Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Bachelor's Program in Computer Science

Bachelor's Thesis

A Sandboxing Infrastructure for Alice ML

submitted by

Andi Scharfstein

on October 31, 2006

Supervisor

Prof. Dr. Gert Smolka

Advisor

Dipl.-Inform. Andreas Rossberg

Reviewers

Prof. Dr. Gert Smolka

Prof. Dr. Andreas Zeller

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, October 31, 2006

Acknowledgment

I would like to thank a number of people for their support during creation of this thesis. To begin with, I am grateful to Andreas Rossberg and Prof. Gert Smolka for offering me this thesis. Andreas deserves special praise for his honest and highly competent evaluations of my thesis, which he always managed to convey in a gracious and friendly manner – a set of qualities which coincide all too rarely.
Thanks go to Georg Neis for offering most helpful advice right when I needed it, to Martin Schreiber for giving many amusingly individualistic (and moreover, valuable) comments on the thesis, and to Jan Christoph for letting me savour the pleasure of looking forward to receiving his opinion for far longer than anyone else did.
I also want to thank my family and my friends for their understanding and patience during the period when I devoted more time to the writing of this thesis than to interacting with them – unfortunately, much as I should have liked to avoid that situation, I did not manage to.
Last but not least, I wish to thank Melina Wack for putting up with me during this time. Thank you for showing more discipline than I had, and for sharing it with me!

Abstract

The essence of open programming is the ability to import code at runtime, in order to allow applications to adapt their behaviour and functionality dynamically. However, when code is acquired from untrusted sources - e.g. some remote Internet domain - security concerns arise. Untrusted code should not be given unrestricted access to local resources. A well-known solution is to run untrusted code in a sandbox, a software environment which dynamically checks all critical operations performed by the code according to some configurable security policy. Java is the most popular language employing this approach.

In this thesis, we develop a model based on an idealised subset of the ML programming language to describe the sandboxing approach. After gaining an understanding of the underlying principles of a sandbox, we show how to apply these to get a working implementation for the Alice ML programming language. Alice ML is a dialect of Standard ML that has been specifically designed to support type-safe open programming. It provides a generic and strongly-typed import/export facility (pickling), which allows processes to exchange or make persistent almost arbitrary language-level data structures and code. For security reasons, resources are precluded from pickling. Instead, it is left to the target site to supply them explicitly to imported code. A flexible notion of component allows respective abstractions to be programmed conveniently. We show how these particular characteristics of Alice ML can be used to create a simple, yet powerful, sandbox design with an emphasis on modularity and extensibility.

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Sandboxes . . . 3
- 1.3 Alice ML . . . 4
- 1.4 Outline . . . 4

2 Important Concepts **5**
- 2.1 Resources . . . 5
- 2.2 Modules and Components . . . 5
- 2.3 System Components . . . 6
- 2.4 Component Manager . . . 6
- 2.5 Access Control . . . 7

3 The Sandbox Model **8**
- 3.1 Resources . . . 8
- 3.2 Modules . . . 8
- 3.3 Components . . . 9
- 3.4 URLs . . . 10
- 3.5 Component Manager . . . 10
- 3.6 Design Space . . . 11
- 3.7 Policy Rules . . . 12
- 3.8 Fundamental Sandbox Design . . . 12
- 3.9 Flexible Policy Checking . . . 16
- 3.10 Policy Rules Revisited . . . 17
- 3.11 Dealing with Rejection . . . 19
- 3.12 Efficient Component Substitution . . . 20
- 3.13 Default Policies . . . 22
- 3.14 Putting it all together . . . 22

4 Implementing the Model **26**
- 4.1 Component Managers . . . 26
- 4.2 Unsafe Resources . . . 29
- 4.3 Design Aspects . . . 31
  - 4.3.1 General Architecture . . . 31
  - 4.3.2 Sandbox Interface . . . 32
  - 4.3.3 Policy Interface . . . 32
- 4.4 Sandbox Usage . . . 35
- 4.5 Sandbox Implementation . . . 37

5 Related Work **41**
- 5.1 Java . . . 41
  - 5.1.1 Resources . . . 41

5.1.2 Policies 41
5.1.3 Component Manager 42
5.1.4 Access Control 42
5.1.5 Certificates 43
5.2 The E Programming Language 43
5.3 Perl 6 44
5.4 Microsoft .NET CAS 45

6 Conclusion **47**

1 Introduction

1.1 Motivation

In the last decade, no technological change had a greater impact on the development of programming languages than the advent of the internet. Modern languages are expected to embrace interconnectivity and deal with the complex issues created by an open environment, even while providing a maximum amount of security. The last point in particular has received a steadily rising amount of attention, since any security leak in a networked environment is a leak to any and all participating computers. In the (not entirely unlikely) case that the machine is connected to the internet, surely a disastrous event.

But even for stand-alone machines, security concerns of a new kind arise in the dynamic contexts created by modern languages: Alice ML [15] for example permits dynamic linking of components to provide new functionality for a program at runtime [16]. What if malicious code downloaded from the internet is accidentally linked against a dangerous system operation? Imagine, for a moment, an insecure browser application that granted applets from any visited site full access to the local file system. The inevitable consequence would be instant abuse: File deletion, acquisition of sensitive data, planting of viruses or trojans... a whole realm of security nightmares. Clearly, there is a need for systems preventing this and other undesirable scenarios from occuring.

1.2 Sandboxes

Spearheaded by Java, the solution that most languages have adopted to address this issue is the so-called *sandbox*.[1] The sandbox model aims to provide a "safe environment" for untrusted code that manages and potentially restricts access to critical system functions, according to some policy guidelines [5, 3].

In that model, the basic idea is to regard applications as heterogeneous compositions of *modules*, where modules can be seen as self-contained chunks of code. Some "main" module serves as a host for the other modules, and provides a "sandboxed" environment for their safe execution. The heterogeneity enters with the actions the different modules can take: While the main module has virtually unlimited power, the hosted modules may be severely restricted in the operations they may perform. As an example, take the browser/applet paradigm popularized by Java. The browser application (the main module) may execute native system code, while the applets (the hosted modules) are restricted to the use of a few selected operations (e.g. drawing an image or performing arithmetic). In this scenario, the system that enforces these restrictions is called the sandbox, and the choice of (dis-)allowing a certain operation is made by accordingly defined *policy* rules.

[1] Besides Java, systems such as Perl 6 [2] and Microsoft .NET choose a sandbox paradigm to ensure safe code execution. Microsoft calls this "Code Access Security" [10], but the underlying model is the same.

1.3 Alice ML

Alice ML [15, 1] is a functional programming language that has been designed to specifically support *typed open programming*. Building on top of Standard ML [12, 14], it incorporates features such as concurrency with data flow synchronisation, higher order modularity and dynamic modules, as well as type-safe pickling. Given its strong inclination towards open programming, it seems vital to also include a facility for the safe management of dynamic components. Indeed, that's the goal of this thesis: To design and implement such a system for Alice ML, in the form of a sandbox.

1.4 Outline

This thesis is organised as follows: First, we try to clarify some ideas about what we are trying to achieve. To this end, we introduce the concepts integral to this thesis in an intuitive way, before going over them again in a more concrete manner. This second chapter is meant to give the reader some context, in order to facilitate understanding for the choices made in subsequent chapters. The third chapter makes these concepts precise in a coherent way, using an untyped pseudo-language based on ML as a modelling tool. In the fourth chapter, we show how to transfer that model to a real-life implementation of the design. We conclude in the fifth chapter by drawing some analogies between our model, our own implementation, and other designs and implementations dealing with the same topics. Finally, chapter six gives a summary of the work done, as well as some ideas for future work.

2 Important Concepts

Our foremost goal is of course to achieve some measure of *security* for our language of choice. Hence, we have to define what we take this to mean. As stated before, security is achieved by the language's ability to limit the set of operations a module can perform. In that sense, it can be argued that by security we mean the *guaranteed absence of unsafe operations* from the pool of operations the module has access to. Consequently, we will also need to take a look at some significant features of Alice ML and try to identify the unsafe ones, as well as potential ways to make them safe again. We'll go about this in a bottom-up fashion, starting with the elementary concepts.

2.1 Resources

One of the most fundamental concepts in this paper is the notion of a *resource*. Simply put, a resource is any part of a computational device that can be used by a program over the course of its operation. This includes hardware entities local to one machine such as mass storage and random access memory, abstractions over the same (e.g. computation capacity provided by the CPU), as well as media with non-local implications, such as the bandwidth provisions granted by a networking device. In fact, most resources are abstractions (or *virtual resources*), such as file handles, sockets or thread ids. The common factor among these is that each can be abused by malicious programs if unlimited access to the resource is given[2]. Hence, we can narrow our definition of safety: A module is unsafe exactly when it can exert unrestricted access to system resources. On the other hand, if all resource access methods of the module are controlled by its environment, we say that the module is *safe*.

2.2 Modules and Components

If programs can be seen as compositions of modules, as we claimed in the introduction, then Alice ML must possess a notion of what exactly a "module" is. This is indeed the case: A module in Alice is in fact nothing more than a datatype that can be used to group a set of values (including submodules) and types into one logical entity. One possible reason to do this may be that the values belong to a common operational theme, as in the case of the `Http` module that groups together some high-level socket functions; however, most of the time, the module will define an implementation of some abstract type together with its basic operations (like `Set` and `Set.union`, or `List` and `List.filter`, `List.map`, `List.exists`, etc.).

While modules provide logical separation, *components* are Alice ML's means for physical separation. Intuitively, they allow partitioning of programs into self-contained units of code (think "files"), which can be compiled on their own and then of course be recomposed to build complex programs – hence, their name. All components are built according to a certain principal structure: First, there is a set of import requests for other components, which are dealt with by the component manager. And second, a

[2] A very nice definition of resources, which sadly extends beyond the scope of this thesis, is given in [6].

listing of one (or possibly more) module definitions that make up the main part of the component. Hence, modules are *contained* by components. In case a component contains exactly one module (an important special case), this component can be thought of as the compiled representation of that module.

Components can also be created dynamically, by "wrapping" modules at run time. Components created this way are called *computed components*[3].

2.3 System Components

In Alice ML, resources and components have a special relationship. Resources are always encapsulated in a particular kind of component, the so-called *system component*. System components provide access methods specific to the resource (for instance, many I/O components have a `write` method or some equivalent thereof), which programs have to use in order to work with that resource.

While system components already stand out as the sole means for resource acquisition, they are special for another reason as well: Since they contain references to media local to one machine (or *site*), they are also referred to as *sited components*. The implication of this is that system components can never be "exported" to other machines, as would be possible with regular components[4]. Obviously, a resource cannot be fabricated out of thin air, so another fact to note is that a program can never dynamically "create" system components. Instead, to obtain a system component, one must **always** *import* it using the language facility provided for this purpose – the so-called *component manager* (see below). Hence, it is evident that system components provide a solid abstraction to reason about resources. We rely on this fact in the creation of our security infrastructure.

By importing a system component, any process can gain access to potentially unsafe operations. In general, if a reference to such an operation is obtained, it is called a *capability* [4], in that the process is then capable of using it.

2.4 Component Manager

For the extraction of modules from a component, one needs a facility for managing the import part of the component (that is to say, the "first" part). In Alice ML, this facility is called the *component manager*. It performs dynamic linking of these imports, which may just mean the loading of a few definitions into local scope – in the case of a regular component – or additionally the acquisition of local resources, in the case of a system component. As we have already seen, controlling access over these system components gives us a handle to control access over potentially unsafe operations, so component managers might be the appropriate tool for this goal.

However, programs are free to create their own component managers by calling the `MkManager` functor (which is basically a function that creates modules) provided by the

[3] Of course strictly speaking, every component has to be computed at some point in time, regardless if by interpreter or compiler.

[4] In Alice ML, this operation is called *pickling*. Other common terms with similar connotations are *marshalling* and *serialisation*.

standard library. Does this mean that component managers do not support the degree of access control required by a sandbox, since they can be circumvented so easily? Thankfully not, as component managers have the desirable property of *restriction inheritance* (even along link chains[5]), which is to say that once a restriction is in place in a component manager, it can never be relaxed by a component manager created inside the more restrictive parent manager. Of course, it can always be made even stricter, but this does not imply any safety concerns.

2.5 Access Control

So far, we have seen that programs are unsafe only if they get unrestricted access to system resources. Resources are always encapsulated by system components, so the access restriction problem for resources can be reduced to one for components. Alice ML natively supports access restriction for components by its component manager system. Thus, we arrive at the conclusion that to meet our original goal of creating a security architecture for Alice ML, all we really have to do is provide a suitably modelled component manager. In this context, such a component manager *is* a sandbox. In the next two chapters, we are aiming to model and implement it.

[5]That is, when several component managers link each other successively.

3 The Sandbox Model

Now that we have gained an intuitive understanding of what we are trying to achieve, we shall make things a little more precise. We develop a model for sandboxes in an idealised subset of ML, which will be easier to understand than the full-blown version of Alice ML used for the implementation. In particular, we choose to work in an untyped context, since, as we shall see, types are irrelevant for our intended course of action (as we must rely on dynamic checks rather than use the static type system). However, we shall sometimes talk about the types of certain constructs in our system. We mean by these the intended types, i.e., the types those constructs would have in a typed context, and hope to evoke a better understanding for what we are trying to achieve by indicating them. However, first things first: Let's iterate over the concepts we already know and see how they can be modelled.

3.1 Resources

Resources are important, as they are the basic building blocks in our access control scheme – basically, a resource is the smallest unit for which one can sensibly define some kind of access control. However, we want to abstract away from differences between individual resources (e.g. file handles vs. sockets), so a sophisticated representation of the various resources would not be of merit. We take the opposite route and define a resource to be just a constant of our model language. That is, for every resource we want to control, we introduce a constant denoting its representation, bearing names like e.g. `stdIO` or `readSocket`. This approach has the advantage that we do not need to care about the different properties of resources, but can focus on how to control their use in a unified manner. Together, the individual constants make up the set of all possible resources, which we call Res. Very importantly, we do not allow modules to just "write down" the denotation of a resource and use it like this – this would defy the whole concept of a sandbox such as we are trying to create. Instead, resources must always be acquired by importing the system components that contain them. This reflects the real situation as witnessed in Alice ML.

3.2 Modules

Complex as they may appear at first glance, modules are really nothing more than named collections of types, functions and values. Functions are first-class values in our language, so this further reduces to a named collection of types and values[6]. However, as we are working in an untyped context, we can just drop the types. Accordingly, modules are modelled as tuples of values, e.g. $(val_1, val_2, ..., val_n)$. We require that no value may occur more than once in a module tuple, so we can introduce the shorthand notation `M.f`. This means the (now uniquely identifiable) element `f` of the module tuple `M` (thereby matching the syntax of Alice ML). For example, if we had

[6] In a slight aberration from reality, we include modules in our definition of values, which is not the exact truth in Alice ML. However, modules can now contain submodules, which is in fact accurate.

a module `TextIO = (stdIO, read, write)`, we could use it in the following way: `TextIO.write (TextIO.stdIO, "Hello world!")`. However, there is a slight twist. In order for `stdIO` to be part of `TextIO`, we must extend the definition for module tuples so they can contain resource designators as well as "real" values. The designators are instantiated with "real" resources provided by the local system during the import phase of linking a component (which we have seen in chapter 2.2, cf. chapter 4.1 for a more detailed discussion). This means that `TextIO` could not have been created by the program itself, but must have been imported by a component manager at some time during its execution. We will largely ignore this issue after mentioning it here in order to ease readability of subsequent chapters, but keep in mind that resources do not just "come for free".

3.3 Components

As explained in the previous chapter, components carry at least one module, or possibly more. For our model, we assume that the latter is never the case, i.e. we are confining ourselves to the special case mentioned before that a component is just the compiled representation of exactly one module. Still, every component needs to be evaluated by some component manager (which is also the one it will subsequently use to load other components itself, if the need arises). Hence, components are modelled as functions that take a component manager as input and return their enclosed module. For example (if we assume that component managers are functions that return modules):

```
fun TextIOComp (componentManager) =
    let
        UnsafeIO = componentManager (x-alice, "/lib/system/UnsafeIO")
    in
        (UnsafeIO.stdIO, UnsafeIO.textRead, UnsafeIO.textWrite)
    end
```

Figure 1: The TextIO component

The `let` part symbolises the import requests mentioned in the introduction, while the return value is simply a module that repackages the relevant parts of `UnsafeIO` for this component. Incidentally, this is exactly the aforementioned "import" part where the process of *linking* happens – and thus, resource acquisition by the component manager[7]. The import part uses the URL denotation of the location of `UnsafeIO`, which we will understand in a second.

[7] The astute reader may have noticed that for one import, the whole transitive closure over all component imports is computed. However, Alice does this lazily [13], so the overhead is not as substantial as might be expected.

3.4 URLs

For our purposes, it suffices to regard URLs as a structure of two concatenated string parts: The first part designates the URL scheme (e.g. "http"), the second part the location of the resource (e.g. "www.ps.uni-sb.de", "/lib/system/TextIO"[8]). Then, URLs are just tuples of (*scheme* × *location*), where *scheme* is one of three constants as defined below, and *location* is just a string. URLs can be used in a program to denote files, system components or web sites – it just depends on the scheme that is used.

These are the three different kinds of schemes we allow, which correspond to the alternatives supported by Alice ML:

file Using the "file" scheme implies that the location part gives a filename in relative or absolute notation, e.g. "../HashTree.alc" or "C:/Alice/lib/data/Stack.alc".

x-alice This scheme is used for library components already provided by the Alice system. It is of special importance, since this is the only possibility for a program to acquire system components (i.e., library components are the only source of potentially unsafe operations).

http The well-known hypertext transfer protocol used to transmit web pages across a network. It will play a very subordinate role in this thesis, as we mainly use the *x-alice* scheme, but will be useful for some examples. Still, it should be mentioned that in principle, non-system components can be retrieved over HTTP just as easily as via the *x-alice* scheme.

3.5 Component Manager

Component managers present a difficulty in that they can be modified at runtime, though only to be more restrictive. This means that they cannot be modelled as constants, but rather have to be some kind of dynamically computed value. As the main purpose of a component manager is to link components (i.e., given some kind of input locating the component, it should return the linked component as output), the proper way to model it is as functional value. This way, it can take a URL as argument and return the respective component located at that URL. But this is still a little more complex than what we really need. Our restriction that components always carry exactly one module allows us to turn the returned value directly into this module. Since "turning a component into a module" for this model amounts to applying the component to any component manager, we use the very component manager in whose context we are operating to do this. This is a bit reminiscent of recursive functions, since the component manager passes itself as an argument in the course of its definition. If we view it that way, the base case is reached whenever an imported component does not need to import any additional components (either because it is self-contained, or all necessary imports have been loaded previously and can be provided from cache).

[8] A local file path.

We assume that there always exists a constant function of the component manager type (*Url* → *Module*), which we call the *root manager*. The root manager plays a special role, as it is needed for bootstrapping. It is used whenever no other manager has been instantiated, but a component must be linked. Hence, it stands at the top of any component manager inheritance scheme.

A very simple component manager could look like this (assuming `SocketComponent` and `HttpComponent` are definitions of the according components):

```
fun componentManager (inputUrl) =
    if inputUrl = (x-alice, "/lib/system/Socket")
    then SocketComponent componentManager
    else
        if inputUrl = (x-alice, "/lib/system/Http")
        then HttpComponent componentManager
        else raise "inputUrl not found!"
```

Real-world component managers load their components from file rather than hard-coding their definition (besides doing lots of other things in a different way, obviously – for example, caching).

Programs We are now ready to understand program execution as well: It is a mere application of the root manager to the location of some component. This component can be thought of as the *root component* or, maybe slightly more familiar to some readers, the *main component* of the application (as in the `main()` function of C/C++ and Java).

3.6 Design Space

The basic concepts are now in place, so we can begin to think about how a sandbox might fit into this design. As we have already seen, it is sufficient to implement our own component manager, but how should its semantics look like? Let us first explore several possibilities, to get a feel for the dimensions we have to balance.

Degree of Control Remember that the stated purpose of our sandbox is to restrict access to certain resources. But which resources are considered safe, which dangerous? As this might vary from one program's context to another's, that information is considered subject to change, and it is the user's[9] task to supply it. The manner in which he can do this, however, is not at all clear: Should he be able to control every single resource access by itself? Such a fine degree of control might get very tedious to handle in larger applications. Or should he only be able to control access to one component at a time? Then it is likely that he will encounter situations in which this proves too inflexible. At any rate, a compromise between these two extremes will have to be found.

[9] In this case, that's the designer of the application hosting the sandbox.

Restriction Behaviour What do we want to happen when access is requested to a resource considered "unsafe"? The process requesting access could be terminated instantly, or we could be content with going on after warning the user that something bad might be happening. This may be another case where the application designer is in a better position to judge the proper reaction to some security error, so another alternative to consider is transferring control to the main application.

Static vs. Dynamic Checks Readers familiar with the ML family of languages know about the advantages provided by a strong static type system. It might be interesting to think for a minute about whether this system can be employed to construct a sandbox mechanism such as we have in mind, which restricts access to certain operations called in a specific context. Indeed, Alice ML could be made to support such a system, so we have to decide if this is the way to go, or if the ramifications of such an approach would make it impractical for real-life use. The alternative of course is inserting dynamic run-time checks at the proper locations, which is known to work, but of course more expensive than the static approach in terms of computation time.

3.7 Policy Rules

As we have seen, a method must be provided that enables users of the sandbox mechanism to define rules which govern access to all potentially unsafe operations. We call these rules *policy rules*. In the trade-off between fine-grained control and simplicity of use, we tend towards maximum flexibility - i.e., policy rules operate on the most primitive elements of our setup, on plain resources. Let's assume for a moment that for any resource, only a simple yes/no decision can be made with regard to access, independent of the intended usage of the resource. Then, it suffices to have one simple policy rule that performs this decision for all resources that can be accessed. We can model it as a function $Res \rightarrow Bool$, that returns for each resource the outcome of that decision. The user merely has to specify his decisions, which the function can then adopt. With this model, it is easy to construct a simple sandbox which shall serve to illustrate the basic idea of its inner workings.

3.8 Fundamental Sandbox Design

Choosing to utilize such a function for our sandboxing system entails the consequence of having to resort to dynamic policy checks instead of using the static type system. This means that we can disregard the latter approach; for our system, we just use dynamic checking. In particular, every method call requiring access to potentially unsafe resources will entail one or more calls to the policy rule function, giving as arguments these resources. If any of these calls should return FALSE, the operation is aborted and control given to the hosting application.

Now we understand the underlying idea for our sandbox component manager: Instead of giving any application unlimited access to all operations, it substitutes unsafe functions by functionally equivalent functions which, prior to committing any actions,

check if the necessary privileges are available. If they lack any of these, the functions immediately return without accessing the respective resources. In a sense, our substitute function transparently wraps the capability of the unsafe function call inside its own function, which can no longer be regarded as a capability since it does not necessarily grant access to unsafe features. The basic shape of a substitute function looks something like this:

```
fun safeWriteIO (text) =
    let UnsafeIO = componentManager (x-alice, "/lib/UnsafeIO")
        Policies = componentManager (x-alice, "Policies")
    in
        if Policies.policyCheck writeIO
        then UnsafeIO.writeIO text
        else ABORT
    end
```

(`ABORT` would likely be implemented by throwing some kind of exception.) In this case, we wrap the unsafe function `UnsafeIO.writeIO` inside a function named `safeWriteIO`. We could use this function to write a new component, `SafeTextIO`. Assuming that this component included the necessary checks for its other constituents as well, it would be used by a sandboxing manager in the following way:

```
fun sandboxComponentManager (inputUrl) =
if inputUrl = (x-alice, "/lib/system/TextIO") then
        SafeTextIO regularComponentManager
    else
        regularComponentManager (inputUrl)
    end
```

This illustrates two important concepts. First, *rewriting*: the sandboxing manager reinterprets the request for `TextIO` as a request for `SafeTextIO`, and serves it accordingly. We will use this technique throughout the course of our design. And second, *delegation*: when the sandbox manager does not have to deal with a request in a special kind of way, it just forwards this request to an ordinary component manager. Also note how `SafeTextIO` is passed the regular component manager as a *delegate manager* to use for its own imports, so it is able to load the unsafe component it needs to work correctly. It is important to understand exactly how this works, so refer back to figure 1 if necessary, to check how it uses the component manager given as argument to load all its imports (in the case of figure 1, that's just `UnsafeIO`).

Indirection Since rewriting is such an important concept, let's spend a little time looking at that mechanism in some more detail (Figure 2). We start at the application requesting access to some kind of unsafe feature. The request goes to the sandboxing component manager, which interprets it as a request for a safe component and loads this instead. The safe component in turn requests access to its unsafe equivalent from the

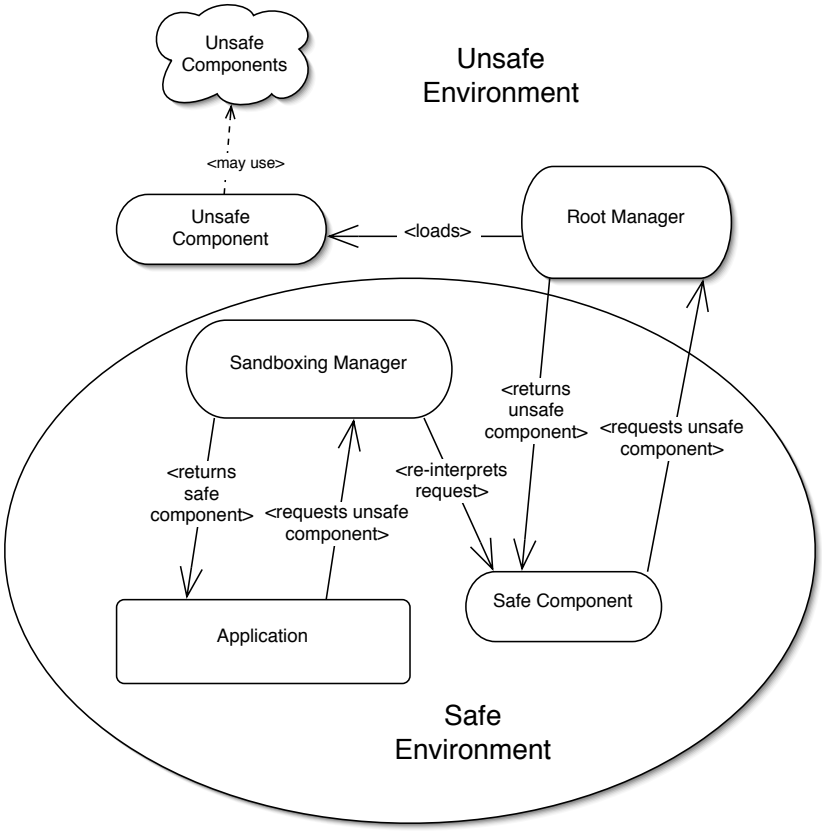

Figure 2: Indirection in component managers

root manager. That manager then loads and returns the unsafe component, which is used to build the safe component. After completion, it is forwarded by the sandboxing manager to the client application.

To be sure, the safe components can use any kind of unsafe feature (as they must, otherwise they would not be of any use) – it is just by the checks we insert that they become "safe". So, these components are trusted by default. It is the use of *indirection*[10] that enables us to nevertheless guarantee safety by using them: Inside the "safe environment" in which the client application is placed, there is no way to get access to a resource other than using a safe component, but from outside, the safe components are inaccessible. Thus, they cannot be corrupted in any way by a malicious user. This means that our trust in these components is justified, since they only access resources in a way we intended them to.

[10]The acquisition of unsafe components by way of the safe components, and not by the direct route.

Returning to our previous discussion, we can combine the two concepts of rewriting and delegation to make our sandbox manager even simpler:

```
fun sandboxComponentManager (inputUrl) =
if inputUrl = (x-alice, "/lib/system/TextIO") then
        regularComponentManager (file, "SafeTextIO")
    else
        regularComponentManager (inputUrl)
    end
```

Now, the definition of SafeTextIO component does not have to occur in the scope of sandboxComponentManager any longer – we just save it to a file and use the capability of any normal component manager to load it. This means that our concept figure must be updated as well: Figure 3 gives the new version. The main difference is that the rewritten

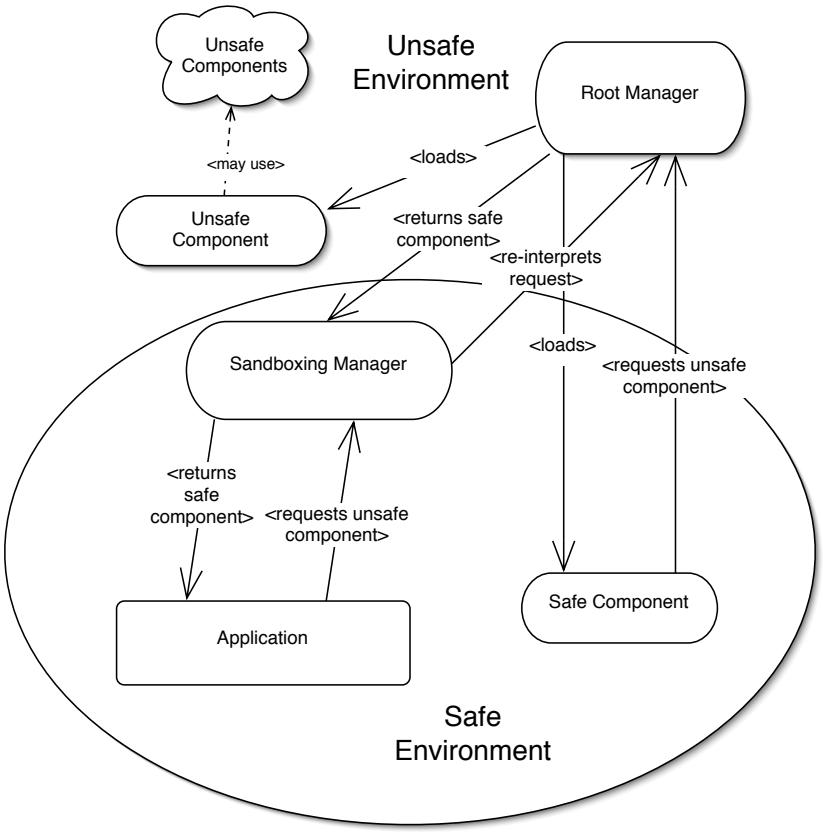

Figure 3: Indirection in component managers, with delegation

requests are passed to the delegate manager (in this case, the root manager), as we have

just seen in code. The substitute component is depicted inside the safe environment to stress that as part of the security infrastructure, it is still trustworthy. In particular, even though it is now implemented completely independent of the sandboxing manager, it remains a library component that can never be substituted by malicious code[11].

Note that using a "regular" delegate manager instead of the sandboxing manager for imports this substitute component performs becomes an absolute necessity, since there could be no import of unsafe system components otherwise. Stated differently: Decoupling the implementations of sandboxing manager and safe components to enable modular, "static" creation of substitute components as self-contained functional units requires usage of an unrestricted component manager for the allocation of unsafe components.

3.9 Flexible Policy Checking

So far, we have just seen a very simple kind of policy rule. Now imagine the following use case: In a browser application designed for a large corporation, the users should only be allowed to surf to work-related sites. If a user tries to access a website that was not cleared for access, he should be redirected to an internal error page that displays a warning. We might try to use a sandbox to implement this behaviour, but probably realize pretty quickly that we will not get farther than this:

```
fun safeLoadUrl (url) =
    if Policies.policyCheck loadUrl
    then loadUrl url
    else loadUrl (file, "warning.html")
```

Of course, this does not work, as our policy check can not know which URL is being requested. What we would like to do is incorporate handling of resource access *dependent* on its intended usage into policy checking – our assumption that this would not be necessary proves unsustainable. Let's accomodate this fact by writing a modified version of `policyCheck` that is used like this:

```
fun safeLoadUrl (url) =
    if Policies.policyCheck (loadUrl, url)
    then loadUrl url
    else loadUrl (file, "warning.html")
```

This looks much better already. Here, `loadUrl` is the resource we want to use, and `url` its argument, signifying the caller's intended usage of the resource. This allows us to *parameterise* resource policies with regard to the context they are used in. Since we are working with an untyped language, we can accomodate any kind of argument the respective resource might be accessed with, including of course the "empty" case. This achieves the necessary decoupling of resources and their respective usage.

Still, working with this version of the sandbox, we find that the implementation of `safeLoadUrl` is unwieldier than what we would normally expect: Since both branches of

[11] Assuming a trustworthy runtime evironment.

the `if`-clause perform a `loadUrl` call (the only difference being its argument), we would like to unify this functionality. The result might look like this:

```
fun safeLoadUrl (url) =
    let
        url' = Policies.rewrite (loadUrl, url)
    in
        loadUrl url'
    end
```

Here, `Policies.rewrite` would leave the `url` unchanged if it was whitelisted for usage with `loadUrl`, but would return the error page in any other case. This mechanism would be useful in other scenarios as well, for example to route HTTP requests through dedicated ports or to perform character escaping in user-submitted text passages.

In fact, there is no downside to this change – `ABORT`s can still be thrown in the rewrite function, so there's a strict gain of functionality compared to the previous version of our sandbox. Really, all we have to do is define the `policyCheck` function of our previous sandbox to perform the work `Policies.rewrite` does at the moment, and we end up with an updated version that's strictly more powerful than the last. The intended type of `policyCheck` is then $(Res \times \alpha) \rightarrow \alpha$: For each request, the proper value is returned[12]. All that's left to do then is to adopt the according shape for our substitute functions as well.

Using such a function as the standard mechanism for policy checks implies another subtle advantage as well: Doing failure handling inside the policy rule (instead of in the definition of the safe substitute function) means greater flexibility for the rule designer (as the restriction to binary decisions ceases to apply), as well as a cutback on boilerplate code in the substitute function.

3.10 Policy Rules Revisited

Let us take a look at the definition of `policyCheck` one more time. After our latest refinement, it is now a function that for each input tuple somehow computes the proper output value. This principle will not be changed anymore for later revisions of the sandbox, however, it needs to be made a bit more specific. In particular, we still need to explain how the computation part works.

Modular Policy Rules In order to understand how `policyCheck` arrives at a computation result for some input value, it is useful to imagine it as a kind of meta-function that for each resource just coordinates a set of subordinate functions which do the real work. Each of these auxiliary functions can be further split into two parts: one which performs *matching*, and another which performs the decision whether to accept or reject a matched value, and also if some kind of *transformation* (or *rewriting*) should be applied

[12]Permitting a value now is tantamount to having `policyCheck` be the *id* function for that value.

in case of acceptance. Matching in this context means a check to see if the subordinate function may not even be applicable.

Let's see a (nonsensical but succinct) example of how these auxiliary functions can be used:

```
fun increasePositiveInts (n) =
    if n > 0
    then (n+1)
    else NOMATCH

fun decreaseNegativeInts (n) =
    if n < 0
    then (n-1)
    else NOMATCH

fun acceptZero (n) =
    if n = 0
    then n
    else NOMATCH
```

The three of these functions can be plugged together into one ruleset, which can be fed any integer number. If it is a positive number, it comes out increased; if it is a negative number, it is decreased. The number 0 passes unchanged. In a working environment, many of these rules may cooperate, but since each rule deals (more or less) only with a special case, no unpleasant interactions occur.

As a result, a very modular system is created, in which policies are checked as follows:

1. `policyCheck` is called with some *(resource, input)* value as argument.
2. From the set of available policy functions belonging to *resource*, `policyCheck` chooses the freshest entry (unless already tested, see step 4).
3. The first part of the chosen function is applied to *input* to check for a matching. Either of two things can happen:
   a) A successful match is found, i.e., the chosen function is responsible for dealing with this value. Then, the second part of the function is consulted. Again, either of two things can happen:
      i. The *input* value is accepted. If a transformation should be needed, it is applied at this point. The resulting *output* value is returned to `policyCheck`.
      ii. The value is rejected. Then, the function immediately aborts by returning a special value signalling rejection to `policyCheck`.
   b) The match is not successful. This means that the function is not responsible for dealing with this kind of value. Then, it aborts without returning any value.

4. If the policy function call yielded a proper value, `policyCheck` returns this value in turn. However, if it signalled rejection, `policyCheck` uses some implementation of `ABORT` to prevent the calling function from accessing the respective resource (i.e., the respective function call would fail). If it did not yield any value, `policyCheck` returns to the second step and tries another policy function.

5. If none of the user-installed policy functions matches, `policyCheck` uses a default policy to deal with the value. This default is of course to reject (using `ABORT`), but it may be overridden by the user using a function we call `setDefaultPolicy`.

Now we can explain what is meant by matching an argument in a little more detail: Let for any type α the extended type $\alpha_{policy} = \alpha \cup \{reject,\ nomatch\}$ (where *reject* and *nomatch* are the types of the respective [identically named] values with the intuitive meaning). We call a function a *matching function*, if for any argument of type α it returns a value of type α_{policy}, i.e., either a proper value of the same type, or either of two special values signalling rejection or non-applicability of the match, respectively. Matching functions have exactly the type expected by `policyCheck` from elements of one of the rule sets for some policy[13], so to fill these sets, we just have to create the respective matching functions.

To summarize: `policyCheck` just iterates through the set of policy rules pertaining to the resource in question, until one is found that matches the given argument. This rule is then applied in order to either accept a (possibly modified) version of the argument, or to reject it entirely. We can imagine the set of rules as stored in a list-like data structure, in order to make it easy to check policy rules in reverse order of their insertion (Insertion always happens at the front of the list). This means that newer rules are always checked before older rules.

3.11 Dealing with Rejection

Until now, we have not said explicitly how we intend to handle `ABORT`s, i.e., explicit rejection of the request to access some resource. In the style of the real implementation in Alice ML, we'll use an exception handling scheme to deal with this issue. Two new language directives are needed for our model to use exceptions: First, a method for raising exceptions, which we call `raise` – and second, a method the catch them again, which we call `handle`. They are used like this:

```
fun fail () = raise (General, "This function call always fails!")
fail() handle (exType, info) => print info
```

Running this small fragment prints "This function call always fails!" to the screen, as expected. The first component of the raised exception tuple indicates the kind of the exception, the second component is a short informational message detailing the reason for throwing the exception. We adopt standard exception semantics: Exceptions are propagated upwards across contexts until they are caught by a `handle` directive.

[13] if we take *nomatch* to mean that no value was returned.

For our sandboxing infrastructure, we introduce a novel global exception kind `Security` which indicates violations of our security guidelines, i.e., a return value of *reject* by a policy rule. Hence, the implementation of `ABORT` used by `policyCheck` is just something along the lines of

```
raise (Security, "Function foo tried to access resource bar,
                  which is restricted.")
```

An exception of this kind can be dealt with on two different levels. A hypothetical third possibility would be to handle them just after they were raised (i.e., in the safe substitute function); a warning would be printed, but other than that, program execution would resume as if nothing had happened. This is of course not what we had in mind for our sandboxing infrastructure, as we wanted to actually *restrict* resource access, so we choose to neglect this option: Substitute functions do not perform exception handling. The first (and lowest) level where exception handling can occur is therefore in the client application hosted by the sandbox, directly after requesting access to a potentially unsafe resource (or after an according function call requested access). The various ways in which the application can deal with such an exception include:

1. Retrying the request with a different argument that might be considered safe by the sandbox.

2. Discarding the request as insubstantial and continuing with the normal course of operation (for example, if a logfile could not be opened, the core application may nevertheless work as intended). Note that the "normal course of operation" may be slightly altered to accomodate the missing resource.

3. Terminating in an orderly way, for example after releasing some locks on global semaphores or clearing up memory. This also affords the opportunity to inform the application's user about what went wrong.

However, if the client does not handle the exception at all (for example, if it did not anticipate that it might be thrown), there is a second level on which the exception can be handled: By the application hosting the sandbox. That is, if the the sandboxed client is wrapped with an exception handler, it will catch any security exception thrown inside the client application. Thus, a security exception need never go unhandled, which avoids terminating the whole process. This is a good thing, since that case would be rather bothersome for the user (just imagine if every browser crashed whenever some applet tried loading a protected file from the system directory of the operating system!).

3.12 Efficient Component Substitution

The model as presented up until now already offers a good approximation of what the final system will look like. However, if we really wrapped every component in a safe counterpart, we would waste a lot of computation time just delegating component import requests around, most of which could in fact be served much quicker. Instead, we

introduce another mechanism enabling *selective component substitution*. As we already discussed in chapter 3.6, it is important to allow the user to specify which components should be wrapped, and which should be loaded directly in a faster way. To this end, we define a function `importPolicy` which for every URL locating a component returns one of three potential results:

1. A value indicating that the component only includes unobjectionable values, i.e. that it is safe to load this component directly. We call this value *accept*.

2. A substitute URL locating a component that should be loaded instead of the original component. In spirit, it is akin to the rewrite function we already saw in the context of resource acquisition, but while that function performed modifications on input arguments for some resource usage, this function only modifies URLs – and in a rather limited way at that. The URLs it rewrites always locate library components, we do not allow any other schemes in this context. Its work can be (mostly) reduced to a simple syntactic transformation, like this:

   ```
   fun rewrite (x-alice, "/lib/system/"^component) =
               (x-alice, "/lib/safe/"^component)
   ```

   This performs pattern matching in the function argument to reduce code cluttering. The intended meaning should be obvious, if we reveal that `^` performs string concatenation.

3. A value indicating that the component can never, under any circumstances, be considered safe. This means that the whole import request should be rejected. Hence, we call this value *reject*.

The sandbox creator can redefine or extend `importPolicy` at any time, for example to accommodate the loading of self-defined resources. Thus, our architecture is extensible and not restricted in any way to the predefined components/resources.

In addition, note that there are now two different layers on which policies can operate: One that permits low-level intervention with regard to primitive resource access, and one that enables higher-level decisions regarding component import. We call rules referring to the lower level *argument policies*, since they always perform decisions based on the given argument. Rules referring to the higher level are called *import policies*. Introduction of this higher level allows us to reduce the overhead of tedious micromanagement of every single resource hinted at in chapter 3.6.

This also seems like a good time to point out a little more clearly when exactly a sandboxed process behaves differently from a regular one. What exactly does change when a sandboxing manager is used to link a component instead of, say, the root manager? We can identify exactly two points where regular program execution is altered:

URL Resolution As seen above, the point where we perform the first alteration is during the link phase of the component manager. With our approach, intervention comes at

a surprisingly early point in time, namely during URL resolution. This is what we refer to as *rewriting*: Instead of loading the specified (unsafe) component, the delegate component manager loads another component entirely. This permits us to interfere as little as possible with the intricate process of linking (described in a little more detail in chapter 4.1), so our sandboxing infrastructre can be integrated into the existing system with minimal overhead. Note that the signature of the substitute component, however, must stay the same with regard to the expectations of the program that imported it[14], so all that changes is the semantics of the respective functions. In particular, linking commences as usual, albeit with a URL different from the intended one.

Dynamic Checks The other alteration point is of course during any function call that involves one of the functions where dynamic checks were inserted. The checks are truly dynamic in the sense that they are performed anew each time the function is called, which sadly cannot be avoided. On the other hand, this enables the switching of policies at run-time, which (although not overtly supported by our system) could be achieved by using side effects.

It is worth pointing out that these local changes are the only ones our system introduces, so it is not hard to reason about if one has some understanding of how the regular system (without the sandbox) works.

3.13 Default Policies

One important issue we have not dealt with explicitly up until now is the decision of the default policies we want to install in our sandbox – i.e., what happens when an application does something the user did not foresee. We are in the glorious position of being able to define almost arbitrary return values for our rules, so the design space must be vast. Or is it? In fact, for a "default policy", trying to define a general return value that is always fitting quickly becomes not only almost, but completely, arbitrary. After some consideration, only two sensible values remain: Unconditional acceptance of the parameter value, and rejection. It is not hard to see that the former is a bad idea, as it completely subverts the sandbox's intention. That is why we choose to be restrictive by default: Whenever faced with the decision to be either user-friendly (with less unexpected exceptions) or safe, we prefer the latter. This should not pose major problems, since only security-conscious users who are likely to agree with this sentiment would be bothering to use a sandbox in any case. Still, it is always possible to override the default setting.

3.14 Putting it all together

The single parts of our sandbox are now in their respective final versions, but the big picture may still be a little foggy. Let's try to get an even better understanding of the system by building one complete application using its single pieces. The target

[14] The Alice ML compiler provides automatic thinning of signatures, so only the types of values actually used count in that regard.

we are aiming for is the corporation-specific browser already mentioned – to restate its purpose: We want to build a system for browsing websites that delegates requests for "unproductive" websites[15] to a page displaying a stern warning to lazy employees. Let's give the corporation the name AnyCorp and assume that its representation on the WWW is located at http://www.anycorp.com.

We begin with an unsafe implementation. For the moment, we assume that employees use the application by calling the `Browser.load` function.

```
fun Browser (compManager) =
    let
        Http = compManager (x-alice, "/lib/net/Http")
        fun load url = print (Http.request (url, 80))
    in
        (load)
    end
```

`Browser` is a component whose module contains just one value, which is the `load` function. Currently, it loads every URL that is requested. We are going to change this by substituting the `Http` component it loads with a safe counterpart. To achieve this, we must execute the unsafe browsing application in a sandbox environment:

```
fun SafeBrowser (compManager) =
    let
        safeCompManager = Sandbox.Manager
    in
        Browser (safeCompManager)
    end
```

That is, we merely pass `Browser` a modified component manager taken from a `Sandbox` module we have yet to write, where the definition of `Browser` is the one from above. From now on, we assume that employees only have access to the `SafeBrowser` component and thus can only access the `SafeBrowser.load` function.

How does the definition of `Sandbox` look like? First of all, let's define its `importPolicy` function. This should be easy, as we only need to care about the `Http` component:

```
fun importPolicy url = if url = (x-alice, "/lib/net/Http")
                       then (x-alice, "/lib/safe/net/Http")
                       else ACCESS
```

This immediately begs the question of how the `/lib/safe/net/Http` component should be defined. The answer is this (assuming existence of the according resource constants):

```
fun Http (compManager) =
    let
        UnsafeHttp = compManager (x-alice, "lib/net/Http")
```

[15] Those that are not whitelisted by the corporation.

```
        fun request (url, port) =
           let
               url' = policyCheck (urlRead, url)
               port' = policyCheck (portRead, port)
           in
               UnsafeHttp.request (url', port')
           end
    in
        (request)
    end
```

Now, we specify our policy regarding ports. Remember that we have to define the set of policy rules responsible for dealing with accesses to the `portRead` resource. Since we never have to rewrite ports (they are defined in the Browser component as the constant 80), we leave this set intentionally empty, and just set the default rule to quiet acceptance:

```
defaultPolicy(portRead, port) = port
```

In contrast, the default policy for URL loading is restrictive. We automatically redirect to the aforementioned warning page hosted on the company's webserver:

```
defaultPolicy(urlRead, url) = (http, "www.anycorp.com/warning.html")
```

With that, we just have to whitelist a few pages the employees are allowed to surf to. First of all, the company's own website is a rather obvious candidate:

```
fun corpMatcher (url) = if hasSuffix (Url.getAuthority url,
                                      "anycorp.com")
                        then url
                        else NOMATCH
```

Let's allow Wikipedia as well:

```
fun wikipediaMatcher (url) = if hasPrefix (url, "www.wikipedia.com")
                             then url
                             else NOMATCH
```

The one thing left to do is to insert these rules into the respective sets for the `urlRead` resource. We have not said explicitly how this is done before, but let's just assume that a call to `insertPolicyRule` is sufficient. Then, we have:

```
Sandbox.insertPolicyRule(urlRead, corpMatcher)
Sandbox.insertPolicyRule(urlRead, wikipediaMatcher)
```

And with this operation complete, we are done! The request for `Http` in `SafeBrowser` will now be rewritten into a request for our safe substitute, which will ask for permission any time it tries to access some URL. If neither of the two whitelisted URLs is requested,

the default policy will be evoked, which rewrites the request into one for the warning page[16].

Take special notice of the fact that we did not need to touch the original `Browser` component at all, but merely had to load it inside a sandbox. This is important in the dynamic context of the sandbox's intended application domain, where the component is linked dynamically, and its source code is of course unknown and unalterable. Component substitution in that case is performed during linking.

This concludes our discussion of the sandbox model. In the next chapter, we'll see how to implement it in a real-world language (and notice that the differences between model and implementation are smaller in nature than might be expected).

[16] Which we will assume is not accessible from outside the corporation's network, since this could reflect on AnyCorp's image in a negative way.

4 Implementing the Model

4.1 Component Managers

In the last chapter, we got a better understanding for the fact that component managers lie at the heart of our efforts to build a secure sandboxing infrastructure for Alice ML. Therefore, it is only fitting to begin this chapter with a discussion on how they are actually implemented. Figure 5 gives an idealised representation of the inner workings of a component manager [16]. As we can see, our introduction of component managers in chapter 2.4 still managed to hide quite a bit of detail. However, this seems somehow appropriate, since in truth, the purpose of component managers in Alice ML is to do just that: Hide away the complex details of component import at runtime. This also enables the programmer to control that process in a more intuitive way, since he can just use the functions provided by the manager. The tasks the component manager has to perform (in sequential order) are these:

1. *URL resolution*, so non-absolute import URLs are interpreted relative to the location of the current component.

2. *Acquisition*, the actual loading of the requested component (e.g. from disk).

3. *Evaluation*, which of course must always be done for freshly loaded components.

4. *Type checking* of the expected signature against the actual exported signature calculated during evaluation of the component.

Each of these can fail independently of the others, in case of which the exception `Component.Failure` is raised. However, if all of the above steps succeed, the component is entered – in its fully evaluated state – into an internal table, where it can be looked up by its URL when needed again later on. This avoids unnecessary (and slow) reloads of components.

The methods provided by the component manager to support control over component import include:

- `acquire`, which performs URL resolution and component acquisition.

- `eval`, which evaluates components, but does not yet enter them into the internal component table.

- `enter`, which enters an evaluated component into the internal table, if it is not already listed. If it is, `Conflict` is raised.

- `lookup`, which performs a lookup of evaluated components in the internal table.

- `link`, which integrates the whole process of linking a component, i.e., all of the above operations.

Figure 4 gives an overview of the signature. The `package` type can be thought of as that of an evaluated component[17], in contrast to the `component` type, which indicates that it is not yet evaluated. The interesting part of the implementation concerns the inheritance

```
signature COMPONENT_MANAGER =
sig
      exception Conflict
      val acquire : url -> component
      val eval : component -> package
      val enter : url * package -> unit
      val lookup : url -> package option
      val link : url -> package
end
```

Figure 4: The signature of a component manager

of restrictions imposed on a component loaded by a particular manager. How is it that these cannot be circumvented, for example by importing a "fresh" `ComponentManager` structure or by using the `MkManager` functor to a similar effect?

Certainly, it does not pose any problem to create new component managers – after all, this is exactly why `MkManager` is provided. These custom managers could have their own resolving strategies, or implement any number of functions differently from the standard component manager, but the one thing they *cannot* do is to import resources. For this, they have to rely on the system components provided by the Alice library, and there simply is no way for a custom component manager to import these other than to delegate this request to the parent manager. Of course, the parent manager is still bound by its sandbox, so there is no chance for security leaks.

What about importing the `ComponentManager` structure, and using its respective `link` function? True, if there was a way to import the original `ComponentManager` (which is used for example to create the root manager), this approach would work, as this manager must of course not be restricted by any sandbox. However, a small trick ensures that this is not possible, either: Any import of `ComponentManager` is served by dynamically repackaging the current component manager (which is the parent manager of the component requesting the structure) into a new structure. In particular, all of the potentially harmful functions are re-used (i.e., copied) from that manager, so they cannot violate any restriction imposed on that manager. This means that newly created component managers are always safe in the sense that they can never get past security guidelines restricting their parent managers. The rest of the implementation of `ComponentManager` is pretty straightforward. For a more in-depth discussion, the interested reader is referred to [16].

[17]The real explanation is outside the scope of this thesis.

```
exception Conflict
val table = ref [] : (url * package) list ref

fun import' parent =
let
    fun acquire url =
        Component.load url handle e => raise Failure (url, e)

    fun lookup "x-alice:/lib/system/ComponentManager" =
        pack ( structure ComponentManager =
        struct
            exception Conflict = Conflict
            val acquire = acquire
            val lookup = lookup    val enter = enter
            val eval = eval' "."   val link = import' "."
        end
        ) : (structure ComponentManager : COMPONENT_MANAGER)
     | lookup url =
          List.find (fn (x, _) => x = url) (!table)

    and enter (url, package) =
         if isSome (lookup url) then raise Conflict
         else table := (url, package) :: !table

    and eval' url component =
         component (import' url) handle e => raise Failure (url, e)

    fun link url =
    let  val url' = resolve (parent, url) in
         case lookup url' of
               SOME package => package
             | NONE =>
               let val package = eval' url' (acquire url') in
                     enter (url', package); package
               end
    end
in
    link
end
```

Figure 5: A canonical component manager

4.2 Unsafe Resources

A significant part of the implementation design is the identification of resources that might be abused in various ways. There are three concerns in particular that have to be considered:

System Security The most obvious category, including for instance the capability to execute unchecked native code or system calls, open network sockets or access the local file system.

User Privacy This includes read access to the file system as well as to various variables set during any run of an Alice program (such as command line parameters or system variables).

General Concerns This category contains any resources that do not fit in either of the other two categories. An example case is the deliberate circumvention of abstraction principles in a program.

Here, then, are the resources from the current Alice ML library (which is based on the Standard ML library [14]) deemed unsafe and thus restricted, grouped by the respective contexts in which they are checked (i.e., the type of both argument and return value of the respective `check` call for that resource). Resources marked with "*" are considered especially dangerous, as they can be exploited to gain unlimited access rights to any other resource.

URL Context

- `componentLoad` (Security). Refers to the capability of calling `Component.load` with the respective URL.
- `readUrl` (Security). Used for checking if read access to some URL should be granted.

String/File Context

In the current implementation of Alice ML, filenames are passed as strings, so these contexts are grouped together.

- `componentSave` (Security). Checked before a component is saved to file (`Component.save`). There exists no reasonable way of checking the semantic content of a component, so this is the only capability that will be requested in that context[18].
- `getApplicationConfigDir` (Privacy). If granted, the canonical directory path where the Alice ML application of the given name can store its (user-specific) configurations is accessible.

[18] Since it wouldn't make sense to create another method to check components as well.

- `getEnv` (Privacy). Given the designator for some environment variable, this represents the right to retrieve its content.
- `readFile` (Security/Privacy). The capability of reading a single file.
- `writeFile`* (Security). The capability of writing/deleting a single file.
- `sysCall`* (Security). The capability of executing the system call given as argument.

If file permissions are set incorrectly, `writeFile` may be used to overwrite the Alice ML interpreter or compiler binary with a custom version, compromising integrity and thus security.

With `sysCall`, one can go even farther and modify the file permissions for the binary file manually before attempting to overwrite it. Extreme caution is advised in granting either of these capabilities.

Int Context

- `readPort` (Security). The capability of opening a TCP port for incoming data, i.e. a *listen* port.
- `writePort` (Security). The capability of opening a TCP port for the purpose of sending data.
- `terminate` (Security). The ability to terminate the whole program thread, including the application hosting the sandbox. The argument refers to the status code returned after terminating.

Empty (Unit) Context

- `MkManager` (Security). If set, users may call `Component.MkManager` to create their own component managers.
- `commandLineName` (Privacy). For retrieving the name of the application with which the current Alice process was started.
- `commandLineArgs` (Privacy). For retrieving the command line arguments with which the current Alice process was started.
- `deepWait` (General). Enables calling of `Store.deepWait`, which breaks abstraction barriers[19].
- `getDir` (Privacy). Grants the right to retrieve the pathname of the current working directory.

[19]This function `awaits` all futures in the representation of some value, even if that value represents an abstract type relying on the maintenance of some unevaluated futures as invariants.

- `getHomeDir` (Privacy). Grants the right to retrieve the pathname of the user's home directory.
- `processWait` (Security). Enables waiting for a system process (`Unix.wait`).
- `processReap` (Security). Determines if a system process may be reaped (`Unix.reap`).
- `readSocket` (Security). Determines if sockets may be opened for receiving data.
- `writeSocket` (Security). Determines if sockets may be opened for sending data.
- `stdIn` (Security). Grants the right to access `stdIn`.
- `stdOut` (Security). Grants the right to access `stdOut`.
- `stdErr` (Security). Grants the right to access `stdErr`.

4.3 Design Aspects

In chapter 3, we have discussed the design approach taken for our sandboxing infrastructure. Now, we'll take a look at how to actually implement it. The implementation was realised during the course of creation of this thesis, and will be available as a part of the Alice ML distribution [1].

4.3.1 General Architecture

By now, it should be a given that no sandbox can work without policy rules. However, policy rules can be created and manipulated independently of sandboxes. For example, two distinct sandboxes could conceivably use the exact same policy set, but host different applications. At any rate, policy rules must always be defined and in place before a sandbox can be deployed. That is why our infrastructure is split in two: On the one hand, there is the component manager providing the actual implementation of the delegation and indirection principles discussed in chapter 3.8. On the other hand, there is a module dealing with the creation and management of policy rules, which is kept separate from the first.

The sandboxing manager is provided by this functor:

```
Sandbox.MkManager (Policy : POLICY)
```

For the reasons just mentioned, it must receive a fixed set of policy rules at creation time. These are created by the `MkPolicy` functor, which does not need any input, but is implemented as a functor nevertheless for reasons of generativity (i.e. every policy set is generated "fresh" from the default settings). The generated policy set should then be customised with appropriate policy rules.

4.3.2 Sandbox Interface

The interface for accessing the sandbox (Figure 6) is very straightforward: The already mentioned `Security` exception is provided, which is raised whenever a policy rule is violated. `MkManager` is of course the functor for creating the sandboxing component manager. After creation, such a manager is used simply by calling its `link` function on URLs locating those components that should be controlled by the sandbox.

```
signature SANDBOX =
sig
    exception Security of string
    functor MkManager (Policy : POLICY) : COMPONENT_MANAGER
end
```

Figure 6: The signature of a sandbox

4.3.3 Policy Interface

As there are a number of ways for implementing policy rules, the interface for doing so (Figure 7) is a little more complex than the sandbox interface. Yet, it is easier to understand than it might appear at first glance. Keep in mind that there are two different kinds of policy rules: Argument policies and import policies. We'll start with the latter, as that concept is not nearly as complex as the former.

Import Policies Instead of the `importPolicy` function used in the model (chapter 3.12), we employ the `import_policy` datatype together with the `setImportPolicy` function to define the policies for component imports. The first argument for this function is the URL of the component that should be imported, the second the intended treatment expressed by an `import_policy` value. These values mirror exactly the possible result values of the `importPolicy` function. If no action is specified for the component, its import is rejected. The `Policy` structure provides reasonable defaults for all system components. Custom components must of course still be entered manually, if they are not to be rejected. As with system components, they must either be substituted by a safe counterpart (⇒ `SUBSTITUTE counterpart`) or marked as harmless by an `ACCEPT`.

Handler Type The `handler` type is the internal representation of policy rules inside a sandbox. Its definition can be extended with new types of any kind, so even user-defined types can be accomodated in a policy rule.

This type represents the main difference between our model and the implementation: Since we are now working in a typed context, we have to indicate to the type system the context of each policy rule. This is the purpose of the `handler` type, to integrate

```
signature POLICY =
sig
    type handler

    datatype import_policy =
        ACCEPT
      | REJECT
      | SUBSTITUTE of Url.t

    val setImportPolicy : Url.t * import_policy -> unit
    val secureComponent : Url.t -> Url.t option

    signature ARG_TYPE =
    sig type t
        val setDefaultPolicy : string * bool -> unit
        val accept : t -> handler
        val reject : t -> handler
        val acceptCustom : (t -> t option) -> handler
        val rejectCustom : (t -> bool) -> handler
        val check : string * string * t -> t
    end

    functor MkArgType (type t) : ARG_TYPE

    structure Unit : ARG_TYPE
    structure Int : ARG_TYPE
    structure String :
    sig include ARG_TYPE
        val acceptPrefix : string -> handler
        val rejectPrefix : string -> handler
        val acceptSubstring : string -> handler
        val rejectSubstring : string -> handler
    end
    structure Url : ARG_TYPE

    val addRule : string * handler -> unit
end
```

Figure 7: The POLICY signature

the "handling" of rules while preserving information about the individual types. That way, the sandbox does not have to care about the context a policy rule works on, but types are respected nevertheless. This is done automatically when using the methods provided by the corresponding `ARG_TYPE` (see below), which in fact affords the only means for `handler` creation. Hence, if the policy rules for some resource perform string rewriting, its default *must* be installed using the `Policy.String.setDefaultPolicy` function (where `Policy.String` is one of the predefined `ARG_TYPE` instantiations), and similarly for the other types. Otherwise, return values could not be properly typed in the case of an *accept*ing default policy!

Similarly, during the creation of policy rules, the respective resource context has to be kept in mind, in order to choose the correct function. However, once a type-dependent handler has been created, it can always be installed using the `addRule` function (which does not have to behave differently across types, since that functionality is now encapsulated in the handler). This function takes a resource descriptor in form of a string and a handler, and installs that handler in the policy rule set for that resource.

ARG_TYPE The `ARG_TYPE` signature collects all elementary operations for working on argument policies (as opposed to component policies), and the `MkArgType` functor provides a convenient method for creating instances of this type. The predefined structures that implement the `ARG_TYPE` signature are listed in the lower part of figure 7. Specifically, each instantiation implements three distinct functionalities:

1. Setting a default policy for a resource via `setDefaultPolicy`. The resource in question can be one of the predefined resources (cf. 4.2), or a custom resource. There is no special syntax for the definition of these, adding rules or default policies "just works" out of the box. The bool argument refers to the two possible defaults, acceptance and rejection of the access request (cf. 3.13) – TRUE and FALSE, respectively.

2. Creating `handlers` for argument policies. These are discussed below.

3. A check function that can be called from safe substitute components. This method takes as arguments the resource designator that should be checked, the name of the function that requested access to the resource, and a value of type `t` that is then rewritten using the policy rules for the specified resource.

The signature offers four methods to create handlers. Remember that in case a handler matches, there always remains the decision of whether to accept the value in question, or whether to reject it – and in case it is accepted, whether it should be modified. Thus, an accepting handler must always specify the value it accepts, while a rejection handler does not. Hence, we use different methods to create accepting and rejecting handlers.

Accepting handlers are set up via `accept` and `acceptCustom`, where `accept` is used for identity matching (i.e. when given x and y, *accept* returns x without modification iff $x = y$, *nomatch* otherwise). To return custom values, `acceptCustom` is used, for which

a return value of the argument function of NONE is equivalent to *nomatch*, and a value of SOME *x'* equivalent to acceptance of *x'*.

Conversely, `reject` rejects the input if it is identical to the given argument, while `rejectCustom` uses a boolean function to test its input. If it returns TRUE, that input is rejected, if it returns FALSE, the function returns *nomatch*.

Note that in our model, there is no way to access handlers directly. After creating them, they have to be entered into the policy rule list for some resource via `addRule`, after which they are accessible for checking (i.e., calling the `check` method on that resource).

URL rewriting Based on the policies thus defined, the function `secureComponent` performs the actual work of deciding the proper treatment for each input URL. In the normal course of sandbox operation, it is only expected to be used by the internal component manager during its resolving phase, but its exposure may nevertheless be useful for debugging purposes.

4.4 Sandbox Usage

After having seen the interface, let's try working with it! We'll revisit our AnyCorp browsing application, but this time, we'll end up with an implementation we could actually use. Again, we begin by writing the unsafe browsing application:

```
(*) /AnyCorp/unsafe/Browser
import structure HttpClient from "x-alice:/lib/system/HttpClient"
structure Browser =
struct
    fun load url = print (#body (HttpClient.get url))
end
```

Yes, that is all we really need if we focus on the essence of the browser and ignore error handling and so on ("`(*)`" defines line comments, here we indicate file location). After all, what could possibly go wrong in loading some site from the internet?

Moving on, we'll load this module inside a sandboxing component manager. To make things more realistic, let's assume that we do not actually know the module definition, but only receive a precompiled component we have to use inside our host application. We begin by creating a fresh `Policy` module that will contain our custom policies:

```
(*) Main File
import structure MkPolicy from "x-alice:/lib/system/MkPolicy"
import structure Sandbox from "x-alice:/lib/system/Sandbox"
structure Policy = MkPolicy()
```

Before we can create our sandbox, we have to finish policy design. From our previous discussion of the browser, we know what we have to do, which is to substitute the `HttpClient` by a safe counterpart:

```
Policy.setImportPolicy (Url.fromString
                            "x-alice:/lib/system/HttpClient",
                        Policy.SUBSTITUTE (Url.fromString
                            "file:/AnyCorp/safe/HttpClient"))
```

We have now set the import policy for `HttpClient` so that it is substituted by the file `/AnyCorp/safe/HttpClient`. Let's write that file:

```
(*) /AnyCorp/safe/HttpClient
import structure Policy from "x-alice:/lib/system/Policy"
import structure HttpClient from "x-alice:/lib/system/HttpClient"
structure HttpClient =
struct
    type document = HttpClient.document
    exception Authority = HttpClient.Authority
    val request = HttpClient.request
    val post = HttpClient.post
    fun get url =
        let val url' =
            Policy.Url.check ("readUrl", "HttpClient.get", url)
        in HttpClient.get url'
        end
end
```

Naturally, in a real implementation, we would run more extensive checks and substitute the other unsafe functions as well[20]. We neglect checking of port 80, since we would need to create our own HTTP request upon receiving a rewritten value, and could not use `HttpClient.get` any longer. The principle, however, should be clear.

Next, we install the default policy for redirecting users to our warning page:

```
fun redirectUrl ( _ : Url.t) =
        SOME (Url.fromString "http://www.anycorp.com/warning.html")
val urlHandler = Policy.acceptUrlCustom redirectUrl
do Policy.addRule ("readUrl", urlHandler)
```

Note that `setDefaultPolicy` does not suffice for our purposes, since it can only accept or reject according to some boolean value. However, such a behaviour is necessary to guarantee that the default rule always matches, since a handler (instead of a bool) might also return *nomatch*. Hence, we instead rely upon the fact that the first rule that is inserted has a *de facto* default status, in that it is of course called whenever no other rule matches. We ensure that this rule always matches by throwing away the argument and replacing it with the warning URL.

[20]See our real implementation of the substitute component for `HttpClient` for further details.

Now we can insert the rules allowing access to certain sites:

```
fun urlPrefix (prefix, url) =
    String.isPrefix (prefix, Url.toString url)

fun acceptInternal url =
    if urlPrefix ("http://www.anycorp.com", url)
    then SOME url
    else NONE

fun acceptWiki url =
    if urlPrefix ("http://www.wikipedia.com", url)
    then SOME url
    else NONE

val internalHandler = Policy.Url.acceptCustom acceptInternal
val wikiHandler = Policy.Url.acceptCustom acceptWiki
val LtUHandler = Policy.Url.accept
                      (Url.fromString "http://lambda-the-ultimate.org")

do Policy.addRule ("readUrl", wikiHandler)
do Policy.addRule ("readUrl", googleHandler)
do Policy.addRule ("readUrl", internalHandler)
```

We added "Lambda the Ultimate" to the set of allowed sites to demonstrate that not every handler requires custom creation. Indeed, the basic policy functions should be sufficient for almost all regular use cases. Notice in the previous example the order of `addRule` calls, which is inverse to the expected match probability (i.e. unlikely matches are inserted first, so they are checked later than others – think of it as a stack).

Now we have to create a sandbox which uses our newly-defined policies, and use it to link the unsafe browser:

```
structure SandboxManager = Sandbox.MkManager (structure Policy = Policy)
val safeBrowser = SandboxManager.link
                      (Url.fromString "file:/AnyCorp/unsafe/Browser")
```

And again, we are done. We can now use `safeBrowser` by unpacking it and using it for surfing the web – AnyCorp style! The question of how desirable this is is left as an exercise for the reader.

4.5 Sandbox Implementation

Safe Substitute Components Although it may be stating the obvious, we nevertheless begin this chapter by mentioning that as a matter of course, the actual sandbox implementation includes safe substitutes for all system components. These mirror the structure of the standard library [14], both in terms of structural hierarchy among the

different components as well as in the layout of single components. In particular, all unsafe functions accessible via the interface of a system component are wrapped in a safe substitute function. The manager provided in the `Sandbox` component always resolves to these safe components per default, which is done by a syntactic transformation of input URLs made possible by the strong correspondence between the structure of the standard library and that of the safe components.

Resolving Policies Not quite so obvious is one difficulty encountered in the implementation of the crucial indirection feature for the `Sandbox` component. We have seen how indirection enables us to wrap unsafe system components with relatively little effort inside a safe counterpart – now, we will see that this method also has its drawbacks. The crux of the matter lies in importing the `Policy` structure in the safe substitute component, which (as we know) is created dynamically by calling the `MkPolicy` functor and specifying the policies we want to use. However, we also know that we definitely have to import `Policy` inside the safe substitute component – otherwise, how could we implement our calls to the proper `check` method?

The situation is like this: The `Policy` structure is created inside some application and dynamically wrapped into a component, so at first it is only visible under "local" scope, i.e., only within the bounds of that application. To make it visible "globally", the structure would need to be entered into a component manager table – then, other structures linked with that component manager could `import` it. Fortunately, component managers offer the `enter` method to deal exactly with this problem: Using it, one can enter dynamically created components under some arbitrary URL into the internal table of the manager. A naïve approach would therefore consist of importing the current component manager and entering `Policy` in its table. The "current component manager" in this context is always the one used to link the application hosting the sandbox.

Doing this would give a first approximation for a solution, as sandboxed components are now able to import `Policy`, which we wanted to achieve. Does this mean that the problem is solved? Unfortunately not, as we have introduced a new difficulty by entering `Policy` into the component manager responsible for the whole host application. You could say that our efforts to move `Policy` from local scope into global scope were "too effective": The one `Policy` component we entered into the manager is now *always* returned whenever this structure is imported by a sandboxed component. In particular, if we wanted to host two different sandboxes with different policies, it would not be possible to install the second `Policy` component. A conflict would be raised at the point of entering the new `Policy` component under the same URL as the old – but using a different URL is not a valid option either, since this would entail writing multiple substitute components that only differ in the URL they import.

So we see that we must enter the `Policy` component somewhere further down the link hierarchy of component managers. The approach that comes to mind almost instantly is of course to enter it inside the sandboxing manager – after all, this would eliminate the

problem of trying to host different sandboxes, as it would preserve the strong correlation between sandbox and policy. Regrettably, this assumption is misguided: Assume for a minute that we actually entered `Policy` into this manager. Now, when we link a component using that sandbox, it immediately delegates this request to its parent manager (possibly after rewriting the URL to point to some substitute component). This means that at the time the actual substitute component is linked, any information contained in the table of the sandboxing manager is completely irrelevant for the imports the substitute component may request – in particular, this is true for the `Policy` import request as well. So it is evident that this cannot be the way to go, either. The solution to our prob-

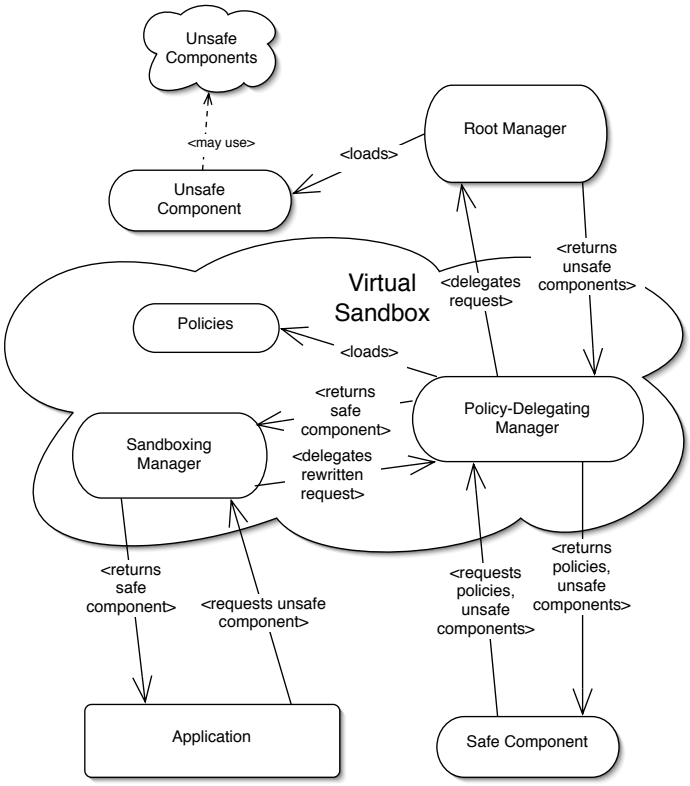

Figure 8: The virtual sandbox

lem, however, is indicated by this last try: If requests are just delegated "up one level" starting at the sandboxing manager, all we have to do is insert a new manager exactly at that point, which will deal with `Policy` import requests. Stated in a different way, we just transparently add another layer of indirection (Figure 8). Rather than modifying the sandboxing manager or passing the current component manager as delegate manager to the safe component, we create a second internal manager inside `Sandbox`, enter the `Policy` there, and delegate import requests from the sandboxed component always to

this manager first. The result is what we refer to in figure 8 as the virtual sandbox, which in fact encompasses the `Policy` module as well as two component managers, but which appears to the user just as the one sandboxing manager. That way, when a safe substitute component tries to import `Policy`, it is resolved by the internal manager. Any other requests are delegated to the parent component manager as before, so there's no change concerning inheritance aspects, and implementation of the new internal manager is straightforward.

Security Violation Component The other interesting aspect in our implementation concerns the default value that is returned when an entire component is outrightly rejected. We do not raise an exception immediately, but use the delegation feature once again to forward the client a dummy component, which raises the exception in our stead. This ensures that program execution is halted only when the client really tries to use the unsafe component, not if it merely imports it but does not use it for some reason (for example, if it is only needed in a conditional branch). The dummy component is depicted in figure 9.

It resides at a location accessible under some URL, which the sandboxing manager delegates to its parent manager. It is linked and returned as the requested component, but as with all components, evaluation is suspended until it is triggered explicitly. This way, the program may still perform useful work before it is interrupted by a security error, or it may even terminate successfully.

```
import structure Sandbox from "Sandbox"
do raise Component.Failure (Url.fromString ".",
             Sandbox.Security "Security Failure")
```

Figure 9: The security violation component

5 Related Work

5.1 Java

The idea of using a sandbox to lock dynamically linked components in a safe space is of course not so new. Of all the languages that use it, Java in particular stands out, since this was the language that introduced and popularised the concept in the mainstream. Let's compare how things work in Java and in our system:

5.1.1 Resources

When we introduced resources, we briefly mentioned that references to functions that access unsafe resources can be considered capabilities. Java makes use of this fact in the definition of *permissions*, which represent access to system resources [18]. There are lots of classes defining various subtypes of permissions, e.g. `FilePermission` and `NetPermission`, with `Permission` as the common superclass. Custom permissions can be created as well, likewise by subclassing `Permission`.

A crucial difference between our concept and the Java approach is the parameterisation: In Java, new permissions are created for each instance of an anticipated usage (e.g. `java.io.FilePermission("/tmp/abc", "read")`), whereas in our system, the resource is defined once, and then equipped with various parametric rules. Java allows instantiation of permissions with predefined wildcards (e.g. for `FilePermission`: [optionally recursive] matching of subdirectories, and a value for matching all files). These wildcards are always specific to one resource and cannot be extended. In our case, this (and more complex functionality) can be easily achieved by defining custom policy functions.

5.1.2 Policies

In our system, each rule defines whether it accepts a given value or not, and if this value should be modified in the process. In Java, each permission just defines whether it accepts a value. As in our system, various permissions pertaining to one resource have to be tried before any assertion can be made about access rejection. They are collected in instances of class implementing the `Policy` interface, which define the *security policy* for any one application. At any time, only one instance of this class can be active; this active instance can be obtained via the `getPolicy` method provided by every class object.

Java has a flexibility advantage in that permissions must always implement the `implies` method, which basically states that if a permission A implies B, then when A is granted, B must be granted as well. For example, general rules like wildcards always imply their specific instances.

How policy configurations can be set in Java is up to the implementation of the `Policy` interface. The reference implementation reads data from configuration files formatted according to a syntax specified in the Java API documentation [18].

5.1.3 Component Manager

In Java, the system most closely corresponding to the component manager is the class loader (accordingly, the root component manager is called *primordial class loader*). Class loaders always work lazily [8]. The meaning of this term in Java encompasses that classes as well as member methods and values are loaded "as late as possible", i.e. on demand.

Each class loader defines its own namespace, so if one class is loaded by two different class loaders, they will be type-incompatible. In Alice, the types will be compatible, although a type sharing constraint may be needed in some cases for the static type system. A discussion of the relative merits of the two approaches is outside the scope of this thesis.

Delegation between class loaders works similarly to the equivalent Alice concept, i.e., class loaders can delegate requests to their parents (or directly to the primordial class loader). However, there is no direct restriction inheritance, as Java employs a different system of policy checking than Alice ML. Instead, the *security context* provided by the `AccessController` class is inherited across processes.

5.1.4 Access Control

The `AccessController` class introduced in version 2 of the Java API is disassociated from the class loader and is used to check for permissions. A sample call may look like this:

```
FilePermission perm = new FilePermission("path/file", "read");
AccessController.checkPermission(perm);
```

That is, `AccessController` is globally accessible and just checks whether the permission is directly or implicitly granted. Thus, the process of policy checking in Java looks something like this:

1. The client application makes a call to some potentially unsafe system function using the Java API.

2. In the API function, various `checkPermission` calls are inserted that check for whatever permissions the function may need.

3. If some permission checked by such a call is not granted, the call raises a security exception called `AccessControlException`. This exception is a subclass of `java.lang.SecurityException`. Otherwise, it returns quietly. The API function then proceeds with its regular operations.

Similarities to our approach lie in the fundamental way the permissions are checked, namely dynamically at call time of the API function, and the way rejection is dealt with: raising a security exception in the respective handler code.

One main difference besides parameterisation (as discussed above) is that API functions include hardcoded policy checks – i.e., there are no substitute functions, but every low-level API function always carries the overhead for policy checking. Our approach

allows acceptance or rejection also at component level, so the basic system functions can be written without regard to such concerns, and are substituted only when needed.

The other difference is of course the use of return values in our system, which can be modified according to the user-specified policy rules. Java simply does not implement anything which could emulate this kind of behaviour.

Another feature of Alice ML to which no equivalent exists in Java is lazy evaluation, which was already mentioned earlier on. As indicated, it enables some very elegant techniques that would have to be reformulated in a manual, ad-hoc approach for use in Java, if at all possible. Consider this fragment taken from the substitute component for `TextIO` (figure 10):

```
val stdIn =
    lazy let
             do Policy.Unit.check ("stdIn", "TextIO.stdIn")
         in
             TextIO.stdIn
         end
```

Figure 10: `TextIO` substitute value for `stdIn`

The use of `lazy` allows us to wrap values with policy checks! This check will be performed once, at the time when `TextIO.stdIn` is first requested (i.e., used in a computation where its value is required to be known). If it fails, the security exception is thrown as usual, otherwise the regular `stdIn` value is put in place of the "placeholder" value. For the semantics of lazy evaluation, see [13].

It is not clear at all how a similar behaviour could be implemented in Java.

5.1.5 Certificates

Java implements a system of *trust*, which is to say that it enables signing of code with certificates by trusted parties. Signed code can thus gain security privileges denied to unsigned code, which basically means that this provides a facility to bypass the security infrastructure – not unsafe per se, but dangerous if any malicious party can gain access to certificates by (potentially illegal) means. Past experiences have shown that this worry is not unfounded.

Alice ML does not provide a system of any such kind, although it might be desirable e.g. in the context of foreign function interfaces.

5.2 The E Programming Language

E [11] is a vocal proponent of the *principle of least authority* (POLA), which states that programs should run with the least amount of rights they absolutely need to work. For

example, in current systems, most applications inherit all the privileges of the particular user running the application. If this user happens to have administrative rights (which is the standard case for a very popular, yet unsafe, desktop OS), the application is practically unbounded with respect to the operations it may perform. No word processing application needs write access to the directory where the OS resides, but usually, this is granted nevertheless – the result is that viruses embedded in "rich" text documents can do considerable damage extending far beyond the environment provided by the word processor [7]. This is just one example of unnecessary dangers that could be avoided by applying the POLA concept.

E has an even stricter focus on capabilities than Java does. While the basic concept is the same, E goes to great lengths to try to ensure that its programs are safe. Examples include the (by now standardly done) exclusion of pointer arithmetic and of static mutable state (citing as reason examples like Java's `System.out` variable that could be replaced in early incarnations of the language due to mutable state) from the language, and a carefully designed API that prevents the "leaking of capabilities" – or at least, that's the claim. This capability leakage concerns authority that is granted to a special object (e.g. a library component which is known to be safe), but that can be derived from that object for general usage with other objects as well. An example given in [17] concerns the Java `InputStream`, which supposedly could be downcast to a `FileInputStream`, which would hold a reference to `File`. This way, control over the entire directory system could be gained.

E does not have an explicit notion of sandbox such as presented in this paper. This is because every E program can be said to be sandboxed anyway, since authority granting is always explicit, instead of implicitly assuming full trust under regular (i.e., non-sandboxed) execution as in the other paradigms. In E, security thus becomes a matter of architecture, since the challenge does not lie in the implementation or usage of a sandboxing infrastructure, but rather in designing the application in such a way that the set of minimal authority it needs to work is kept as small as possible. In [17], a range of design patterns is offered that help to support that goal. A discussion of these, however, is outside the scope of this thesis.

5.3 Perl 6

Since Perl 6 is still in its conceptual phase (albeit a very sophisticated one), not very much can be said about its sandboxing mechanism. The intention documented in the relevant RFC [2] is to use a *tainting* mechanism familiar from previous versions. This mechanism treats specific data as tainted, i.e., potentially unsafe, and forbids most operations on it. The data primarily concerned in previous versions was user-input, but version 6 extends the concept to other data as well, like file content from the local file system and data gained from socket interaction. In this respect, it is not unlike our resource system – a rough characterisation would be to say that usage of resources results in tainted data.

Tainted data must be explicitly *untainted* by the programmer before the restriction on the operations a program may perform on this data are lifted. Hence, the operations that are allowed concern checks and alterations the programmer may perform to satisfy

himself that the data is secure (in the sense relating to the application domain of his program) before untainting it. Of course, this is a rather general mechanism, and as such it is both much more complex to use than our system, and also much more potent. For example, alteration of data is no issue at all (it is just done), and the dynamic run-time checks that Perl uses to make up for its lack of a static type system are of course less prohibitive than our ML-based system (and, needless to mention, more error-prone).

The Perl approach differs significantly from other approaches in one main aspect: The context in which code is assumed to be executed. Most languages presented here assume that unsafe code is loaded dynamically inside some host application, which must restrict access to certain unsafe language features. In Perl, this is somehow muddled by the notion that the program can always lift *taint* restrictions on its own, instead of being granted capabilities by some "greater power". The source of this lies in the fact that Perl programs are assumed to need sandboxing primarily in a CGI context, i.e. when the Perl program is hosted and executed by a web server accessible to the public. Hence, the sandboxing feature helps the programmer recognise and deal with dangers resulting from resource use that would not be relevant in a non-public context. That is, the source of mistrust is shifted away from the code and the programmer (which is known to behave benignly), and towards interactions of the program with sensitive resources that might be exposed to the public (and thus, potentially malicious users) inadvertently.

It is interesting to note that the RFC proposes to inherit restriction contexts to `eval()` calls as well, such that these calls cannot perform operations more powerful than their enclosing block allows. In Alice ML, this functionality comes for free due to the nature of our implementation.

5.4 Microsoft .NET CAS

The *Code Access Security* ([9, 10]) technology employed in Microsoft's .NET framework adopts the same view as E (chapter 5.2), namely that the POLA principle should be applied to code. Sandboxes are understood as a set of policies that apply to code of a special *identity*. This identity is made up of the application context the code appears in (for example, whether it was loaded from some site on the internet, or whether it is a plugin for some application provided by the distributor of that application), and certificates that determine who wrote the code, and who signed it. In a significant deviation from the path chosen by both Alice ML and Java, code thus does not automatically run with all the privileges of the user that executed the respective application, but with the rights granted to its own identity. That is, security is not based on user identy, but on code identity. Two qualifications have to be made to this statement, however: The first objection is that user rights still provide an upper bound for rights that can be obtained by an application, which is of course quite sensible. The other qualification is that if the code does not have a specific identity as assigned by the application, it is still run under full trust, i.e. user rights.

As in Java, resources in CAS come in the guise of permissions, such as the ever present `FileIOPermission`. Again, as in Java, these are parameterised by predefined operations and arguments relating to these, for example `PathDiscovery` and `C:\Test`. There is no

option for altering values, but new permissions can be created as easily as in Java.

Microsoft encourages the introduction of declarative *permission requests* into client applications that are expected to run in a sandboxed environment. They are extracted at compile-time and integrated in the assembly code, to be parsed by the host application when the assembly is loaded. The requested permissions should symbolise the set of minimal authorities the application needs to work correctly. This way, applications which would not be allowed to execute properly anyway can be rejected before they are actually run. Of course, this approach is directly opposed to ours, which is to let applications run as long as possible before they have to be terminated.

The difference in these approaches is probably due to the more conservative context in which CAS is applied, and it can indeed be argued that in a production setting, immediate rejection is the appropriate behaviour. In Alice, this could be implemented by dummy calls to the respective API functions, and checks to see if these go through.

6 Conclusion

The dynamic nature of modern applications, supported by the open programming style offered by many languages, has led to new risks and concerns. These are placated by the rise of appropriate techniques to counter many of those risks, with sandboxing as one such technique to guarantee safety by encapsulating dangerous processes in a controlled environment. Sandboxes work by enforcing a set of configuration rules called "policies", which can be specified by the creator of the sandbox. Policies govern the use of so-called resources, which designate critical parts of a computing system. Unrestricted, these resources could be used for malicious purposes such as deletion of files, acquisition or publication of private data, installation of viruses, abuse of computation power or net bandwidth, and many more.

In this thesis, we have presented a model describing sandboxes and policy rules on basis of an untyped, ML-like programming language. We have shown how delegation and rewriting suffice to define a modular sandboxing infrastructure, if a component system similar to the one used in Alice ML is available. We have given a working implementation of this model in Alice ML, and discussed certain pitfalls and how to avoid them. The resulting system is modular and extensible, and just as powerful as sandboxes provided by popular languages like Java or the Microsoft .NET framework. In our system, policy rules can return and modify values, a characteristic not found in any other implementation. Use of advanced Alice ML features like laziness enables us to express certain idioms succinctly and efficiently, that would otherwise be hard to implement.

There are some ideas for future work: For example, it would be desirable to integrate a system for the grouping of certain related policy rules into higher-level structures, so it would be possible to reason about sets of policies instead of just single policies. Some predefined sets would be distributed with releases of Alice ML, to provide users with a reasonable basic rule set to extend or modifiy as they see fit. Possible sets that come to mind for this purpose would be *browser applet*, *intra-corporation application* or *server-side application*, which could define some sensible default policies alongside more specific rules. In fact, this is one extension we are currently working on.

Another point would be to create a declarative-style syntax for the definition of policies in configuration files, and the implementation of a parser to read those files and generate appropriate `Policy` structures, as done in Java. This would facilitate use of widely applicable policies provided for multiple users, which could be distributed over some kind of intranet.

Lastly, a fascinating technique not properly explored in this thesis is *signature thinning*: Here, not the component itself is altered, but its signature is modified in such a way that only safe function calls are possible. For example, a general I/O component could be reduced to just performing reading operations. However, it is unclear how to communicate signature alterations to programs expecting a particular version of the system API. This problem would have to be solved, and a language facility would have to be created to allow the dynamic building of component signatures, analogous to functors which dynamically create components.

References

[1] Alice Team. The Alice System, 2006.
http://www.ps.uni-sb.de/alice/

[2] Matthew Byng-Maddick. A Sandboxing mechanism for Perl 6.
RFC, 2006. http://dev.perl.org/perl6/rfc/353.html

[3] Drew Dean, Edward Felten and Dan Wallach. Java security: from HotJava to Netscape and beyond.
Symposium on Security and Privacy pages 190–200, Oakland, USA, IEEE Computer Society Press, May 1996.

[4] Li Gong. A Secure Identity-Based Capability System.
IEEE Symposium on Security and Privacy pages 56–65, 1989.

[5] Li Gong. New Security Architectural Directions for Java.
IEEE COMPCON pages 97–102, IEEE Computer Press, 1997.

[6] Ulrich Lang. Access policies for middleware.
PhD Thesis – Technical Report 564, University of Cambridge, UK, 2003.
http://www.objectsecurity.com/ulrichlang.com/Diss.htm

[7] Xin Li. Computer Viruses: The Threat Today and the Expected Future.
Master's Thesis, Linköpings University, Sweden, 2003.
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-1998

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

[9] Microsoft Corporation. .NET Framework Developer's Guide: Code Access Security.
Technical Reference, 2006. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcodeaccesssecurity.asp

[10] Microsoft Corporation. Code Access Security in the .NET Framework 2.0.
MSDN Magazine Article, 2005.
http://msdn.microsoft.com/msdnmag/issues/05/11/CodeAccessSecurity/default.aspx

[11] Mark Samuel Miller. Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.
PhD Thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.
http://www.erights.org/talks/thesis/

[12] Robin Milner, Mads Tofte, Robert Harper and David McQueen. Definition of Standard ML (revised). *The MIT Press*, 1997.

[13] Georg Neis. A Semantics for Lazy Types.
Bachelor's Thesis, Saarland University, Saarbrücken, Germany, September 2006.
http://www.ps.uni-sb.de/∼georg/bachelor.html

[14] John H. Reppy and Emden R. Gansner (Editors). The Standard ML Basis Library. *Cambridge University Press*, October 2004. http://www.standardml.org/Basis/

[15] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus and Gert Smolka. Alice ML through the looking glass. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming*, Vol. 5, Munich, Germany, 2005. Intellect.

[16] Andreas Rossberg. The Missing Link - Dynamic Components for ML. *11th International Conference on Functional Programming*, Portland, Oregon, USA, ACM Press 2006.

[17] Marc Stiegler. The E Language in a Walnut. *Book Draft*, 2000. http://www.skyhunter.com/marcs/ewalnut.html

[18] Sun Microsystems. Java 2 Platform SE 5.0 API Documentation. *Technical Reference*, http://java.sun.com/j2se/1.5.0/docs/api/