Modular Internet Programming with Cells

Ran Rinat and Scott Smith

Department of Computer Science
The Johns Hopkins University
Baltimore, Maryland, USA
{rinat, scott}@cs.jhu.edu
http://www.cs.jhu.edu/~scott/cells

Abstract. The success of Java in recent years is largely due to its targeting as a language for the Internet. Many of the network-related features of Java however are not part of the core language design. In this paper we focus on the design of a more parsimonious Internet programming language, which supports network integration smoothly and coherently as part of its core specification.
The key idea is to center these extensions around the unified notion of a *cell*. Cells are deployable containers of objects and code, which may import (*plugin*) and export (*plugout*) classes and operations. They may be dynamically linked and unlinked, locally or across the network. Cells may be dynamically loaded, unloaded, copied, and moved, and serve as units of security. At first approximation, cells can be thought of as a hybrid between modules and components. Here we concentrate on the design of *JCells*, a language which builds cells on top of the fundamental Java notions of class, object, and virtual machine.

1 Introduction

In the history of programming languages, fundamentally new concepts have never been implemented correctly the first time or even the first several times. FORTRAN was very difficult to parse because parsing theory was not developed. Lisp first implemented higher-order functions, but with dynamic scoping; Scheme was created partly to correct this flaw in the Lisp design. In a similar spirit, our goal is to take a second look at important new features that have arisen in the Java JDK, and to create a more parsimonious language expressing these features smoothly and coherently as part of the core specification. Features of Java which are relatively new include explicit dynamic class loading, object serialization and RMI, an explicit security architecture, Java Bean components, and mobile Applet code.

At the core of our quest for an improved language platform is the concept of a *cell*. Here is a brief definition.

> **Cells** are deployable containers of objects and code. They expose typed linking interfaces (*connectors*) that may import (*plugin*) and export (*plugout*) classes and operations. Via these interfaces, cells may be dynamically linked and unlinked, locally or across the network. Standard client-server

style interfaces (*services*) are also provided for local or remote invocations. Cells may be dynamically loaded, unloaded, copied, and moved. Cells serve as principals to which security policies may be applied.

In terms of existing concepts, cells can be thought of as an internet-aware, security-geared hybrid between the concepts of *component* and *module*.

In this paper we define *JCells*, a language which builds cells on top of the fundamental Java notions of class, object, and virtual machine. We build on top of Java here for two purposes: we need not detail many aspects of JCells since it is identical to Java; and, by implementing on top of Java we can more easily get a prototype. We aim to get to examples as quickly as possible to define what is meant by the above description. First we help set the stage by relating cells to known concepts.

1.1 Relating Cells to Existing Concepts

Cells combine ideas from disparate domains of research, so they need to be viewed from several angles to see their merit. As already mentioned, cells aim to capture aspects of existing module and component systems, but also aspects of distributed object systems, mobile code systems, persistent object systems, and distributed security architectures.

Cells are similar to some module interface designs [MFH01,FF98,CDG+88]. Like modules, cells have fixed interfaces ("connectors") to the outside which import and export ("plug-in" and "plug-out") items, and cells are persistently linked to one another. However, unlike modules, cells separate the notions of loading and linking. Cells are loaded in unlinked form, and then are explicitly linked by the JCells `link ...at ...` command. They may also later be unlinked and unloaded. Cells may link across the network; module linking is confined to a single process. Cells may contain objects and so have state and thus a run-time presence; modules are purely code with little if any run-time presence. The module aspects of cells were partly inspired by Units [FF98], which recently have been implemented for Java as Jiazzi [MFH01].

Cells are related to component systems [Szy98] such as Corba [Gro01] and COM/DCOM: they are larger-grained than objects and have run-time interfaces (*connectors and services*) for interaction. Like Corba and COM/DCOM objects, cells support local and distributed service invocations, a client-server mode of interaction. Unlike components, they are designed to make long-term interactions explicit by forming persistent peer-to-peer links with one another, via connectors. Cells also "own" a collection of objects and thus have state; components don't. Several considerations led us to make cells stateful. Cells can dynamically link and unlink, and as such must have the state of the connections kept at run-time. The modularity of components and module systems is "shallow" in the sense that it exists at compile-time only; cells are "deeply" modular in that run-time objects are also grouped. We believe that when engineering software, this extra dimension of modularity will help factor functionality and clarify meaning. Lastly, migrating cells

can directly "walk to" their new locations without the need for an additional protocol to move their state; this is similar to the design of other mobile code systems, discussed next.

Cells support a notion of mobile code which is a generalization of Java Applets: mobile code packages are *serialized cells*, which may include objects as well as code, and cells may actively migrate from one host to another. Here we have been inspired by the numerous mobile object systems also aimed to support such generalized functionality [JLH90,Car95,Whi94,VT97,LO98]. Third-party references to a cell are maintained after a cell moves.

Cells support a clean messaging protocol between virtual machines; in this sense they are related to Java's RMI [Sun] and other research in distributed object-passing systems [JLH90,BNOW93,Gro01]. In contrast to RMI, we believe that remote objects should not generally be used to support services invoked by non-local users: objects are too light-weight to support the sizable burden of remote invocation. Instead we advocate the use of cells in the place of remote objects: one cell may easily invoke a service on a remote cell to which it holds a reference. In this sense cells act like a distributed Corba or DCOM component. We still support the notion of object method invocation across the network (via what we call a *modulated reference*) for cases of lightweight servers implemented as objects.

Security is a main focus in the cell architecture. First, the act of bringing cells together—linking—is subject to a runtime authentication protocol. Secondly, cells aim to isolate objects that should not directly interact with one another. Our approach to object isolation is related to J-Kernel [HCC+98]; other related work includes [BR00,WCC+74]. Cells also support a capability-based layer of security across the network, as in [vDABW96]. The goal of object isolation inside a cell wall also has parallels in several abstract theories of distributed system security [Car99,VC99].

Cells are designed to cleanly support persistence: cells *own* their objects, and so a cell can be archived by serializing it and all of "its" objects. Persistent object database systems [ZM90] generally allow any object to be persistent; we instead have a per-cell notion of persistence. Like object databases, cells are a good level of granularity on which to place transaction processing protocols.

Cells also are related to prototype-based object languages such as Self [US87] in the sense that they can in some ways be viewed as heavyweight prototype-style objects: cells have no notion of inheritance, and copies can be generated from a prototype cell. Self's transporter mechanism for serializing objects is also related to how cells serialize their "own" objects, but ownership is more *ad hoc* in Self since without something like cells it is hard to define what objects should own what other objects. A more elegant self transporter mechanism is stated as future work [Ung95]; we believe cells make a step toward such an improvement.

In terms of the general goals, cells are also intended to provide open internet services on a wide-area network. Such services have clear, published modes for interaction, and directories for looking up services. In this sense cells can be viewed as a toolkit with a similar purpose to the XML/SOAP/UDDI/WSDL standards now being developed. Cells are more expressive than the above technologies, but

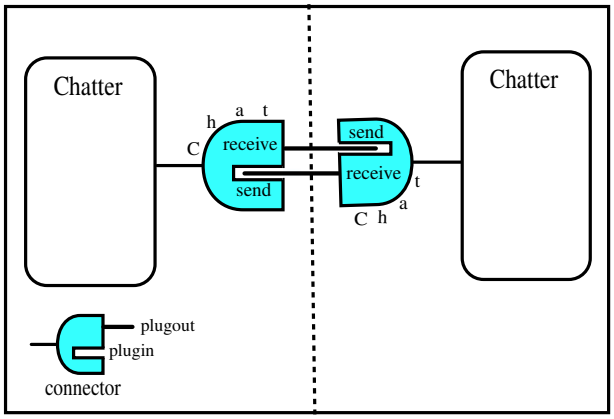

Fig. 1. Two chatters communicating over the network

the above are at the bottom text-based messaging protocols and so are more widely adaptable across platforms. So the two address a different target set of problems. For loosely-coupled systems a simple protocol is often preferable, but for more tightly-coupled systems, the additional expressivity of cells may be necessary.

There are many projects in addition to those named above that share at least a fraction of our goals. JavaSeal [BV01] is built around the concept of a *seal*, which encapsulates objects and code much like cells do. Seals own their objects, just like cells. Our focus is more on module/component technology, and individual cells can be large immobile components consisting of many classes, whereas seals are more intended for mobile units. Our module focus means we support cross-network linking and our component focus means we support remote service invocation. Serialized cells are similar to passive seals. Seals may be nested; we may add nesting to cells in the future.

ArchJava [ACN02] is another module/component system extension to Java which supports code-embedded architectural design and guarantees the compatibility of implementations. ArchJava components have *ports*, which are similar to our plugins and plugouts. Their focus however is more on static architectural aspects and less on runtime and networking.

2 Cells by Example

In this section we introduce JCells through a running `Chatter` example modeled on an internet chat system such as ICQ, Odigo, or Yahoo Messenger. The base scenario is two `Chatter` cells running in different virtual machines which link to one another across the network to carry out a chat conversation; see Fig. 1.

Each `Chatter` has a *connector* named `Chat` with send and receive operations. The `receive` operation is a *plugout*, i.e. provided to anyone who connects to it, and `send`

is a *plugin*, i.e. expected from connecting cells. When the individual `Chatter` cells are first loaded, their connectors are not linked. Later, an explicit `link` command will connect them across the network (the dashed line in the figure is the network boundary), linking the `send` of one to the `receive` of the other. From this time on, one cell calling its plugin `send` will invoke the plugout `receive` of the other `Chatter`, passing the message as parameter. When the two cells are done chatting, they can be `unlink`ed.

The `Chatter` is expressed by the following JCells syntax.

```
// File Chatter.csc

cell Chatter {
{ // begin cell header

//------------------Cell Connector Declaration -----------------

connector Chat {  // connector which another chatter links on to chat
    plugins {
        void send (String aMessage);
    }
    plugouts {
        void receive (String aMessage);
    }
}

} // end cell header

//====================== Cell Chatter Body ====================

{ // begin cell body

// ------------------------- Cell Operations--------------------
// (Operations defined for the cell; some, e.g. receive, may be plugged out)

void linkToAnotherChatter(String userID) {
    cell Chatter otherChatter = ... get chatter for userID;  detailed later ...
    link otherChatter at Chat [receive->send, send<-receive] ;

void UnLinkFromOtherChatter() {
    unlink at Chat;
}

// sendMessage is activated by the GUI's send button event handler
void sendMessage() {
    String msg = ... // get message from GUI window
    send (msg) ;     // call send plugin. receive of linked chatter is thus invoked.

}

void receive (String aMessage) {
    ... implementation for plugout operation receive: display aMessage ...
}

...

}  // end cell body

}   // end cell Chatter
```

The code above is in a cell source file `Chatter.csc`. A `.csc` file contains the definition of a single cell, in analogy to a `.java` file holding one class. A cell's definition consists of a *header* and a *body*. The header contains declarations, and the body contains implementations. Every plugout must be implemented in the body

(or, itself plugged in on some other connector). The only declaration in the header here is of the `Chat` connector with the `send` operation as plugin and `receive` as plugout. The cell body contains the operation implementations. `receive` implements the plugout, `linkToAnotherChatter` is a local cell operation responsible for linking this chatter to another chatter, and `sendMessage` sends a message by invoking the `send` plugin. Note that a call to a plugin is the same syntax as a call to a local operation.

The body of operations mostly follows standard Java syntax, with only a few modifications. In `linkToAnotherChatter`, the variable `otherCell` has *cell type* `cell Chatter`. The keyword `cell` is used to distinguish cell types from class and interface names. As with classes, the definition of cell `Chatter` also defines a type by the same name. The statement

```
link otherChatter at Chat [receive->send, send<-receive]
```

is a JCells extension to Java syntax which dynamically links the current cell with `otherChatter` via connector `Chat`, and hooks `receive` into `send` in both sides. For the chat application, this link will be across the network, but local and nonlocal linking share the same syntax and largely the same semantics. Even though linking is dynamic, we do not want to lose the advantages of static types and typchecking: since `send` and `receive` are linked, we require their types to be the same. In general, a link at a connector requires the types in all connected plugin/plugout pairs to correspond.

Cells execute in a *Cell Virtual Machine* (CVM), which is similar to a Java JVM. One important property of cells is cross-network linking: the two chatter cells are are running in different CVMs at different network locations, but still can be linked. The `Chatter.csc` file above is compiled into a `Chatter.cell` file which can then be loaded into two different CVMs; one chatter then can `link` to the other to form the configuration of Fig. 1. A `.cell` file is at first approximation analogous to a Java `.class` file, but it contains a cell *including* its classes and objects.

CVMs only execute cells, so every object and operation must be part of some cell. There can be many cells loaded into a given CVM.

Library Linking Cells serve both as components and as modules, thus code libraries are also cells. `Chatter` will need to plug in libraries, in this example an audio-visual support library extending the chatter's basic text capability. The audio-visual cell is likely to arrive to the local machine as a downloaded `AV_Extension.cell` file, perhaps some time after the basic `Chatter.cell` system has been installed. The `AV_Extension.cell` is a mobile cell, which generalizes Java's Applets: it may include state (in the form of objects) along with code.

We now expand the above example to link to an audio-visual library, pictured in Fig. 2 (cell `ChatCentral` at the bottom of the figure is introduced later). The syntax is as follows.

```
cell Chatter {

{ // Begin Cell Header
```

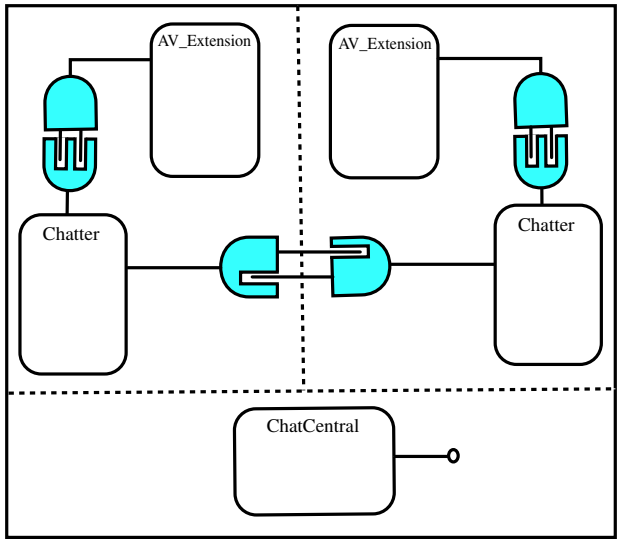

Fig. 2. The complete chatter example

```
//---------------Class Signature Declarations------------
// (Class signatures declared here for typechecking purposes)

class MediaPlayer {
    void play();       // (No code is given in a class signature)
    void pause();
    ...
}

interface Cert {
    ... Java interface for certificates ...
}

//----------- Cell Signature Declarations ------------
// (Like class signatures, need to declare types of other cells for typechecking)

cell AV_Extension {
    connector AudioVideo {
        plugouts { MediaPlayer, ... }
    }
    service Cert getCertificate();

}


//----------- Connectors for Cell Chatter ------------

connector AudioVideo {  // For plugging in audio-video library code
    plugins {
        MediaPlayer,  ...  // MediaPlayer is a class plugin
    }
    // special verify predicate: verify the extension comes from a reliable source
    boolean verify(cell AV_Extension other) {
        Cert cert = other <- getCertificate();
        ... return true only if cert is satisfactory ...
    }
```

```
}

connector Chat { ... as before ... }

} // End Cell Header

//================ Cell Chatter Body =================

{ // Begin Cell Body

//-----------------Cell Fields-------------------------
// (Cell fields allow cells to have their own state)

cell AV_Extension theAV_Extension ;

//------------------Cell Operations------------------

void onLoad() {  // this hook is invoked when the cell is loaded
    theAV_Extension = (cell AV_Extension) lookup("AV_Extension");

    link theAV_Extension at AudioVideo;  // links plugins with like-named plugouts

}

void linkToAnotherChatter(String userID) { ... as before... }
void UnLinkFromOtherChatter() { ... as before ... }
void receive(String aMessage) { ... as before ... }

//--------------Cell Internal Class Definitions-------------
// (Cells can define their own classes either for internal use
// or for plugging out and use by others)

class ChatMediaPlayer extends MediaPlayer { // internal class extending a plugin class
    ....
}

}  // End Cell Body

}   // End Cell Chatter
```

Looking first at `Chatter`'s connectors declared above, a new connector for the `AV_Extension` library cell is defined. This connector declares plugins through which various classes such as `MediaPlayer` are plugged-in, showing how classes as well as operations may appear on connectors. In the providing cell `AV_Extension` (whose code we don't give), these classes should have been declared as plugouts and defined in its body. In `Chatter`'s body, there is now a class `ChatMediaPlayer` which extends the plugin class `MediaPlayer`—cross-cell inheritance is an important feature of JCells.

One important security aspect of cell connections is that link time is a point at which authentication should occur. This is one of the points in allowing persistent connections via `link`: security checks happen only at `link` time, not at every invocation. The connector `AudioVideo` includes a special `verify` predicate for this purpose. `verify` is called by the CVM whenever a link at this connector is initiated. This is one example of the per-cell approach we take to security. Note that the `Chat` connector should certainly also be verified in a real implementation since it is a cross-network link. In this example, verification of a library is sensible because the A/V library is a downloaded plug-in which cannot always be trusted.

In the code of `verify`, `other <- getCertificate()` invokes a *service* on the other cell. Services are direct cell operations which one can use without linking;

they are used for more transient interactions. Services are described in more detail below. The signature for `AV_Extension` declares the service `getCertificate()`.

Upon loading a cell, the CVM calls the hook operation `onLoad()` if it is defined. Here `onLoad()` looks up and then links the `AV_Extension` library cell via `link theAV_extension at AudioVideo`. Since here plugouts are plugged into plugins by the same name, there is no need for a plugging map (`[...<-...]`). `lookup` is a *built-in* operation, found on all cells in analogy to the methods of `Object` which all objects respond to. `lookup` is a nameserver which maps strings to cells.

Cell source files are compilation units, and so they must know the types of all cells and classes they interact with. For simplicity, we now require each cell to directly declare the signatures of all these cells and classes. Class and cell signature information is given above in the header of the cell definition. A class signature is a class without method bodies, and similarly a cell signature is a cell without a body. Note that these declarations are solely for compile time typechecking; in order for actual code linking to occur, cells must explicitly link their connectors at runtime. Realistic cells will have very long signature declaration blocks, and a mechanism for abbreviating and sharing signatures is a topic for future work.

A full Chatter system We now flesh out the example above by showing how a central server (`ChatCentral`) is used to complete the picture of a chat framework, by allowing individual `Chatter` cells to learn about one another. In this full example, we also show how song archives may be traded amongst chatters, without having to send entire archives across the network, and without compromising security. Lastly, the example also illustrates cell persistence: a `Chatter` can be unloaded along with its state, and then reloaded at a later time.

The full architecture is shown in Fig. 2. It includes the `ChatCentral` cell, on which we only invoke services. Here is the code for the full example.

```
cell Chatter {

{ // Begin Cell Header

//--------------Class/interface Signature Declarations------------

 ... as before plus:

interface User {           // interface for chat user data
    String getID();
    String getName();
    String getIntention();  // romance, friendship, chat etc.
    String getHobbies();
    int getAge();
    ...
}

interface SongArchive {    // for song archives shared amongst chatters
    setGenre(String musicGenre);
    setArtist(String anArtist);
    setLanguage(String aLanguage);
    Collection getSongs(); // returns songs matching criteria
    ...
}

//------------ Cell Signature Declarations --------------

cell AV_Extension { ... as before ... }
```

```
cell ChatCentral {    // Clearinghouse for chatters to find other users to connect to
    // declaration of ChatCentral's services
    service String registerNewChatter(copy User aUser, cell Chatter aCell);
    service void logon (cell Chatter aChatter);
    service void logoff(cell Chatter aChatter);
    service copy User getUser (String userID);
    service void updateUserDetails(User aUser);
    service cell Chatter getChatter(String userID);
    service Collection getUsersByCriteria(String Criteria); // returns set of users

}

//----------- Connectors for Cell Chatter --------------------

 ... connectors Windowing, AudioVideo as before ...

// Chat connector expanded to include song archive capability

connector Chat {
    plugins {
        void send (String aMessage);
        modulate songArchive requestSongs(); // modulated (proxy) reference returned
    }
    plugouts {
        void receive (String aMessage);
        modulate songArchive provideSongs();
    }
}


} // End Cell Header

//================ Cell Chatter Body =================

{ // Begin Cell Body

//-----------------Cell Fields-----------------------

cell ChatCentral theChatCentral;
cell GUI theGUI;
cell AV_Extension theAV_Extension ;

String myUserID;
User me;                     // an object holding this user's details
SongArchive mySongArchive;   // this user's SongArchive

boolean firstLoad = true;

//-----------------Cell Operations ------------------

void onLoad() {

    // get cell references via lookup

    theChatCentral  = (cell ChatCentral) lookup("ChatCentral");
    theAV_Extension = (cell AV_Extension) lookup("AV_Extension");

    // link to library

    link theAV_Extension at AudioVideo;

    // if first time, get user details and register with ChatCentral

    if (firstTime) {
        me = getUserDetails();     // obtain user details by input from user
        // register new user via ChatCentral service ; fresh user ID is returned:
        myUserID = theChatCentral <- registerNewUser(me, thisCell);
```

```
        firstTime = false;
    }

    // logon to chatCentral
    theChatCentral <- logon(thisCell);   // let ChatCentral know we're ready to chat

void onUnload() {     // a hook run just before cell is unloaded

    // logoff upon unload since user will not be available for chat
    theChatCentral <- logoff(thisCell);
}

void linkToAnotherChatter(String userID) {

    cell Chatter otherChatter = theChatCentral <- getChatter(userID);
    // link chatters as before but more plugins and plugouts to map:
    link otherChatter at Chat [receive->send, send<-receive,
                               provideSongArchive->requestSongArchive,
                               requestSongArchive<-provideSongArchive] ;

void UnLinkFromOtherChatter() { ... as before ... }

// operations which implement plugouts

void receive(String aMessage) { ... }

modulate songArchive provideSongs { .... };

//--------------Cell Internal Class Definitions-------------

// As before, plus:

class SongArchiveObj implements SongArchive {
    ...
}

class UserObj implements User {
    ...
}

}   // End Cell Body

}   // End Cell Chatter
```

`ChatCentral` is a server cell that exclusively provides *services*. Services are the other means of communication with cells: they are operations that users can invoke without linking, in a client-server fashion. The group of services of a given cell act like a COM/Corba interface on the cell. Like connectors, services must be listed in the cell's header, and like plugouts, they must be implemented in the body or listed as plugins.

Each loaded cell has a *cell identifier* (cid) which identifies it globally across all CVMs. Cell variables at run-time hold cid's. Users wishing to join the chatting network proceed as follows: first they obtain the `Chatter.cell` file with no cid, an *anonymous* cell. When it is then loaded, the CVM automatically generates a fresh cid for the cell and executes its `onLoad()` hook. Using a boolean flag `firstTime`, `onLoad()` recognizes that it is the first time this cell is being loaded, and so asks the user for his/her details, including name, age, hobbies, etc. It then invokes the service `registerNewUser()` on `ChatCentral` to register this new user. This service returns a fresh `userID` generated by `ChatCentral`, which identifies the user in the chatting

network. Next, `onLoad()` calls the service `logon()` on `ChatCentral`, letting it know that it is active and willing to chat.

At some later point the `Chatter` cell may be unloaded into a `.cell` file by invoking `unload(aStream)`; the resulting `.cell` file will now contain the cell's cid, as well as its state, including the `userID`, the fields `me` and `mySongArchive`, etc. The full unload process is as follows: the CVM first unlinks all connectors, the code in the `onUnload()` hook is run (which in this case logs the user off from `ChatCentral`), and the whole cell including its state is then serialized into a `.cell` file. When the same cell is later loaded into the CVM, its state is fully restored and cid preserved.

While logged on, `ChatCentral` allows individual chatters to learn about each other and to update their own details via `ChatCentral` services. If one chatter wants to chat with another, it obtains a reference to it (i.e., a cid) from `ChatCentral` using `getChatter(String userID)`, where `userID` has been obtained previously, for example by using `ChatCentral`'s `getUsersByCriteria(aCriteria)` service. The chatting itself is done by the two cells linking directly at connector `Chat`. (Note that `Chatter` cells could have alternatively used a persistent connector to interact with `ChatCentral`; we chose a service interface here to illustrate cell services.)

The result type of `getUser()` is declared to be `copy User`; the keyword `copy` indicates a copy of the result is returned. Since this service is invoked from another network node, the link would fail without this declaration; unlike RMI, there is no implicit serialization of objects upon remote invocation.

This example also demonstrates a use of *modulated* object references, another means by which parameters can be passed and results returned across the network. A modulated reference is an RMI-like proxy through which one cell may refer to an object inside another cell. Each cell implicitly maintains a *modulation table* holding the objects it is letting outside cells access; these references can be revoked when the `Chat` connector is unlinked. Revocability of modulated references is an important security feature—when one cell is done interacting with another, backdoor channels via modulated references may be closed. Modulation is, unlike RMI, designed for either local or remote access; locally, the stronger security properties make it useful for *e.g.* agent interaction.

Here, a plugin `requestSongArchive` and a plugout `provideSongArchive` have been added to the `Chat` connector. The return value of these operations is a modulated `SongArchive` object which one chatter passes into the other while chatting: the keyword `modulate` in the connector declaration indicates a modulated proxy is to be returned and not the actual object. While linked, the receiving chatter may browse the other chatter's song archive by invoking methods on the modulated `SongArchive` object it holds. If the song archive were instead returned by `copy`, the whole archive would have to be copied across the network.

3 JCells Programs

This section describes JCells programs, already introduced informally in the example of the previous section. In the subsequent section we describe the run-time behavior of cells.

3.1 Cell Source Files and Static Scoping

Each cell is defined in its own distinct `.csc` (cell source code) file. Cells are the unit of source definition in JCells; each class is defined as part of some cell, and so `.csc` files replace `.java` files. A `.csc` file is compiled to a `.cell` file with no cid and with an empty state (see Sec. 4.3).

Cell definitions are statically closed name spaces on two levels. At the first level, every identifier mentioned in the header must be declared in the header. So, if a service, a plugin, or a plugout has a parameter of type A, A must be declared in the header. At the second level, identifiers mentioned in the cell's body must either be declared in the header or defined in the body.

All top level identifiers declared inside a cell are unique and thus referable anywhere within the cell without qualification. The top level identifiers of a cell are its name, the names of cell signatures declared inside, connector names, plugin/plugout names, services, and the names of all elements defined in the body. The uniqueness requirement entails that there may not be two plugins with the same name (even on different connectors), and the names of all internal classes, operations, and interfaces must be different from all plugin names. Plugouts and services must be defined in the body, and a given plugout name may be listed in more than one connector (but implemented by one element in the body).

3.2 JCells Syntax

The syntax is very similar to Java, with the `cell` definition, `link`/`unlink` statements, and service invocations `aCell <- aService()` being the primary new syntax. We informally described this new syntax when presenting the example.

A cell definition consists of a *header* and a *body*. The header mainly contains declarations of connectors and services, but also cell, class, and interface signatures for compile-time typechecking. Connectors list plugins and plugouts which may be classes or operations. Services are always operations. All plugouts and services must either be defined within the body or declared as plugins.

The body defines fields, classes, operations, and interfaces, some of which implement the plugouts and services, and some of which are purely internal. In the body, plugins are referable as if they are defined internally. In particular, an internal class may inherit from a plugin class.

The built-in operations are listed in Appendix A. The electronic version of this paper contains a link to the full JCells grammar.

3.3 Typing

Cells are strongly typed, meaning that "message not understood" errors do not arise at run-time. Additionally, cell linking never fails due to plugin-plugout type mismatch. Cell type casting *may* fail dynamically, just like object downcasting may fail at run-time in the JVM.

Cell Types and Subtyping The only new types introduced in JCells are *cell types*. A definition of a cell implicitly derives a cell type by the same name with connectors and services as specified in the cell's header, much in analogue with how class definitions in Java implicitly define a type. Cell types may in addition be declared directly via *signatures*, see below.

Cell types are referred to in the program as `cell ACellName`, with the prefix `cell` added to distinguish them from class types.

A *subtyping* relation is supported between cell types. Unlike class/interface subtyping in Java, cell type subtyping is structural, not by name: one cell is a subtype of another if every service and every connector appearing in the supertype cell also appears in the subtype cell, and with the same or more plugouts and the same or fewer plugins on each connector. Structural subtyping is important because of the open-ended nature of cells: it is possible to interact with an external cell for which only one connector type is known; the types of all connectors are not needed. The empty cell type `cell Cell` is a supertype of all cell types.

Casting on cell types is also supported, but in contrast with Java typecasting, it is of a purely *structural* nature: a cell may be cast to a type consistent with its header. That is, the cell at run-time must have all the connectors and services specified by the cell type it is being cast to. By subtyping (subsumption), it may have more. As with Java type casts, upcasts entail no runtime overhead but downcasts require a run-time verification that the cell indeed has the indicated and so-typed connectors and services. Type `cell Cell` is useful in analogy to `Object` in that polymorphism on cells can be crudely modeled by using type `cell Cell` and then casting as appropriate. For instance, the built-in operation `load` to load a `.cell` file returns a cell of type `cell Cell` which then must be cast to be used.

Type Signatures in Cells Because cells are statically-typed closed namespaces, a cell definition must declare the types of all cells and all classes/interfaces it refers to. This is done by declaring cell and class/interface type *signatures*. Cell type signatures are simply cells without a body. Class signatures are analogously classes without method bodies. Signatures may appear only within a cell's header. Because cell signatures are defined within an enclosing cell, they may refer to other signatures declared in that cell. This makes it possible to declare cell types that recursively reference each other. One property of this requirement is that there will be many class signatures written out in cell definitions and cell types. We plan on adding type abbreviation mechanisms to the language in the future.

Type checking cell-related statements A simple link statement

`link` *aCell* `at` *aConnector*

is statically typechecked: both the cell containing the link command and *aCell* need to have a connector named *aConnector*, with compatible types. That is, the connector *aConnector* on each cell must have a plugout A for every plugin A on the other cell's *aConnector*, and with identical type. There may be unused plugouts on either connector. In addition, the type `cell A` of the argument to `verify(cell A`

`aCell)` in each cell must be a supertype of the other cell's type. `unlink` statements are typechecked similarly to `link`.

A service invocation

```
aCell <- s (args)
```

is typechecked w.r.t *aCell*'s type in analogy to how object method invocation is typechecked. The concepts underlying cell types are generally well-understood: import/export interfaces subtyping, and typecast. So, we do plan on evantually proving the system sound, but is it not a priority.

4 JCells Semantics

In this section we outline the semantics of cells, elaborating on what was presented in the example. Since all of the concepts are interlinked, this section by necessity contains many cross-references.

4.1 Cells and the Cell Virtual Machine

Each cell executes within a *Cell Virtual Machine* (CVM). CVMs are responsible for loading, linking, and executing cells—much like JVMs w.r.t. classes. The major difference is that cell linking is triggered by explicit `link` commands in the program and is not done automatically upon loading: directly after being loaded, a cell is not connected to any other cells. In addition, linking is performed through cell connectors, which act as explicit interfaces to cells. In our implementation of JCells, each CVM is implemented by a JVM. CVMs contain cells exclusively—there is no possibility of a class or an object to be in a CVM but not part of some cell.

A cell in a CVM interacts with other cells via its connectors and services, and internally contains classes, operations, and objects, see Fig. 3. Connectors list plugins and plugouts, and may be used to link to cells with compatible connectors (Secs. 4.8,3.3). Linking will typically be used for more long term, security sensitive interactions. After linking, references to plugins in one cell are resolved to the connected plugout on the other. Services are operations which may be invoked directly, in a client-server fashion. The set of operations of a given cell act like a COM/CORBA interface to the cell. Services will typically be used for more light-weight, short term, and less security sensitive interactions.

Internally, a cell has code in the form of classes and operations, and state consisting of the state of its cell fields, the objects it owns (4.5), and the modulation table for its modulated references (4.4).

4.2 Cell Identifiers (cid's)

When a cell is loaded into a CVM, it is assigned a *cell identifier* (cid). The cid identifies a CVM and a cell within that CVM, so it identifies a cell uniquely and globally across all CVMs. Any `cell` variable at run-time is in fact holding a cid. We elaborate more on the need for and use of cid's in Sec. 5, where distributed cells are discussed.

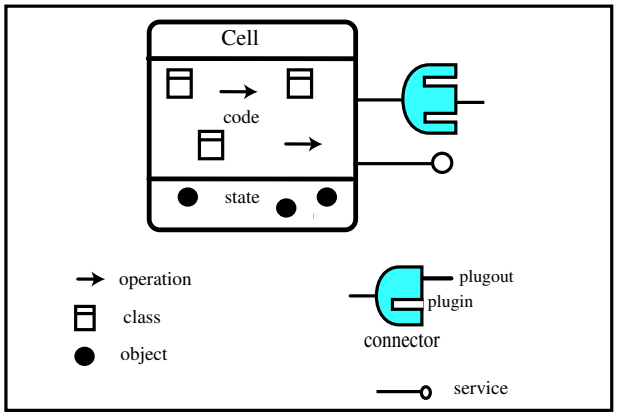

Fig. 3. A cell at run-time

4.3 .cell Files

A cell is loaded into a CVM from a stream called a *`.cell` file* (In analogy with .class files, we use this terminology even though the stream may not actually come from a file). A `.cell` file contains all the information needed to create a cell in memory. This includes the static part of the cell (functions, classes, Java interfaces) as well as the dynamic part (cell fields, owned objects, table of modulated objects). A `.cell` file may also contain a cid identifying the cell; if there is no cid the cell is *anonymous*.

Compiling and serializing a cell both result in a `.cell` file. In this sense, `.cell` files unify two distinct ideas in Java, the `.class` file and a file of serialized objects. Compiling a `.csc` file results in an anonymous `.cell` file with `null` state. Serializing a cell (see Sec. 4.6) results in a `.cell` file that contains a cid and the cell state.

4.4 Object References

CVMs support two kinds of references to objects: *hard* and *modulated* references. A hard reference is a normal Java object reference: a pointer to an object in memory. Hard references can span cell boundaries, but cannot span CVM boundaries. They allow direct and full access to the referenced object.

A modulated reference is a proxy to an object which may be in a different cell, see Fig. 4. Internally, a modulated reference has the form (cid, oid) where cid is the cid of the modulating cell and oid is the identity of the object within the modulating cell. From a programmer's point of view, modulated references are transparent: they are accessed as if they were local, like RMI objects. Behind the scenes, the proxy forwards method calls to the modulating cell which in turn forwards it to the real object.

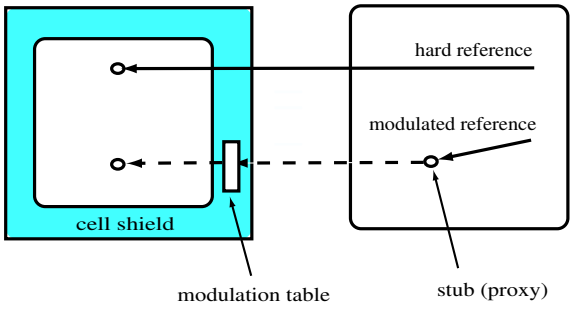

Fig. 4. Hard versus modulated references between cells

The modulation of references is an important security construct: a cell may invalidate a reference it is modulating at any time. This is typically programmed to happen when a connector through which the reference was passed is unlinked; Unlinking a connector signifies the end of a tightly-coupled relationship, and object-object connections made between the two cells via the connector should be broken. The idea of revokable proxy object access is a common theme of object security models [HCC+98]. The co-existence of hard and modulated references allows the programmer to control the degree of isolation of a given cell.

In the implementation of modulated references, each cell holds a *modulation table* mapping the oids of objects it is modulating to their actual locations in memory.

4.5 Cells as Containers: Object Ownership

Cells are both containers of static elements—functions, classes, interfaces—and dynamic elements—objects. The static elements contained by a given cell are those textually contained by it. The objects contained in a cell are given by an *ownership* relation between cells and objects: every object is owned by a unique cell, its *owner*. The owner of an object is set at object creation time to be the cell containing the `new` command that instantiated the object. Ownership is important for cell serialization purposes (see Sec. 4.6): when a cell is serialized, all the objects it owns come along.

4.6 Serialization

One of the fundamental features of cells is the ability to control them *as a whole*. Cells can be unloaded to be re-loaded later, they can be copied, and they can be moved. Underlying all these operations is the *serialization* process, i.e., turning a cell with everything it contains into a `.cell` file. All cells are in principle serializable.

When a cell is serialized, both its static elements—functions, classes, interfaces—and its dynamic elements—fields, owned objects, modulation table—are streamed into a `.cell` file. Streaming the object state consists of deeply serializing all cell

fields and objects owned by the cell until either a *basic value* (including integer, boolean, cell reference, modulated reference) is reached, or an object not owned by the cell is reached. There are three built-in operations for serialization, which differ in the way they handle the latter case. `serialize(aStream)` serializes a cell and into `aStream` and raises an exception if a non-owned object is encountered. `serializeWithNull(aStream)` serializes a cell putting `null` in fields of non-owned objects. `serializeByModulate(aStream)` serializes a cell and requests that all non-owned objects be modulated by their owner (by calling `o.getOwner()` ←`modulate(o)`). All of these operations fail if the cell has any active connections at the time of serialization.

4.7 Cell Loading and Unloading

A cell is loaded into a CVM from a `.cell` file. Loading is accomplished by the built-in service operations `load(aStream)` and `loadWithoutCid(aStream)`, differing on whether a fresh cid is assigned to the cell upon loading or not. `load(aStream)` loads the cell into memory. If the stream contains a cid, it is assigned to the loaded cell; an exception is raised if there is already a loaded cell with that cid. If the stream is anonymous, i.e., does not contain a cid, `load()` generates a fresh one for it. `loadWithoutCid(aStream)` loads a cell into memory, generating a fresh cid for the cell even if `aStream` contains a cid. Loading a cell entails loading the cell's classes, the objects owned by the cell, and the modulation table of the cell from the `.cell` file into memory. Once the cell is loaded, the cell's hook `onLoad()` is invoked if present. If a `.cell` file with cid is loaded via `load(aStream)`, other cells who knew this cell's cid may immediately begin interacting with it now that it is (again) loaded.

Cells may also be unloaded, which means removing them from the CVM, while possibly serializing into a stream. Built-in operation `unload(aStream)` unloads `thisCell`. If `aStream` is not `null`, the cell is also serialized onto `aStream` by invoking built-in operation `serialize(aStream)`. A Cell may be unloaded only if it has no active connections. So, `unload` first `unlink`s any linked connections. Before a cell is unloaded, the hook `onUnload()` is run.

Unloading a cell entails also unloading all objects owned by the cell. If an object that is loaded holds a reference to an object that has been unloaded, the object appears in a *zombie* state: all object access results in an "object unloaded" exception being thrown. Zombie objects cannot be restored; if the cell is re-loaded its objects are restored, but at new locations. A cell may however hold *modulated* references to objects in unloaded cells, and these references will function upon re-loading of the cell, even if at a different location.

4.8 Cell Linking and Unlinking

Linking of cells is an operation which is not implicitly implied by loading. In order for two cells to be linked, each must already be running in some CVM. Linking can be either an intra- or inter-CVM operation, *i.e.*, the two cells being linked may be in

different CVMs. intra- and inter-CVM linking is similar; here we define intra-CVM linking, and Sec. 5 describes how inter-CVM linking differs.

The cell linking protocol is as follows.

1. Cell connection is initiated by the atomic syntax `link` *`linkee`* `at` *`aConnector`* being invoked from within the *linker* cell. We use the terms *linker/linkee* to distinguish the two parties involved.
2. Multiple connections can be established at a given connector. However, the connection attempt aborts if as a result of the linking there would exist a plugin with more than one connected plugout (i.e. a plugout can be linked to multiple plugins, but a plugin must be linked to a unique plugout.).
3. To verify the connection passes security checks, the `verify` predicate of the linkee is first run, and if it succeeds, the `verify` predicate of the linker is run. Both the linker and the linkee have a chance to refuse the linking via the `verify` predicate.
4. If both `verify` operations succeed, the plugin/plugout connections are made, the `onConnect()` operation of the linkee connector is run, followed by the `onConnect()` operation of the linker connector.
5. Lastly, references to plugin classes, operations, and interfaces on either side of the connection are resolved to refer to the connected plugout on the other cell and the `link` command is complete.

The cell link protocol is asymmetric; for a perfectly symmetric model where both cells ask for the connection (as in e.g. process algebra synchronization) each cell must have its own thread in which each can simultaneously request a connection. Since each additional thread complicates the architure, we do not assume a thread-per-cell model. So, one cell—the linker—initiates the process and the other—the linkee—responds.

A connector may have more than one cell linked to it, *provided* item 2. above is not violated. This supports reuse of a library-style cell with only plugouts by multiple cells. Cells are unlinked by the JCells command `unlink` *linkee* `at aConnector`. This first runs the `onDisconnect()` hooks of the linkee and linker in turn and then un-resolves the references to plugged-in operations. So, after disconnection is complete, any reference to plugin operations on the disconnected connector will raise an exception. Classes that are unplugged will also get an exception upon `new`.

Class loading and linking In Java, the loading of classes is performed by the class loader. Because a run-time type in Java is determined by the combination of class loader and class name, class loaders introduce multiple name spaces into Java ([LB98]). This way of managing class names and their actual code is low-level and is not related to the structure of the system at hand.

In JCells, on the other hand, there is only a cell loader, and not a class loader: classes are loaded as part of some cell, which acts as its complete namespace. Different cells may have different internal definitions of the same-named class, which causes no conflict. Classes also can be shared between cells by explicit links which plug in/out the classes. Cells thus hold classes both statically, by their definition within a cell, and dynamically, by how class names are resolved at run-time.

Type consistency across cells As explained above, different cells may have different interpretations for a given class identifier. This is a problem when an instance of class A is copied from one cell to another one, which has a different notion of A. This problem is analogous to an RMI A-instance being sent to another location in the network, where there may be a different A. This inconsistency will be detected at runtime similarly to how RMI handles type compatibilty: the receiving cell matches its type against a type hash value carried by the copied object.

4.9 Parameter passing and modulation

Each object parameter or returned value on a method or operation can be passed in one of three ways:

1. by hard reference: the usual and the default case of Java.
2. by `copy`: the object and all objects it refers to, transitively, are copied.
3. by `modulate`: the cell holding the reference modulates it and passes this modulated reference; modulated references are defined in Sec. 4.4.

A `copy` parameter is deeply copied at method/operation invocation time. Copying is similar to serialization, but does not stop at non-owned objects.

5 Distribution

JCells is explicitly designed to support distribution of cells; in particular, CVM processes may be running on different nodes on a network, and cells in one CVM may directly hold references to cells in other CVMs. Additionally, cells may be copied or moved across the network, cells in different locations may be `link`ed across the network, and cells in one location can hold (modulated) references to objects in other locations. In what follows, we use the terms *CVM* and *location* interchangeably.

Each cell is universally addressable, and has a *home CVM*—the CVM that initially loaded it and generated its cid. We can elaborate on cids in this context: a cid is a bit sequence which is a pair `(CVM_locator,id)`, where `CVM_locator` globally identifies a CVM, and `id` identifies a cell with that CVM as its home. The `id` portion of a cid is random and "long enough" to make it, and thus cell identifiers, practically unguessable. For example, a cid of a cell in cvm1 in machine.cs.jhu.edu could have cid

```
(cvm1.machine.cs.jhu.edu,203483204835591720340548553 2639)
```

Within one cell, a `Cell` variable holds a cid which may either be local, *i.e.*, refers to a cell at home on the executing CVM, or remote, i.e., refers to a cell which has its home on another CVM, which in turn may be on another machine. Built-in service `home()` may be used to dynamically determine any cell's home CVM. Every CVM has a running thread for handling remote cell operations. The protocol for linking to and invoking services on remote cells is the same as the local in-CVM protocol except for the issues which we now outline.

Inter-CVM Connections One of the main points of the cell `link` protocol is that it can be used to persistently link two cells in different locations on the internet. As such it is a high-level analogue to `ftp` and other persistent socket-based internet protocols. An *interlocation* connector is one which is suitable for interlocation, that is inter-CVM, linking. Not all connectors are so suitable: hard object references are not sensible as parameters between CVMs, and so the connector must not contain such references. Additionally, we disallow classes being plugged in across CVMs: code for an object must be local, so a remote class reference is not sensible.

An *interlocation* connector is thus a connector with the following properties:

1. It has no class plugins or plugouts.
2. All non-basic parameters on operation plugins or plugouts are either `copy` or `modulate` parameters; similarly for return values.

An *interlocation* service is one in which all non-basic parameters are passed by `copy` or `modulate`. `copy` object parameters may be serialized to the remote CVM, and modulated parameters may be sent as `(cid,oid)` bit sequences.

The semantics of interlocation linking is the same as intra-location linking, except the connectors must first be verified to be interlocation connectors. This check is dynamic because `cell MyCell aCell` may either be local or not; this information is not declared in the type and the cell could have in fact initially been local and later moved to another CVM.

Loading and Cell Movement Cells can be moved between locations by `unload`ing and then re-`load`ing them in a different location. In order for references to the moved cell to be transparent, the home CVM must be notified upon re-load at a new location. It then forwards any reference to that cid to the CVM where the cell is currently loaded. The home CVM is again notified if the cell is later unloaded. The home CVM is thus responsible for tracking the cell, wherever it might be loaded, throughout its lifetime.

Security In a distributed setting, security is of particular concern. Cells were designed from the beginning as units on which security policies can be defined; this was in fact one of the inspirations of the design. Classes and objects are arguably too fine-grained for every class and object to hold an advanced security policy; components and modules are just code and directly support code-level security only.

Recall that cells mutually authenticate each other at link time via `verify` clauses, and the linking succeeds only if both sides verify the connection is legitimate. Verification is particularly important for inter-CVM linking. Cells are also secured by cid: cid's are unguessable and so cells know only other cells they have been explicitly told about. Finally, remote object references in JCells take the form of modulated references, which are dynamically revokable. These principles are however only the infrastructure on which a detailed security policy is to be placed. We are currently developing a complete policy built on the SDSI/SPKI open standard [EFL+97], where each cell is a SDSI/SPKI principal. Under any policy it is still unavoidable that some trust must be placed in the CVM by cells running on it. If the CVM has not itself been tampered with, integrity of individual cells running on the CVM is guaranteed.

6 Conclusions

Contributions This work makes a number of contributions, which we now summarize. We unify the compile-time and run-time notions of module/component: modules are well-structured at compile time but largely disappear at run-time, whereas components have relatively less structure at compile-time but have clear interfaces at run-time; cells importantly have both a strong compile-time and run-time presence. Module principles of code reuse are also preserved by cells: a class in one cell may inherit from a class it has plugged in.

Java allows some programmer control over loading via class loaders; we complete what Java started by separating loading from linking and by putting both fully under high-level programmatic control. Cells support unlinking, a function absent from module systems. We extend the notion of module linking to a network context by allowing two cells running on different network nodes to link with one another.

Cells give an excellent foundation for a new security architecture because each cell itself is sensibly a principal to be secured; in particular, cell reference and cell linking are both privileged operations which may be restricted in JCells. The concept of a modulated reference allows for security-controlled access to objects in other, possibly remote cells.

We introduce the concept of multiple object spaces within a single virtual machine at run-time by requiring each object to be "owned" by a particular cell; this helps focus policies of security and persistence. We work persistence more directly into the design by having a standard definition of how cells and their objects may be serialized.

We believe the above elements in the cell architecture result in a more parsimonious language, in which modular Internet programming is supported as part of the core design. In particular, Java's RMI, `ClassLoader`, security manager, packages, Java Beans, and applets are not needed in JCells because cell functionality subsumes their responsibilities: cell references and modulated references replace RMI (and cell nameserver `lookup` replaces the RMI registry), cell loading replaces class loading, the cell security policy (currently under development) will replace the Java security manager, cells replace Java Beans and applets, and cells are also used in place of Java packages. The cell parallel of java package importing is future work.

Implementation The design outlined in this paper clearly needs to be implemented to verify its soundness and workability in practice. So far we have implemented a limited prototype [Lu02] on top of an unmodified JVM. We have implemented cells with connections, a `.cell` file format with an XML manifest, loading of cells from `.cell` files, and dynamic linking and unlinking of cells. We have yet to implement ownership of objects by cells, serialization, modulated references, and distributed cells. Reflection is needed to implement plugin operations. A more complete implementation of JCells is currently in progress.

Future Work This is a large project and we were forced to leave out topics that were not critical to the core architecture. A full cell security policy is currently under development. An explicit protocol for sharing cell signatures is needed, possibly via

cell signature nameservers. This will provide the Java-like package import functionality for cells. Currently we lack any notion of cell version control, a critical issue for component systems.

Acknowledgements The authors would like to thank the ECOOP reviewers for helpful comments on this work.

References

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: Connecting software architecture to implementation. In *ICSE 2002*, 2002.

[BNOW93] A. Birell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *14th ACM Symposium on Operating System Principles*, pages 217–230, 1993.

[BR00] Ciaran Bryce and Chrislain Razafimahefa. An approach to safe object sharing. In *OOPSLA*, pages 367–381, 2000.

[BV01] Ciaran Bryce and Jan Vitek. The javaseal mobile agent kernel. *Autonomous Agents and Multi-Agent Systems*, 4:359–384, 2001.

[Car95] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995.

[Car99] Luca Cardelli. Abstractions for mobile computation. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 51–94. Springer, 1999.

[CDG+88] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report. Technical Report 31, Digital Equipment Corporation, Systems Research Center, August 1988.

[EFL+97] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. Simple public key certificate. Internet Engineering Task Force Draft IETF, July 1997.

[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.

[Gro01] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, revision 2.5 edition, September 2001.

[HCC+98] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *1998 USENIX Annual Technical Conference*, pages 259–270, New Orleans, LA, 1998.

[JLH90] Eric Jul, Henry Levy, and Norman Hutchinson. Fine-grained mobility in the emerald system. In *Readings in Object Oriented Databases*, pages 317–328. ACM, 1990.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. In *OOPSLA '98*, pages 36–44, 1998.

[LO98] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[Lu02] Xiaoqi Lu. Report on the cell prototype project. (Internal Report), March 2002.

[MFH01] Sean McDirmid, Matthew Flatt, and Wilson Hsieh. Jiazzi: New age components for old fashioned java. In *OOPSLA 2001*, 2001.

[Sun] Sun Microsystems. Java RMI. `http://java.sun.com/products/jdk/rmi/`.

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

[Ung95] D. Ungar. How to program self 4.0 a guide to the programming environment, June 1995. `http://research.sun.com/self/release/documentation.html`.

[US87] D. Ungar and R. Smith. Self: the power of simplicity. In *OOPSLA 1987*, pages 227–241, 1987.

[VC99] Jan Vitek and Giuseppe Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, 1999.

[vDABW96] L. van Doorn, M. Abadi, M. Burrows, and E. Wobber. Secure network objects. In *IEEE Symposium on Security and Privacy*, May 1996.

[VT97] J. Vitek and C. Tschudin. *Mobile Objects Systems: Towards the Programmable Internet*, volume 1222. Springer-Verlag, Berlin, Germany, 1997.

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor. *Communications of the ACM*, 17(6):337–345, 1974.

[Whi94] J. E. White. Telescript technology: The foundation for the electronic marketplace. White paper, General Magic, Inc., 2465 Latham Street, Mountain View, CA 94040, 1994.

[ZM90] S. Zdonik and D. Maier. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, San Mateo, CA, 1990.

A Built-in Protocols

Cells have built-in services and operations for important functions, and the `Object` protocol is also enriched in JCells. In this Appendix we summarize the protocols.

Built-in services Cells have several built-in services to support system functions. This is in analogy to class `Object` public methods. Several of them have been already described above. Here we list the full metaprotocol available.

- `cell Cell clone()`—returns of a copy of the cell.
- `cell Cell modulate(Object o)`—returns a modulated reference to `o`.
- `copy CVM home()`—returns the home CVM of the cell.
- `copy CVM location()`—returns the current CVM where the cell is located.

Built-in operations Some in-cell built-in operations are also supported. These may not be invoked by outsiders, i.e. they are not services.

- `Cell cell [] cellsAt(String aConnector)`—returns the set of cells connected at `thisCell`'s `aConnector`.
- `cell Cell invoker()`—the cell which invoked the currently executing cell operation.
- `cell Cell revokeModulationOn(Object mo)`—removes modulated reference `mo` from `thisCell`'s modulation table.
- `cell Cell load(aStream)`—cell load preserving the cid
- `cell Cell loadWithoutCid(Stream s)`—a cell is loaded off of stream s, ignoring cid.
- `void unload(Stream s)`—`thisCell` is serialized to `s` and then unloaded from the CVM. If `s` is `null`, the cell is just unloaded.
- `void serialize(Stream s)`—serialize `thisCell` unto the stream `s`; variants `serializeWithNull(Stream s)` and `void serializeByModulate(Stream s)`.
- `cell Cell lookup(String s)`—lookup a cell via name server.
- `cell Cell register(String s)`—register this cell with the name server as `s`.

Supported hooks The following hooks are supported by the CVM:

- `onUnload()`—This operation, if it is defined, is invoked just before a cell is unloaded.
- `onLoad()`—This operation, if it is defined, is invoked immediately after a cell is loaded off a stream, and can for instance be used to re-link the cell to libraries or other services.

Object Protocol There are also a few additional object protocols, messages which all objects in a CVM respond to.

1. `cell Cell o.getOwner()` returns the cell owning object `o`.
2. `Boolean o.isModulated()` returns true iff `o` is a modulated reference.
3. `Boolean o.isZombie()` returns true iff `o` is a zombie.