are prepared to exchange. We refer to this as the zone contract.

2.2 Zone Model

A DRASTIC system is decomposed into a number of largely autonomous spaces called zones. This decomposition is made at design time, and remains explicit throughout the software life-cycle. At run-time a zone is a named, logical collection of processes distributed over a number of hosts. Zones support the recognition and exploitation of the almost-disjoint application subdomains which are known to developers and users, but which cannot be cogently described and supported at the system level on current platforms. In our example, one zone might contain all the hosts in the X-Ray department of the hospital. However, zones do not have to follow such a rigid physical organisation. It is possible both to have a zone containing processes that are physically separated by large distances, eg. with hosts in Scotland, Finland and the United States, and for processes on the same host to belong to different zones.

All the objects in a zone evolve as a single unit, thus the zone is the unit of evolution in the DRASTIC system. A DRASTIC process combines an optional store, the DRASTIC support platform and the language run-time. A process is considered to be a single address space that encapsulates objects and supports their execution. DRASTIC processes contain application objects. Objects hold references to other, potentially remote, objects and the DRASTIC platform intercepts these references at the process and zone boundaries to support software evolution. Zones exist from design time through to becoming explicit system components in the run-time system. When a reference traverses a zone

boundary it passes through processes that bridge two zones. The zone boundary processes (ZBPs) handle any transformations that are required when objects migrate between two zones or when invocations are performed along a reference that crosses a zone boundary.

Zones constrain the effects of evolution. Without them, a change to a software component might be visible throughout the entire system. Allowing a change to propagate throughout an entire system usually requires that each software component be recompiled, or if not recompiled, at least examined to ascertain the impact of the change. This creates a lot of work for the software engineer. Encapsulating change allows collections of software components to evolve largely autonomously, reducing component recompilation requirements and the need to check code.

Within DRASTIC, a process is required to reside in precisely one zone. To ensure a process can only be in one zone at any one time, DRASTIC zones do not overlap and it is not possible to nest one inside another. This requirement makes the design of the system much easier. If a process were allowed to reside in two zones simultaneously, a possible ambiguity could arise when evolving objects of a particular type. Consider two zones, `A` and `B`, and a process `P` which consists of a single object of type `t`. Assume that `P` is allowed to reside simultaneously in both zones. If zone `A` and zone `B` evolve type `t`, but evolve it differently, two different types now exist, $\mathtt{t}_A$ and $\mathtt{t}_B$. Without adding considerable complexity to the model and run-time architecture there is no way of deciding how `P`'s object of type `t` should evolve.

2.3 Contract Model

The zone contract is a pair-wise agreement between two distinct zones. It describes which types can flow between zones, the permitted invocations and what needs to be done to transform objects at the zone boundary. For the purposes of implementation the contract is a description of the transformations to be applied to objects and references that are exported from one zone and imported into another (figure 1a). When viewed by a software engineer, a contract between any two zones, for example A and B (figure 1b), is seen from the point of view of what a zone imports and exports. Zone A exports objects of type X and Y to zone B and imports from it objects of type X and Z.

In figure 1a, zone A's contract specifies it will export to zone B objects of type X and Y and zone B will import from zone A objects of type X' and Y'. Transformations between the two zones protect the software in one zone from changes made to the software in the other zone.

Zone B exports objects of type Z which zone A imports; no transformation is applied. It is necessary to specify this in the contract because the two zones exchange this data and because, over time, one zone's definition of Z may change. Without explicitly capturing this information in the contract, the knowledge required to permit change to either Z is easily lost. By specifying a seemingly redundant transformation greater control can be exercised over subsequent changes made to the system.

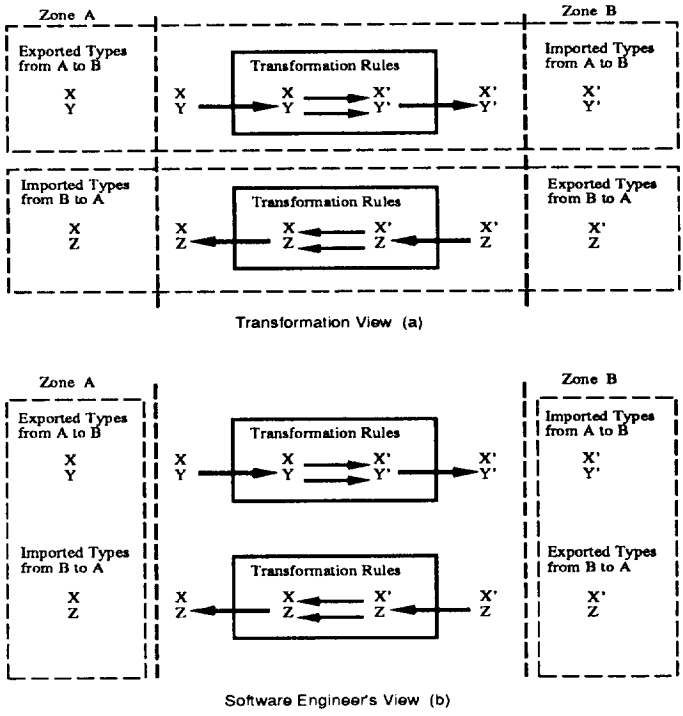

Fig. 1. Zone A and B's Contract

A zone contract allows the software architect to describe the interactions between two zones. As a complex system evolves, these contracts will need to be reviewed and periodically updated. For example, the contract between the X-ray zone and the morgue zone described in section 2.1 would allow a patient record object to move between the two zones. The contract therefore provides the software engineer with greater control over the consistency of their system. For example, if one zone exports a type to another zone, that second zone should import it. Perhaps, as a result of evolution, that type should no longer be exported. However, the importing zone may still rely on it. Capturing this kind of information is important in managing large systems and can provide valuable feedback to the software engineer when a system is being modified.

It is possible to extend the idea of contracts to be more than just pair-wise, capturing information for a set of zones; however such a description seems to get extremely complex very quickly. It may be possible to take a global view of the entire system, checking for consistency in a way that is not possible with our pair-wise approach, but this, also, seems to be difficult to realise at run-time. Pair-wise agreements appear to capture sufficient information in a manageable way to allow realistic systems to be modelled using them. Extensions to this approach, with global consistency checking of contracts, are an area for future work.

2.4 Evolution Model

To effectively maintain modern systems, support for evolution is essential. Critical applications need to be capable of being evolved at run-time. For complex applications, evolution should be limitable to major sub-divisions of the system. DRASTIC's zones divide the application into more manageable parts, such that one zone can be evolved independently of others. The transformation rules contained within a contract are realised at run-time in software components called change absorbers. A change absorber is code provided by the software engineer which can transform objects of one type into objects of another type. DRASTIC allows processes in one zone to be evolved while allowing other processes in other zones to continue execution, even when these processes hold references to each other. This is done by intercepting application-level object references and placing change absorbers along the reference chain between the invoking object and the object being invoked.

The DRASTIC zone is the unit of evolution. In the current DRASTIC design, all software and data in a zone must be evolved at the same time. If some instances of a type in one zone were allowed to evolve while others remained at the previous version, this would make the run-time platform much more complicated. In future DRASTIC prototypes, this issue of selective evolution may be considered. When evolving a zone in its entirety, any databases the zone may contain may require some form of schema modification to take place. The DRASTIC project does not directly address this issue, instead providing an appropriate framework within which schemata can be changed.

2.5 Type Identity

Types are named entities in the DRASTIC system, to permit them to be referred to in contracts and during evolution. Within a single zone each type is uniquely identified, with the type name-spaces being zone-specific. The same identifier may be used for distinct types in two different zones, and identical types may be given distinct names in different zones. The zone contracts are written using the names from the two participating zones, with the type names being implicitly extended by their zone names. Furthermore, the association of names to types within a zone can change over time. When a type is evolved, changing its definition, it may, but is not required to, retain the same name.

2.6 Object Identity

Objects have identity in the DRASTIC system, and within the DRASTIC platform object identifiers are used. Again, the object name spaces are zone specific, with objects being uniquely identified within their zone. Implicit extension of the object name by the zone name provides unique names for objects within

a persistent application system, since zones have distinct names within a given complex application.

Since objects may be transformed during evolution, and can be migrated, it is not necessarily possible to rely on the implementation language object naming mechanisms. In our current implementation, for example, migration of an object relies on the construction of a new proxy and actual object in the recipient process. The Java run-time views these as new language-level objects. However, within the DRASTIC system the object is the same, it has just moved location. Any code the programmer has that depends upon the language-level identity remaining unchanged would now fail.

To solve the object identity problem, the DRASTIC pre-processor ensures that all migratable objects support an identity checking operation. This operation relies on an identity field in the object which is not affected by migration. It is safe to do this as the original actual object will be removed from the system if the migration is successful, so only one object with that identity is visible at any one time.

2.7 Process Identity

Processes in a distributed system contain objects that need to refer to other objects which are potentially in other processes. Within a DRASTIC process, objects refer to each other using language-level object references. Within a single persistent application system (PAS), globally available services used by DRASTIC, such as name servers to allow the system to boot, are at well known locations. Each persistent application system within DRASTIC is given a unique name which does not change over its lifetime. Each zone within a persistent application system is represented by a name which is unique within that PAS. Processes within a particular zone are also uniquely named within that zone by attaching the name to the store that the process is booted over (processes with no store are given new names when they are started). In the current implementation, DRASTIC uses simple strings to represent names. These names are part of a process' persistent state and different instantiations of the same process over time are distinguished by a monotonically increasing value. Using the persistent application, zone and process name together, a process can be uniquely identified in a DRASTIC system. More compact and richer name representations may be explored in future versions of the platform.

2.8 Distributed Object Model

Distribution is central to building modern software systems. Many modern systems have to be distributed as the people involved in the enterprise are themselves spread over large geographic distances. The underlying distribution model adopted by DRASTIC is similar to that provided by the Emerald [BHJ+87] system. Objects in DRASTIC are described by classes, containing an interface and an implementation either of which may evolve in any way. For example, a

method's implementation may change or a method signature may evolve (expand or contract) over time.

Information describing the current system configuration, such as initialisation files and program source code, also tends to be geographically distributed. Managing these repositories of information requires a distributed system support layer in which the current system description can be modified and the system subsequently reconfigured. The relationship between DRASTIC and other systems which focus solely on this reconfiguration problem is addressed in §7.

Failure Model Distributed systems exhibit partial failure. At any one time, some subset of a distributed system's components may be faulty and these faults can be visible at the application programming level. For example, processes may suddenly crash and inter-process messages may be lost or duplicated. DRASTIC's process failure model assumes any process may crash at any time but that when a process crashes it does so without exhibiting Byzantine behaviour. Message delivery is assumed to be at most once.

The DRASTIC run-time system offers the system architect some assistance with failure (eg. the ability to restart a crashed process) but it does not attempt to mask all errors related to distribution as, in the general case, this is not possible. For example, consider a non-replicated server: if the server is actually down when a client wishes to use it, due to hardware modification, there is no alternative but to report an error at the client side. DRASTIC allows failures at the granularity of an object, as an invocation through an object reference may fail due to a network error. Such partial failures may lead to method call failure, but within DRASTIC it is never possible to invoke the wrong object as a result of failures. This gives programmers a form of referential integrity that is compatible with a distributed system's failure semantics.

2.9 Persistence Model

It is a common requirement when building such large, long-lived, distributed systems that objects need to out-live the process that created them, typically by being written to hard disk. DRASTIC supports this using the concept of orthogonal persistence by reachability [AM95]. This is implemented by transitive reachability from a distinguished object, a persistent root, that is known to the persistence mechanism. Thus, any object in a DRASTIC process that is reachable, either directly or indirectly, from the persistent root is periodically made persistent automatically. In our current implementation orthogonal persistence is provided by Persistent Java [AJDS96].

Without support for orthogonal persistence, the programmer would be required to explicitly transfer to and from stable storage data-structures that should out-live the process that manipulates them. This means that programmers spend a lot of time flattening their complex data-structures into other data-structures, such as streams of bytes, that are easily written to and read from disk. Orthogonal persistence makes this explicit conversion unnecessary,

allowing the software engineer to concentrate on the semantics of their application.

3 Extended Example

This section describes the example that will be used throughout the rest of the paper. Before the example is introduced, the view of the system as seen by the software architect is presented together with how they could decompose a system into zones. The programmer's view of the DRASTIC system is then described.

3.1 The Software Architect's View

System decomposition is a powerful tool in the design of complex systems. Using zones, DRASTIC allows the software architect to capture this decomposition and explicitly express it in a uniform manner from design time through to run-time. Zones also allow the software architect to encapsulate components, such as legacy systems. Contracts allow the software architect to explicitly describe which objects can flow across zone boundaries, increasing system modularity. They also specify which object transformations are required at zone boundaries, increasing system autonomy.

The software architect using DRASTIC will describe their distributed system as a collection of zones. Each zone will have contracts with a number of other zones. The software architect will provide change absorbers which are placed between zones to protect one zone's software from changes in other zones. It is the goal of DRASTIC to provide as much automation for this as possible. For example, once the change absorbers have been defined, DRASTIC automatically inserts them where necessary in the zone boundaries.

Choosing an Appropriate Decomposition So far, how the software engineer chooses an appropriate system decomposition has not been described. The goal of the DRASTIC project is not to define a methodology for the development of a zoned system, rather it is to build a prototypical run-time system that supports the construction of systems that can evolve dynamically. Within our research group a related project called ZEST is underway. The ZEST project focusses on the software engineering issues raised by the idea of decomposing a system into a collection of zones. DRASTIC is intended to be one possible run-time support for systems designed using approaches like that being developed within ZEST.

3.2 Programmer's Model

The programmer using DRASTIC uses the programming model provided by their chosen implementation language. If more than one language is used in an application, other problems may arise, for example, when comparing two

object references for equality. These problems must be addressed and DRASTIC is intended, in principle, to support language heterogeneity, although in the current implementation only Java [GM96] is supported. Supporting multiple implementation languages is not the main focus of this work and the implications for the run-time system are not discussed in this paper.

3.3 The Example

The example system used in this paper is that of a hospital. A hospital can be divided into a number of zones where, for example, one zone would be responsible for dealing with X-rays, another the hospital catering, a third the morgue and so on.

The software in each of these zones will tend to evolve independently of the software in other zones. Two zones may have originally shared the source code for a given type, but when the type is evolved in one zone, a copy of the source code may be taken and updated.

Imagine a simple hospital system that has been decomposed by the software engineer into two zones, the X-ray zone and the morgue zone (figure 2).

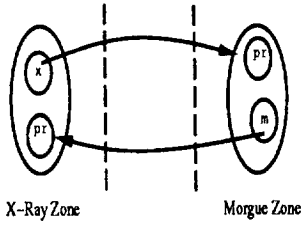

Fig. 2. The Initial System Architecture

The software engineer has defined one main type, the patient record (PR), that both zones understand. Two objects of this type exist in processes that are executing in the zones, one object (`pr`) per process. Both processes also contain one other object with a single reference to the patient record in the other zone, `x` in the X-Ray zone and `m` in the Morgue zone. Figure 3 contains the initial definition of the patient record type and the original contract between the two zones. It is assumed that each field has two associated methods (not shown) that get and set the particular field. For example, the `id` field will have methods `int getId()` and `void setId(int i)` defined.

A variety of changes are possible, such as modifying an existing method's signature, adding or removing a field from the class' state and reorganising the class inheritance hierarchy. The software architect is free to make such changes to a zone whenever they feel it is appropriate. Therefore, at some point in the future, the X-Ray zone may have its PR type updated, creating a new type PR_X. At some other time, the PR type in the morgue zone may also be updated, to

```
class PR
{
  int      id;
  Name     name;
  Address  address;
}
```

Direction of Transfer	Exported Types	Transformations Required	Imported Types
X–Ray to Morgue	PR	None	PR
Morgue to X–Ray	PR	None	PR

Fig. 3. Initial Definition of the Patient Record and X-Ray and Morgue Zones Contract

another type, different to both the PR and the PR_X types, resulting in another type, PR_M.

The changes that will be considered in this paper are illustrated on figure 4. In this new configuation, the PR type has been changed in both zones, leading to PR_X and PR_M (note that neither PR_X nor PR_M is a subtype of PR). Any of these objects moving between zones will now need to be transformed, as will any method invocations. The zone contract will also have to be changed.

```
class PR
{
  int      id;
  Name     name;
  Address  address;
}
```

X-Ray Zone Evolution

```
class PR_X
{
  String   id;
  Name     name;
  Address  address;
  XRH      xray_history
}
```

Morgue Zone Evolution

```
class PatientRecord
{
  Name     name;
  Address  address;
}
class PR_M extends PatientRecord
{
  NOK      next_of_kin;
}
```

Direction of Transfer	Exported Types	Transformations Required	Imported Types
X–Ray to Morgue	PR_X	PR_X to PR_M	PR_M
Morgue to X–Ray	PR_M	PR_M to PR_X	PR_X

Fig. 4. Updated Definition of the Patient Record X-Ray and Morgue Zones Contract

The new type in the X-Ray zone, PR_X, uses a new representation for the patients record's identifier. It has been changed from an integer to a String. A new field to capture a patient's X-Ray history has also been added. The PR type in the morgue zone has undergone a more extensive change. A patient in the morgue zone is now identified by a combination of their name and address and this has been abstracted out into a class PatientRecord. PR_M extends PatientRecord, which itself supports fewer methods and holds less state than PR, providing an extra field to capture a patient's next of kin.

4 DRASTIC's Architecture

This section describes in detail the DRASTIC platform's high-level support data-structures. Firstly, the architecture of an application process is described. This is followed by a top-down description of the rest of the DRASTIC architecture, starting at the system-wide level, moving down through the multi-zone architecture, ending with the architecture of a single zone.

4.1 Within an Application Process

When a process boots over the DRASTIC platform, the process acquires several additional objects (figure 5). The two objects that handle incoming references are the `InHandler` and the incoming reference table (`IRT`). The `InHandler` handles incoming migrating objects and invocations to objects in this process that are remotely accessible. The `InHandler` has a reference to the `IRT` which contains references to application-level objects as well as a weighted reference count for distributed garbage collection.[1] The outgoing reference table and `RemoteStoreDescriptors` handle out-going object references. Application level objects logically hold references to the target object. In practise the reference is pointing at a slot in the `ORT`. The `ORT` handles the indirection of references from this process and this usually involves referring to the `RemoteStoreDescriptor` object. The `RemoteStoreDescriptor` is responsible for packaging up outgoing object migrations and method invocations into a format suitable for transfer across a network. The `IRT` and `ORT` tables allow DRASTIC to construct chains of references that, logically, lead from one application object to another application object.

All inter-process references that an application level process creates are indirected through the DRASTIC platform. This is so the DRASTIC platform can, by inserting other objects along these references, implement run-time software evolution by intercepting application level method invocations. A process' `InHandler` also provides a way for other processes to contact it.

DRASTIC evolves sets of processes as a unit. Working at a finer granularity would require intercepting intra-process invocations. This is expensive, slowing down local invocations which are much more numerous than remote calls. Intercepting intra-process invocations would also involve changing the language run-time system or the compiler.

`CallHandlers` are objects that are provided automatically by the DRASTIC platform, after pre-processing the application's source code. A `CallHandler` contains code to call a particular type's methods should an object of that type be remotely invokable. For example, if the `pr` object was to migrate from its process in the morgue zone to the process in the X-Ray zone, a mechanism would be required so that methods of the migrated object could be invoked. This is provided by the `CallHandler`. It is not enough to use a conventional RPC mechanism at this level, as a remote method invocation would pass straight through

[1] The distributed garbage collection algorithm used in DRASTIC is outside the scope of this paper. The interested reader is referred to [Dic92].

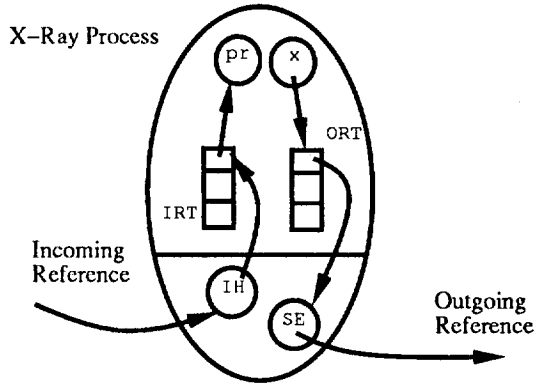

Fig. 5. A Process' Main Data-Structures

from caller to callee, bypassing DRASTIC and, in particular, the zone boundaries. The ability to intercept remote invocations is the key to our approach to supporting run-time software evolution.

Most DRASTIC processes are booted over their own persistent store. All objects that are transitively reachable from a persistent root are regularly checkpointed (stabilised) and placed on disk, thus the store becomes a snapshot of a process' state. DRASTIC processes are normally defined to be either running over their store or not running, and a store can have either zero or one process running over it. If one process requires access to another store that does not currently have a process running over it, DRASTIC's daemon processes will attempt to boot a process over that store. However, not all processes require the services of a persistent store, for example, the client in a client/server architecture. DRASTIC, therefore, allows a process to boot without a persistent store.

Should a process crash, when another process is started over the store all persistent objects are reinitialised to contain their last saved state. Orthogonally persistent systems aid in the construction of such fault tolerant systems. Crashed processes can be restarted and will fault in their objects during the normal course of execution. No special code has to be written to retrieve the last known process state and execution can be rapidly resumed because objects are faulted-in as needed rather than requiring all of the previously saved state to be read before resuming execution.

Having a snapshot of a process on disk facilitates the simple evolution of a process. When a change is made to a type that is contained in a particular process, that process can be terminated, causing its state to be written to the store. The store can then be traversed, changing old objects for objects instantiated

from the new type. A new process, containing code that understands the new type can then be booted over the manipulated store.[2]

4.2 System Wide

A DRASTIC system consists of many zones and, within a zone, processes need to contact each other. DRASTIC provides two levels of name server. The "Name Server" in figure 6 allows application level objects to contact each other. The zone specific process manager (`ZSPMdaemon`) allows DRASTIC platforms to contact each other (see §4.4). At the system wide level, daemons within zones need to find out about other processes in other zones. This is done through the zone boundary name servers (see figure 7).

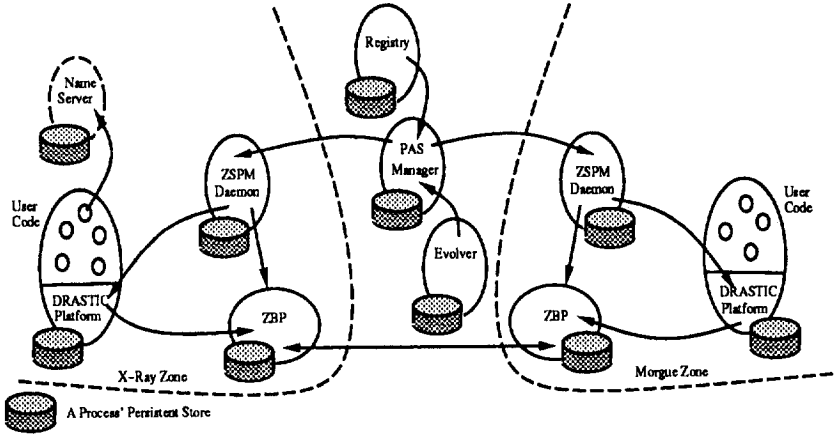

Fig. 6. System Wide Process

A multi-zone application description is contained in a persistent application systems manager (`PASManager`). The `PASManager` contains information on the persistent application system's contracts as well as references to each `ZSPMdaemon`. When booted, the `PASManager` is used by each `ZBP` process to initialise its transformation rules, which are embodied in code provided by the software engineers. When the software architect wants to evolve zones, an `evolver` process causes changes to take place in the `PASManager`. The `evolver` manages and checks the contract information, deriving change absorber code when possible. It then informs the affected zones that they are to be evolved, using an API present at the `ZSPMdaemons`.

Ideally, the `PASManager` would be replicated for protection against faults and there could be many `evolver` processes running over a DRASTIC platform. In

[2] Booting a process over a store and reinitialising a process after a crash both have implications for DRASTIC's fault tolerant remote method invocation protocol. A discussion of this protocol is outside the scope of this paper.

our current implementation, the **PASManager** is a centralized server and there is at most one **evolver** process at any given time.

The registry process is the lowest-level name server in the DRASTIC platform and it is at a well known location. The **PASManager** registers with it and each **ZSPMdaemon** gets a reference to the **PASManager** from the registry.

The **PASManager** and the **evolver** are separate processes as they capture distinct roles within an evolving system. The **PASManager** stores the contracts and is a name server for the **ZSPMdaemons**, whereas the **evolver** contains information about what needs to be done when evolution is required. The **evolver** calculates which zones need to be amended for a given change and how the **ZBPs** and contracts are affected. This information is passed to the **PASManager** for execution.

4.3 Between Zones

At the boundary between two distinct zones are a pair of processes that handle references between these two zones (figure 7). They contain tables very similar to a process' incoming and outgoing reference tables (the **IRTs** and **ORTs**). Now, however, there are two sets, one handles intra-zone references to and from the **ZBP** process and the other handles inter-zone references, to and from the corresponding **ZBP** in the other zone. The additional tables are the zone incoming reference table and zone outgoing reference table (the **ZIRT** and **ZORT**).

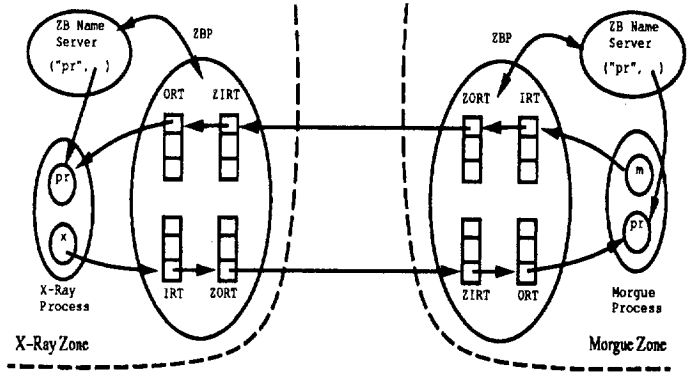

Fig. 7. Handling Inter-Zone References

When a **ZBP** is started for a given zone it is started in the context of a pair of zones, requiring another **ZBP** in the target zone. In a system with a large number of zone contracts this may result in a large number of **ZBP** processes per zone. However, it is important in such a system to be able to effectively manage the references that cross zone boundaries. This is facilitated by having one **ZBP** process per contract on either side of the zone boundary.

The ZBP process at one side of a pair enforces the contract on that side of the pair. If an application program tries to pass an object, or a reference to an object, from one zone to another, without the appropriate type being described in the contract, an exception is raised in the invoking code.

Over time it is possible for many change absorbers to be inserted along an individual inter-zone reference. The series of change absorbers can be contained in either, or both, ZBP processes. DRASTIC implements a stacking discipline for the management of the change absorbers, so that it is always possible to transform an object from its exported type to its imported type, through a series of (possibly zero) transformations. If a change absorber is removed because, for example, the type it transforms to no longer exists in the system, new change absorbers may be necessary to ensure a complete transformation.

A separate zone boundary name server (ZBNS) is provided so that application level objects in one zone can be made visible to application level objects in other zones. A process wishing to advertise an object to other processes outside its zone places a named reference to that object in its local ZBNS. For example, both the X-Ray process and the morgue process have advertised their pr objects with their respective ZBNSs in figure 7. If another process in the morgue zone wanted a reference to the pr object in the X-Ray zone, without obtaining it from the local process, it would contact the ZBNS in its local zone passing the name ("pr") to a method provided by the ZBNS for querying the remote ZBNS. The morgue zone ZBNS then contacts the X-Ray zone's ZBNS, via the ZBPs, asking for a reference to an object with that name. The query between the two ZBNSs is conducted through the two ZBPs so any reference that is passed back is subject to the necessary transformations. Within the context of the contract between the two zones, such a reference is guaranteed to be legal.

4.4 Within a Zone

When creating a zone, several daemon processes are started. The ZSPMdaemon is used by the DRASTIC platforms, which are created when starting application level processes over them. The ZSPMdaemon is a name server that contains a mapping from a process' name to the process' InHandler. When a DRASTIC process is booted it registers itself with the ZSPMdaemon, allowing other processes in that zone to obtain references to it.

Providing name servers for use within a zone decreases the amount of dependency one zone has on another, increasing its autonomy. It would be possible to have name servers directly accessible from outside their zone. For the current implementation and development of the architecture, however, strict separation of concerns is proving very helpful, hence the provision of the zone boundary name servers, corresponding to each contract, in each zone.

Five sorts of name server are used in the current implementation of DRASTIC: the Registry, PASManager, evolver, ZSPMDaemon and ZBNS. It would be possible to simplify the system by combining the Registry, PASManager and evolver into one service and the ZSPMDaemon and ZBNS into another and in-

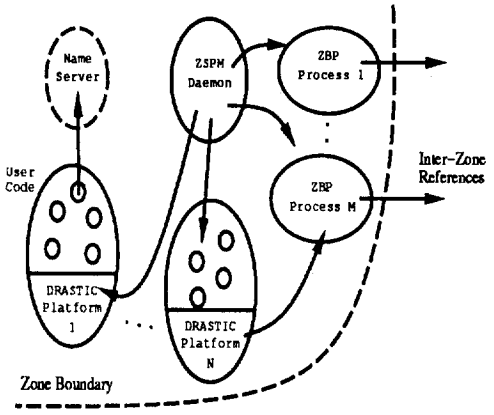

Fig. 8. Processes Booted within a Zone

creasing their functionality. This would make the services more like traders, capable of supporting sophisticated queries.

All of the DRASTIC platform's daemons and name servers can be restarted, providing a simple form of fault tolerance. Processes with references to these servers need to rebind, although within the DRASTIC platform this is done automatically by the code used to access such servers.

4.5 Implementation Details

Space limitations prevent a discussion of some of the implementation details of the DRASTIC platform. The interested reader is referred to [ED97tr] which is a more detailed version of this paper.

5 Evolution within DRASTIC

This section describes in detail how the DRASTIC platform supports computation within a system that undergoes evolution. The description is broken into four sections: firstly, the computation before evolution is described; then what occurs during system evolution is presented; how computation is performed after the change is described next; with a discussion of the differences in the system before and after evolution concluding this section.

5.1 Computation Before Evolution

The physical architecture of the example system before any evolution occurs is given in figure 9, which is a more detailed version of figure 2.

Assume that object `m` in the morgue zone wants to set the id number of the `pr` object in the X-Ray zone:

```
pr.setId(946);
```

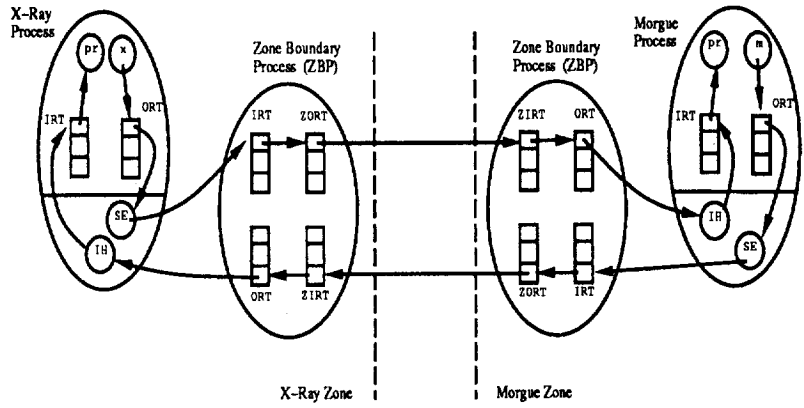

Fig. 9. Physical Architecture Before Evolution

The application-level invocation on `pr` is intercepted by the DRASTIC platform as the object being invoked is remote to this process. The `StoreExemplar` for this process pickles[3] the `int` value of 946 and forwards the call request with the pickled arguments to the `ZBP` process from the morgue zone to the X-Ray zone. The morgue `ZBP` looks up the necessary transformations for calls to PR objects invoking the `setId` method. Before evolution takes place, no conversions are necessary (see figure 3) and the morgue zone's `ZBP` just passes the call through to the `ZBP` process at the X-Ray zone where the lookup is done again. No conversion is required at this side so the call is passed to the `InHandler` associated with the process that contains the `pr` object. The `InHandler` passes the pickled arguments to the object's `CallHandler` which unpickles them and executes the `setId` method, passing the integer as an argument. If there were any results from this method call, the `CallHandler` would pickle them and pass this pickle to the `InHandler` for returning to the invoking object.

5.2 Evolving the System

The software engineer plays an integral role in the evolution of the software in a DRASTIC system. Figure 10 shows how the PR type has been evolved by both zones, leading to three new types, PR_X, PatientRecord and PR_M.

After evolution, when both processes have had their PR typed object evolved to their new type, the example application looks like figure 11.

[3] Pickling takes an object and converts it into a form suitable for transfer across a network. The pickled object contains enough information for the object to be unpickled into its original form. Pickling is also referred to as "serialization" or "marshalling".

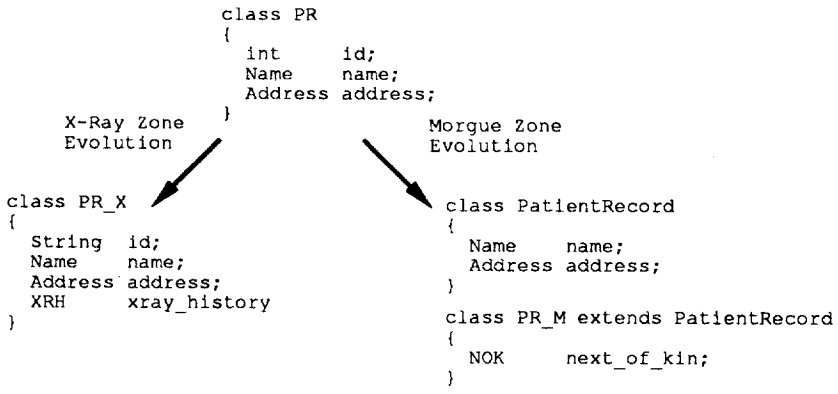

Fig. 10. The Updated Patient Record

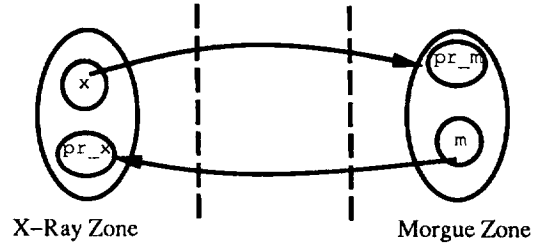

Fig. 11. The System Architecture After Evolution

To evolve both zones in this way, the software architect will have changed the zones' contents and the code as above. This change triggers changes to the zone contract and requires that new or amended change absorbers be provided.

The software engineer has to provide change absorbers to handle the disparity that has come about because of the evolution. After evolution, the **m** object in the morgue zone has a reference to an object of type PR_M as all **pr** typed objects in the morgue zone have been evolved to be objects instantiated from type PR_M. However, the object it actually refers to is of type PR_X (figure 11). The code in the morgue zone will not be able to call the new methods to get and set the X-Ray history. However, it does have the ability to call the **get** and **set** methods on the name and address fields, which the morgue zone is using to identify a PatientRecord. As the morgue zone is now using two fields to identify a patient's record, it makes sense for the software engineer to provide two methods to abstract over this implementation detail. Therefore, the PatientRecord class is assumed to also support these methods:

The **x** object experiences a similar disparity. After evolution, it holds a reference to a PR_X typed object as all PR typed objects in the X-Ray zone have been evolved to be instantiated from PR_X. However, the object it actually refers to is

Direction of Transfer	Exported Types	Transformations Required	Imported Types
X-Ray to Morgue	PR	None	PR
Morgue to X-Ray	PR	None	PR

Direction of Transfer	Exported Types	Transformations Required	Imported Types
X-Ray to Morgue	PR_X	PR_X to PR_M	PR_M
Morgue to X-Ray	PR_M	PR_M to PR_X	PR_X

Fig. 12. The X-Ray Morgue Zone Contract Before and After Evolution

```
String getId();
void setId(String name, String addr);
```

Fig. 13. Extra PatientRecord Methods to Hide Implementation Details

of type PR_M (figure 11). PR_X objects identify a patient using a single `String id`. Therefore, `x` can attempt to call `String getId()` and `void setId(String id)`. It is the rôle of the change absorbers at the zone boundary to handle the transformations between the PR_X and PR_M typed objects.

Inserting the change absorbers into the run-time system is handled automatically by DRASTIC. The system wide `evolver` process (figure 6) is aware of which zone boundary processes contain sets of change absorbers that need to be updated. The update is handled automatically through an API presented by each zone boundary process.

5.3 Evolving the Zone

In the current implementation, a zone is evolved in four stages. The four stages are executed automatically by the DRASTIC system once the software engineer has provided all the necessary updated and new software components.

1. Freezing the zone
2. Updating the objects
3. Updating the change absorbers
4. Thawing the zone

In a more sophisticated system these changes would be performed incrementally, rather than freezing the entire zone until all objects have been evolved.

Freezing the Zone Freezing the evolving zone will temporarily prevent objects in other zones from contacting objects in the evolving zone. To freeze a zone, a message is sent by the `ZSPMDaemon` to all the processes executing in that zone. On receipt of this message a process executes an identified piece of

code, supplied by the software engineer, which executes the necessary actions to safely terminate this process. References from objects in other zones to objects in this zone are still allowed. Terminating a process does not imply that any references to objects it embodied will fail, but if method invocations were forced to prematurely return, the software engineer would have to add code to handle methods failing due to evolution. This is clearly undesirable and one of the goals of this architecture is to ensure that references into the evolving zone from external zones are only temporarily blocked; the invocation should only be delayed, rather than responding with an error due to ongoing evolution. If an object external to the currently evolving zone tries to invoke a method belonging to an object that temporarily now no longer exists due to evolution, the invocation will not be rejected, it will be noted by the DRASTIC system and blocked. Once the evolution of the zone has completed and the process that contains the evolved object has been restarted, the invocation will be allowed to proceed.

All objects in the evolving zone will also be prevented from contacting objects outside of their zone.

The piece of code that terminates the process safely is required as only the software engineer knows how to safely terminate a process. It is possible for some methods to be executing infinite loops, eg. servers continually waiting on incoming requests. If the DRASTIC system merely waited for all currently executing methods to finish before attempting evolution, processes that had methods with infinite loops would never become available. Where such a loop is necessary, the software engineer is required to add code to allow termination of the loop when required for evolution.

Updating the Objects All objects in the X-Ray and morgue zones are updated to be of their respective types. This is realised in the current DRASTIC implementation by terminating each process, so that an up-to-date copy of its persistent data is residing on disk (as described above). Each affected object in the persistent store is then converted to be an instance of the new type. This may require writing code that takes the state of a PR object and loads it into a PR_X object, for example. A similar conversion is done in the morgue zone to produce PR_M objects. If a process which has been booted without a persistent store is terminated and brought back it is assumed to be capable of resuming from where it left off; in effect we assume such processes are stateless.

Updating Change Absorbers Both `ZBP` processes need to have change absorbers added to convert objects and method invocations from the incoming type to the type used locally. During the lifetime of a complex system, many change absorbers may be inserted along a given reference. For example, the patient record may go through several amendments, requiring one change absorber per change. These are placed in the `ZBP` in a stack, ensuring that the transformation is coherent.

Thawing the Zone Once all the change absorbers are in place and all the conversions have been completed, the zone is thawed and invocations can proceed to objects of the new type.

5.4 Computation After Evolution

This section is divided in two: first we describe a method invocation from the morgue zone to the X-Ray zone and then a method invocation in the opposite direction.

Morgue Zone to X-Ray Zone Method Invocation After the system has been evolved, the system's physical architecture is as in figure 14. The two processes contain objects of the new type and the ZBPs contain the necessary change absorbers.

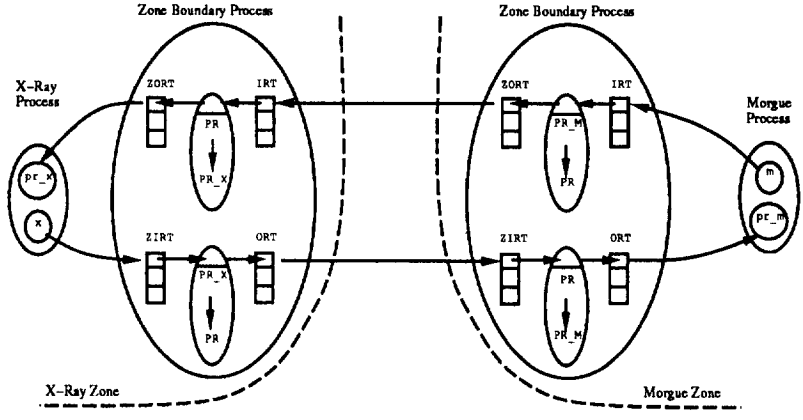

Fig. 14. Physical Architecture After Evolution

The code in the morgue process has been changed to make use of the PR_M type. Therefore, the call made by the `m` object to set the identifier of the `pr_x` object could become:

```
Name    name  = new Name("Joe", "Bloggs");
Address addr  = new Address("17", "Lilybank Gardens");

m.setId(name, addr);
```

The actions performed at the morgue zone require the `setId()` call on the PR_M typed object to be transformed into a call on to a PR typed object. This is handled by the PR_M to PR change absorber in the `ZBP` in the morgue zone. This change absorber could look something like that in figure 15.

```
class PR_M_to_PR
{
  PR pr;
}
```

```
void setId(Name name, Address addr)
{
  int trans_id = transformId(name, addr);

  pr.setId(trans_id);
}
```

Fig. 15. Simplified PR_M to PR Change Absorber

The `setId()` method of the change absorber is the same as the PR_M typed object. The `name` and `addr` objects are used by the `transformId` method to generate an integer used in the call forward on `pr.setId(trans_id)`.

When this forwarded call is received by the X-Ray zone's `ZBP` the call is found to be following a reference containing one or more change absorbers. This requires that the method invocation is passed through the PR to PR_X change absorber.

The PR to PR_X change absorber in the X-Ray zone takes the incoming method invocation on `setId(int id)` which was called from the PR_M to PR change absorber above and the unpickled argument, which is the value generated from the above `transformId`. The argument was unpickled by a `CallHandler` contained in the `ZBP`.

```
class PR_to_PR_X
{
  PR_X pr_x;
}
```

```
void setId(int id)
{
  String s = transformId(id);
  pr_x.setId(s);
}
```

Fig. 16. Simplified PR to PR_X Change Absorber

The class that handles the change from PR to PR_X typed objects (figure 16) will have a method that has the same signature as the PR `setId` method. In that method the integer is converted to a String and then the `setId` method on the `pr_x` object is called. The call to `pr_x.setId(s)` would then be a call through to the `pr_x` object in the X-Ray zone and would be handled as any other incoming invocation through the process' `InHandler`, `CallHandler` and IRT.

The `transformId` methods in both of these change absorbers are more than just conversions from one representation to another as they may depend on the semantics of the invoked object. This understanding is provided by the software engineer, who is the only one who knows the desired semantics of the application.

X-Ray to Morgue Zone Method Invocation The `x` object in the X-Ray zone invokes the `String getId()` method on its reference. On the X-Ray zone side the PR_X to PR change absorber is invoked to convert the String to an integer. The change absorber for this will look similar to that in figure 17.

The call `pr.getId()` in figure 17 is forwarded to the `ZBP` in the morgue zone which has a PR to PR_M change absorber. This change absorber has a method

```
class PR_X_to_PR
{
  PR pr;
}
```

```
String getId()
{
  int id = pr.getId();

  return transformId(id);
}
```

Fig. 17. Simplified PR_X to PR Change Absorber

that matches the `int getId()` method signature and it calls on to the `pr_m` object. This could be implemented in the PR to PR_M change absorber given in figure 18.

```
class PR_TO_PR_M
{
  Map       id_map;

  PR_M      pr_m;
}
```

```
int getId()
{
  Name name    = pr_m.getName();
  Address addr = pr_m.getAddress();

  // '+' is overloaded to produce a suitable
  // argument for the get method

  return id_map.get(name + addr);
}
```

Fig. 18. Simplified PR to PR_M Change Absorber

As the change in the morgue zone has caused the `int id` field to be removed from PR_M typed objects, some form of conversion is required between the integer value that has to be returned and the name and address fields used by PR_M typed objects.

The software architect has decided to do this by defining a map from the patient's name and address to the integer value that should be returned. Using a map is purely an implementation decision on the part of the software architect. They have decided that the semantics embodied in the system are such that this mapping will always result in the same `int id` value being returned for a given `name` and `address` pair.

It must be stressed that the conversion from the `name` and `address` pairs to integers in this way is an application-level issue. There are many other ways of converting between the two patient record identifiers. The software architect must ensure two things: that there is an object in the change absorber list in the morgue zone with a method that has the same signature as was called by code in the change absorber in the X-Ray zone[4], and that for a given `name` and `address` pair the same `int id` will always be returned. How the conversion is performed to protect the change in one zone from the change in the other is something that only the software engineer can know. The same argument applies to the two kinds of `transformId` methods used in the other change absorbers.

Adding the `setId()` and `getId()` methods to the PatientRecord class is more

[4] The DRASTIC platform provides some support to allow this to be checked.

than just an abstraction over the implementation of that type. The identifier for a PatientRecord in the morgue zone consists of a name and address field. Both of these fields are required to identify a particular patient record. If `setId()` was not defined, a call to `getName()`, for example, would only return the name of the patient. As the identifier for a patient record in the morgue zone is defined using both fields, a method is required so both fields are used when retrieving the patient's id. The need to define this method is a semantic issue as only the software engineer knows that both fields must be used to identify a particular patient record.

Semantic Change Absorbers When an object passes across a zone boundary it undergoes a series of transformations. During these transformations, data may be lost or gained as the intermediate objects could contain differing numbers of fields, as in the example in section 5.4. The software engineer could define a change absorber that provided default values for object fields that are added when transforming an object.

A more sophisticated change absorber could save and restore some of an object's data during its outward and return journeys. This requires that the change absorber has access to the `ZBP`'s persistent store. If state saved on the outward journey is required on the object's return journey then two or more change absorbers may need to share state. If the software engineer wants to do this, they must write code that can identify particular objects as they travel across the change absorber in either direction, hence the need for DRASTIC object identifiers to be exposed to the application programmer.

This assumes that the object travels through the change absorber in the same `ZBP` pair on the outward and return journeys. It is possible for an object to leave a zone via one pair of `ZBP`s and return via a different pair, eg. by moving from zone A to zone B and then from B to zone C before returning to zone A. If this were to happen, any object state that was stored on the outward journey would be present in the A to B pair of `ZBP`s. The object is, however, travelling back to zone A from zone C and so will pass through the pair of `ZBP`s for the zone C to zone A contract, missing any state saved on the outward journey. This is an extremely complex problem to solve at the application-level. The DRASTIC platform allows the application programmer to do this, although it is not recommended. Building support into the platform to solve this problem requires taking a higher-level view than the current pair-wise view of zones and this is not an area that the current DRASTIC platform and design addresses.

5.5 Architectural Differences after Evolution

The main architectural difference after evolution is the addition of the change absorbers. When a method invocation is performed along this reference, the method argument has to be unpickled and transformed, potentially using other change absorbers. This argument is then passed to a method on the change absorber which has the same signature as the method being called at the invoking

process. The change absorber's method will then call on to the `pr_x` object in the X-Ray process. When the PR_X object's method returns, it passes any results back to the calling change absorber. The change absorber will then pass the result back to the invoking process.

Another difference is the inheritance hierarchy in the morgue zone consists of two classes, the PatientRecord and PR_M class. The change absorber in the `ZBP` at the morgue zone protects the software in the X-Ray zone from being aware of this change. The change absorber presents a method that has the same signature as called by the `x` object. Inside this method was the call onto the evolved object, thus hiding the change from the X-Ray zone.

The system's configuration repositories are now different. For example, the `PASManager`'s contents have changed because the X-Ray and morgue zone contract has been updated.

Other processes in other zones may have experienced a delay while the X-Ray zone was changed. Objects migrating into the X-Ray zone and requests to invoke object methods were temporarily delayed during evolution. An object migration that was started but not completed by the end of evolution may be rejected as the new contract may not allow transfer of the object's type between the two zones. The information that an attempted object migration has failed will be returned in the same way as errors are reported due to communication failure. It is up to the software engineer to decide whether or not to distinguish between these two sorts of problem.

The `m` object and `x` objects are not aware there has been a change in the other zone, only that communication with that zone was temporarily delayed.

6 Performance Measurements

This section presents some preliminary performance measurements of our unoptimised DRASTIC system. Section 6.1 gives timings for a null method invocation within a single Java virtual machine and between two processes using RMI, section 6.2 gives figures for a null method invocation within the DRASTIC system.

The measurements were conducted using a 10Mbit/s Ethernet and lightly loaded Sun SPARCstation 20 workstations running Sparc Solaris. The Java virtual machine used was version 1.0.2 of Sun Microsystem's JDK which does not use any just in time (JIT) compiler technology. Two mechanisms were used for the timings. One, incurring an overhead of $4\mu s$, permitted the measurement of events known to take less than one second. An alternative approach for longer timers introduced an overhead of $34\mu s$, but allocated memory and could therefore introduce occasional delays of $340\mu s$ due to extension of the Java heap.

All measurements were conducted repeatedly, and in every case the variance in measurements was negligible. The timer overheads above have been subtracted from the figures presented below. Values exceeding 1ms have been rounded to two significant figures or 1ms, whichever is the greater accuracy.

6.1 Timing Java

A null method invocation was executed in a single Java virtual machine and in the context of RMI to establish a base from which to compare the other timings.

Java executed a null method invocation in $2\mu s$, however the first call took approximately $70\mu s$. The decrease is due to the optimisations that are performed by the Java virtual machine on the first invocation of an object's method. Subsequent calls are much faster as the virtual machine has an optimised path to finding the method to execute.

A null method invocation between two different processes was conducted using RMI. The first test was from an object in one process to a different object in another process with both processes running on the same machine. The average elapsed time was $6200\mu s$. When the processes were run on two different machines, the cost increased to $8200\mu s$.

6.2 Timing DRASTIC

The timings for DRASTIC are divided into two sections. First figures for communication between processes in the same zone are presented, then for processes in two different zones.

Intra-Zone Measurements The time taken to invoke the null method on a local object that exists in a process booted over the DRASTIC platform is $2\mu s$. This is the same as the local Java case as objects in a DRASTIC process that are not capable of being migrated are the same as conventional Java objects i.e. DRASTIC adds no overhead in this case.

If the software engineer requires an object to be migratable the source code for that object's class must be preprocessed to add a proxy object which is called by the software engineer's code. The proxy then forwards the call to the actual object. In the local case where both the proxy and the actual object are in the same process the null method invocation takes $32\mu s$. The majority of this cost, $28\mu s$, is because the proxy is synchronizing on an object before and after the call is forwarded. These synchronizations are done to ensure the proxy's state is consistent, as a concurrently executed `migrate` method call to move the object elsewhere could invalidate the object's state if no synchronization was used. Without the overhead of the synchronization, the forwarded call would cost about $4\mu s$, which is to be expected as two method invocations are required: one to the proxy and one from the proxy to the actual object.

The next two tests involve DRASTIC remote method invocations through proxies to objects in other processes which are running on the same or different hosts in the same zone. To conduct this test a number of DRASTIC name servers need to be running. This is so objects in different processes can gain references to other objects placed in the name servers under well known names. Running the name servers contributes some load to the machines, but this is not appreciable in the figures.

The first DRASTIC remote method invocation is performed between two objects residing in different processes running on the same machine. This invocation takes 128,000μs. In this case, the reference that the invoking side is using was retrieved from its local zone boundary name server (ZBNS) and therefore the invocation chain leads from the caller to the callee through the ZBNS. This means the call has to be forwarded using two RMI calls. An optimised case with the call going directly from caller to callee reduces the cost to 55,000μs. When running the two processes on different machines the unoptimised cost is 146,000μs, reducing to 84,000μs for the optimised case. If these figures are compared to the RMI values given earlier it can be seen that the added functionality provided by DRASTIC, the ability to migrate objects and evolve objects, adds a factor of approximately twelve over a conventional RMI call. However, the DRASTIC platform has not been optimised; in particular, several strings are passed as arguments even in a notionally null remote call. Profiling suggests that the marshalling costs of handling these strings may dominate the DRASTIC remote call costs, so we are now investigating more efficient representations for type and method names.

Inter-Zone Measurements The final test performed was between two processes running on different machines which are placed in different DRASTIC zones. This requires that a zone boundary process (ZBP) and a zone boundary name server (ZBNS) be run in each zone. A reference to the object to be invoked is placed in its local ZBNS. A different object in a process running in the other zone obtains a copy of that reference by querying its local ZBNS. The local ZBNS contacts the remote ZBNS via the ZBPs for that reference. It is then returned through both ZBPs and ZBNSs. In the unoptimised case, this generates a reference similar to the case above, that leads from one process to another via four other processes. The optimised case misses out each ZBNS on either side, reducing the number of other processes involved to two, the two ZBPs. An unrestricted zone contract was active during this test. The ZBP, ZBNS and the process being measured in each zone were run on the same machine.

To invoke the null method in one process from another process running in a different zone on a different machine, using the unoptimised path of four intermediate processes cost 382,000μs. Using the optimised path, the cost is reduced to 241,000μs.

The optimised intra-zone call takes 84,000μs and the optimised call across the zone boundary costs 241,000μs. The factor of three increase is to be expected as the inter-zone call has to travel through the two ZBP processes. The cost of calling across the zone boundary is 236,000μs (382,000μs$-$146,000μs) in the unoptimised case and 157,000μs (241,000μs$-$84,000μs) in the optimised case.

Timing with Java JDK1.1 The DRASTIC system has recently been ported to version 1.1 of Sun's JDK, with an encouraging increase in performance. For example, in the case of a Java RMI from two different processes running on the same machine the mean elapsed time has reduced from 6200μs to 5400μs. When

the processes are running on two different machines the elapsed time reduces from 8200μs to 7100μs. Under JDK1.1 the cost of a DRASTIC inter-zone call that leads through the two ZBPs has reduced from 241,000μs to 136,000μs.

This increase in speed can largely be attributed to the use of assembler code in the main interpreter loop of JDK1.1.

7 Related Work

It is not possible to give a comprehensive treatment of other work due to space limitations. The interested reader is referred to [ED97tr].

7.1 CORBA

CORBA does not directly address the issue of evolution. CORBA separates an object's interface and implementation and provides two repositories where object interfaces and implementations may be stored. The primary role of the interface repository (the implementation repository is similar) is to manage and provide access to a collection of object definitions specified in OMG IDL. A CORBA ORB provides the means by which client processes make requests and receive responses to and from servers. A critical use of the interface repository is for connecting ORBs together. When an object is passed from one ORB to another, it may be necessary to create a new object to represent the passed object in the receiving ORB. This may require locating the interface information in an interface repository in the receiving ORB. By getting the object's repository id from a repository in the sending ORB, it is possible to look up the interface in a repository in the receiving ORB. To succeed, however, the interface for that object must be installed in both repositories with the same repository id [Obj95b]. If the remote interface is evolved it may be given a new repository id and CORBA does not provide a means of tracking such an evolution, leaving the programmer to manage this complexity.

When adding a new interface to a repository it is possible that external processes may see an incoherent repository. A coherent repository is one whose contents can be expressed as a valid collection of OMG IDL definitions [Obj95c]. The repository does not ensure that it contains coherent information and it is possible to enter information that does not make sense, although those errors that are detected are reported. The expectation in [Obj95c] is that a combination of conventions, administrative controls and tools that add information to the repository will work to create a coherent view of the repository information. The programmer has to manage this and there is no means of bounding the effects of evolution as the contents of a repository are visible throughout the system.

8 Current Status

The DRASTIC project is half-way through its initial funding and this paper has, therefore, presented work in progress.

The basic DRASTIC architecture was first implemented with Modula-3 [CDJ+89] using Network Objects [ABW95] for distribution. The current implementation uses PJava [AJDS96] and Java's [GM96] remote method invocation package, RMI [WRD96]. Porting from one language to another was relatively easy, suggesting that DRASTIC's architecture and design may indeed be somewhat language independent.

Object migration and distributed reference management were completed during the Modula-3 prototype. The current implementation has built on this architecture and it now supports multiple zones and persistence.

9 Further Work

Much work remains to be done in the DRASTIC project, including providing a more general framework in which object migration can take place. We are currently experimenting with several different object migration APIs, one of which will allow groups of objects to migrate atomically.

Extension and optimisation of the DRASTIC platform is an ongoing effort to support more efficient run-time type evolution and to allow us to conduct additional experiments.

More tools are required to support the software engineer. For example, a more complete pre-processor for the generation of migratable objects is needed. A tool to assist the software engineer in checking the coherency of a series of change absorbers may prove useful.

Manipulations of the Java virtual machine to provide basic support for class evolution at a level below the DRASTIC platform is also being investigated, to permit the construction of more challenging demonstrations.

9.1 Freezing a Zone

The most significant area for future work is in dealing with currently executing object migrations when freezing a zone. Freezing an entire zone before it can be evolved is inefficient and potentially problematical as it unnecessarily keeps objects frozen after their evolution. A more incremental approach must be taken where objects are thawed immediately after they have been evolved. The process of freezing a zone causes all the currently executing processes to terminate to ensure a consistent version of their state is on disk. This problem is illustrated below with an example.

Invocations that do not Return Consider a zone called P with a single process in it called `p`. Process `p` contains an object `o` with a method `m1` which executes an infinite loop. Another process, `q`, which is in another zone, has a reference to `o` and it is currently executing `m1`.

When zone P is evolved the `ZSPMDaemon` will send process `p` the **`evolve`** message. On receipt of this message, process `p` will terminate. However, process

q is executing the m1 method. When the evolved process p is started again, m1 must be restarted at the point where it left off to ensure correct behaviour. The current DRASTIC platform does not support this and it is left up to the programmer to start m1 running again for all references to object o from outside the evolved zone.

Involving the Software Engineer The above example illustrates why a more sophisticated approach to freezing a zone is required. The simplistic freezing of an entire zone is only being used at the moment to gain experience in this area. We do not believe that this problem can be solved automatically, in the general case. We are therefore seeking to find clean models which permit the software architect to apply their knowledge of the application's semantics to this issue.

9.2 Minimising Zone Evolution Time

Evolving a zone is a costly operation as the store for each process has to be manipulated to transform objects from one type to another. In a distributed, persistent environment the services that processes provide need to be highly available. It will not always be possible to terminate a process, evolve the contents of its store and restart it as evolving the store will cause the process to be unavailable for an unacceptably long period of time. To minimise the amount of time a process is not available can be accomplished by doing as much of the evolution as possible off-line, but this raises other difficult questions.

10 Summary

In this paper we have presented DRASTIC's model and run-time architecture. DRASTIC uses zones to decompose a system into collections of processes that evolve as a single unit. Zones are concepts at design time which are realised as explicit system components at run-time. Contracts are pair-wise agreements between zones that describe which types can and cannot flow between any two zones, what needs to be done to transform objects as they move from one zone to another and how to transform inter-zone method invocations. It has been shown, through an extended example, how zones increase system modularity by encapsulating software and how contracts increase autonomy by explicitly describing the degree of coupling between zones. Change absorbers, provided by the software engineer, are automatically inserted at the zone boundary to protect one zone's software from changes made to software in other zones. How the software architect would build a system using DRASTIC has been described in some detail.

At run-time the DRASTIC platform intercepts all inter-process application-level object references. The ability to do this is the key to our approach to supporting run-time software evolution.

Acknowledgements

The DRASTIC project is supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under its Architectures for Integrated Knowledge-Manipulation Systems programme (AIKMS). The project also receives support from Digital and ICL.

The authors thank the referees for their helpful comments.

References

[ABW95] S. Owicki A. Birrell, G. Nelson and E. Wobber. Network objects. Technical Report 115, DEC SRC, December 1995.

[AJDS96] M. Atkinson, M. Jordan, L. Daynès, and S. Spence. Design issues for persistent Java: a type-safe, object-oriented, orthogonally persistent system. In *Proceedings of The Seventh International Workshop on Persistent Obje ct Systems*, Cape May, New Jersey, USA, May 1996.

[AM95] M. Atkinson and R. Morrison. Orthogonal persistent object systems. *VLDB Journal*, 4(3):319–401, 1995.

[BHJ+87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract data types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.

[CDJ+89] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The Modula-3 type system. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas*, pages 202–212. ACM, January 1989.

[Dic92] P. Dickman. *Distributed Object Management in a Non-Small Graph of Autonomous Networks With Few Failures*. PhD thesis, University of Cambridge, September 1992.

[ED97tr] Huw Evans and Peter Dickman. Drastic: A run-time architecture for evolving, distributed, persistent systems. Technical report, Department of Computing Science, Glasgow University, 1997.

[GM96] J. Gosling and H. McGilton. The Java language environment: A white paper. Technical report, Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, USA, October 1996.

[Obj95a] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 1995. Version 2.0.

[Obj95b] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 1995. Version 2.0, page 6-3.

[Obj95c] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 1995. Version 2.0, page 6-4.

[WRD96] A. Wollrath, R. Riggs, and C. Darke. Java(tm) remote method invocation specification: Prebeta draft, version 1.1. Technical report, Sun Microsystems, Inc, 2550 Garcia Avenue, Mountain View, CA 94043, USA, November (C) 1996.