Runtime Modularity in Complex Structures: A Component Model for Fine Grained Runtime Adaptation

Barry Porter
School of Computing and Communications
Lancaster University
Lancaster, UK
b.f.porter@lancaster.ac.uk

ABSTRACT

Online modular adaptation and self-adaptation techniques have demonstrated significant benefits in coarse-grained software, enabling agile and high-performance deployments. We are studying the same kinds of runtime adaptation applied to fine-grained software such as graphical user interfaces and web server implementations. However, this kind of software is defined by pervasive use of *behaviourally-driven structure*. Existing runtime component models fail to capture this necessity due to their exclusive reliance on externally-driven structural composition. In this paper we present a novel runtime component model that both satisfies the need to externally manage software structure, enabling runtime adaptation and self-adaptation, while also satisfying the need for fine-grained software to create elements of its own structure based on application-specific system behaviour. We present the key details of our model along with an initial evaluation.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**] Coding Tools and Techniques – *Component models*

1. INTRODUCTION

Online modular adaptation and self-adaptation methods, in which software can evolve its composition at runtime, have demonstrated significant benefits in coarse-grained software [4, 13, 8, 16, 12]. In detail, these approaches propose that software continually observes its external environment, while also measuring its own behaviour and performance, to constantly adapt itself towards optimal compositions.

We are studying the use of online component-based adaptation and self-adaptation in *fine-grained* software systems. By fine-grained we refer to software such as graphical user interfaces, web server implementations, and database implementations. We are interested in decomposing this kind of software into its smallest re-usable units and making those units into strongly separated components, the composition of which we can then reason about and adapt at runtime. By doing this we are able to investigate similar levels of performance enhancement, agility, and complexity management to that seen in the application of runtime adaptation to coarser-grained modular software as mentioned above.

The current crop of runtime component models, however, fails to apply to fine-grained software. In particular, all runtime component models that we are aware of strictly enforce and manipulate software structure *externally to* the components that make up a system's behaviour [6]. The composition of system structure is therefore the exclusive domain of a 'meta-level' or 'composition level' which lies outside the application-specific behaviour of the software system. This approach works well at coarse software granularity and indeed is an important enabler of modular runtime adaptation.

Unfortunately, at fine software granularity, it is very difficult to design systems whose structures are entirely defined by such an external entity. This is because (i) 'instances' must often be created *as a direct result of* system behaviour; and (ii) these instances must interact with each other using (correspondingly) *behaviourally-driven structure*. Examples of behaviour-based instance cardinality include populating a window with graphical widgets such as file or directory entries in a file browser; and instance-per-client architectures in network server implementations. Example inter-instance structures include observer patterns in event-based logic or polymorphic iterators in centralised control dispatch logic.

In this paper we present a novel runtime component model that both satisfies the need to externally manage software structure, supporting runtime software adaptation and self-adaptation, while also satisfying the need for fine-grained software to create elements of its *own* structure based on its application-specific system behaviour. Our model links these two distinct structural levels (externally-imposed and internally-driven) together into a fully generalised runtime adaptable paradigm for fine-grained software.

In detail, we propose a runtime component model with:

1. A traditional **macro-level** of strongly-separated software components. Each component exports provided and required interfaces that are wired / re-wired by an external meta-level to drive runtime adaptation.
2. A novel **micro-level** in which components freely create *micro-component* instances from each required interface based on their application-specific behaviour.
3. A concept of **gateways** to support *referential persistence* for micro-components, linking freeform micro-structure to meta-level adaptation of macro-structure.

In the remainder of this paper we first present our component model in Sec. 2. We then present an initial evaluation of the key aspects of our model in Sec. 3, followed by a survey of the literature in Sec. 4. We offer conclusions in Sec. 5.

CBSE'14, June 30–July 4, 2014, Marcq-en-Baroeul, France.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2577-6/14/06 ...$15.00.
http://dx.doi.org/10.1145/2602458.2602471.

2. THE DANA COMPONENT MODEL

Our component model is realised within a full-featured, purpose-built programming language called *Dana* [1]. In the majority of this paper we focus on the runtime component model on which Dana is founded, enabling runtime adaptation in fine-grained software structures. At a language level all functionality in Dana is expressed within strongly-separated components that are independently deployable. It is an imperative, procedural, interpreted language, is multi-threaded, and features only interface, record and primitive types. It is syntactically similar to contemporary languages like Java and its source code library includes over 100 components from networking and data manipulation to graphics and system-level APIs. In the remainder of this section we first describe the Dana component model in detail, then we provide an example of its use in fine-grained structures, and finally we describe its runtime adaptation mechanics.

2.1 Component model details

A software component in our model is defined as a unit of functionality that expresses one or more 'provided' interfaces and zero or more 'required' interfaces. An interface is a list of function prototypes and each interface type may inherit from one other interface type. Interfaces that do not explicitly inherit from another type automatically inherit from a common base type which includes the functions `equal()`, `clone()` and `toString()`. Each component may only use external functionality via its required interfaces.

A system is constructed from a composition of components, created by loading each desired component into memory and interconnecting each required interface of each component to a type-compatible provided interface on another component. This composition is performed by a third-party 'meta-program' outside the behaviour of the components that make up the system itself. Aside from interface inheritance, this core design is similar to classic runtime component models (e.g. [3, 5]) and enables a meta-program to compose systems from the desired units of functionality and later adapt those systems online to meet changing needs.

Beyond these fundamentals however our component model diverges significantly from its contemporaries as follows.

Firstly, we introduce the concept of a 'micro-component'. In detail, each provided interface that a component advertises actually represents an *instantiable micro-component* (or 'microcom'). Each such microcom has one primary interface, which maps to a component's advertised provided interface as described above, and zero or more secondary interfaces. Each microcom may also have its own per-instance state. These structural elements are illustrated in Fig. 1. Secondary interfaces, and any per-instance state, used by a microcom are a part of its implementation detail. Two components can thus provide the same interface type which internally maps to two different microcom implementations with different secondary interfaces and different instance state. The syntax with which components declare provided interfaces is exemplified in Fig. 3(a); this component provides an instantiable microcom with a primary interface of type `FileBrowser` (i.e. a provided interface of the component) and a secondary interface of type `ClickListener`.

Symmetrically, each required interface that a component declares in fact therefore represents an instantiable microcom sourced from the component to which that required interface is currently connected.

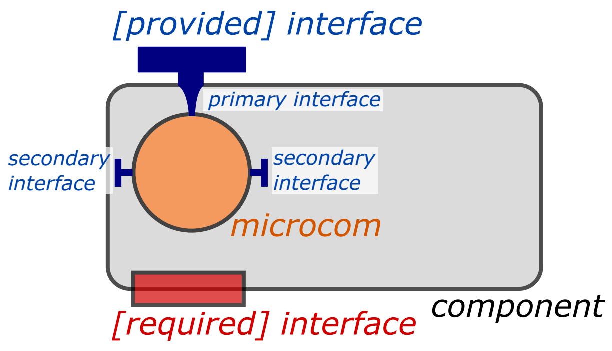


Figure 1: The main structural elements of our component model. Secondary interfaces of microcoms, and any per-instance state fields, are internal implementation details that are not outwardly visible.

Microcoms can be instantiated in a completely freeform manner (such as instantiating many graphical, clickable file representations) according to the particular behaviour of a given component. Furthermore, references to microcoms (specifically to one of their interfaces) can be handed around to other microcoms in a similarly freeform fashion by passing them as parameters to functions. Complete micro-structures (such as a observer patterns or polymorphic collections) of microcom instances and inter-instance references can thus be formed as a result of system behaviour.

The second novel feature of our model is then the way in which macro-structure (components and inter-component wiring) is linked to micro-structure (microcoms and inter-microcom reference graphs) in the face of externally-driven runtime adaptations in the former.

This is achieved with the concept of *gateways*. In detail, whenever a component C uses one of its required interfaces to create a new microcom in another component, a gateway is created in C which serves as an indirection to that microcom. A gateway adopts the interface type of the corresponding required interface (which is respectively the provided interface type to which it is connected, and is by extension the primary interface of the microcom) and so serves as a gateway to a microcom whose primary interface is of that type. Whenever a component C hands out a reference to a microcom that it created, it actually hands out a reference to its gateway for that microcom. When a required interface of C is being re-wired to a different provider component, all of C's gateways from that required interface stay in place, while the microcoms behind it are switched to instances from the new provider component. We return to runtime adaptation mechanics in more detail in Sec. 2.3.

At the macro level our components exhibit all the features of traditional runtime component models [15, 10]. They are independently deployable, using only explicitly-declared dependencies, making them highly re-usable and their compositions highly re-configurable. They also offer the strong discretisation of software that makes component-based compositions particularly conducive to introspection, self-analysis and self-adaptation [10]. At the micro-level, meanwhile, components can freely create microcom instances and can freely pass around references to them, knowing that these instances and inter-instance reference graphs will remain in place despite implementation changes caused by runtime adaptation at the macro level. In the next section we use an example GUI system to demonstrate these concepts.

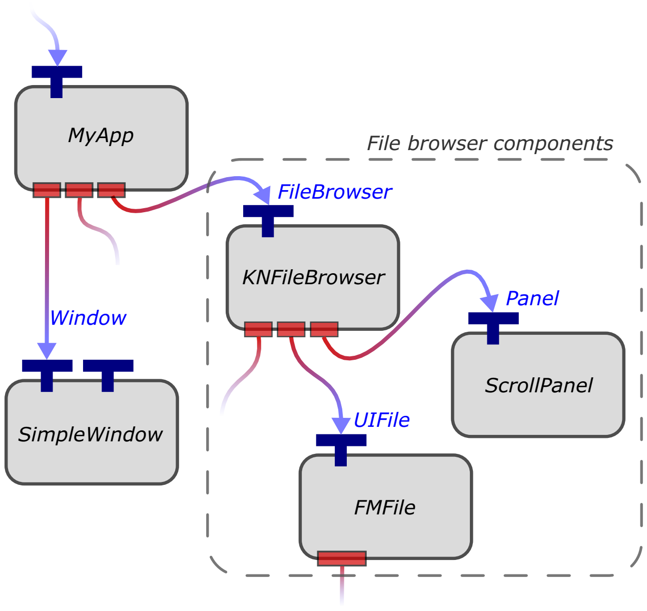


Figure 2: Component architecture of our example system. As shown in Fig. 1, note that an instantiable microcom implementation lies behind each provided interface (not shown here to simplify the diagram).

2.2 Example system

To better illustrate our component model we now present a simple example using part of a file browser dialog. This is taken from one of our graphical user interface systems and demonstrates both behaviourally-driven microcom instance populations and inter-instance reference graphs (via an observer pattern and a polymorphic collection).

The functionality of the file browser is simply to be given a directory D and to display each directory and file that D contains. The user can click on a displayed directory D' to move into it, thereby showing the contents of D' instead.

At the macro level the file browser is composed of a file browser component, a panel component which arranges the file display artefacts, and a graphical file representation component which provides the visual appearance of files and directories. The composition can be reconfigured online to change the way in which files are displayed (as a simple scrolling list or as a variety of alternatives) depending on user preferences, accessibility considerations, and the number and types of files in a particular directory.

The macro architecture is illustrated in Fig. 2 and source code extracts of the components are shown in Fig. 3.

Within the outer macro level is a complex, behaviourally-driven population of microcom instances and inter-instance references forming a dynamic micro-structure. Aided by the use of gateways as detailed earlier, this micro-structure stays in place despite runtime adaptations affecting the macro-structure when required interfaces are re-wired to alternative components. These forms of micro-structure pervade the kinds of fine-grained software that we target and demonstrate the novel abilities of our component model.

For example, in Fig. 3(a) we see a function `setDir()` which needs to instantiate many 'UIFile' microcoms (line 8), each representing one directory or file in the current directory. Similar patterns are found in other kinds of systems software such as instance-per-client patterns and worker pools.

```
 1  component provides FileBrowser(ClickListener)
              requires UIFile, Panel, FileSystem fs{
 2     Panel myPanel;
 3
 4     void FileBrowser:setDir(char path[]){
 5        myPanel.clear();
 6        FileEntry files[] = fs.getFiles(path);
 7        for (int i; i < files.arrayLength; i++){
 8           UIFile nf = new UIFile(files[i].path);
 9           nf.addListener(this);
10           myPanel.addObject(nf);
11           }
12        }
13
14     void ClickListener:click(Object o){
15        UIFile f = o;
16        if (f.isDirectory())
17           setDir(f.getPath());
18        }
19     }                                     KNFileBrowser
```

(a)

```
 1  component provides Panel{
 2     GraphicsObject objects[];
 3
 4     void Panel:addObject(GraphicsObject go){
 5        objects += go;
 6        /* ... position the object ... */
 7        }
 8
 9     void Panel:paint(Canvas c){
10        c.drawRect(...);
11        /* ... now draw my objects ... */
12        }
13     }                                       ScrollPanel
```

(b)

Figure 3: Source code extracts from our example system. We do not have space to show the interfaces used here; note however that the `UIFile` interface for example inherits from `ClickableObject` which in turn inherits from `GraphicsObject`. All source code from this example system is available online [2].

The same function drives two other kinds of common micro-structure in the form of an observer pattern and a polymorphic collection. The observer pattern is used on line 9 of Fig. 3(a). Here the `FileBrowser` microcom registers itself, using its internal secondary interface of type `ClickListener`, as a callback target with each `UIFile` microcom. Similar fine-grained software patterns include system up-calls and dynamic event-condition-action logic.

The polymorphic collection is used on line 10 of Fig. 3(a). Here the 'panel' microcom is provided with a reference to each file microcom created by the file browser. As shown in Fig. 3(b) (line 4) the panel component accepts as a parameter to this function any microcom with an interface that is a subtype of `GraphicsObject`. Key to our model is that even if the implementation of the UIFile microcoms is changed at runtime, gateways ensure that all references to those microcoms held by the panel *will still be valid* (and will have been automatically routed to equivalent microcoms of the new implementation). Similar polymorphic collection patterns are found in other kinds of systems software like queueing modules and centrally-orchestrated task execution managers.

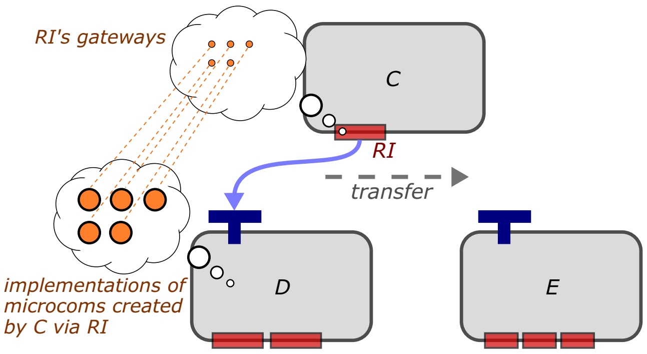
2.3 Runtime adaptation mechanics

Runtime adaptation allows us to adjust software to its current deployment environment by selecting between different components that are judged to be best suited to current conditions. *Self*-adaptation occurs when a software system analyses its own behaviour and performance and uses runtime adaptation to optimise that performance without human intervention. This approach fosters agile and high-performance systems in changing environments such as variable client workloads or fluctuations in system resources.

In this section we present Dana's runtime adaptation mechanics. Runtime adaptation in general has two important elements: the mechanism(s) by which adaptation is achieved; and the way in which continuity of overall system integrity (or state) is maintained when adaptation occurs.

2.3.1 *Adaptation mechanics*

The single mechanism by which Dana supports runtime adaptation is by allowing a meta program to re-wire a selected required interface of a component C from its current resolution against component D to instead be resolved against a different component E. This is illustrated below. Wider architectural change occurs when component E itself has a different set of required interfaces than D. Component E could for example be a completely different implementation of D, implying different behaviour and performance characteristics; or could be an interceptor component that is actually itself connected to D through one if its own required interfaces and which simply performs some pre-processing before calls arrive at D; or could even be acting as a proxy to a remote version of D running on a different host.



Unlike contemporary runtime component models, Dana uses each required interface as a source of microcom instances, created and destroyed as needed by the component declaring that required interface. Many microcom instances sourced from that interface may thus be active at any time.

The runtime adaptation procedure therefore works in the following stages. First, we isolate (or 'pause') the required interface, stopping further interaction through it and so preventing further externally-driven state transitions in the associated microcoms. This works simply by waiting for any existing function calls passing through a required interface to complete, and holding any new function calls at the required interface boundary. Once paused we then iterate over *each* microcom that is active over that required interface, sourced from component D, and transfer each such microcom to instead be sourced from component E (including state transfer). Once this is complete the selected required interface is finally 'resumed', re-enabling interaction through it (and allowing any calls held at the required interface to proceed).

```
void UIFile:clone(Object o) {
   UIFile prev = o;  UIFile(prev.getFile());
   Point p = prev.getPosition();
   setPosition(p.x, p.y);
   listeners = prev.getClickListeners();
   }
```

2.3.2 *System integrity across adaptation*

The way that we maintain continuity of overall system integrity (or state) across adaptations is then as follows. As part of the process of iterating over each microcom active over a required interface, we perform *state transfer* to microcom instances from the new implementing component. In detail, for each microcom instance i active over RI of component C we use Dana's 'transfer' procedure to migrate $microcom_i$ to its new implementation in component E. This transfer procedure first creates a new microcom instance using the corresponding provided interface of component E and then invokes the `clone()` function (from the common interface base type mentioned in Sec. 2.1) on that new microcom, passing in a reference to the existing microcom instance sourced from component D. From within its `clone()` function, the new microcom instance sourced from component E is then able to use the functions from the microcom's *primary interface* (which both the existing and new microcom must necessarily have in common) to manually extract any necessary state from the existing instance (i.e. using 'getter' functions) and apply that state to itself.

It is important to note that the actual implementation of `clone()` functions is left entirely to the programmer to write as appropriate, including designing necessary 'getter' functions in interfaces. An example is shown above. We are not therefore concerned with any semantic or structural issues of inter-component state compatibility that occur with many automated or transparent state transfer mechanisms.

Finally, as detailed in Sec. 2.1, our component model uses *gateways* to provide sufficient indirection to ensure that any *references* to microcoms that are migrated to a new implementation remain valid despite the fact that their implementation (and internal state representation) has changed. After each state transfer operation, the Dana runtime thus completes the overall transfer procedure by updating the gateway for that particular microcom instance so that it points at the new instance in the new component (component E). It is this mechanism that links freeform micro-structure to externally-controlled and adapted macro-structure.

In detail, recall that whenever a microcom is instantiated, a gateway for that microcom is made in its creator. Any references to a microcom that are then handed out by its creator are actually references to that gateway, which lies in the creator's internal state rather than in the implementing component of the microcom. As described above, these gateways stay in place despite changes to the component that is implementing the microcoms behind those gateways.

Not *all* forms of microcom reference enjoy persistence across adaptation, however. In detail we say that microcom references that are handed out *by the creator* of a microcom will persist across evolutions of that microcom's implementation, since the creator is not the subject of the evolution (i.e. it is staying in the system) and as such any effects of its implementation logic should persist. This applies for example to the references to `UIFile` instances (via `nf`) that are handed out to the `Panel` instance on line 10 of Fig. 3(a).

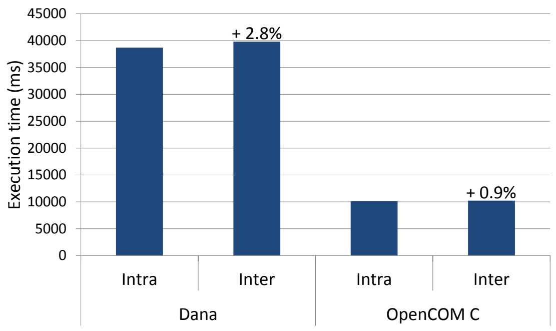


Figure 4: Differences in execution speed for 1 million calls when using internal component (local/intra) calls versus inter-component calls.

In contrast, microcom references that are handed out *by a microcom instance itself* (i.e. self-references derived using the 'this' notation) do not persist across runtime evolutions of that microcom's implementation. Instead such references are removed from the system after transfer is complete. This is done to avoid the need to make any assumptions between two different components implementing the same interface. Specifically, the incoming implementation can safely assume that it is starting from a 'clean slate' when state is transferred into microcom instances that it implements. It can then choose when and to whom any self-references should be handed out according only to the logic of its own implementation, safe in the knowledge that no other self-passed references to its microcoms are left over from the logic performed by other implementations. In this way components can be designed as black boxes and will operate in a consistent way with respect to runtime adaptation procedures.

3. INITIAL EVALUATION

In this section we provide an initial evaluation of the main elements of our component model's baseline performance characteristics. Our aim is to demonstrate a capable level of performance primarily in terms of execution speed (we consider program size less important for modern computers).

The main factor involved is the extra level of indirection that gateways create for any references used by, and given out by, the creators of microcoms. In addition to this there is a minor added overhead in making inter-component calls due to the need for our runtime to check whether or not a required interface is currently paused. All experiments were conducted on a 1.8Ghz Intel Atom Z2760 with Windows 8.1.

Fig. 4 shows the differences in call speed between local function calls (within a component) and inter-component calls. This experiment was performed both with Dana and, as a point of comparison, for the C implementation of OpenCOM [5], a traditional runtime component model that does not support fine-grained sub-structure. The results demonstrate the expected performance delta between local and inter-component calls. As noted above, we expect that the difference is greater in Dana due to the extra gateway indirection along with checks on required interface paused status. OpenCOM, by comparison, does not use gateway-like mechanics because it lacks support for fine-grained behaviourally driven structures, and has no in-built 'pause' feature to aid with seamless runtime adaptation.

Examining the ratio of local to inter-component microcom calls, our simple GUI file browser example performs 2164 local calls during startup and 25999 inter-component microcom calls; 92.3% of all calls in this example are therefore microcom calls. Such a ratio is expected due to the finer granularity of component in our model, though in more complex systems we might tend to find lower ratios when individual functions carry out more work. Inter-component calls in this kind of model may therefore be a particular target for some form of just-in-time compilation.

On a qualitative level, our graphical user interface systems demonstrate sufficiently high levels of execution speed to provide an interactive user experience. The above overheads do not therefore appear to noticeably harm overall system performance impressions on modern commodity hardware.

In future we expect the performance figures reported here to improve further in successive iterations of our implementation. In future work we also intend to perform a more extensive evaluation in multiple different kinds of example system to provide a more complete picture.

4. RELATED WORK

In this section we examine related work in three categories. First, in Sec. 4.1, we discuss existing runtime component models. In Sec. 4.2 we then examine runtime update approaches in the Java platform in particular. Finally in Sec. 4.3 we discuss the rationale for using a new language.

4.1 Runtime component models

All existing runtime component models that we are aware of lack support for the behaviourally-driven sub-structural relationships that we have described throughout this paper and which pervade fine-grained software. Examples of such component models include [5, 11, 3]; see [6] for a recent survey of work in this field. Specifically, these models focus on wiring components together as instances and have no feature for those components to then go on to create sub-structural constructs as a factor of their behaviour (and pass sub-structural references around to create entire micro-structural graphs driven by that behaviour).

Because these behavioural factors are far too application-specific for a runtime meta-model to describe, these models are therefore limited either to relatively simple software structures or else tend towards coarse-grained components.

Our aim to apply runtime component-based adaptation to finer-grained, structurally complex systems functionality therefore required a rethink of the component paradigm. This has resulted in the model described in this paper in which traditional macro-level components can create micro-components on demand as a factor of their behaviour.

4.2 Runtime update approaches in Java

The popularity of the Java language platform has created significant interest in approaches to increase the reconfigurability of Java programs. There are two main bodies of work in this area: component models like OSGi and EJB, and runtime update approaches like Javeleon and JRebel.

The component models were designed with very different requirements to ours and so work in correspondingly different ways. EJB is a server-side technology for client/server interactions with the intention of separating 'container' concerns (like security and access/interaction technology) from functional behaviour. Components in EJB sit in tightly defined categories such as 'stateful' and 'message-driven' and have corresponding constraints on their interactions. It is not therefore a general-purpose system building technology. OSGi, meanwhile, is primarily designed to support plug-in architectures. It works at a coarse granularity by arranging Java classes into jar 'bundles' and using the concept of a 'bundle scope' – a private classloader scope which hides internal classes of the bundle aside from selected exceptions that form the exported services of a component. While the amount of inter-bundle micro-structure is unbounded (as all of the usual Java features are available) there is no link from this micro-structure to a meta-level controlled macro-structure. This effectively prevents post-initialisation online architecture adaptation over a system's execution.

The runtime update approaches (like JRebel [9], Javeleon [7] and JavAdaptor [14]) are designed to transparently support dynamic differential patching of Java programs while they execute. As input they take the current system architecture (often examining its source code) along with the updated system architecture's source code and calculate a way to transform the former to the latter without taking the program offline. These approaches work for some difference patterns but will fail if the scale of difference is too great, for example with major changes to the internal member fields of a particular class. In addition to these limitations, the main drawback of these approaches in terms of our requirements is that they are designed only for administratively-instructed updates. This makes them unsuitable for self-adaptation approaches which require strong discretisation of software into 'components' and 'connections' as first-class concepts [10].

4.3 Language comparison

Developing a new language as part of a component model is a larger undertaking than developing a language-targeted runtime component model of the kind mentioned in Sec. 4.1. We took this route to provide a clean and simple programming model and also to avoid some of the pitfalls of using popular languages like C and Java. For C, these limitations include a lack of desirable constructs like 'instances' and 'interfaces' as well as the difficulty of intercepting multi-threaded inter-component interactions when runtime adaptation takes place. Other drawbacks include robustness (one faulty component can bring down the entire system) and multi-platform portability / interaction. In Java, while many drawbacks of C are improved, the introduction of typed classes makes fully generalised runtime loading and unloading of classes very difficult (see [7, 14]), and the unconstrained mixing of data and behaviour in 'objects' significantly hinders well-structured runtime adaptation.

5. CONCLUSION AND OUTLOOK

In this paper a novel runtime component model is proposed that supports the expression of behaviourally-driven micro-structural relationships. This is achieved by introducing the concept of a 'micro-component' for freeform instantiation and inter-instance referencing. At the same time our model supports unconstrained adaptation of the macro-level component graph and uses a new concept of 'gateways' to link the micro- and macro-levels together across adaptation.

Our model is highly adept at implementing finer-grained software in which this kind of micro-structure is pervasive. This in turn supports our study of runtime adaptation and self-adaptation in such systems to examine the performance improvements and higher-level reasoning that can be gained.

In future work we intend to provide a more extensive evaluation, both quantitative and qualitative, of our component model, as well as continue to explore the benefits of runtime adaptation in fine-grained software such as web server implementations, databases and graphical user interfaces.

6. REFERENCES

[1] Dana language: http://www.projectdana.com/.

[2] Demos and code from this paper with instructions: http://research.projectdana.com/cbse2014porter.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. An open component model and its support in java. In *Component-Based Software Engineering*, volume 3054 of *LNCS*, pages 7–22. Springer Berlin Heidelberg, 2004.

[4] S. Chaitanya, D. Vijayakumar, B. Urgaonkar, and A. Sivasubramaniam. Middleware for a re-configurable distributed archival store based on secret sharing. In *Proceedings of the ACM 11th International Conference on Middleware*, Middleware '10, pages 107–127, Berlin, Heidelberg, 2010. Springer-Verlag.

[5] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Trans. on Comp. Systems*, 26(1):1:1–1:42, Mar. 2008.

[6] I. Crnković, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron. A classification framework for software component models. *Software Engineering, IEEE Transactions on*, 37(5):593–615, 2011.

[7] A. R. Gregersen and B. N. Jørgensen. Dynamic update of java applications: Balancing change flexibility vs programming transparency. *Journal of Software Maintenance and Evolution*, 21(2):81–112, Mar. 2009.

[8] D. Hughes, P. Greenwood, G. Blair, G. Coulson, P. Grace, F. Pappenberger, P. Smith, and K. Beven. An experiment with reflective middleware to support grid-based flood monitoring. *Concurrency and Compututation: Practice and Experience*, 20(11):1303–1316, Aug. 2008.

[9] J. Kabanov. JRebel tool demo. *Electron. Notes Theor. Comput. Sci.*, 264(4):51–57, Feb. 2011.

[10] F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, June 2002.

[11] J. Magee, J. Kramer, and M. Sloman. Constructing distributed systems in conic. *Software Engineering, IEEE Transactions on*, 15(6):663–675, 1989.

[12] D. A. Menasce, J. a. P. Sousa, S. Malek, and H. Gomaa. Qos architectural patterns for self-architecting software systems. In *Proc. of the 7th Int. Conf. on Autonomic computing*, ICAC '10, pages 195–204, New York, NY, USA, 2010. ACM.

[13] J. Philippe, N. De Palma, F. Boyer, and O. Gruber. Self-adapting service level in java enterprise edition. In *Proceedings of the 10th ACM International Conference on Middleware*, Middleware '09, pages 8:1–8:20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[14] M. Pukall, C. Kästner, W. Cazzola, S. Götz, A. Grebhahn, R. Schröter, and G. Saake. Javadaptor : flexible runtime updates of java applications. *Software, practice and experience*, 43(2):153 – 185, 02 2013.

[15] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming.* Acm Press Series. ACM Press, 2002.

[16] W. Wang and G. Huang. Pattern-driven performance optimization at runtime: experiment on JEE systems. In *Proc. of the 9th Workshop on Adaptive and Reflective Middleware*, pages 39–45. ACM, 2010.