# The Cell Project: Component Technology for the Internet

Ran Rinat Scott F. Smith
Department of Computer Science
The Johns Hopkins University
{rinat, scott}@cs.jhu.edu

June 1, 2001

## 1 Introduction

Component-based software systems are currently receiving significant attention in both industrial and research environments. One fact that is not brought out in current component architectures is that there are two different layers at which co-operating components are tied together. In the *component link* layer, the *code* that comprises the cooperating components is *linked* in a way that enables them to interact. In the *component invocation* layer, the components interact by dynamically invoking operations defined in component interfaces.

In existing technologies—including COM, Corba and Java Beans—components are primarily constructed around the invocation layer above: COM components are in fact a certain kind of object that provides services to clients via COM interfaces. Code-wise, COM objects are instances of COM classes, which ship inside a DLL or an executable file. That DLL or executable needs to be linked with the client's program before its COM classes can be instantiated, and its objects accessed. In other words, the linking mechanism is outside the scope of the component technology (COM). In addition, DLLs are binary entities created from object files. Consequently, their "interfaces" are internal computer-generated symbol tables, not programmer-declared interfaces.

*Module* systems, on the other hand, are concerned with the link layer, i.e. with the way program units developed separately may be combined in a declarative, interface-oriented fashion. Once a system has been constructed from its constituent modules, the modules themselves cease to exist: they are only code- and link-time abstractions. Because modules have no runtime existence, there can be no programmatic manipulation of modules. For example, it makes no sense to unload, or "unlink", a module, and it makes no sense to ship a module at runtime to another location.

Keeping these two layers of component interaction apart has some historical justification: before applets and electronically shipped plugins became prominent, linking functioned as "binary glue": there just had to be a way of putting together various compilation results (statically or dynamically). This apparently has nothing to do with the runtime elements (COM objects) put to life by these binaries (DLLs). However, an applet coming over the internet is much more than an innocent binary: it is a component written by someone, somewhere, that wants to enter the local domain and do its thing. As such, its identity and intentions need to be examined and authorized by the hosting environment. In addition, if instead of the Java-specific applet construct we consider the *applet metaphor* – i.e., the notion of code being copied from a remote server to be executed locally – then the ability to carry *state* (i.e. data) gains important flexibility. For instance, applets/plugins could be specialized to the target client before shipping.

All this suggests that much can be gained by treating shipped code, not the object it instantiates, as a *dynamic entity*: a stateful first-class runtime entity that can be accessed and manipulated in a systematic way via explicit interfaces—much like a COM object. While it should be possible for a given system to support two different component technologies—one module-like for the code linking purposes and one COM-like for object interaction—the similarity of concerns strongly suggests that the two layers are better off unified.

In this position paper we describe progress in constructing a new component model that combines both interface-driven code linking and service invocation in one streamlined architecture. It is based around the concept of a *cell* (the name is inspired by the biological notion of cell). Cells are units of linking and deployment that exist as first-class entities in the program runtime environment. They are *containers* of both passive code, in the form of classes and functions, and of executing stateful entities, in the form of objects. In this sense, they correspond both to DLLs and to COM objects. The cells themselves—not the objects inside—correspond to COM objects: in our model, objects are only used to implement cells. In contrast with DLLs, cells are linked via explicit interfaces. Like DLLs, but unlike modules, they can be de-linked at a later time. Linking and delinking is done programmatically at run-time via a *plugging interface.* This notion is closely related to connection-oriented programming interfaces [Szy98]. Figure 1 shows a cell containing classes and objects with a plugging interfaces ready to be linked to other cells.

On top of this fundamental unification of component layers in cells, it is possible to build a broader architecture incorporating component distribution and security. The unified notion of component gives in turn a more unified notion of distribution and security, where for instance persistent network connections can be achieved by the linking of components, and where security of linking and security of component invocation are one and not two problems.

**Distribution** Cells are designed to seamlessly support distributed computing. This is in analogy to e.g. DCOM or Java's RMI, but the distribution protocols are closely tied to the component architecture and so are cleaner. Every cell exists at some network location, and any cell in any location can in theory refer to any other cell on the network. Program linking has traditionally been a purely local

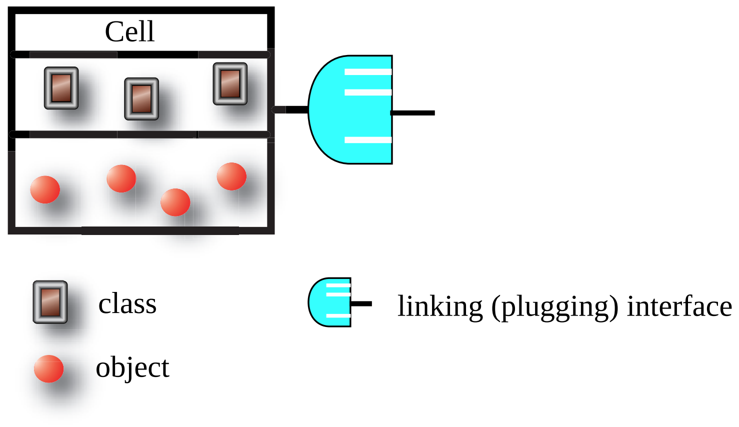

Figure 1: A cell containing classes and objects ready to link

operation, but cells can directly link *across the network*, just by applying the cell linking operation, *connect*, between cells on different nodes. Such persistent inter-node connections bear a strong analogy to the persistent connections of internet protocols such as ftp and ssh, but are at a higher level of programming abstraction. What this means is that persistent service connections, which must be performed in an ad-hoc manner in other languages, can be coded directly in the cell language.

Also within the context of distribution is the applet metaphor mentioned above: cells containing classes, functions, and (stateful) objects may be *copied* from one network location to another, to be linked and/or invoked locally. In addition, cells may *move* from one network location to another, supporting a form of roving agent.

**Security** Cells are units of security in that they are designed to be an important element in a distributed environment for which security policies can be established. Cell boundaries are walls protecting the contained classes and objects, which may be accessed only via the explicit interfaces. We propose here two simple levels of security. The key point of our proposal is how naturally it fits into the existing cell architecture. The existing security architectures in COM, Microsoft's .NET, and Java are based on external access control lists maintained in files, and are not as tightly integrated into the design.

On the first level, cell references are unforgeable, and so serve as capabilities for accessing the cell: whoever holds a reference to a cell has the capability of invoking services on any of its service interfaces. Without a reference, a cell is completely inaccessible and so is secured. Cell references sent across the network are implicitly encrypted by the cell run-time system, so a reference cannot leak unless it was leaked by an insider (assuming the cell run-time system was not itself hacked). This layer of security requires no access control lists, no stack inspection/walking, no extra syntax, and extremely little extra run-time overhead. Yet it provides a powerful layer of protection; so, we believe it is a good thing to include.

On the second level, there is a natural synergy between linking and security: in a distributed environment, linking is an operation between code units potentially written by different parties, and it is a natural place at which to impose security checks. So, at cell connection time the two cells mutually authenticate each other. In the cross-network cell connection case outlined above, the analogy between persistent internet connections and cell connections is reinforced: internet protocols such as ssh and ftp start with a security check just as cross-network cell connections start with a security check. Once the check is passed, a persistent connection is opened and generally no further security checks are needed. We believe this is a powerful analogy, and perhaps cells can be called the first unified internet programming language proposal.

Although the long-range goal of the project is to proceed with both foundations and implementation, the project now focuses on foundations. In this position paper we sketch the constructs and protocols supported by cells, via a series of examples.

## 2 The Cell Model

We illustrate the cell model in two stages: first the basic concepts, and then the distribution layer. The examples in this section are written in the cell language, which is defined along with its operational semantics and typing rules in [RS01].

### 2.1 Basic Cell Concepts

Cells interact via explicitly declared interfaces. There are two kinds of interfaces, *plugging* interfaces and *service* interfaces, through which two modes of interaction are supported:

1. Seamless interaction following linking via a *plugging* interface, a module-style connection; and,

2. Explicit interaction via a particular *service* interface, a COM-style interaction.

Two cells may interact seamlessly (to be explained in a moment) after they connect (link) via a plugging interface, see Fig. 2. A plugging interface on a cell lists a series of imports and exports, the *plugins* and *plugouts.* To interact with cell $c$ via plugging interface $I$, another cell must be *connected at* $I$. That cell must also have a dual plugging interface named $I$, in which the plugins and plugouts trade places. Once a connection between two cells is established, calls to a plugin within

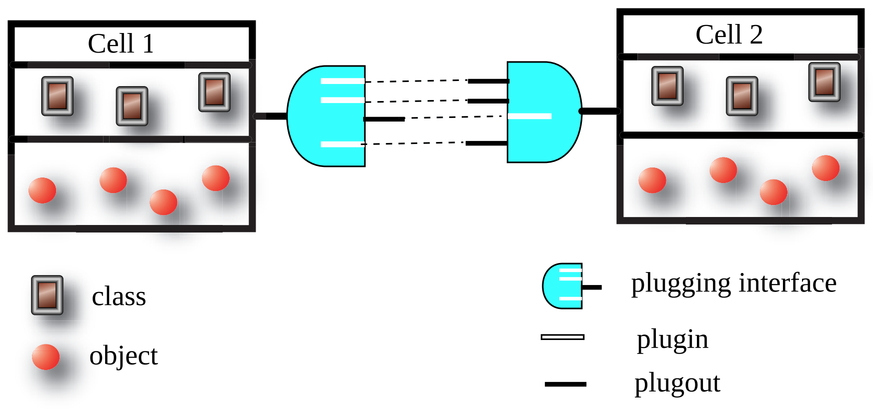

Figure 2: Linking cells via plugging interfaces

the implementation of cell $c$ invoke the implementation for the plugout by the same name in the connected cell. Thus, calls to plugins within a given cell look like a call to an external function whose implementation will be provided via linking. The interaction is therefore seamless from the point of view of the cell authors. This notion of interface matching is close to module interface matching, only plugging connections are more dynamic: two connected cells may be disconnected at a later time.

Plugins and plugouts can be either functions ("operations") or classes. Declaring a class plugin allows a class in one cell to inherit a class in another cell. This powerful feature allows, for instance, shipped code to inherit from local libraries, as often is the case with applets using a local GUI library via inheritance.

Cell interaction via plugging interfaces is as dynamic as operation invocation on COM objects, but it is peer-to-peer: the interaction is between two *cells*, not an unspecified *client* interacting with a COM *server*. Because interaction occurs between two concrete parties, it is possible to perform a nontrivial security check to verify that a mutual trust relationship holds. Cells contain a special *trust* predicate that allows arbitrary security checks to be coded. This is illustrated below.

The second mode of interaction is via service interfaces. A service interface lists a series of *services* which can be invoked by clients. This is done by first obtaining an handle to the interface, and then invoking the service on the handle, much like in COM. In contrast with plugins and plugouts, services are always operations (functions): they cannot be classes. A given cell can have both plugging and service interfaces.

**Example: Explicit interaction via service interfaces** A first example shows how a COM-like purely service-oriented cell can be defined. The cell has no plugging interfaces.

```
nameserver =
  cell interface NameServer
    lookup  : String -> Int
    setRoot : String -> Void
  impl
    toLowerCase = fun(str) ...
    theRoot = new(class (s) inst root = nil meth ... )
    lookup = fun(str) ... // look up a name, returning its number
    setRoot = fun(newRoot) theRoot.root := newRoot // set default domain
```

In this example, the nameserver cell declaration has one interface, `NameServer`, which is a service interface with two services: `lookup` and `setRoot`. Their implementations follow the interface. In the implementation there is a private function, `toLowerCase`, which the other implementations can invoke, and `theRoot`, a private object belonging to the cell. This object, which was created at cell creation time, holds the current root value. The presence of objects in cells is important because it allows cells to have state. Within the cell, the private elements can be referred to directly by name.

Continuing with this example, the following client code grabs the `NameServer` interface as `ns`, and invokes some services on that interface ($\triangleleft$ is used both for obtaining a handle to a service interface and for invoking a service on a handle).

```
 ns = nameserver ◁ getInterface[NameServer] // obtain handle to interface NameSever
 ns ◁ setRoot(".jhu.edu");  // invoke setRoot service to set default domain
 ns ◁ lookup("cs")  // uses default domain ".jhu.edu"
```

**Example: Seamless interaction following linking via a plugging interface** The next example repeats the same functionality as the previous one, but using a plugging interface instead of a service interface to explicitly contrast the differences between the two.

```
pluggingnameserver =
  cell interface NameServer
  plugouts
    lookup : String -> Int
    setRoot : String -> Void
  impl
    ... as in nameserver above ...

nameuser =
  cell interface NameServer
  plugins
    lookup : String -> Int
    setRoot : String -> Void
  impl
    ... setRoot("jhu.edu") ...
    ... lookup("cs") ...
    ... lookup("www") ...
```

```
connect(nameuser,pluggingnameserver) at NameServer;  // link cells
...
disconnect(nameuser,pluggingnameserver) at NameServer  // unlink at a later time
```

In this example, the `pluggingnameserver` is similar to the `nameserver` above, but the interface is declared a *plugging* interface by the presence of `plugouts`. These plugouts are the services the `pluggingnameserver` provides. The user cell `nameserver` has the dual set of `plugin` declarations on its `NameServer` interface.

The two are dynamically linked by the `connect` statement. The act of linking reserves the `NameServer` interface for the particular client until the subsequent `disconnect`, so conflicts between users with different default domains are avoided. The `setRoot` and `lookup` operations can be directly invoked from inside `nameuser`, because those services were plugged in. If `lookup` were invoked before the connection to the name server was established, a run-time error would result.

**Example: A Secure Name Server** If the server is serving sensitive data, it may be desirable to restrict the parties that can plug into it by requiring a *trust* relationship.

```
securenameserver =
  cell interface NameServer
  plugouts
    lookup : String -> Int
    setRoot : String -> Void
  impl
    ... as above ...

  trust(x) plugcase NameServer =
    cert = (x ◁ getInterface[Cert]) ◁ certificate()
       ... check that cert is a valid certificate for access
           return true iff the certificate is valid ...

secureuser =
  ... nameuser, with in addition service interface
  interface Cert
    certificate : Void -> String
... and implementation
  impl
    certificate = fun(x)  ... return the certificate for this cell ...
```

The `securenameserver` cell above includes a trust predicate of the form `trust(x) plugcase Nameserver....` The plugcase defines code to be executed when any cell plugs into the `NameServer` interface. The `secureuser` cell includes a service interface `Cert` which the `securenameserver` uses in its trust predicate to determine if the user is authorized to connect. The `securenameserver-secureuser` connection succeeds only if the `securenameserver` trust predicate returns `true`. (In the general case, `secureuser` could also have its own trust predicate for mutual authentication.)

## 2.2 Cells in Distributed Environments

As illustrated in the previous section, cells can be used in a local setting for modular and secure programming. However, a main focus is for cells to serve as a cross-network abstraction for distributed environments. This means that cells are global entities: every cell has an identity which is unique over the network (like COM GUIDs). References to cells are valid across the network, and reference holders use them in the same way regardless of whether they refer to local or to remote cells (like CORBA Inter-operable Object References (IORs)). This is the basis for the following supported functionality:

1. Cells may be copied and shipped across the network, to be linked/invoked locally (the applet metaphor).
2. Cells may move from one network location to another (a form of roving agents).
3. Cells may be linked across the network, that is a cell in one location can be connected to a cell in another location via a plugging interface.
4. A client may transparently invoke operations on a remote cell via a service interface.

From the four points above, only the last one (remote invocation) is integrally supported in existing technologies. As explained in the introduction, copying a cell containing code and stateful objects from one location to another realizes the applet metaphor in a way that allows for systematic authorization and invocation on the client side. Linking cells across the network allows for persistent peer-to-peer connections to be made at a high level of abstraction rather than at the low level of internet protocols. Consequently, instead of performing security checks on a per-operation-invocation basis, such checks will typically be performed on a one-time per-connection basis. This is a crucial point for security-sensitive distributed applications that need to maintain reasonable performance. Component technologies not supporting the component linking concept cannot be expected to provide this kind of security mechanism as an integral part of the technology.

We now elaborate on some of the details involved in the design. The network can be viewed as a set of *locations*, each with an associated address space. Every cell is in some location. References to cells are meaningful across locations, so one location can hold a reference to a cell in another location. By contrast, an object reference is valid only within the location containing the object. Cells can communicate across the network via plugging interfaces or service interfaces in exactly the same way as locally. Copying (moving) a cell means copying (moving) its static code part, i.e., classes and functions declared inside, as well as its current state, i.e., the objects to which it holds a reference, each in its current state.

Since copied or moved objects should execute in the new location, their code (i.e. their classes) must exist in that location. In general, a cell can hold a reference to an object whose class is defined in another cell. If that class is a plugin for the cell, then the cell can be sensibly copied or moved, assuming that an appropriate cell will be available for plugging in the new location. If however this is not the case, for example if the reference was obtained via a service/plugin/plugout parameter, then there will be no way of executing the object in the new location.

We therefore define *encapsulated* cells, as ones which own all object references they hold. A cell *owns* an object reference if it created the object. Only encapsulated cells can be moved or copied, where the check of encapsulation is performed at runtime. Ownership implies having the class for the object either as part of the cell or as a plugin, so restricting copy and move to encapsulated cells guarantees that objects within a cell can execute on the new location. (Some initial connection to library code may be required).

A fundamental requirement that we put on object execution is that objects always run locally, meaning that the class for the object and all its superclasses must be in the same location as the object itself. This is still not guaranteed by restricting copy/move to encapsulated cells because a superclass might be plugged-in remotely. To avoid that, and also in order to be explicit about networking, we require that interfaces – both plugging and service – be declared as `networked` to be used remotely. Networked interfaces have certain restrictions which guarantee locality of code: they cannot have classes as plugins or plugouts.

**Example: Cross-Network Linking** Consider the `pluggingnameserver` and `securenameserver` examples above. These examples can be explicitly implemented with the user and server cells being on different locations with minimal change in the code. First, the interface `NameServer` in both cells must defined as `networked`. Then, supposing a reference for the server cell has been obtained in the user's location, the exact same `connect` command is coded, and the interaction proceeds transparently over the network. The service interface example `nameserver` can also be made distributed, just by defining the interface `NameServer` as `networked`.

**Example: Applets** Applets are cells that come from other locations on the network. In Java, an applet arrives at a location with requirements for several system libraries, such as `java.awt` and `java.applet`. In a cell implementation of the applet concept, there must be a cell for each such system library, and the applet will then negotiate a connection (a plugging) with such a library. Applets with insufficient authorization will not be allowed to plug into the system libraries. The library cells in the client's location are of the form

```
awtlib = cell
  interface AWTRoot plugouts Canvas : Class(...) ... other java.awt root class headers ...
  interface AWTEvent plugouts ... the java.awt.event class headers ...
  ...
  impl
    ... code for all AWT classes ...
  trust(x) ... authorization policy to be enforced on library users ...
appletlib = cell
  interface AppletRoot plugouts Applet : Class(...) ... other java.applet class headers ...
  impl
    ... code for applet classes ...
  trust(x)  ... authorization policy for applet library users ...

...
```

In the server's location, an applet cell of the following form is declared:

```
myapplet = cell
  // import all the relevant libraries
  interface AWTRoot plugins ... Canvas : Class(...)  ...
  interface AWTEvent plugins ... the java.awt.event class headers ...
  interface AppletRoot plugins ... Applet : Class(...) ...
  ...
  interface Run
    main : ...
  impl
    MyApplet = class extends Applet ... // Applet imported from AppletRoot
    MyCanvas = class extends Canvas ... // Canvas imported from AWTRoot
    main = fun(x) ...
```

Assuming the client obtained a reference for that cell, stored in `remoteapplet`, it could copy it and then use it locally. After connecting to the client's local library cells, the plugins `Applet` and `Canvas` will refer to these local classes. The following code copies and then links the applet dynamically with the libraries:

```
// link
myapplet = copy(remoteapplet)()
connect(myapplet,awtlib) at AWTRoot;
connect(myapplet,awtlib) at AWTEvent;
connect(myapplet,appletlib) at AppletRoot;
// run the applet
(myapplet ◁ getInterface[Run]) ◁ main()
```

Note that since the library cells `awtlib` and `appletlib` are plugged into `myapplet`, these cells cannot be used by other parties. New cells should be created for other applets. One consequence of this is that the libraries may themselves hold state information. For instance, the `appletlib` cell could keep track of the applet currently connected to it, and keep an audit trail of critical library functions accessed. Also note that there is no need to declare any `networked` interfaces here since all connections are made locally (after copying).

One important feature of this example is that it shows how a class in one cell can inherit from a class in another cell. When a `MyApplet` object is created, some of its code is in the `myapplet` cell, and some is in the `appletlib` cell due to inheritance from `Applet`. This does not violate the requirement that objects have their code locally, since all cells involved are at the same location.

In addition to code, the imported applet could contain some data, as part of its state. For instance, it could contain relevant web links dynamically kept up-to-date in the server's location. The server could even pick links which are appropriate to the target client.

## 3 Related Work

We have already given the broad picture of how cells relate to existing component technologies in the previous sections; here we focus this comparison further, and also discuss related work in the component research community.

**Existing Component Frameworks** We must start with a disclaimer: it is not quite right to compare our proposal here with COM/Corba/Java because these technologies are implemented working systems, whereas the cell technology is still at the conceptual design stage. At this time, we have a toy language with a precise operational semantics as a proof of concept, and we are about to start an implementation project aiming to prove the concept at a more practical level. The comparison below is therefore on a conceptual level rather than on the level of technical particulars.

Cells are designed to unify in a single abstraction the properties of (1) COM/CORBA/Java-like components, (2) Modula-like modules, and (3) DLL-, or Java .class-like dynamically linkable entities. We believe that this kind of unification is the right thing to do in order to support the challenges of the Internet distributed computing environment. We gave several concrete advantages earlier in the paper, including the synergistic advantages this approach gives to the resulting security and distributed programming architectures.

The idea that a compilation result is directly a dynamically linkable unit is not new; in Java all linking is in fact a dynamic load of .class files generated directly by the compiler. The concept of linking via explicit interfaces is also not new: it is the main concern of module systems, and is also similar to C++ included .h files; but the combination of modules or C++ object files is done statically by tools outside the scope of the runtime execution engine. DLLs on the other hand are linked dynamically; but they have no explicit interfaces. They are at a different level of abstraction than object files, and their relationship to the object files from which they were created is a secret of the operating system.

Cells incorporate the above properties in one programming abstraction. The Java model, where classes in .class form can move around the network to be loaded dynamically by the JVM, is an enabling technology for the Internet. However, there is no systematic, interface-driven, upfront way by which these classes (usually applets) are plugged into the local environment. Consequently, security checks – a major issue in this setting – are embedded in the code of library functions. Moreover, the data for these security checks – the policy files – are kept separately in files that have nothing to do with the class being loaded. This has two problems: first it is hard, if not impossible, to write class-specific security policies. Second, programming for security is cumbersome for the programmer, who needs to insert explicit calls to checkPermission() in his program.

Another drawback of the Java loading mechanism is that the objects instantiated from the loaded classes exist in the general JVM space, not within some space devoted for the loaded class. As a result it is hard to control the imported component *as a whole*.

In contrast, COM and Corba have all it takes for a component to act as an isolated entity, systematically accessible only via interfaces. However, because they are not concerned with the *code* that runs the components (which are COM/Corba *objects*), they are not directly suitable for the internet, and a different layer – outside the scope of these technologies – needs to be used to move them around as code-components. In the new Microsoft internet architecture, these concepts are divided between the COM+ component architecture and the .NET runtime architecture, each with independently designed security and distribution policies. By putting together the advantages of the existing technologies into a single entity, cells, we believe a simpler, more synergistic architecture is produced which will lead to faster production of more reliable and secure code.

**Other Related Work** The importance of modeling the dynamic aspect of component connections has been addressed by several researchers. Connection-oriented programming as discussed in [Szy98] is closely related to our plugging interface connections, which can be viewed as a concrete realization of that idea.

There are several foundational component frameworks that have been developed. Piccola [AN00, LAN00] is based on a core calculus for components. It is a glue scripting langauge for putting together software. Importantly, it explicitly models the dynamic nature of component connection. It does not include 1-1 linking/unlinking via plugging/unplugging or trust relationships and is untyped. Several purely static component calculi have also been developed. [SC00] models typed static component composition. Some module systems are also very close to static component systems; one such system is Units [FF98], which allows some dynamic module linking but is lacking any service interfaces or statefulness.

## References

[AN00] Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology.* Kluwer, 2000. to appear.

[FF98] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[LAN00] Markus Lumpe, Franz Achermann, and Oscar Nierstrasz. A Formal Language for Composition. In Gary Leavens and Murali Sitaraman, editors, *Foundations of Component Based Systems*, pages 69–90. Cambridge University Press, 2000.

[RS01] Ran Rinat and Scott Smith. Cells: Dynamic trusted components. Draft, available at http://www.cs.jhu.edu/~scott/pll/cells/, March 2001.

[SC00] Joao Costa Seco and Luis Caires. A basic model of typed components. In *ECOOP*, pages 108–128, 2000.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New York, NY, 1998.