# Framework for Dynamic and Automatic Connectivity in Hierarchical Component Environments

Gábor Paller
Nokia Research Center, Köztelek str. 6, Budapest, 1092, Hungary, email: gabor.paller@nokia.com

## Abstract

*Component frameworks have been receiving continuous attention in the last decade. Dynamic component-based composition plays central role in the area of reflective software where the structure of the application and the supporting middleware can be changed during the execution to adapt the software to environmental changes. Selection, instantiation and wiring of the components are generally the tasks of "reconfiguration managers" that are able to interpret context changes and to create the appropriate component network. This centralized approach is not satisfactory for more complex systems because the reconfiguration managers must be aware of all the relevant context combinations and the relationships to component configurations. Even in hierarchical component networks the complexity of reconfiguration managers could quickly get out of control as the number of components and context states grow.*

*This paper proposes a more distributed approach for dynamic, hierarchical component composition where components themselves influence component instantiation and wiring. The proposal is based on an analogy with biological cell transfer processes. The paper presents this approach and demonstrates its use on a Fractal-based demo implementation motivated by a mobile application use case.*

## 1. Introduction

Systems that change their architectures dynamically emerged during the analysis of many domains. Reflective middleware [2],[7] identified the need for such systems for middleware serving applications in dynamically changing environments. It was pointed out that dynamic architectures can be efficiently used for self-healing/self-management systems [4]. The relationship between reflective systems and self-healing systems was analyzed in [6]. One main objective of IBM's Autonomic Computing initiative [1] is the following: "An autonomic computing system must configure and reconfigure itself under varying (and in the future, even unpredictable) conditions." All these problem statements lead to dynamic architectures.

Today the research scene is dominated by dedicated reconfiguration managers that listen to context changes and adapt the target software configuration accordingly [3],[4],[5],[8]. The event-conditions-action pattern is frequently used and the action part is often described by some script that reconfigures the component network. [15]. Some approaches allow for the fine-tuning of the rule-set by learning [9] but the rule-set is still enforced by the dedicated configuration manager.

The vision of the Dynacomp framework presented in this paper is an architecture that automatically rearranges the components according to component constraints when a component instance is inserted, removed or its constraints are changed. There is no specialized reconfiguration manager, the introduction of a component into the component network is enough in itself to trigger reconfiguration. Similar systems have already been presented, e.g. [11] and Service Binder [13]. These systems, however, are able to work in non-hierarchical component frameworks only and in Service Binder ambiguity resolution is inadequate.

The Dynacomp framework overcomes these limitations. Dynacomp differs from Service Binder in the following fundamental ways:

- Richer meta-information attached to components that allows them to influence the reconnection process more precisely.
- Ability to work in component hierarchies.

Dynacomp components can also be compared to classpects [14] of the aspect-oriented world. Like classpects, Dynacomp components are self-contained because the functionality of the components and the way the component changes its environment are combined into one unit. The fact that component frameworks may be used to implement aspect-like behavior has already been pointed out in [16].

Dynacomp's support of hierarchical component architectures is important because the number of connections

[1] http://www.research.ibm.com/autonomic/

is lower in component hierarchies therefore the automatic connection can finish faster (see appendix for discussion).

Dynacomp was inspired in many ways by the internal processes of a biological organism. Similarly to biological organisms, components are arranged hierarchically. Components can be created and destroyed continuously while components can reach the location where they are eventually used by means of transfer processes. Unlike in biological organisms, component transfer in Dynacomp serves mainly the purpose of architectural reconfiguration, while data transfer is accomplished by the usual mechanism (e.g. method calls). Dynacomp's component transfer also differs from data-driven approaches e.g. architectures based on Chemical Abstract Machine (CHAM) [10] the principles of which are very similar to those of Dynacomp. CHAM, however, uses its "molecule" and "transfer process" abstractions to describe computations, while in Dynacomp these abstractions are used solely for component network adaptation purposes. Applying biological principles to the self-healing problem was already discussed in [18] although the architecture presented in the paper is still based on dedicated reconfiguration managers.

## 2. The dynamic composition framework

Dynacomp is built on top of the Fractal framework [1]. The Dynacomp framework uses two types of components that extend Fractal's composite and primitive component types. Additionally to Fractal's features the following functionalities are provided by Dynacomp component types.

**Dynacomp primitive components** are able to expose their constraints to be used for dynamic composition.

**Dynacomp composite components** are able to rearrange other components contained in the composite component and to participate in transferring components into and out of the composite component. The collection of components managed by Dynacomp composite component is also called the *domain* of the composite component, while the composite component containing other component is referred to as the *container* of the components it contains. Composite components are able to expose their constraints in the same way as primitive components do.

As described previously, containers arrange components in their domain according to constraints. The constraints are provided by individual component instances thus component instances control the reconnection procedure directly by means of these constraints. In a border-line case expressing constraints needs a programmatic approach with full access to the component to be connected. However, a much simpler declarative approach was chosen which was found

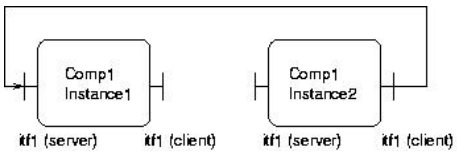


**Figure 1. Bidirectional interfaces**

to be satisfactory during the case study implementation. This declarative approach adds further meta-information to the - quite limited - meta-information already available in the Fractal framework. The following meta-information is proposed by this paper:

**Interface name** Each interface is named and only interfaces with the same name can be connected. The interface name convention comes directly from Fractal.

**Interface direction** Beside Fractal's client (outgoing) and server (incoming) interfaces, the dynamic composition framework introduces the bidirectional interface type. A component exposing bidirectional interface has one client and one server interface with logically the same name (due to a Fractal restriction, the real names can not be identical). One bidirectional interface can be connected only to another bidirectional interface. The bidirectional interface is connected at a random direction, e.g. the client interface of the first component will be connected to the server interface of the second component. The server interface on the first component and the client interface of the second component will be left unconnected (see figure 2). Bidirectional interfaces are able to express the statement that certain component instances of same type have to be connected with each other. Without the notion of a bidirectional interface, this statement would create loop in the connection graph hence the component network could not be started.

**Interface multiplicity** Fractal differentiates between the singleton and collection cardinality. Automatic composition needs more precise description of allowed interface connections. Dynacomp adopts the cardinality description of Service Binder [13]. The cardinality of the interface (both client and server) can be one, one or more, zero or one, zero or more connections. Cardinality can be specified for both client and server interfaces.

**Interface priority** Priority is assigned to all component interfaces. Priority determines which component interface to select for connection in case there are multiple component interfaces satisfying the search criteria. The priorities of the interfaces to be connected are

added and the interface pair with the largest sum is selected for connection.

**Component and interface properties** Similarly to Service Binder, it is possible to narrow component instance candidates by means of matching component instance or interface properties. According to the OSGi [2] service property convention, Service Binder allows properties to be assigned to individual interfaces of the component. Dynacomp assigns properties primarily to components. Those properties can be overridden by properties assigned to interfaces. The properties are matched according to a very simple pattern language. This language is able to describe that the other component must have a certain property with a certain value (mandatory property). It can also show that the other component may have a certain property in which case the value must be the same as the value of the component to match (optional property). Finally the kanguage can describe that the other component is not allowed to have a certain property with certain value (forbidden property). This pattern language is significantly simpler than OSGi's LDAP filter although the optional property feature cannot be expressed with the LDAP filter mechanism. The need for the optional property feature emerged during the case study implementations. A more advanced constraint language was presented in [11]. The constraint language is a footprint-complexity tradeoff. A more complex constraint language is favorable but it means more memory and longer reconnection time. The features of Dynacomp's simple constraint language were found to be enough for the prototypes.

**Superproperties** Components can have superproperties that determine which container this component can be placed into. The component can be imported into the container only if its superproperties match the properties of the container. The matching rules are identical to those of component and interface properties.

**Export and import lists** Containers can have export and import lists controlling which components can leave the domain of this container and which components can be moved into this container. CHAM [10] has similar mechanism but CHAM applies it to data items instead of components. The export and import list contain interface names. If an unconnected component has an interface with the name on the export or import list, it can be exported to the higher (enclosing) domain or a component from the higher domain can be imported into this domain. The import happens only if the superpropeties of the component match the properties of the domain.

**Timeout** Unconnected components "expire" after a timeout and are removed from the storage by the framework.

**Creation time** The framework places a timestamp on components when created. This timestamp can be used to implement a conflict resolution logic based on component age, e.g. if a component detects that it is in conflict with some other component in the domain, the surviving component is the younger one.

**Creation list** Each component has an assigned a creation list which is a list of component names. The list may also contain labels. The creation list can be considered as a scenario about how the domain should be rearranged by inserting a set of components into the domain. When a certain event occurs, the event handler can look up a certain label in the creation list and create all the components on the list from this label up to the next label. This activity can be carried out by the component itself (using its own creation list) or by components specializing on processing the creation lists of other components.

While interface name, direction, multiplicity and priority are uniform for each component of the same type, other meta-information can be set per component instance.

When a component is inserted into the domain of the container, the domain is rearranged. Components are stopped, connected according to their constraints, the connected components are started in dependency order and the unconnected components are placed into the storage of the domain. Components in the storage may be exported into higher domain, imported into the domain of a container in this domain or deleted if the unconnected component's timeout has expired. The exact algorithm is described below. For the purpose of this description the set of components already existing in the system is denoted as $C$. The component to insert is denoted as $c_i$ and the component whose domain it is inserted into is marked as $c_p$. The set of components in the domain of $c$ are denoted as $child(c)$ and the container in which $c$ resides is denoted as $parent(c)$. $parent(c)$ is theoretically a set but Dynacomp does not support the Fractal feature that one component can be part of multiple domains. In the Dynacomp model $\mid parent(c) \mid \equiv 1$ for any $c \in C$.

- The reconnection procedure gets an identification number. This reconnection ID must be unique (successive reconnection passes must have different IDs) so that each container can recognize whether it was visited by this reconnection pass.

[2] Open Service Gateway Initiative, http://www.osgi.org

- $c_i$ is added as a subcomponent of $c_p$ at Fractal level.
- Every $c \mid c \in child(c_p)$ are stopped and components are disconnected.
- The components in the domain of $c_p$ (including those in the storage) are reconnected. When connecting interfaces, the framework considers interface names, properties, direction and multiplicity. Note that in order to minimize reconnection time, the algorithm is greedy: it connects interfaces with the highest priority and does not consider whether more components could be eventually connected with non-greedy strategy. After the connection pass, components are considered to be *resolved* when each mandatory interface of the component is connected. Unresolved components are put into the storage.
- The storage is propagated downward, toward the composite subcomponents of $c_p$. Each composite subcomponent is examined whether it is able to import any component in the storage (any component in the storage has an interface the name of which is in the import list of the composite subcomponent) and whether the superproperties of the components in the storage match those of the composite subcomponent. If it is found that any component can be imported into a composite subcomponent, that subcomponent is reconnected after the import. As a side-effect, the composite subcomponent may become resolved which requires another reconnection pass in the domain of $c_p$.
- The components in the storage are checked for timeout. When a component enters the storage, its base time for timeout is set. This base time is used for the calculation of timeout. If the timeout of the component expires, the framework removes the component.
- If any composite subcomponent becomes resolved, the components are disconnected again and another iteration occurs. Note that this may be an endless loop. This is considered to be the programmer's fault who is expected to create component networks that can be resolved by this algorithm.
- All the components in the domain of $c_p$ are started in dependency order. Simple ASAP schedule [17] is calculated and components with lower ASAP time are started first. During this process composite subcomponents are started recursively - when the composite subcomponent is started, every subcomponent in its domain is started as well.
- The component network may change when the components are started. For example the startFc() method of some component may delete itself or some other component, hence connected components may become unresolved. If any component becomes unresolved during the start of the component network, the whole reconnection procedure restarts.
- Components still in the storage are propagated up to $parent(c_p)$. This process is controlled by the export list of $c_p$, the superproperties of $parent(c_p)$ and the properties of the component to export. If the export is successful, the exported component is removed from the domain of $c_p$ and is placed into the domain of $parent(c_p)$ which is then reconnected. By using the reconnection ID the framework guarantees that the exported component will not be reimported into the domain of $c_p$ during this reconnection pass even if this reimport were possible. This prevents "playing ping-pong" with a component between a parent and a child container.

It must be noted that the optimization criteria of the algorithm only approximates the optimal global component arrangement. Greedy connection algorithm connects as many components as it can to high-priority interfaces. If a component can be connected in a domain, the framework will never try to connect this component in another domain even though the component could be exported or imported into that domain and the priority sum in that other domain could be higher. Component developers must be aware of this fact and use the component meta-information to create the connections they want.

## 3. Case study: the Java2D demo

The dynamic composition will be demonstrated on the familiar Java2D demo application. Java2D is a Swing application shipped with every Sun Java Development Kit as example program for demoing the graphical capabilities of the standard Java 2D API. Java2D demo and the underlying Java 2D API are implemented entirely in Java and the highly dynamic demo puts a significant load onto the garbage collector. The memory needed on the heap is not simply the sum of the sizes of live objects but several times more memory is needed in order to achieve realistic performance [12]. Our motivation is to componentize Java2D so that users can instantiate only the services they actually need. Unused services will not be instantiated thus memory can be saved. I will show several cases of component instantiation and reconnection occuring in the refactored Java2D to demonstrate how dynamic composition enables this vision.

The first interaction allows the user to add demos to the demo set he is able to use (figure 2). Demos are grouped into demo groups and one demo group is rendered on one

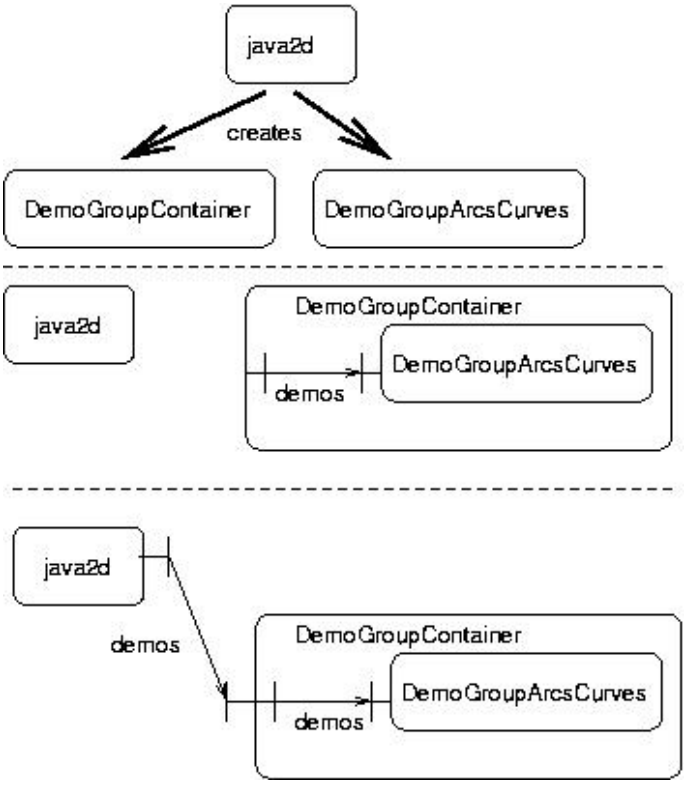


**Figure 2. Creating demo groups**

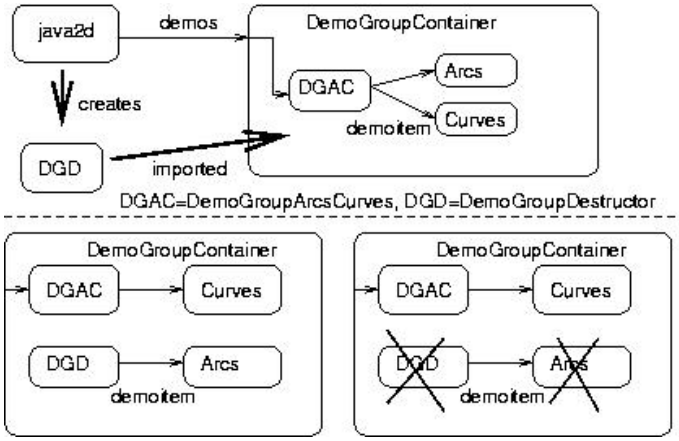


**Figure 3. Deleting individual demo components**

tab. Java2D core application (main frame and intro) is represented by one instance of *java2d* component. First the demo group is created. One demo group is represented by one composite *DemoGroupContainer*. This composite component contains one demo group component instance (*DemoGroupArcsCurves* in our case) and zero or more demo component instances. First the *DemoGroupContainer* and *DemoGroupArcsCurves* are created into the domain where *java2d* is. There is a superproperty for the demo group and a property for the container. (demogroup=arcscurves). This means that the demo group can only be placed into a container with the same demogroup property. After the two components are created, reconnection starts. None of the components can be connected because the internal interfaces of the container are not connected and the demo group superproperty does not match the properties of *java2d's* parent. As a result both are placed into the storage. The container, however can import components with *demos* interface so the demo group is imported into the container and the container is reconnected. This time the connection succeeds, the mandatory internal interfaces of the container are connected, and the container gets resolved. This, in turn, triggers reconnection at *java2d's* level: *java2d* and the container get connected over their *demos* interfaces. When the components are restarted, *java2d* checks the connection vector of its *demos* client interface and if a new *demos* connection is detected, *java2d* queries the name of the group, creates the tab and adds the AWT container created by the demo group component to the tab.

Demo components are placed into demo groups in a similar way. However, the interaction taking place when a demo component is removed from the group is interesting. (figure 3). In this case *java2d* creates an instance of *DemoItemDestructor* component into its domain. Then it sets its properties in such a way that the destructor component will be imported into the *DemoGroupContainer* containing the demo component and will be connected only to the demo component to be removed (demogroup and demoitem properties are used to select first the container then the demo component). The priority of the destructor's *demoitem* interface is increased so if the destructor is propagated into the domain of the demo component, it will "steal" the demo component from the demo group component. When the components are restarted, the destructor destroys both the demo component and itself, which triggers reconnection. The effect is that the demo component ceases to exist.

A feature of demo groups is that they are able to render demos either in a grid on the same surface or individually on tabs of a tabbed pane. When Java2D was componentized, this functionality was refactored by means of inserting a *DemoGroupTabbedPane* component instance between the demo group and the demo components (figure 4). *DemoGroupTabbedPane* has both client and server *demoitem* interfaces [3] with higher priorities than either the demo group or the demo components. This means that after reconnection the demo group client interface will be con-

[3]Fractal does not allow components to have interfaces with the same name. Dynacomp provides name mangling to create logically same names.

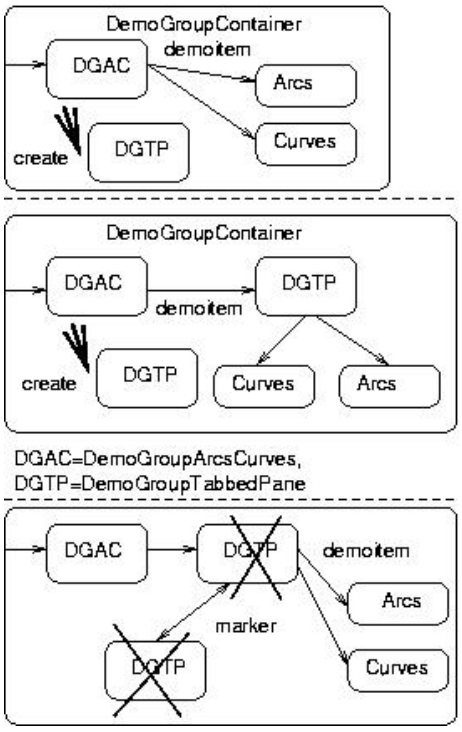


**Figure 4. Switching between grid and tabbed view**

nected to *DemoGroupTabbedPane*'s server interface and all the demo components will be connected to *DemoGroupTabbedPane*'s client interface. This allows *DemoGroupTabbedPane* to display and hide demo components using the tabbed pane. Switching back to grid view just requires the destruction of the *DemoGroupTabbedPane* component then the reconnection of the domain because these steps will connect the demo components to their demo group. This is accomplished by creating another *DemoGroupTabbedPane*. The *DemoGroupTabbedPane* has an optional bidirectional *marker* interface. If there is one *DemoGroupTabbedPane* in the domain, the *marker* interface is unconnected. If, however, there is another *DemoGroupTabbedPane*, the *marker* interfaces are connected. When *DemoGroupTabbedPane* is started and discovers that there is another component connected to its *marker* interface, it destroys both that component and itself. After reconnection, *DemoGroupTabbedPane*s are gone and demos are connected back to their demo group.

Java2D has a performance monitor feature. The demos measure their own performance (e.g frame per second) and send this data to the performance monitor which displays the performance data. Enabling and disabling the performance monitor is done by replacing a dummy implementation (that has only stub code and does not display anything)

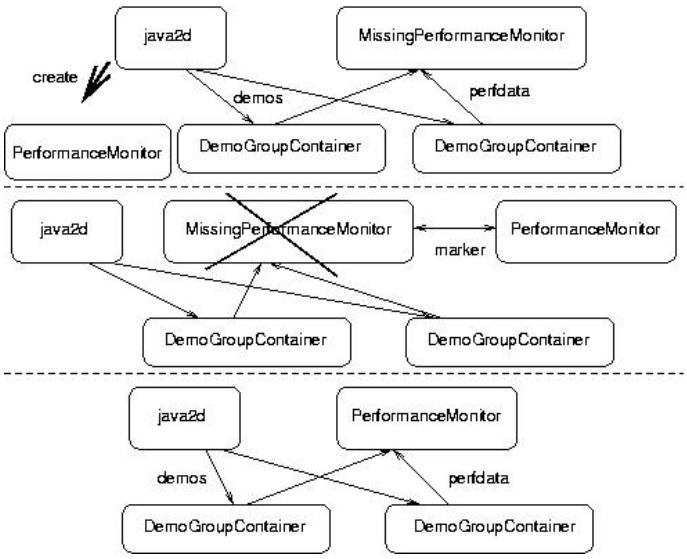


**Figure 5. Activating performance monitor**

with the real implementation and vice versa. Originally the *MissingPerformanceMonitor* component (the dummy implementation) is instantiated and connected. When the user enables the performance monitor, an instance of *PerformanceMonitor* is created. The two performance monitors sense each other's presence by means of a *marker* interface, similarly as *DemoGroupTabbedPane* instances did in the previous scenario. In this case, however, the component checks the creation time of the component connected through the *marker* interface. The creation time is maintained by the component framework. The component destroys either itself or the other component depending on which one is the older. The components are reconnected and the newly created performance monitor becomes operational.

In order to demonstrate the power of dynamic composition in hierarchical component structures, a new feature was added to Java2D. This feature allows for moving the current demo group tab to another Java2D main frame. This is realized by instantiating another *java2d* in a different domain (figure 6). having a different *appcontainer* property. The export and import filters are set up in such a way that unconnected demo group containers are moved into this second domain and get connected with the second *java2d* instance. To do this, it is enough to modify the *appcontainer* superproperty of the container component holding all the components of a demo group (demo group and demo components) and request reconnection. By the time the component network is started, the entire demo group component set is moved into the second domain and is connected to the second *java2d* instance.

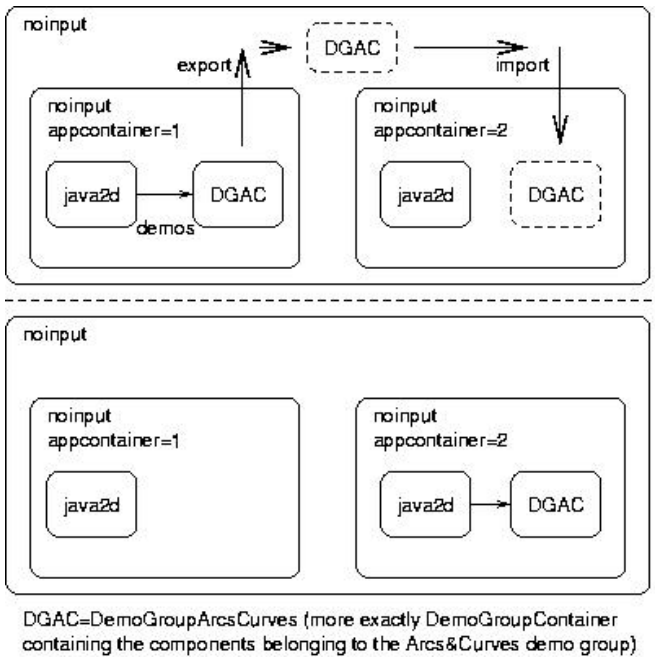


**Figure 6. Moving component groups in hierarchy**

## 4. Implementation and results

Dynacomp was implemented on top of Fractal 2.0-3 specification using the Julia 2.1 implementation. Standard 1.4.2 version of the JRE was used for benchmark measurements on an operating system based on Linux 2.4.20 kernel. The test machine was a 600MHz Intel Pentium II machine equipped with 256MByte RAM. The prototype implementation can be downloaded from http://javasite.bme.hu/ ∼paller/ common/ dynacomp.tar.gz.

The Dynacomp framework implementation uses Julia's mixin-based controller architecture thus it depends on the Julia implementation. Components are described in XML file format which is not based on Fractal ADL. In Dynacomp, several functionalities of Fractal ADL (e.g. interface connections) are handled by the components themselves. In addition, components have more meta-information compared to Fractal. It seemed simpler to devise a separate format for the purpose of prototype implementation. This was, however, a convenience decision not based on deeper considerations regarding the Fractal ADL.

The Java2D demo needed a serious amount of refactoring because the original software was quite monolithic - for example many objects were connected through object static fields. Two relatively minor features from the original software (cloning and printing) were left out of the Dynacomp version which in turn has two additional features (second main frame and remote monitoring).

| Demo | Orig (kB) | DC (kB) | DCall (kB) |
|---|---|---|---|
| Arcs&Curves | 6135 | 3675 | 7305 |
| Clipping | 6081 | 4130 | 7302 |
| Colors | 6339 | 3904 | 7617 |
| Composite | 6309 | 4149 | 7854 |

**Table 1. RAM footprint of individual demos**

The total class file size of Java2D's original version is 387964 bytes, the size of the Dynacomp version is 376769 bytes. The sizes are comparable, despite that the Dynacomp version contains code for component binding and dynamic connection handling. The Dynacomp framework (including Fractal API and Julia) is 458825 bytes (397729 bytes from Fractal API+Julia and 61096 bytes from the dynamic reconnection facility) but the framework code can be shared by multiple applications. Note that Julia is the Fractal reference implementation optimized for flexibility rather than class file size footprint.

The Dynacomp version of Java2D can run with less RAM because features can be switched off selectively, and the user "pays" only for the features he uses. Java2D is written in such a way that its RAM footprint depends on the active demo tab. In table 1 the RAM footprint of selected demos are shown. In the case of the original version, the entire application is loaded into the memory (Orig column). The DC column shows the case when only the selected demo group, its demos and the memory monitor feature were activated in the Dynacomp version. The DCall column shows the scenario when all the demo groups, demos, controls and monitors are activated in the Dynacomp version. Differences between the Orig and DCall columns relate to the memory footprint of the Dynacomp framework.

The total size of active objects is shown in the table made by Java2D's own memory monitor based on the freeMemory() and totalMemory() methods of the java. lang. Runtime class thus the size of allocated objects (active plus waiting for garbage collection) are shown. As the memory usage fluctuates with the characteristic "saw tooth" pattern of garbage-collected systems, the maximum size is presented.

The numbers demonstrate that it is indeed possible to save memory by loading only features the user uses. It also confirms the belief that the flexibility of the Julia implementation comes with significant footprint cost. Note that although every effort was made to keep the original and refactored Java2D implementations semantically identical, there were high amount of code changes which may also be responsible for some memory footprint differences.

Table 2 shows the time needed for certain reconfiguration tasks. This is the time needed for the entire recompo-

| Activity | Time (msec) |
|---|---|
| Inserting an empty demo group as first demo group | 361 |
| Inserting a demo into an empty demo group | 736 |
| Inserting a demo as the 4th demo in the group | 471 |
| Inserting 4 demo groups with a total of 16 demos | 6007 |
| Switching from grid to tabbed view | 418 |
| Switching from tabbed to grid view | 497 |

**Table 2. Recomposition time in different scenarios**

sition process including component stopping, reconfiguration, component creation, component startup, etc. thus it measures well the "blackout period", the time when the application is not available due to recomposition.

The measurements show that the dynamic recomposition in a hierarchical arrangement is able to provide low blackout times for end-user applications. The blackout effect exists, however, and may be a serious obstacle in some high-availability applications.

## 5. Conclusions

Dynamic adaptation of the application and middleware architectures is a promising approach to handle changing environmental conditions. Component models provide an attractive solution to the adaptation problem. In order to realize the adaptation by means of the components, the component framework must be able to react to the environmental changes and to establish a new component network by reconnecting components, creating new components and removing old components.

Former approaches used centralized reconfiguration managers, which are hard to analyze and extend. This paper proposes a more distributed approach when a general reconfiguration manager is able to do the adaptation based on the meta-information exposed by the components and own component manipulation activities of the components' code. The components themselves can also participate in the manipulation of the component network. This results in a similar adaptation logic found in biological systems where injecting new components is enough to change the behavior of the system. The approach is related to autonomous computing, self-healing systems and reflective middleware initiatives.

An implementation of the idea was presented. The prototype was built on the top of the Fractal framework. Many ideas were taken from earlier Service Binder work. This paper extends the meta-information presented in Sevice Binder and its reconnection algorithm. The resulting framework is able to provide automatic reconnection in component hierarchies.

The analysis of the prototype proved that it is indeed possible to save memory by componentizing the application and instantiating only the components the user actually uses. The blackout time (reconfiguration time when the application is not available) was found to be low in general cases. The Julia implementation of the Fractal API was found quite memory-intensive.

Minimizing the blackout time, the memory footprint and addressing security issues are attractive research directions for the future.

## References


[1] E. BRUNETON AND T. COUPAYE AND J. STEFANI, Recursive and dynamic software composition with sharing, *In Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02), Malaga (Spain), June 2002.*

[2] L. CAPRA, W. EMMERICH AND C. MASCOLO, Reflective Middleware Solutions for Context-Aware Applications, *Metalevel Architectures and Separation of Crosscutting Concerns: Third International Conference, REFLECTION 2001, Kyoto, Japan, Sept. 2001*

[3] O. LAYAÏDA AND D. HAGIMONT, Designing Self-Adaptive Multimedia Applications through Hierarchical Reconfiguration, *5th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), Athens, Greece, June 2005*

[4] E. M. DASHOFY, A. VAN DER HOEK AND I. R. N. TAYLOR, Towards architecture-based self-healing systems, *Proceedings of the first workshop on Self-healing systems, Charleston, South Carolina, Nov. 2002*

[5] W. CAZZOLA, A. GHONEIM AND G. SAAKE, RAMSES: a Reflective Middleware for Software Evolution, *ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution, Oslo, Norway, June 2004*

[6] G. S. BLAIR, G. COULSON, L. BLAIR, H. DURAN-LIMON, P. GRACE AND R. MOREIRA, Reflection, Self-Awareness and Self-Healing in OpenORB, *Proceedings of the first workshop on Self-healing systems, Charleston, South Carolina, Nov. 2002*

[7] F. KON, T. YAMANE, C. K. HESS, R. H. CAMPBELL AND M. D. MICKUNAS, Dynamic Resource Management and Automatic Configuration of Distributed Com-

ponent Systems, *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems, San Antonio, Texas, USA, Feb. 2001*

[8] J. DOWLING AND V. CAHILL, Dynamic Software Evolution and The K-Component Model, *Workshop on Software Evolution, OOPSLA, Tampa, Florida, USA, Oct. 2001*

[9] J. DOWLING AND V. CAHILL, Self-Managed Decentralised Systems using K-Components and Collaborative Reinforcement Learning, *Proceedings of the Workshop on Self-Managed Systems, Newport Beach,California, USA, Oct. 2004*

[10] P. INVERARDI, A.L. WOLF, Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model, *IEEE Transactions on Software Engineering, volume 21, no. 4, 1995*

[11] I. GEORGIADIS, J. MAGEE AND JEFF KRAMER, Self-Organising Software Architectures for Distributed Systems, *Proceedings of the first workshop on Self-healing systems, Charleston, South Carolina, USA, 2002*

[12] G. PALLER, Increasing Java Performance in Memory-Constrained Environments Using Explicit Memory Deallocation, *International Workshop on Mobility Aware Computing, Erfurt, Germany, Sept. 2005*

[13] H. CERVANTES AND R. S. HALL, Automating Service Dependency Management in a Service-Oriented Component Model, *ICSE CBSE6 Workshop, 2003*

[14] H. RAJAN AND K. J. SULLIVAN, Classpects: Unifying Aspect- and Object-Oriented Language Design, *27th International Conference on Software Engineering (ICSE'05), St. Louis, USA, May 2005*

[15] P.-C. DAVID AND T. LEDOUX, Towards a Framework for Self-Adaptive Component-Based Applications, *4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), Paris, France, Nov. 2003*

[16] N. PESSEMIER, L. SEINTURIER AND LAURENCE DUCHIEN, Components, ADL & AOP: Towards a Common Approach, *ECOOP'2004, Workshop on Reflection, AOP and Meta-Data for Software Evolution Oslo, Norway, June 2004*

[17] T. C. HU, Parallel Sequencing and Assembly Line Problems, *Operations Research 9 (6), 1961*

[18] W. CAN, L. YANG AND B. JIANJUN, A biological formal architecture of self-healing system, *IEEE International Conference on Systems, Man and Cybernetics, The Hague, The Netherlands, Oct. 2004*

# Appendix

One can feel intuitively that the number of connections consequently the execution time of the reconnection algorithm can be decreased if components are arranged hierarchically instead of a "flat" configuration. In this section an approximation of the exact solution is presented.

The execution time of the reconnection algorithm will be approximated by the number of connections among components. The number of connections depends on components in the component network. For example it can occur that no connection can be made among the components. For this reason the worst-case scenario, when every component can be connected to every other component unidirectionally, will be considered. This means that if two components are connected in one direction, they will not be connected in the other direction as it would create a loop in the dependency graph. Let $N$ be the number of components in the component network, $C$ the set of components and $c_i \in C, 1 \leq i \leq N$ be the components. Let us assume a fully connected component network where every $c_i$ is connected with every $c_n \in C, i < n \leq N$. As the connections always point from the components with lower index to ones with higher index, there is no loop in this graph. It is easy to see that the total number of connections is

$$\frac{N-1}{2}N$$

Now let us assume that the components are arranged into a two-level hierarchy where the components are partitioned into $0 < g \leq N$ groups, each having $n_g = \frac{N}{g}$ components (the components are distributed evenly among the groups). Connections are made at two levels: inside the groups and among the groups. This yields the total number of connections in this hierarchy as:

$$\frac{n_g - 1}{2} n_g g + \frac{g-1}{2} g = \frac{N^2}{2g} - \frac{N}{2} + \frac{g^2}{2} - \frac{g}{2}$$

We would like to prove that

$$\frac{N-1}{2}N > \frac{N^2}{2g} - \frac{N}{2} + \frac{g^2}{2} - \frac{g}{2}$$

This yields $N^2 > g^2, g > 1$ which is true if $1 < g < N$. The number of connections is thus lower in this hierarchical arrangement for any non-trivial grouping (so that groups consist of more than one component and there are more than one groups).