Karlsruher Institut für Technologie

Desktop Integration Management for Portable, Zero-Install and Virtualized Applications

Bachelor Thesis
by

Bastian Eicher

at the Department of Computer Science
System Architecture Group
Karlsruhe Institute of Technology

Supervisor: Prof. Dr. Frank Bellosa
Supervising Research Assistant: Dipl.-Inform. Konrad Miller

Created during: April 6th 2011 – August 4th 2011

I hereby declare that this thesis is my own original work which I created without illegitimate help by others, that I have not used any other sources or resources than the ones indicated and that due acknowledgment is given where reference is made to the work of others.

Karlsruhe, August 4th 2011

Abstract

Classic installation systems suffer from a number of disadvantages, such as version conflicts and a lack of user control. A number of approaches to mitigate these disadvantages exist, for example portable applications, zero-install systems and application virtualization. However, these approaches do not provide the same end-user experience since they lack proper desktop integration.

In this work we determine the improvements necessary to allow a zero-install system to provide the same level of desktop integration users are accustomed to from classic installation systems while providing more user control and retaining a conflict-free environment.

We developed a domain-specific language, extending the application-metadata language of an existing zero-install system as a proof of concept as well as a real-world-suitable system. The accompanying tool set is capable of capturing systems changes made by classic installation systems and automatically generating desktop integration metadata from them.

Deutsche Zusammenfassung

Klassische Installationssysteme weisen eine Reihe von Nachteilen auf, wie z.B. Versionskonflikten und einem Mangel an Benutzerkontrolle. Es existieren mehrere Ansätze um diesen Problemen entgegen zu wirken, z.B. portable Anwendungen, installationsfreie Systeme (zero-install) und Anwendungsvirtualisierung. Diese Ansätze bieten jedoch keine gleichwertige Arbeitsumgebung für Endanwender, da ihnen die notwendige Desktopintegration fehlt.

Zero-install ist eine Methode zur Softwareverteilung, bei der zur Bereitstellung von neuen Anwendungen lediglich Dateien kopiert werden und keine Konfigurationsänderungen notwendig sind. Diese Arbeit baut auf einem zero-install System namens *Zero Install* auf.

Portable Anwendungen sind ein Sonderfall von zero-install Anwendungen, welche zusätzlich darauf ausgelegt sind, sämtliche zur Laufzeit anfallenden Daten in ihrem Installationsverzeichnis abzulegen. Auf diese Weise können die Anwendungen auf tragbaren Datenträgern gespeichert und an unterschiedlichen Rechnern verwendet werden.

Anwendungsvirtualisierung ist eine Methode, um bestehende Anwendungen automatisiert in portable Anwendungen zu überführen. Hierzu werden Zugriffe auf das Betriebssystem teilweise in eine virtuelle Umgebung umgeleitet.

Das Ziel dieser Arbeit ist es, die Desktopintegration klassischer Installationssysteme zu analysieren und *Zero Install* dahingehend zu erweitern Endanwendern eine gleichwertige Benutzeroberfläche beim Starten von Anwendungen, Öffnen von Dateien usw. zu bietet.

Wir untersuchen schwerpunktmäßig die Möglichkeiten zur Desktopintegration, die von der *Windows* Plattform angeboten werden. Hierzu bauen wir primär auf der offiziellen Dokumentation auf, analysieren aber zusätzlich die Systemänderungen, die von Installationssystemen bekannter Anwendungen vorgenommen werden, um sicherzustellen, dass diese Daten für den alltäglichen Einsatz repräsentativ sind.

Zero Install beschreibt Softwarekomponenten und Abhängigkeiten zwischen ihnen in einem XML Format, dass *Feed Format* genannt wird. Diese Arbeit zeigt den Entwurf einer zusätzlichen domänenspezifischen Sprache zur Beschreibung von Desktopintegration, welche in das *Feed Format* eingebettet wird, auf.

Um Desktopintegration klassischer Installationssysteme schneller abbilden zu können, haben wir ein Tool entwickelt, welches Systemänderungen automatisiert erfasst und in der domänenspezifischen Sprache abspeichert.

Den Kern der Implementierung bildet eine Erweiterung des *Windows*-Ports von *Zero Install*. Diese Erweiterung kann in *Feeds* eingebettete Desktopintegrationsdaten verwenden, um Anwendungen auf Anfrage des Benutzers ins System einzubinden. Sämtliche vorgenommenen Änderungen werden aufgezeichnet und können somit rückgängig gemacht oder mit anderen Computern synchronisiert werden.

Um sicherzustellen, dass Anwendungen, die über *Zero Install* gestartet werden, während ihrer Laufzeit keine ungewollten Änderungen an der Desktopintegration vornehmen, fangen wir Zugriffe auf die System API ab und passen die Methodenparameter bei Bedarf an.

Die Evaluierung der Implementierung hat gezeigt, dass bei alltäglichen Anwendungsfällen keine sichtbaren Unterschiede zwischen der Desktopintegration durch klassische Installationsprogramme und durch *Zero Install* vorliegen.

Die aktuelle Version der Implementierung verlangsamt API Aufrufe für die *Windows* Registrierung erheblich. In der durchschnittlichen Startdauer und Reaktionsgeschwindigkeit der getesteten Anwendungen schlägt sich dies jedoch nicht merklich nieder.

Bei einigen Anwendungen führt das Abfangen der System APIs zu Speicherzugriffsverletzungen. Daher ist dieses Feature in seiner derzeitigen Form noch nicht für den Alltagseinsatz geeignet. Zukünftige Versionen könnten diesem Problem mit Techniken aus dem Bereich der Anwendungsvirtualisierung begegnen.

Weitere Möglichkeiten für zukünftige Verbesserungen sind Unterstützung für *Linux* und *Mac OS X* sowie eine Nutzung der Desktopintegration zur Umleitung von Anwendungsstarts in virtuelle Maschinen oder auf Terminalserver.

Contents

Abstract v

Deutsche Zusammenfassung vii

Contents ix

1 Introduction 1
- 1.1 Problem definition 1
- 1.2 Objectives 3
- 1.3 Methodology 3
- 1.4 Contribution 4

2 Background 5
- 2.1 Common terms 5
- 2.2 Zero Install 8
- 2.3 Related work 11

3 Analysis 13
- 3.1 Platform documentation 13
- 3.2 Installation monitoring 18
- 3.3 Definitions 20
- 3.4 Capturing 21

4 Design 23
- 4.1 Goals 23
- 4.2 Domain-specific languages 24
- 4.3 Capturing 26
- 4.4 Applying desktop integration 26
- 4.5 Synchronization 27

5 Implementation 29

5.1 Tools and libraries 29
5.2 Installation capturing 30
5.3 Applying desktop integration 31
5.4 Stubs 32
5.5 API hooking 33

6 Evaluation **37**
6.1 Methodology 37
6.2 Use cases 38
6.3 Benchmarks 39
6.4 Results 40
6.5 Discussion 40

7 Conclusion **45**
7.1 Future work 45

A Screenshots **47**

B XML formats **55**

C API hooking **59**

Acknowledgments **63**

Bibliography **65**

Chapter 1

Introduction

This chapter gives you an overview of the limitations of current installation systems and the methods we used to mitigate them.

1.1 Problem definition

Classic installation systems such as *Windows* installers and *Linux* package managers (e.g., *RPM* and *DPKG*) suffer from a number of disadvantages:

User control Changes made to the desktop environment happen as a side-effect of installing an application instead of explicitly at the user's request. This means that users must rely on application packagers to provide sensible integration defaults.

Uninstall *Windows* applications are installed in individual sub-directories of the *Program Files* directory. However, it is also common to place shared library files in the *Windows* directory and other locations. Therefore, the complete removal of an installed application depends on the correct behavior of its uninstall tool, which is not enforced by the operating system.

Linux distributions following the *File System Hierarchy* standard distribute applications' files across various directories. These directories are intended to group files based on their type (e.g., executable or library) and their shareability (e.g., current user, local system or network-wide). However, they do not provide any means for determining which files belong to which application. Therefore, only the package manager can determine which files need

to be removed when uninstalling an application. A manual cleanup in case of a damaged package manager database is generally not feasible. [1] [10]

Multiple versions Since *Windows* applications are installed into separate directories they are inherently relocatable[1]. Therefore, multiple versions of an application can be installed in different directories simultaneously. However, the desktop integration performed by the last version to be installed will take precedence over the others.

Linux distributions traditionally place the executables for all installed applications in the *search path*[2]. This means that only a single version of an application carrying a specific name can be installed at any given time.

Shared libraries Shared libraries are called *DLL files* (*Dynamic-Link Library*) on the *Windows* platform. When an application requires a specific library *Windows* searches for it in a number of locations, retrieving the first file that matches the name. [2]

Since this method provides no versioning support installing multiple applications that require different versions of the same library leads to conflicts. This problem is colloquially referred to as *DLL hell*. *Windows XP* introduced a new feature to mitigate this problem called *Side-by-side (SxS) assemblies*. A *SxS assembly* combines a library and any additional data files it may require with versioning metadata. Applications can use a so-called *application manifests* (usually embedded within the executable) to specify precisely which version of a library they require. While this prevents DLL conflicts it requires existing applications and libraries to be updated to use *SxS*. [3]

On *Linux* systems library filenames contain version numbers allowing multiple version to be installed in parallel. A field in the library files called `soname` contains a string identifier denoting compatible versions. This is used to create *symbolic links* between different versions of a library that implement the same interface. [4]

The aforementioned problems can mostly be traced back to a few root causes:

- Installation processes are, at their core, sequential scripts instead of declarative data stores. This limits the amount of control an outside system (i.e., package manager) can excerpt over the installation details. [11]

[1] A *relocatable* application is an application that can be installed to different locations within the file system without requiring to be recompiled.

[2] The *search path* is an environment variable specifying a set of directories the operating system will search in order to locate an executable file that was specified without an explicit path.

- Installing new components is seen as a modification of system state instead of a non-invasive addition of a component to a list. Therefore side-effect-free installations are inherently impossible.

Portable applications, zero-install systems and application virtualization aim to mitigate these problems. However, these approaches do not provide the same end-user experience since they lack proper desktop integration (registering as handler for file types, context menu entries, etc.).

1.2 Objectives

The objective of this work is to analyze integration points used by classic installation systems and to enhance an existing zero-install system so it can replace them. The final result should provide an equivalent end-user experience when launching applications, opening files, etc..

The new system should work with a wide range of unmodified applications (as released by the original developers). This work will use the following applications as a guiding sample: *Mozilla Firefox* as the default browser, *Mozilla Thunderbird* as the default mail client, *Pidgin* as an instant messenger and the *OpenOffice.org* office suite.

1.3 Methodology

To determine what constitutes *desktop integration* in everyday usage, we selected a number of applications using classic installation systems and observed their installation and runtime behavior.

We developed a domain-specific language for describing an application's desktop integration as well as tools for capturing and reapplying such data. These features are built upon the existing *Zero Install* system (see chapter 2.2).

The evaluation compares the end-user experience as well as installation and runtime performance of applications installed using their normal installation system with applications accessed via a zero-install system augmented by the aforementioned tools.

1.4 Contribution

The contribution of this work is an analysis of system modifications performed during software installations on *Windows* systems and a zero-install system capable of reproducing the same effects without suffering from the problems described in Chapter 1.1.

This makes it possible for users to combine traditional workflows with new options, such as opening files via double-click before the corresponding application is even stored on the local disk or switching the version of the default browser on the fly.

Chapter 2

Background

This chapter defines common terms used in this work and introduces the software the implementation is based upon. It also presents related work and different approaches to improving software installation while handling desktop integration.

2.1 Common terms

Zero-install A zero-install application is an application that requires no invasive modification to the system state in order to be executable. Installation only consists of copying a directory structure to an arbitrary location. In the *Windows* world this kind of installation is known as xcopy[1] deployment.

The term is not be confused with *Zero Install*, which is a tool used for managing zero-install applications (see Chapter 2.2).

Portable A portable application is a zero-install application that stores user preferences and any other run-time generated data in locations relative to its installation directory instead of the user profile. This kind of application can be carried around by users as a self-contained working environment on a portable medium such as a USB stick.

Windows applications are often available in modified portable versions. A popular website offering such applications is PortableApps.com. The unmodified binaries of the applications are bundled with a tool called the *PortableApps.com Launcher*, which temporarily changes the system envi-

[1] `xcopy` is a *DOS/Windows* command-line tool for copying entire directory structures.

ronment before launching the application and restores the original system state after the application terminates[5].

The term *portable* carries a certain ambiguity since it may also refer to an application being able to run on multiple operating systems and/or hardware architectures. To avoid confusion this work will instead use the term *cross-platform* to distinguish such applications.

Virtualized A virtualized application is a regular application that was turned into a zero-install or portable application using an application virtualization system.

Application virtualization isolates an application from the underlying operating system by redirecting system calls to a virtualization layer. This layer will usually overlay the operating system's actual file system (and registry) with virtual stores containing required libraries, configuration files, etc.. This also makes it possible to protect the operating system from unwanted changes while selectively allowing modifications to certain areas (e.g., saving files).

Commercial application virtualization products include *VMware ThinApp*, *Microsoft App-V* and *InstallFree Bridge*. These products all enable the user to package existing applications into stand-alone executables[2].

Similar effects can sometimes be achieved by running applications within full-fledged virtual machines with their own operating systems while providing a communication channel between the physical and virtual machine. This method can be more reliable than application virtualization because it provides a much more complete runtime environment for the virtualized application. However, since this requires an additional operating system to be running concurrently it is far less efficient.

Native An application that is not a zero-install application, i.e., an application that modifies the system state during installation, is referred to as a native application.

This term is also potentially ambiguous, since *native* can refer to an application running directly on the system's hardware as opposed to a Virtual Machine (VM) as used in *Java* and the *.NET Framework*. Since VM-based languages are not the focus of this work, *native applications* can henceforth be assumed to mean *non-zero-install applications* in this work.

[2] A stand-alone executable is an executable that does not depend on any external libraries other than those provided by the operating system itself.

Desktop integration Apart from copying files to a specific directory, the installation of a native application usually consists of a number of modifications to configuration files associated with the desktop environment (*GNOME*, *KDE*, the *Windows* shell[3], etc.). The purpose is to provide easy access to the application's features (e.g., open certain file types, browse the web) or to integrate it with other tools (e.g., add a "burn to CD" function to a file manager). This kind of modification is called *desktop integration*. It is distinct from modification necessary for an application to run in the first place (e.g., registering shared libraries).

In the *Windows* world the term *shell integration* is often used as a synonym for *desktop integration*.

Figure 2.1 recapitulates the relations between the aforementioned terms.

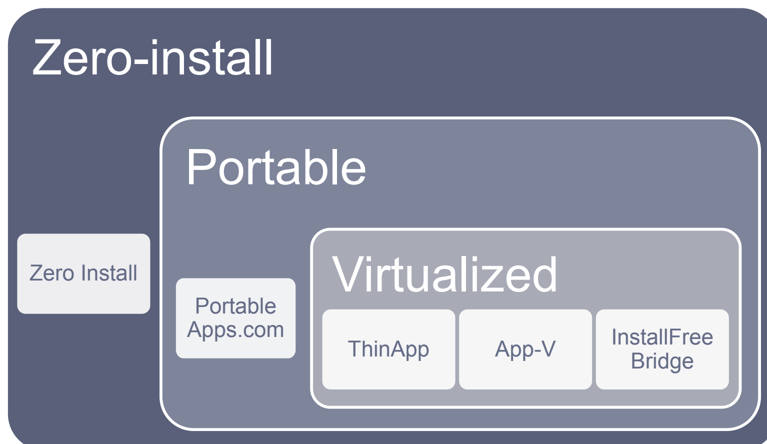

Figure 2.1: Virtualized applications are a subset of portable applications, which in turn are a subset of zero-install applications.

[3]The *Windows* shell comprises the desktop, the start menu and any *Windows Explorer* windows. Upon login an instance of `explorer.exe` is automatically loaded in order to provide the shell. While it can be replaced by another executable, the *Windows* shell is considered to be an integral part of the operating system.

2.2 Zero Install

Zero Install is a decentralized platform-independent software-installation system. It makes it possible to publish individual applications via their project websites instead of using central repositories while still providing features such as shared libraries, automatic updates and digital signatures.

Zero Install specifies an XML metadata format for describing software components and relationships between them. A single metadata file can contain data for multiple platforms (e.g., *Windows*, *Linux* and *Mac OS X*).

Zero Install also provides features that go beyond native installation systems. While it reuses shared libraries whenever possible, it can also handle multiple versions of a component on a single system at the same time. Unlike *Windows*' *SxS*, *Zero Install* does not require components to be modified to support this. The user can manually control which library versions *Zero Install* shall use for a specific instance of an application. [6]

Installations have no side effects since each software component is extracted into a separate directory. This also enabled atomic updates, negating the risk of corrupting the system in case of a system crash or power failure during an installation. Users can install new applications without requiring an administrator. [7]

2.2.1 Metadata format

The XML metadata format used by *Zero Install* is often called the *feed format* since these XML files are used in a fashion similar to RSS feeds: The file is reachable via a fixed URL and is regularly updated with new releases.

An *interface* is an HTTP URI that uniquely identifies a *Zero Install* software component, while a *feed* is the actual XML metadata file containing information about the component. Usually the interface URI[4] is identical to the feed URL[5]; however, additional feeds can be registered for an interface. This can, for example, be used to provide an alternative build of an application as a drop-in replacement. A specific version of a software component listed in a feed is called an *implementation* of the interface.

A software component identified by a *Zero Install* interface can either be an executable application or a library. Feeds for the former specify a default launch

[4] A *Uniform Resource Identifier* (URI) specifies the identity of a resource on the internet.

[5] A *Uniform Resource Locater* (URL) specifies the location of a resource on the internet as well as a method of retrieving it. Every URL is a URI but not vice versa.

command while feeds for the latter do not. Feeds can also contain alternative launch commands, accommodating scenarios where a single software component consists of multiple executables (e.g., *OpenOffice.org*). [8]

Figure B.1 in the appendix shows the complete structure of the feed format.

2.2.2 Launching an application

When *Zero Install* is instructed to launch an application it processes the request in three phases:

`select` phase In this phase the *Zero Install* solver[6] consults all relevant feeds in order to select a set of implementations that best satisfy the user's request (the specified interface URI and optionally additional requirements such as specific version numbers or platforms) as well as the dependencies between the components.

`download` phase In this phase *Zero Install* downloads any required implementations. See Chapter 2.2.3 for details.

`run` phase In this phase *Zero Install* combines the selected and downloaded implementations to form an executable environment. Based on the principle of dependency injection[7] implementations are made discoverable by each other by adding their paths to per-process environment variables without affecting the rest of the system.

Figure 2.2: The *Zero Install* launch process is split into the phases `select`, `download` and `run`. Each phase depends on the results of the previous one.

[6] *Zero Install* uses a SAT-solver based on MiniSat.

[7] *Dependency injection* is a design principle wherein components do not locate their dependencies themselves. Instead, an external dependency manager locates and then *injects* them.

2.2.3 Retrieving implementations

Zero Install identifies implementations via cryptographic hashes[8] of their contents. This makes it possible to verify downloads and share them among users safely.[9] These hashes are called *manifest digests*.

After the solver has selected a set of implementations to satisfy the user's request *Zero Install* checks a local cache directory called the *implementation cache*. Each subdirectory represents one implementation and carries its manifest digest as the name. If a required implementation is missing from the cache *Zero Install* consults the feed for instructions on how to retrieve it from the internet.

An archive (e.g., `ZIP` or `TAR.GZ`) is downloaded from an HTTP or FTP server and then extracted to a temporary directory. Next, a so called *manifest* file – listing the complete directory structure (sub directories, file names, timestamps, etc.) as well as hashes of each of the files' contents – is generated. Finally, a hash of the manifest file (i.e., *manifest digest*) is calculated. Figure 2.3 depicts this process.

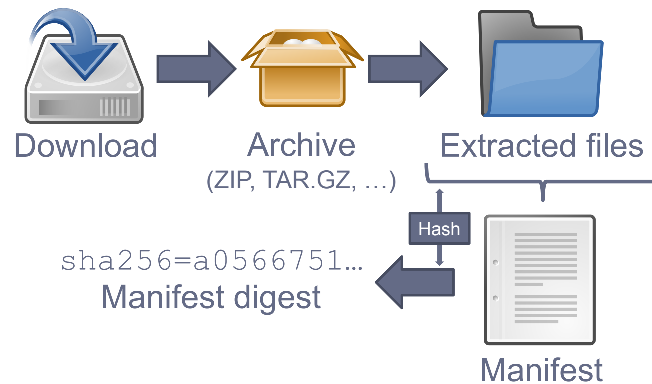

Figure 2.3: *Zero Install* retrieves implementations by downloading an archive, extracting it, generating a manifest of the contents and hashing the manifest. This figure contains content from the Tango Icon Library placed in the public domain.

Zero Install also supports alternative methods for retrieving implementations, such as combining the contents of multiple archives, but the method for generating manifest digests stays the same.

[8] As of version 1.2 *Zero Install* supports the hashing algorithms *SHA-1* and *SHA-256*.

2.3 Related work

Most systems for software installation on desktop operating systems need to deal with desktop integration in some fashion. Native installation systems usually modify the desktop environment as a step in the installation process. Zero-install systems need to handle this in a different way.

2.3.1 Alternative *Linux* distributions

The *Linux* distribution *GoboLinux* installs each version of a component in a self-contained directory much like *Zero Install*. This provides a database-less package management system. To maintain backwards compatibility with traditional *Linux* file system layouts the classic directories for executables and libraries still exist and are populated with symbolic links to the currently active version of each component. These symbolic links also determine which version of an application is used for desktop integration. [10]

NixOS is an experimental *Linux* distribution based on the package manager *Nix*. Unlike most other package managers *Nix* describes packages and system states using a functional language. This enables atomic upgrades and rollbacks. On *NixOS* new software is not *installed*. Instead a new system configuration is described and then activated. Activating a configuration creates symbolic links and desktop integration entries for the selected components in a fashion similar to *GoboLinux*. [11]

Both *GoboLinux* and *NixOS* provide many of the same advantages as *Zero Install*. However, applying desktop integration is still handled as a side-effect of installing or activating an application. Both systems require applications to be recompiled to support their directory naming schemes.

2.3.2 Application virtualization tools

VMware ThinApp[12], *Microsoft App-V*[15] and *InstallFree Bridge*[16] isolate the operating system from any changes virtualized applications attempt to make. If an application performed any modifications for desktop integration during its installation phase it will still see them as applied within its virtual environment, however they will not be visible to other applications.

VMware ThinApp compares snapshots of the system before and after an application was installed. It uses the differences between these two snapshots to build a

virtual environment for the application. It also extracts changes relevant for desktop integration (mainly file associations) and stores them as metadata accessible outside the virtual environment. A command-line tool named `thinreg` reads this metadata and creates shortcuts and entries in the real registry to point to the virtualized executable file. [12]

Microsoft App-V uses a similar snapshot technique. It stores information about file associations using a modified version of the now obsolete *Open Software Description Format*[13][14]. When an application is deployed (using the `SFTMIME` command-line tool) shortcuts and registry entries are created. [15]

InstallFree Bridge uses a different technique for capturing application installations: Setup programs run within a virtualized environment and any changes they make are recorded directly. Desktop integration is handled in a fashion equivalent to the other solution: Shortcuts, file associations, etc., are stored in the so-called *Shell Shadow* and applied to the real system by the *Bridge Virtual Agent*. [16]

Application virtualization usually aims at reproducing the work environments users are already accustomed to while removing the deep entanglement between the applications and the operating system in order to simplify administrative tasks such as installing and upgrading. In the most basic case, deploying a virtualized application consists of copying a single executable file to the target computer and creating a desktop shortcut pointing to that executable file. In order to accommodate the aforementioned methods for additional desktop integration this process is usually wrapped inside a native installation system.

In the end, deploying a virtualized application looks remarkably similar to installing a regular application. Some form of setup EXE or a *Windows Installer* package is executed. However, since these installers were created by the administrator (within the confines of the virtualization system) it is easier to keep track of the system changes being made.

Application virtualization systems provide solutions for desktop integration, however they do not handle desktop integration changes attempted at runtime. Such changes will simply vanish into the virtual environment, not having any effect on the desktop environment. Unlike the solution presented in this work, they only provide the packager/administrator, not the user, with the opportunity to control which elements of the desktop integration they wish to apply. They do not attempt to handle conflicts between integration attempts made by different installations.

Application virtualization is generally not intended for use by upstream developers. Instead, it is geared towards use by in-house IT departments. Non-virtualized zero-install applications, on the other hand, can be published directly by their developers.

Chapter 3

Analysis

This chapter analyzes how desktop integration works on the *Windows* platform. First, relevant vendor documentation is referenced. Then, real-world installations are analyzed in order to determine whether all relevant aspects were covered.

3.1 Platform documentation

The *MSDN Library* is the primary documentation source for *Microsoft* products such as *Windows*. It covers the following aspects relevant for desktop integration.

3.1.1 Shortcuts

The simplest form of desktop integration on the *Windows* platform is the creation of shortcut files. Shortcut files are files with the file ending `.lnk`. They store a target path and optionally an icon path, a working directory path and a short description. They are handled by the `explorer.exe` shell and appear to the user in a fashion similar to symbolic links. However, other applications see them as normal files with binary content. Shortcuts can be placed anywhere, but they are most commonly located in the *Start menu* and on the desktop.

Windows 98 introduced the *Quick Launch bar* as an additional location for shortcuts. It is usually displayed next to the *Start menu* button on the taskbar. *Windows 7* replaced the *Quick Launch bar* with the concept of pinnable applications, which are also based on shortcut files.

3.1.2 File associations

Windows differentiates file types based on their extension. A list of known file types is stored in the HKEY_CLASSES_ROOT subtree of the registry. Each extension is associated with a *programmatic identifier*, which in turn contains information about the file type. Multiple extensions can point to a single programmatic identifier, reflecting the possibility of multiple extensions being in use for a single file type (e.g., .htm and .html for HTML).

A programmatic identifier specifies a human-readable description of the file type, an icon path and a set of *verb*s. A verb is an operation that can be performed on a file such as *open* or *edit*. Each verb maps to a specific command-line into which the path of the file in question is inserted.

An extension can reference additional programmatic identifiers to allow for multiple applications to handle a single file type. This is represented by the *Open with* dialog in the shell. (See Figure A.3 in the appendix for a screenshot.) However, only the primary programmatic identifier controls the file type description and icon displayed in the shell. [17]

Figure 3.1 illustrates the aforementioned relations.

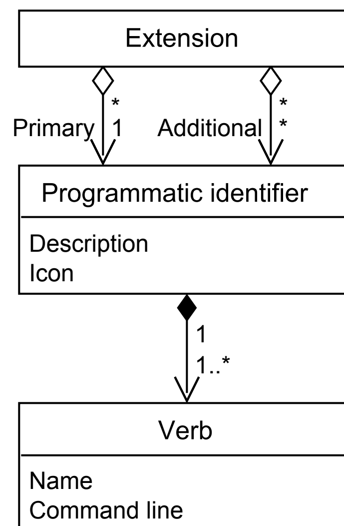

Figure 3.1: In the *Windows* registry every file extension points to one primary and an arbitrary number of additional programmatic identifiers. Multiple extensions can share a single programmatic identifier. Every programmatic identifier specifies one or more verbs.

3.1.3 URL protocols

A URL protocol is the first part of an absolute URL such as `http` or `ftp`. It usually denotes the communication protocol to be used when contacting a remote server, however some URL protocols describe an action instead (e.g., `mailto` instructs a mail client to create a new mail but not to send it yet).

Windows registers URL protocols directly as *programmatic identifiers*. In order to support multiple applications for a single protocol *Windows Vista* introduced an additional per-user setting selecting the programmatic identifiers to use for certain well-known protocols. [18]

3.1.4 Context menus

Windows aggregates data from numerous different sources to populate the context menu displayed for a file. A so-called *association array* specifies a set of registry keys which are queried for data about a particular file type. Among these are the special programmatic identifiers `AllFilesystemObjects` representing all files and directories and * representing all files but not directories. [19]

If an application provides a service applicable to all types of files or entire directories (e.g., backup tools or virus scanners) it can provide access to that service by adding a verb to one of the aforementioned programmatic identifiers. [20] (See Figure A.2 in the appendix for a screenshot.)

In addition to static menu entries that always appear for specific file types, *Windows* provides a feature called *dynamic verbs*. These are context menu entries generated on-demand by COM objects loaded into the shell. These entries can vary based on a file's content and not only its extension. [21]

3.1.5 Sent to menu

Windows displays a *Send To* sub-menu in the context menus for files and directories. (See Figure A.1 in the appendix for a screenshot.) The entries in this menu are determined by the shortcut files located in the *SendTo* directory. Clicking on one of them launches the according application and passes the selected file or directory as a command-line argument.

3.1.6 Default programs

Set Program Access and Computer Defaults (*SPAD*) is a user interface introduced in *Windows 2000 Service Pack 3* and *Windows XP Service Pack 1* to provide the user with a central location for choosing a default browser, mail client, instant messenger and media player. (See Figure A.5 in the appendix for a screenshot.)

This feature does not perform any configuration changes by itself. Instead, it allows applications to register commands that will be called when the user selects new defaults. The applications are then responsible for registering themselves appropriately. Any changes made via this user interface are applied system-wide and therefore require administrative privileges. [22]

The *SPAD* registry entries for mail clients contain information on how to handle requests to the `mailto:` URI scheme. The entries for browsers and mail clients also control the *Browser* and *Mail* icons displayed in the start menu up until *Windows Vista*. [23]

In *Windows Vista SPAD* was superseded by the *Set Your Default Programs* (*SYDP*) user interface. The old *SPAD* interface is still available, however changes made via the *SYDP* interface are effective on a per-user level and override the system-wide entries created by the *SPAD* interface. (See Figure A.6 in the appendix for a screenshot.)

The *SYDP* interface requires applications to explicitly register their capabilities (i.e., the file types they can open and whether they can act as the default browser or the default mail client). It allows the user to select which capabilities to apply as defaults in detail. (See Figure A.7 in the appendix for a screenshot.) The modifications are applied by the operating system itself instead of delegating the task to the applications. [24]

3.1.7 AutoPlay handlers

When *Windows* detects a new medium such as an USB stick or a blank DVD it automatically displays a dialog box offering the user a number of operations applicable to that type of medium. (See Figure A.4 in the appendix for a screenshot.) This feature is called *AutoPlay*.

The arrival of a new medium is called an *AutoPlay event*. An application can register itself as a handler for such an event, specifying a description of the operation it will perform (e.g., "Burn data on the CD"), an icon and the command-line to be executed, should the user chose this option. [25]

3.1.8 COM servers

The *Component Object Model* (*COM*) is an object-oriented system for creating software components that can interact within and across process boundaries. [26]

Some applications provide automation interfaces for their services to be consumed by other applications via *COM*. For example a media player could provide an API that allows other applications to query and manipulate its database of songs.

Such interfaces are provided via an *In-Process COM server* (loading dynamically linked libraries at runtime) or an *Out-of-Process COM server* (using inter-process communication). *Windows* uses the registry to locate the necessary files.

3.1.9 Games Explorer

Windows Vista introduced a new user interface called the *Games Explorer*. In addition to being a central listing of all installed games it can display game statistics, provide parental controls, etc.. In order for an application to be listed here it must provide a *game definition file* (*GDF*) embedded within an executable file or a DLL. [27]

3.2 Installation monitoring

We selected a number of popular applications, listed in Table 3.1, covering all forms of desktop integration mentioned in Chapter 3.1. We installed each of these applications on both a *Windows XP* and a *Windows 7* virtual machine and monitored the changes their installers made using *Sysinternals Process Monitor*[1][28].

We applied the following rules when installing to ensure consistent results:

- Install any required runtime environments such as the *.NET Framework* or the *Java Runtime Environment* before beginning the actual analysis, since they are not the subject of this work.
- If the installer offers the possibility to install either "For the current user" or "For all users" perform the entire analysis once for each option.
- Enable all file type associations provided by the installer.
- Enable any additional integration options (e.g., create shortcuts or add installation directory to PATH) provided by the installer.
- Never install any bundled third-party applications.
- If the installer offers the possibility to install the application in a "portable mode", storing its settings in its installation directory instead of the user profile, do not select this option.

All modifications we detected pertinent to desktop integration are covered by Chapter 3.1.

Some of the installers performed different modification based on the *Windows* version they were executed on. They created registry entries for features that are only available on newer *Windows* versions, such as *SYDP* (see Chapter 3.1.6), only when the current operating system supported them. Other installers simply created registry entries for all features they supported.

[1]"*Process Monitor* is an advanced monitoring tool for *Windows* that shows real-time file system, Registry and process/thread activity."[28]

Application name	**Version**	**License**	**Detected modifications**
7-zip	9.20	GNU LGPL	Context menu (Chapter 3.1.4)
Audacity	1.2.6	GNU GPL	
Blender	2.5.7	GNU GPL	
CDBurnerXP	4.3.8.2523	Freeware	*SYDP* (Chapter 3.1.6), *AutoPlay* (Chapter 3.1.7)
foobar2000	1.1.6	Freeware	*SYDP* (Chapter 3.1.6)
Inkscape	0.48.1-2	GNU GPL	
IrfanView	4.28	Freeware	*SYDP* (Chapter 3.1.6)
iTunes	10.2.3	Apple EULA	Browser plug-in, *SPAD* (Chapter 3.1.6), *SYDP* (Chapter 3.1.6), *AutoPlay* (Chapter 3.1.7), COM server (Chapter 3.1.8)
KeePass	1.19b	GNU GPL	
KeePass	2.15	GNU GPL	
LibreOffice	3.4.0	GNU LGPL	*SYDP* (Chapter 3.1.6), Browser plug-in
Mozilla Firefox	5.0	MPL	*SPAD* (Chapter 3.1.6), *SYDP* (Chapter 3.1.6)
Mozilla Thunderbird	5.0	MPL	*SPAD* (Chapter 3.1.6), *SYDP* (Chapter 3.1.6)
Notepad++	5.9	GNU GPL	Context menu (Chapter 3.1.4)
OpenOffice.org	3.3.0	GNU LGPL	*SYDP* (Chapter 3.1.6), Browser plug-in
Opera	11.10	Opera EULA	*SPAD* (Chapter 3.1.6), *SYDP* (Chapter 3.1.6)
Pidgin	2.7.11	GNU GPL	URL protocols (Chapter 3.1.3)
Skype	5.3	Skype EULA	*SendTo* menu (Chapter 3.1.5), *SPAD* (Chapter 3.1.6), *SYDP* (Chapter 3.1.6)
Texmaker	3.0.2	GNU GPL	
VLC	1.1.9	GNU GPL	Browser plug-in, *SYDP* (Chapter 3.1.6)
WinMerge	2.12.4	GNU GPL	Context menu (Chapter 3.1.4), Add to system `PATH`
WinRAR	4.0	Shareware	Context menu (Chapter 3.1.4), *SYDP* (Chapter 3.1.6)
yEd	3.7	Freeware	

Table 3.1: This table lists applications we installed to analyze the system modifications their installers performed. It highlights any modifications going beyond the registration of file types.

3.3 Definitions

In everyday usage multiple applications often compete as handlers for a single element (e.g., default handler for a file type). By separating *capabilities* from *access points* user intentions can be expressed more clearly.

Capabilities A capability tells the desktop environment what an application can do and in which fashion this can be represented to the user. It does not change the behavior of existing UI elements.

Examples:

- Add an application to the "Open with" list for a file type, but do not change the default application for opening it
- Show an application as a candidate for default web browser, mail client, news reader, etc. but do not set it as such

Access points Access points build upon capabilities and represent more invasive changes to the desktop environment's UI.

Examples:

- Register as default application to open a file type with
- Register as default web browser, mail client, news reader, etc.
- Create a desktop shortcut

An exhaustive list of capabilities and access points can be found in Chapter 4.2.

Classic installation systems usually do not discriminate between capabilities and access points. Modern desktop environments themselves do, to a certain degree (e.g., *Windows*' *Default Programs* (see Chapter 3.1.6) and *GNOME*'s *Preferred applications*[29]).

3.4 Capturing

Desktop integration on the *Windows* platform is mainly governed by the creation of new registry entries and shortcut files. In order to add desktop integration support to a zero-install system, we need to extract these data and store them in a sufficiently abstract form to accommodate for the differences between launching a native applications and using a zero-install system.

There are a number of different methods commonly used for capturing system modifications:

Snapshots Snapshot capturing works by recording a snapshot of the system state before the installation and comparing it with the state afterwards. Cloning the entire system for this purpose is usually not practical. Instead, a combination of directory listings, file hashes and registry exports is employed.

This method easily handles installations that require system reboots or involve inter-process communication, but it may pick up a lot of background noise[2] from other system modifications. Modifications might be lost if they were already applied before the initial snapshot was taken. Therefore, one should use a clean operating system install in a virtual machine as a baseline.

VMware ThinApp[12] and *Microsoft App-V*[14] use this method to build virtual environments from application installations.

Syscall hooks System-call hooks redirect API calls of a specific process. This makes it possible to specifically monitor the modifications made by a single executable, eliminating any background noise.

Simulating parts of the operating system's API is a relatively complex task and somewhat error prone. Handling special cases such as communication with the system-wide *Windows Installer* service may be prohibitively complicated. Installations that require system reboots are also difficult to handle since the hooks need to be reinstalled if the installer resumes after the reboot.

InstallFree Bridge uses a similar method. Installers are executed in an isolated virtual environment called the *IFV Engine Virtual Environment*. Any attempted modifications are recorded instead of being applied to the real system.[16]

[2] On a typical *Windows* system there are hundreds of file system and registry accesses every second, even when the system is seemingly idle.

System APIs *Windows* offers APIs for monitoring the file system[30] and the registry[31] for modifications. While these are reliable and easy to use, they do not discriminate between modifications made by different processes. Thus, inter-process communication during installation is not an issue while background noise is. In the case of reboots the monitoring application needs to be resumed before the installer.

Table 3.2 compares the three capturing methods.

Method	Isolates processes	Handles reboots easily	Easy to use
Snapshots	No	Yes	Yes
Syscall hooks	Yes	No	No
System APIs	No	No	Yes

Table 3.2: This table compares different methods for capturing system modifications. Methods that isolate processes reduce background noise but make handling inter-process communication more difficult.

Chapter 4

Design

This chapter describes the design of a domain-specific language (DSL) for storing applications' desktop integration capabilities. It also presents methods for automatically recording such capabilities by capturing changes made by native installers. Finally, we elaborate on methods for making capabilities accessible to the user via *Zero Install*.

4.1 Goals

The analysis in Chapter 3 produced a comprehensive list of methods used to provide desktop integration on the *Windows* platform. A desktop integration solution for a zero-install system should be able to handle all applicable integration methods without sacrificing features such as adding new applications without administrative rights. Any method a user might normally employ to launch an application should be **tunneled through the zero-install system**.

New operating system versions often introduce new methods for interacting with applications (e.g., the *Windows 7* taskbar). The desktop integration solution should be **easily extensible** in order to accommodate such new features.

Zero-install systems are inherently more declarative than classic installation systems. A classic installer might check the current operating system version, the CPU type and the user interface language upon startup to determine which binaries to deploy. A zero-install system will defer these decisions to runtime. However, when applying desktop integration localization decisions (e.g., should a context menu entry be named "Open" or "Öffnen") need be handled at deployment time.

The desktop integration solution should therefore provide **localization mechanisms** for any user-visible metadata it may store.

While the focus of this work is on desktop integration for the *Windows* platform, many of the explored concepts are common to all current desktop operating systems. *Windows*, *GNOME*, *KDE* and *Mac OS X* all associate file types with applications that can handle them (although some systems do not exclusively use file endings to determine file types). These systems also share concepts such as having a *default browser*. The desktop integration solution should therefore consider **cross-platform support** where applicable.

One of the major advantages of using *Zero Install* is the ability to deploy new software without requiring administrative rights. Therefore, the desktop integration solution should apply all changes **per-user** by default. Administrators should also be able to apply settings system-wide.

4.2 Domain-specific languages

We designed two closely related DSLs to augment the existing *Zero Install* feed format (see Chapter 2.2.1):

Capabilities The *Capabilities DSL* is embedded within *Zero Install* feeds. The existing feed format describes how to launch commands provided by an application. The new DSL specifies what these commands can be used for and how this can be presented to the user.

We have also introduced a new concept to the feed format itself complementing the existing *commands*: An *entry point* is a command provided by most (usually all) implementations of an interface, decorated with additional information such as a localizable name and description and an icon.

Each feed can contain zero or more *capability lists*. A capability list groups a set of capabilities (see Chapter 3.3) that are applicable to a specific architecture (operating system and/or CPU type).

Access points The *Application list and access points DSL* is used to locally store user preferences for desktop integration. The application list contains an entry for each application the user requested desktop integration for.

Each application entry contains a copy of the capability lists from the feed. This ensures that updates made to the application, and thus the feed, cause no unexpected changes or inconsistencies in the desktop integration unless

the user explicitly requests an update. The application itself is still kept up-to-date automatically by *Zero Install*.

Each application entry contains a list of access points (see Chapter 3.3) representing the user's integration choices. Each access point either points to a capability (usually registering a default handler of some kind) or to an entry point (e.g., adding a new icon or menu entry).

Figure 4.1 illustrates the aforementioned relationships between the different DSLs. Table 4.1 enumerates all capabilities and access points supported by the DSLs and displays the relationships between them where present.

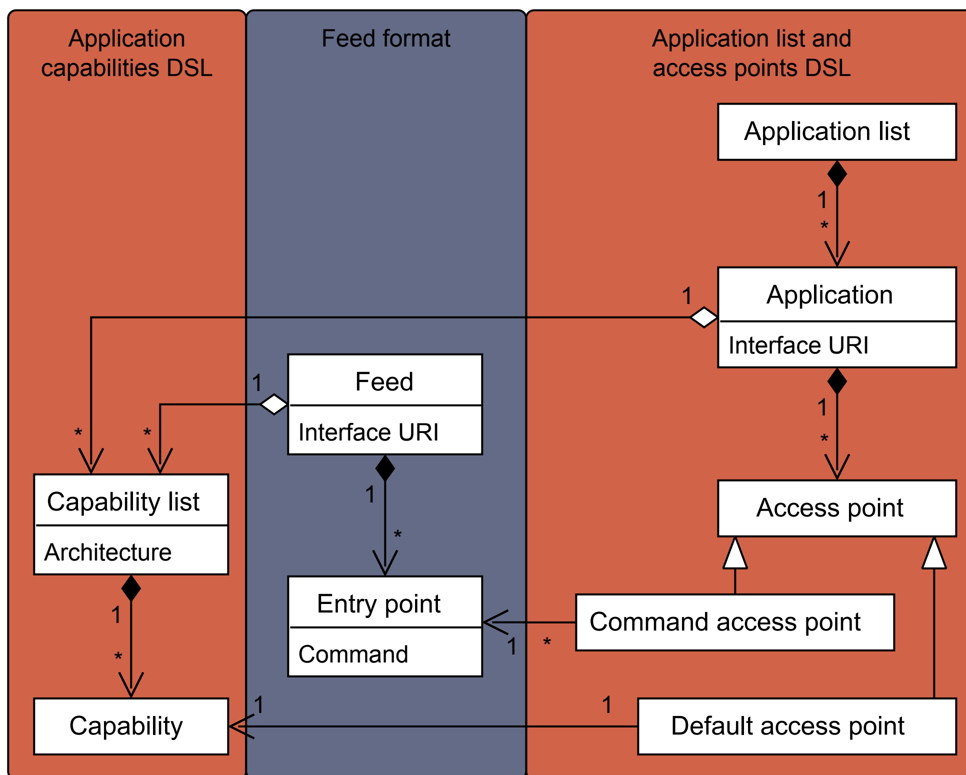

Figure 4.1: The *Capabilities DSL* is embedded within *Zero Install* feeds. The *Application list and access points DSL* contains access points and copies of capabilities. Access points either reference capabilities or entry points.

Figure B.2 in the appendix shows the complete structure of the *Capabilities DSL*. Figure B.3 in the appendix shows the complete structure of the *Application list and access points DSL*.

Capability	Access point
Register file type handler	Set as default handler
Register URL protocol handler	Set as default handler
Register *AutoPlay* handler	Set as default handler
Register default program candidate	Set as default program
Store context menu entry	Register context menu entry
-	Menu entry
-	Desktop icon
-	*Quick Launch bar* icon
-	*Send To* menu entry
-	Command-line alias

Table 4.1: Every capability has a corresponding access point the user can choose to create. Some access points do not refer to specific capabilities.

4.3 Capturing

When developing an application with the intention of publishing it as a *Zero Install* feed (see Chapter 2.2.1) an author can encode capabilities directly in the *Capabilities DSL* (see Chapter 4.2). However, *Zero Install* is often used to publish existing applications, which usually already have an installer to perform the integration. Reproducing the steps these installers perform manually would be cumbersome. Therefore, we wish to capture the changes an installer makes and isolate those relevant for desktop integration.

Of the capturing methods listed in Chapter 3.4 we decided to use Snapshots, since they are the only method that can handle setups that require a reboot and are easy to implement. The problem with background noise can be countered sufficiently by only monitoring very specific locations, such as the list of registered file types or the common shortcut locations.

4.4 Applying desktop integration

Data stored in the *Capabilities DSL* (see Chapter 4.2) are *applied* to the current system when the user decides to add/register/install an application via *Zero Install*. As a part of this process the user choses which access points to create/apply.

When applying capabilities and access points two entries can potentially be in conflict with each other. For example there cannot be two different desktop icons with the same name and two applications cannot both be the default handler for a

file type. In order to reliably detect such conflicts *Zero Install* calculates a *conflict ID* for each capability and access point.

All conflict IDs exist within a common namespace. If two elements (possibly from different applications) have an identical conflict ID they cannot be simultaneously applied to a single system. Conflict IDs are usually a concatenation of the element type and its regular ID (thus preventing things like two menu entries with the same name) however they can also detect conflicts between elements of different types (e.g., a file type and a URL protocol handler sharing a common registry location).

After confirming that integration changes requested by the user do not cause any conflicts, *Zero Install* applies the new settings to the system and stores the according metadata using the *Application list and access points DSL* (see Chapter 4.2).

4.5 Synchronization

The *Application list and access points DSL* (see Chapter 4.2) stores all desktop integration choices made by the user and can therefore be used to reproduce a working environment on another machine.

Zero Install uses this to provide a server-based synchronization feature. The application list is mirrored from and to an HTTP server using *HTTP GET* and *HTTP PUT* [1]. All user data is encrypted locally before transmission with AES-128 using a private key. This ensures both data confidentiality[2] and data consistency[3] even if the server is compromised.

The synchronization algorithm works as follows:

1. Download existing data from the server if present and decrypt it.
2. Merge the data from the server into the local application list using a three-way merge[4].
3. Apply any changes made to the application list to the desktop environment.
4. Encrypt and upload the new application list.

[1] HTTP defines multiple *request methods*. The most common are *GET*, used to retrieve data, and *POST*, used to submit data for processing. *PUT*, used to set data in an idempotent fashion, is less common.

[2] In the context of computer security data confidentiality describes the ability of a data storage or transmission system to prevent unauthorized users from reading data.

[3] In the context of computer security data consistency describes the ability of a data storage or transmission system to prevent unauthorized users from modifying data without being detected.

[4] A three-way merge combines changes from two files that share a common ancestor.

5. Store a copy of the application list locally for use as a baseline for the next merge.

Figure 4.2 illustrates the aforementioned design.

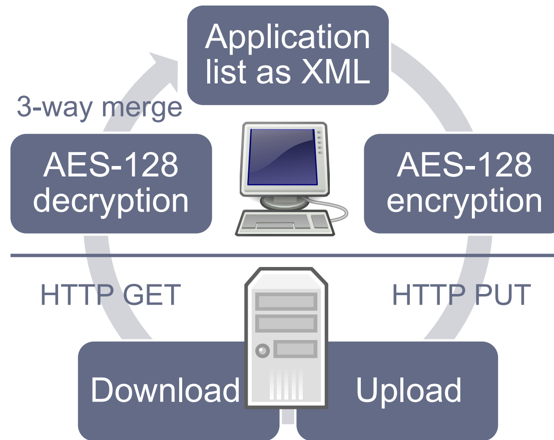

Figure 4.2: *Zero Install* synchronizes the application list between the local machine and an HTTP server using a three-way merge. All user data is encrypted locally with AES-128. This figure contains content from the Tango Icon Library placed in the public domain.

Chapter 5

Implementation

This chapter details how we implemented the design decisions from Chapter 4 and what difficulties we came across in doing so.

5.1 Tools and libraries

The original *Zero Install* system introduced in Chapter 2.2 was developed using *Python* and was intended for use on *POSIX* systems such as *Linux* and *Mac OS X*. Outside the scope of this work, we have developed a *Windows* port of *Zero Install* using the existing *Python* code for solving dependencies but rewriting the downloading, file-system access and user interface code in *C#*.

The implementation presented in this work builds upon the *Windows* port of *Zero Install*.

We used the following tools and libraries for our implementation:

- *C# 2.0* as the main programming language,
  Visual Studio 2010 as an integrated development environment
- *C5 Generic Collection Library 1.1.0* for advanced collection data structures,
  INI File Parser v1.5.5 to store non-XML configuration data,
  NDesk.Options 0.2.1 for parsing command-line arguments,
  NUnit 2.5.10 for unit tests
- *SharpZipLib 0.86.0* for *ZIP* and *TAR* compression support,
  LZMA SDK 4.65 for *LZMA* compression support
- *EasyHook 2.6* for API hooking (see Chapter 5.5)

5.2 Installation capturing

We implemented the design for capturing desktop integration data chosen in Chapter 4.3 in a command-line tool called `0capture`.

`0capture` stores all data related to a capture process in a working directory called the *capture directory*.

When you instruct `0capture` to take snapshots of the system state before and after installing a new application it will collect a list of registry keys relevant to desktop integration as well as a list of all subdirectories of the *Program Files* directory. This data is stored on-disk in a binary format to allow the process to span system reboots.

In the final step `0capture` compares the two snapshots in order to detect newly created registry keys and file system directories. It then collects data from these registry keys and the installation directory to generate a *Zero Install* feed file with embedded desktop integration data.

Figure 5.1 depicts the aforementioned work-flow for capturing an application installation.

Figure 5.1: The `0capture` command provides a set of sub-commands that are called in a specific sequence to capture the installation of an application.

`0capture` uses heuristics to locate an application's main executable and description. Therefore, the generated feed files are only intended as starting points that save a developer the work of manually creating capability entries for numerous file type handlers, etc.. Download sources, missing descriptions and localizations need to be added manually.

`0capture` is not intended to automatically make native applications portable. While many applications can be redistributed by packaging the contents of their installation directory shared libraries installed to other locations may cause problems. One possible strategy for such cases is to use a native installer version of an application to capture its desktop integration but to apply the captured data to a portable version of the same application.

5.3 Applying desktop integration

Our extended version of *Zero Install* provides the following commands for managing desktop integration:

- `0install add-app` adds an application to the application list without applying any desktop integration.
- `0install integrate-app` registers an application's capabilities and allows the user to specify any desired access points.
- `0install remove-app` removes an application from the application list and removes all capabilities and access points from the system.
- `0install add-alias` / `0alias` creates a command-line alias access point.

You can instruct *Zero Install* to launch an application with a command line like:
`0install run` *`Interface URI`*
When applying desktop integration *Zero Install* creates shortcuts and registry entries pointing to such command lines.

Zero Install keeps track of any applied desktop integration changes in a file called `app-list.xml` using the *Application list and access points DSL* (see Chapter 4.2). If an error occurs during any part of an integration operation all already applied changes are rolled back. Since this does not protect against system-wide crashes desktop integration operations are not truly atomic.

In order to prevent race conditions *Zero Install* uses a *Windows* mutex[33] to prevent multiple desktop integration processes from running simultaneously.

5.3.1 Limitations

Our current implementation does not support *COM server* (see Chapter 3.1.8) or *Games Explorer* (see Chapter 3.1.9) registration.

In-Process COM servers cannot be provided via *Zero Install* since it is unable to inject dependencies into running processes. Application virtualization systems such as *ThinApp* suffer from the same problem. [12]

Out-of-Process COM servers and the *Games Explorer* both require direct access to the application's binaries. Since *Zero Install*'s desktop integration is independent from specific implementations *Zero Install* would need to generate some form of surrogate files.

5.4 Stubs

Windows user interfaces that provide a choice of multiple applications (e.g., the *Open with* dialog) extract the application names and icons from the executable files referenced by the according registry entry. (See Figure A.3 in the appendix for a screenshot.) If *Zero Install* created registry entries pointing to
`0install run` *`interface URI`*
Windows would always display the *Zero Install* icon and name instead of the metadata of the actual application being launched.

We solved this problem by generating *stubs*. A stub is a small executable that simply executes another command and passes through any command-line arguments. For each `0install` command that needs to be added to the registry we generate an according stub and embed the name and icon of the target application.

Zero Install uses the *.NET Framework*'s built-in runtime *C#* compiler[32] to generate stubs on demand. In order to reduce the number of stubs that need to be compiled *Zero Install* employs a caching scheme: Reusable stubs are identified by a cryptographic hash of the interface URI and command name used to start the target application. They are placed in directories named after this hash. The executable files themselves are named like the executables of the target applications to make their processes more recognizable in the task manager. Figure 5.2 illustrates the composition of a stub's directory and file name.

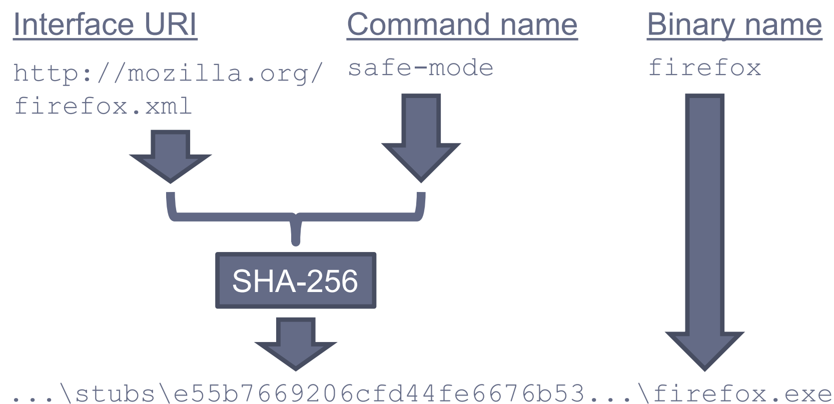

Figure 5.2: Each stub is placed in a separate directory whose name is generated by hashing the interface URI and the command name. The file name of the stub itself is determined by the binary name specified by the entry point.

Stubs are also used to provide command-line aliases. In order to make a stub discoverable without its complete path *Zero Install* adds an *App Path* entry to the registry. [34]

5.5 API hooking

An application launched via *Zero Install* may attempt to perform its own desktop integration. It would incorrectly assume that it could be relaunched in future simply by executing the path of its main executable located in the implementation cache (see Chapter 2.2.3). Registry entries created by such an application would bypass *Zero Install* on startup. This would lead to a number of problems:

- If an application has dependencies *Zero Install*'s dependency injection (see Chapter 2.2.2) would not be able to provide them.
- If a new version of an application becomes available *Zero Install* will place it in a new directory in the implementation cache. The registry entries would continue to point to the old version.

We handle this issue by intercepting registry access attempted by applications launched via *Zero Install* and transparently replacing paths of binaries located within the implementation cache with paths of stubs.

5.5.1 Hooking library

Most libraries for intercepting/hooking system calls use a method called *DLL injection*. Hereby, a process is forced to load an external DLL and execute code from it. This code then replaces the methods to be hooked with new code. Usually, the hooked methods perform additional operations or modify parameters and then pass the control on to the original method. [35]

Our implementation uses the *EasyHook* library, which provides *DLL injection* and API hooking for the *.NET Framework*, to hook a number of *Win32* API[1] methods. [36]

5.5.2 Filtering registry access

In order to filter an application's view of the *Windows* registry we need to hook the following *Win32* API methods:

- `RegQueryValueEx()` for read access to the registry [37]
- `RegSetValueEx()` for write access to the registry [38]

[1] The *Win32* API is the API provided by modern *Windows* versions, starting with *Windows 95*.

Before starting the target process *Zero Install* builds a string replacement table mapping each executable within the chosen implementations to its corresponding `0install` stub.

Whenever the application attempts to write a string to the registry the API hook checks to determine whether the string starts with an entry listed in the replacement table and performs a substitution if a match is found. On registry reads returning a string the API hook performs the converse replacement. Any application checking to see whether it is currently integrated with the desktop environment will therefore not see the changes made by *Zero Install*.

Since `RegQueryValueEx()` writes to a buffer provided by the calling application and may require buffer resizing modifying result data is non-trivial. Figure C.1 in the appendix contains a sequence diagram illustrating the algorithm we employ to handle this.

5.5.3 Child processes

Zero Install must ensure that the API hooks are also applied to any child processes launched from within the same implementation. In order to detect the creation of new processes we need to hook the following *Win32* API method:

- `CreateProcess()` [39]

When the application attempts to start a new process the API hook determines whether the target executable is located within the same implementation directory. If this is the case the new process is created with a suspended main thread, enabling the installation of the API hooks before any application code is executed. Figure C.2 in the appendix contains a sequence diagram illustrating the algorithm we employ to handle this.

If an application launches a child process with elevated privileges via *User Account Control* (introduced in *Windows Vista*) *Zero Install* will not be able to install the API hooks there. This is because injecting a DLL into a foreign process requires the `PROCESS_CREATE_THREAD` right. A process running with a *Normal* integrity level does not have this right for processes with a *High* integrity level (set by *UAC* upon elevation). [40]

5.5.4 Windows 7 taskbar

Windows 7 introduced a new taskbar that allows the user to *pin* a running application. This creates a shortcut to the application that takes its place once it terminates and allows the application to be relaunched in future. [41]

By default, *Windows* creates shortcuts that point to the executables the processes were launched from. This leads to a problem analogous to the previously discussion registry issue. However, *Windows* allows applications to specify alternative relaunch commands to use for these shortcuts for each window.[42]

In order to detect the creation of new windows we need to hook the following *Win32* API method:

- `CreateWindowEx()`[43]

Whenever a new window is created we set the relaunch command to the appropriate `0install` command. Additionally, we add entries to application's *jump list*[44] that allow the user to invoke the *Zero Install* user interface to select a different version of the application.

Chapter 6

Evaluation

This chapter evaluates the implementation under the following aspects:

- Ability of `0capture` to collect data relevant to desktop integration
- User-visible differences between desktop integration performed by *Zero Install* and native installers
- Performance impact caused by using *Zero Install*
- Reliability of API hooking

It concludes with a discussion of the results in the context of the real-world applicability of *Zero Install*.

6.1 Methodology

We used `0capture` to capture the system modifications performed by a subset of the applications listed in Table 3.1. To determine user-visible differences between desktop integration performed by *Zero Install* and native installers we created a set of use-case scenarios for these applications. These use cases were then each executed once using applications installed natively and once using the same applications integrated by *Zero Install*.

We measured the performance impact caused by *Zero Install* by comparing the average startup time of applications when launched directly with the startup times when launching them via *Zero Install* (both with and without API hooking).

In order to assess the reliability of the API hooking we compared the runtime behavior of applications with API hooking turned on and off. If an application behaved correctly when API hooking was active we checked whether all desktop integration attempts performed at runtime were transformed into references to stubs (see Chapter 5.4).

6.2 Use cases

The following use cases assume that all involved applications have already been integrated into the desktop environment and set as default handlers for their respective capabilities. Unless specified otherwise they are intended to be executed on *Windows XP*.

UC1a Launch *Audacity* via the start menu,
save a new `.aup` project file to the disk,
close *Audacity*,
locate the new file and double-click on it (should reopen *Audacity*).

UC1b Launch *Blender* via the start menu,
save a new `.blend` file to the disk,
close *Blender*,
locate the new file and double-click on it (should reopen *Blender*).

UC1c Launch *OpenOffice.org Writer* via the start menu,
save an empty `.odt` document,
close *OpenOffice.org Writer*,
locate the new file and double-click on it (should reopen *OpenOffice.org Writer*).

UC2a Launch *Mozilla Firefox* via the *Internet* button in the start menu,
browse to a website containing a `mailto:` link and click on it (should open *Mozilla Thunderbird*).

UC2b Launch *Mozilla Thunderbird* via the *E-Mail* button in the start menu,
open a mail containing an `http:` link and click on it (should open *Mozilla Firefox*).

UC3 Launch *WinMerge* via the command-line (prepare with `0alias`),
open the *About* dialog and click on the *WinMerge homepage* link (should open *Mozilla Firefox*),
browse to a website containing an `xmpp:` link and click on it (should open *Pidgin*).

UC4a Open the *Windows SPAD* dialog (see Chapter 3.1.6),
set *Internet Explorer* and *Outlook Express* as default browser and mail client respectively,
set *Mozilla Firefox* and *Mozilla Thunderbird* as default browser and mail client respectively,
execute **UC2a** and **UC2b**.

UC4b Execute this on *Windows 7*:
Open the *Windows SYDP* dialog (see Chapter 3.1.6),
set *Windows Media Player* as default handler for all music formats,
set *foobar2000* as default handler for all music formats,
double-click on a music file (should open *foobar2000*).

UC5 Insert a blank DVD into a DVD-R drive,
select *Burn a disc using CDBurnerXP* in the *AutoPlay* dialog (should open *CDBurnerXP*).

6.3 Benchmarks

We used the tool *PassMark AppTimer 1.0*[45] to measure the startup times of applications. We configured the tool to detect an application as fully launched as soon as its main window started accepting user input.

To ensure consistent measurements with warm caches (reducing the IO impact and highlighting the actual processing time) we launched each configuration six times consecutively, discarding the first measurement and averaging the remaining results.

Using *Sysinternals Process Monitor*[28] we determined the number of calls each application made to hooked API methods (see Chapter 5.5) during its startup process.

We performed all measurements on a system with the following specifications:

Component	Model/Specification
CPU	AMD Phenom II X6 1055T (6x 2.8 GHz)
Memory	4.0 GB
Operating System	Windows 7 Service Pack 1 x64

6.4 Results

Table 6.1 contains a list of applications we captured using `0caputre`. It details what was captured automatically and what we added manually. Automatic capturing does not detect icons for applications or their file types. These were always added manually.

All use cases listed in Chapter 6.2 provided identical user experiences with and without *Zero Install* except for a *Zero Install* dialog and a short delay whenever a new application was launched.

Table 6.2 and Table 6.3 contain the results of our performance benchmarks.

Table 6.4 shows the results for our analysis of the API hooking reliability.

6.5 Discussion

The results indicate that *Zero Install*'s desktop integration works but that there are a number of areas of possible improvement.

6.5.1 Capturing

`0capture` can capture most application capabilities supported by *Zero Install* automatically. Icons currently always need to be added manually. This could be automated in a future version by extracting icon resources from executables.

Zero Install does not support advanced context menus since they require *In-Process COM servers* (see Chapter 5.3.1). In some cases equivalent functionality can be provided by using normal context menus instead. However, context menus that vary based on the file's content are not possible. This makes this method of desktop integration less suitable for applications like file compression utilities.

The installation of browser plug-ins is not handled. Supporting this would require *Zero Install* to be extended with a feature for externally specifying additional implementations to be injected into applications (i.e., a plug-in mode).

Application	**Captured**	**Manual**	**Unable to map**
7-zip	File types	Context menu	Adv. context menu
Audacity	File types		
Blender	File types		
CDBurnerXP	File types, *SYDP*, *AutoPlay*		
foobar2000	File types, *SYDP*,		
Inkscape	File types		
IrfanView	File types, *SYDP*,		
KeePass	File types		
LibreOffice	File types, *SYDP*		Browser plug-in
Mozilla Firefox	Default browser, *SPAD*, *SYDP*		
Mozilla Thunderbird	Default mail client, *SPAD*, *SYDP*		Newsreader
Notepad++		File types, Context menu	(Adv. context menu)
OpenOffice.org	File types, *SYDP*		Browser plug-in
Pidgin	URL protocols		
Texmaker	File types		
VLC	File types, *SYDP*		Browser plug-in
WinMerge		Context menu	(Adv. context menu)
WinRAR	File types	Context menu	Adv. context menu
yEd	File types		

Table 6.1: This table lists applications we captured with `0catpure`. The second column enumerates capabilities that were automatically captured. The third column contains capabilities we added to the generated *feeds* manually. The last column lists capabilities we were unable to map with *Zero Install*. Capabilities written in parentheses were replaced with other capabilities providing equivalent functionality.

Application	Startup time			API calls
	Native	Zero Install		
		Without Hooking	With hooking	
Audacity	195 ms	1570 ms	1910 ms	387
CDBurnerXP	760 ms	2103 ms	2290 ms	472
Mozilla Firefox	335 ms	1738 ms	1934 ms	1066
VLC	167 ms	1660 ms	1978 ms	444
WinMerge	233 ms	1506 ms	1800 ms	728

Table 6.2: This table shows the startup times we measured for a set of sample applications. The last column indicates the number of calls each application made to hooked API methods (see Chapter 5.5) during its startup process.

Application	Solving time	Injection time	Injection time per API call
Audacity	1375 ms	240 ms	620 ns
CDBurnerXP	1343 ms	187 ms	396 ns
Mozilla Firefox	1403 ms	196 ms	184 ns
VLC	1493 ms	318 ms	716 ns
WinMerge	1273 ms	294 ms	404 ns
Average	1377 ms	247 ms	464 ns

Table 6.3: This table contains values derived from Table 6.2. The second column's values are calculated by subtracting the native startup times from the startup times with *Zero Install* without API hooking. The third column's values are calculated by subtracting the *Zero Install* startup times without API hooking from the startup times with API hooking. The last column's values are calculated by dividing the values from the second column by the number of hooked API calls during startup.

Application	Crashes	Handled	Not handled
Audacity		File types	
foobar2000	`RegQueryEx`	File types	
LibreOffice	`CreateWindowEx`		Browser plug-in
Mozilla Firefox	`RegQueryEx`	Default browser	
Notepad++		File types	Adv. context menu
OpenOffice.org	`CreateWindowEx`		Browser plug-in
WinMerge		Installation path	

Table 6.4: This table lists a set of applications that perform desktop integration operations at runtime. The second column indicates what hooked API methods cause the applications to crash. The third column lists integrations successfully handled by the hooking DLL. The last column lists integrations not handled by the hooking DLL.

6.5.2 Use cases

The execution of the use cases was not impaired by *Zero Install*'s desktop integration.

- The results for use cases **UC1a**, **UC1b** and **UC1c** show that *Zero Install* is capable of handling **file type associations** both with single-entry-point and multi-entry-point applications.
- The use case **UC2a** and **UC2b** results demonstrate that registration as **default programs** via *Zero Install* works.
- Use case **UC3** combines **command-line aliases** and the registration of custom **URL protocol handlers**.
- The use cases **UC4a** and **UC4b** demonstrate *Zero Install*'s integration with *Windows*' own user interfaces for **managing default programs**.
- The results for use case **UC5** show that the *Zero Install* ***AutoPlay*** support works.

6.5.3 Performance

The values in Table 6.3 derived from the performance benchmarks show that the startup of an application is slowed down by *Zero Install* by approximately 1,4 seconds on average (assuming all required implementations have already been downloaded). This value may be improved in future by optimizing the code used to solve dependencies, which currently uses inter-process communication to delegate work to a Python process.

The API hooking increases the average startup time of applications by less than 250 ms. While this delay lies beyond the 0.1 s threshold for what users commonly experience as instantaneous, it is well below the 1 s threshold indicating that additional user feedback is necessary to retain the impression of responsiveness. [46]

The API hooking delay is caused in part by additional work performed by *Zero Install* to inject the hooking DLL and in part by *Win32* API methods slowed down by hooking. Therefore, dividing this delay by the number of hooked API calls during startup yields an upper bound for the delay caused by a single call to a hooked method. With an average delay per call of 464 ns a single user operation would have to trigger more than 215 hooked API calls to cause a noticeable delay (beyond 100 ms).

According to the *EasyHook* documentation[36], delays caused by API hooks written in a *managed* (*.NET*-based) language can sometimes cause delays in the millisecond area, whereas hooks written in *C* are several orders of a magnitude faster. This is mainly due to the parameter conversions .NET requires for interaction with *C* code (called *marshalling*). Our API hooking implementation code be rewritten using *C* to gain such performance improvements.

6.5.4 Stability

The API hooking properly handles all runtime integration attempts that are supported by *Zero Install*. Unsupported capabilities such as advanced context menu entries and browser plug-ins are ignored.

A number of applications crash unexpectedly when hooked API methods are invoked. Preliminary analyses suggest that these applications allocate memory for buffers passed to the API methods without allowing write-access for regular processes. Presumably, the system's kernel is able to write to these buffers while the hooking DLL is not able to. Further analysis, preferably with access to the source code of the affected applications, is needed to confirm this.

When filtering data being written to the registry the hooking API can replace the input string buffer with a new one. This has shown to work reliably. However, when filtering data read from the registry strings need to be written to buffers provided by the application, thus causing the aforementioned crashes.

Chapter 7

Conclusion

The main objective of this work was to improve the state of desktop integration for zero-install systems in comparison with native installation systems. For that purpose we designed a domain-specific language to describe desktop integration and extended *Zero Install* to use data stored this language.

The implementation at hand can be used to automatically capture the installation of numerous productivity applications and convert them into *Zero Install feeds* without losing their desktop integration.

The evaluation showed that desktop integration performed by *Zero Install* can provide a user experience equivalent to that provided by native installers for most use cases. However, some scenarios requiring API hooking are not handled reliably by the current implementation. Future versions could handle this using techniques from the domain of application virtualization.

7.1 Future work

Our implementation could be improved to handle the registration of *out-of-process COM servers* by providing a proxy *COM server*. This proxy would perform *Zero Install*'s usual implementation startup process and then pass all method calls through to the actual application.

Support for the *Windows Games Explorer* could be implemented by generating the required embedded *game definition file* on demand instead of providing a precompiled one with the application's implementation.

The *Desktop Entry Specification*[47] published by the *freedesktop.org* project provides a common method for integrating applications with multiple *Linux* desktop environments. A `.desktop` file stores information about an application in a key-value store, such as an icon path and localized descriptions. The existing *Zero Install* version for *Linux* uses this to create menu entries for *feeds*. It could be extended to use data stored in the *Capabilities DSL* to add information about supported file types to the `.desktop` files and the *Shared MIME-info Database*[48].

Mac OS X applications are distributed as so called *application bundles*. These are self-contained directories that combine the binaries necessary to execute an application with metadata describing it, including the file types it supports. The desktop environment automatically detects the presence of such bundles and performs appropriate desktop integration. [49] Zero Install could be enhanced to create *application bundles* with embedded desktop integration data as launcher stubs.

Launching an application via *Zero Install* always entails a call to the command `0install run`. Our implementation ensures that all available methods of desktop integration use to this single point of entry. This could also be used to redirect local application launches into virtual machines, terminal servers, etc. without loss of desktop integration.

Appendix A

Screenshots

This appendix contains screenshots taken from various versions of the *Microsoft Windows* operating system, used to illustrate interfaces relevant for desktop integration.

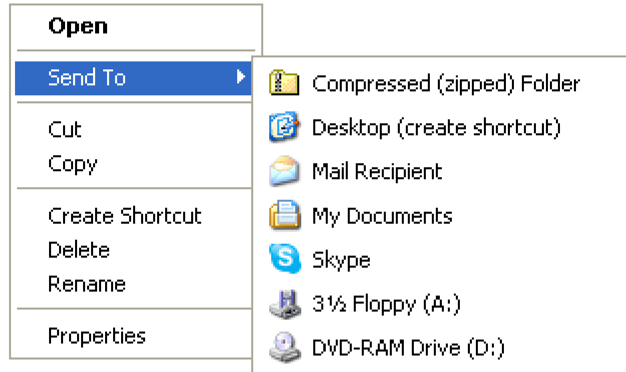

Figure A.1: This screenshot shows the *Send To* menu displayed for a file on *Windows XP*. It contains an entry created by the installation of *Skype*. Clicking on this entry would pass the file's path to *Skype* for transmission.

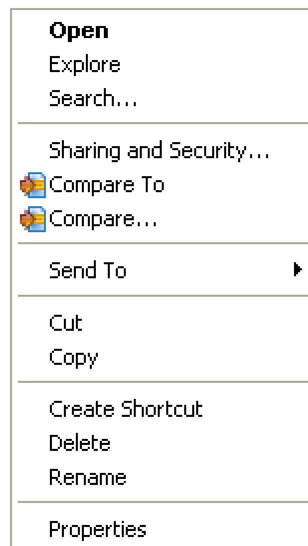

Figure A.2: This screenshot shows the *context menu* displayed for a file on *Windows XP*. It contains entries created by the installation of *WinMerge*. Clicking on one of these entries would pass the file's path to *WinMerge* for comparison.

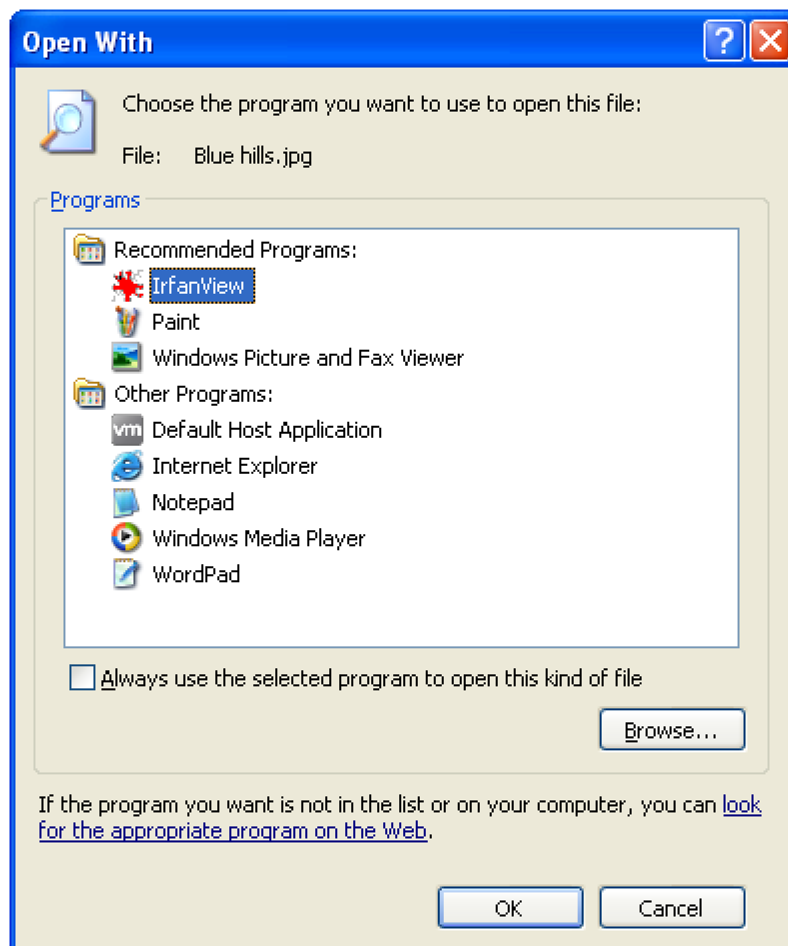

Figure A.3: This screenshot shows the *Open with* dialog displayed for a JPG file on *Windows XP*. It lists applications registered as handlers for JPG files as well as other known applications. The user can use this dialog to temporarily override the default handler or permanently change it.

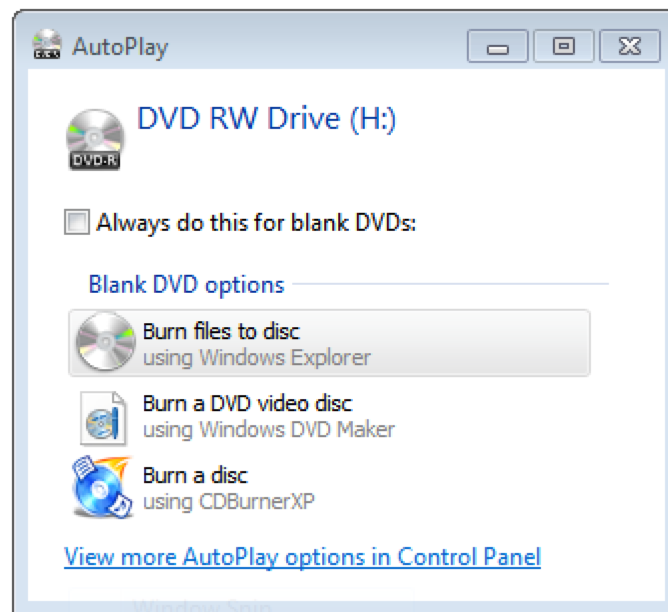

Figure A.4: This screenshot shows the *AutoPlay* dialog on *Windows 7* when a blank DVD is inserted, listing possible actions. It contains an entry created by the installation of *CDBurnerXP*. Clicking on this entry would pass the drive letter to *CDBurnerXP*.

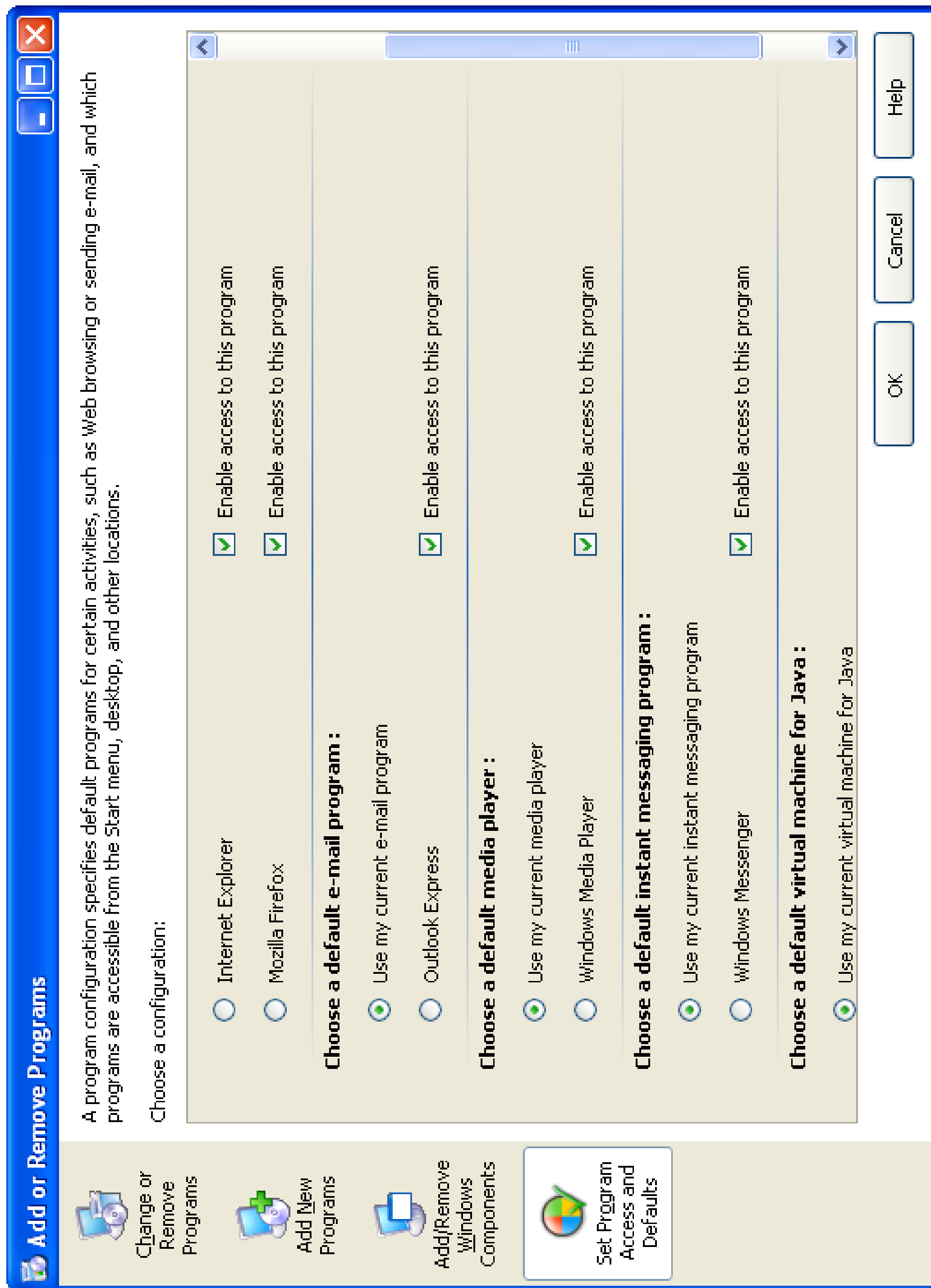

Figure A.5: This screenshot shows the *Set Program Access and Computer Defaults* dialog on *Windows XP*. It contains an entry created by the installation of *Mozilla Firefox*. Selecting this entry would invoke a helper application provided by *Mozilla Firefox*.

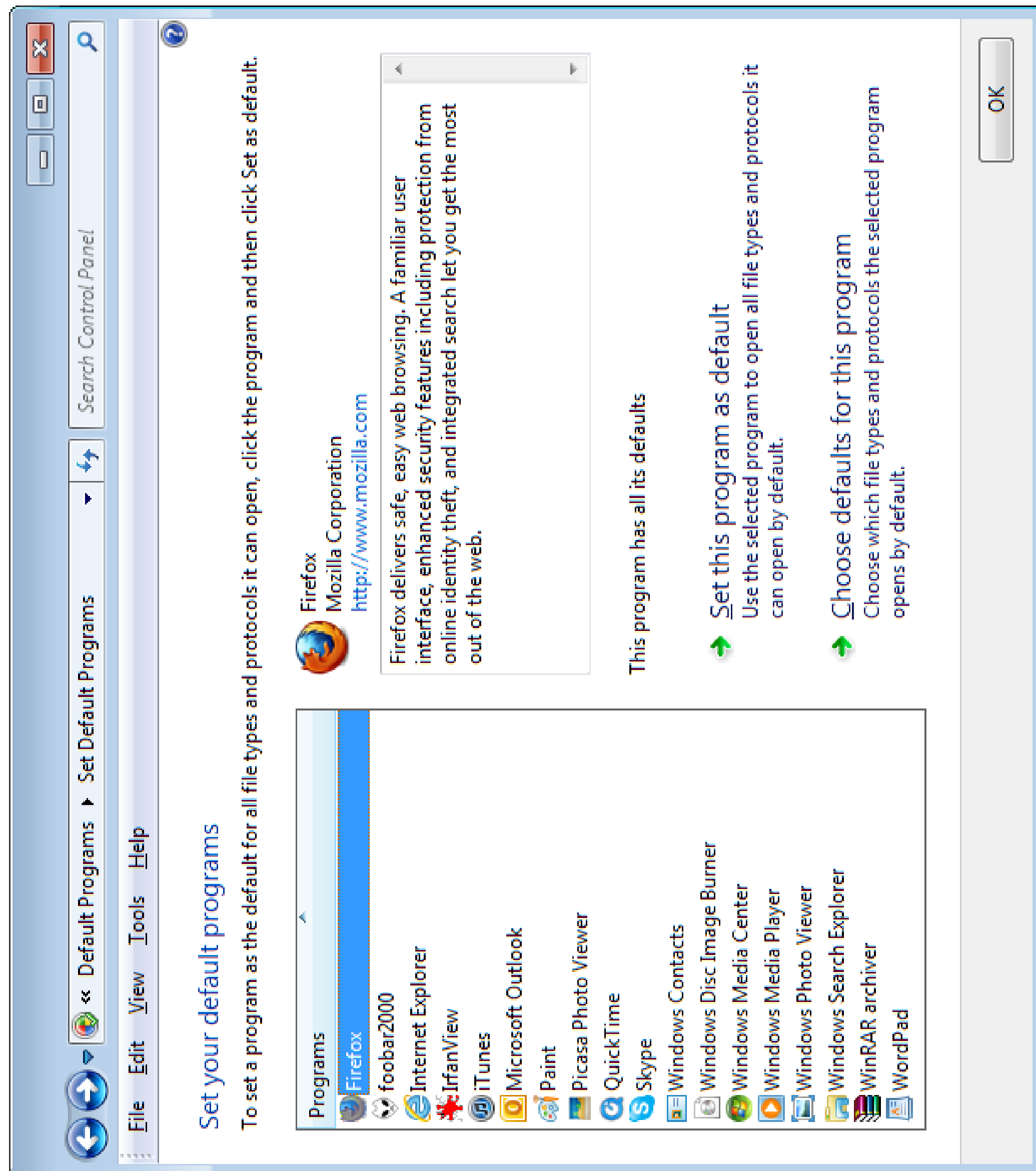

Figure A.6: This screenshot shows the *Set Your Default Programs* dialog on *Windows 7*. It contains entries created by the installation of *Mozilla Firefox*, *foobar2000*, *IrfanView*, *iTunes*, *Skype* and *WinRAR*. Clicking on *Set this program as default* would cause *Windows* to register any applicable defaults for the current user. Clicking on *Choose defaults for this program* would lead to the dialog depicted in A.7.

Figure A.7: This screenshot shows the *Set Your Default Programs* sub-dialog for selecting specific file type and URL protocol handlers to register. This example depicts the options provided for *Mozilla Firefox*.

Appendix B

XML formats

This appendix contains class diagrams describing various XML-based domain-specific languages. A class corresponds to an XML tag, a field corresponds to an XML attribute and an aggregation indicates that one tag is nested within another.

Complete specifications as well as XML Schema files for the languages can be found here:

- *Zero Install* feed format (Figure B.1):
  [8] and `http://0install.de/schema/injector/interface/`

  XML namespace:
  `http://zero-install.sourceforge.net/2004/injector/interface`

- Capabilities DSL (Figure B.2):
  `http://0install.de/schema/desktop-integration/capabilities/`

  XML namespace:
  `http://0install.de/schema/desktop-integration/capabilities`

- Application list and access points DSL (Figure B.3):
  `http://0install.de/schema/desktop-integration/app-list/`

  XML namespace:
  `http://0install.de/schema/desktop-integration/app-list`

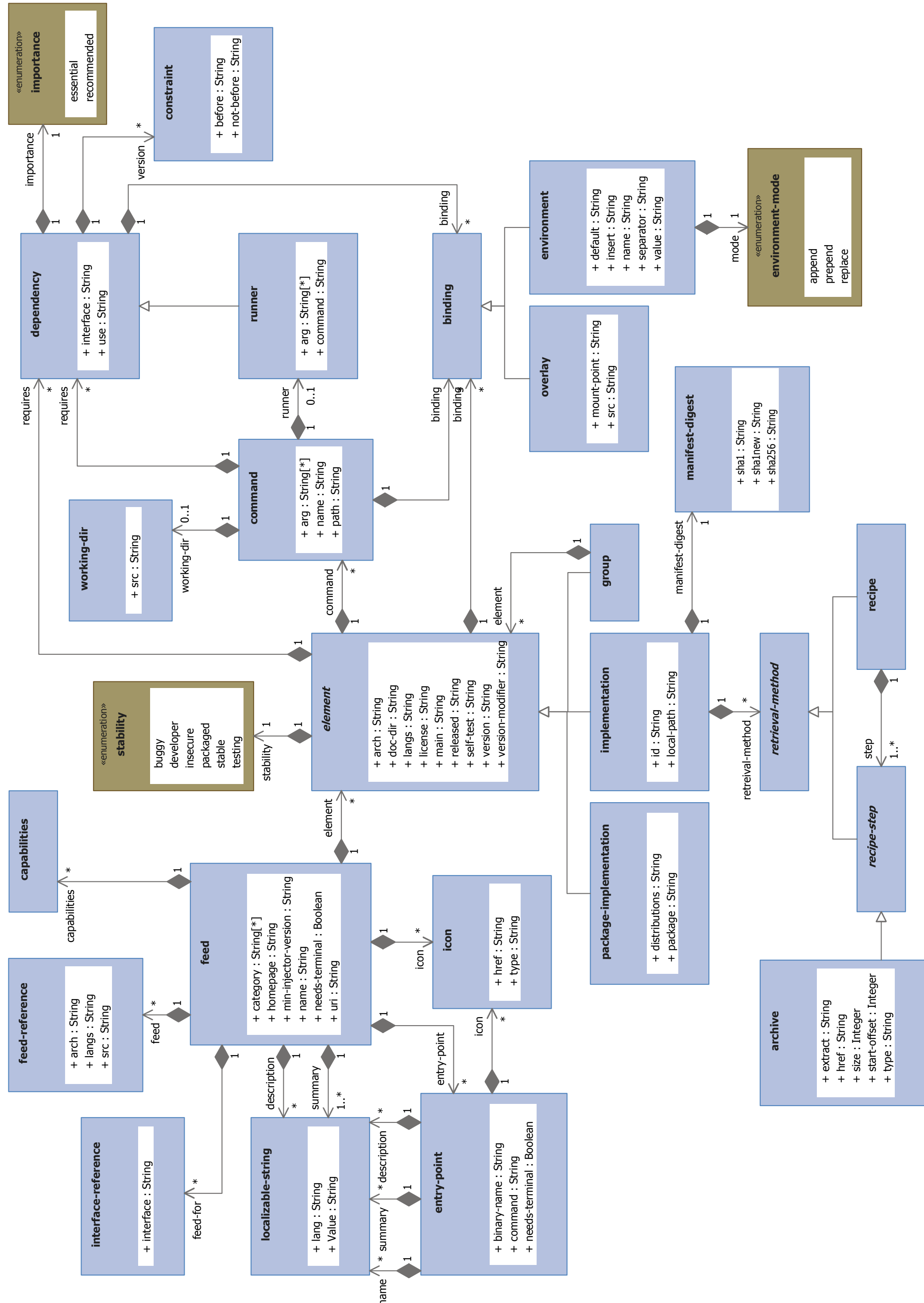

Figure B.1: *Zero Install* feed format

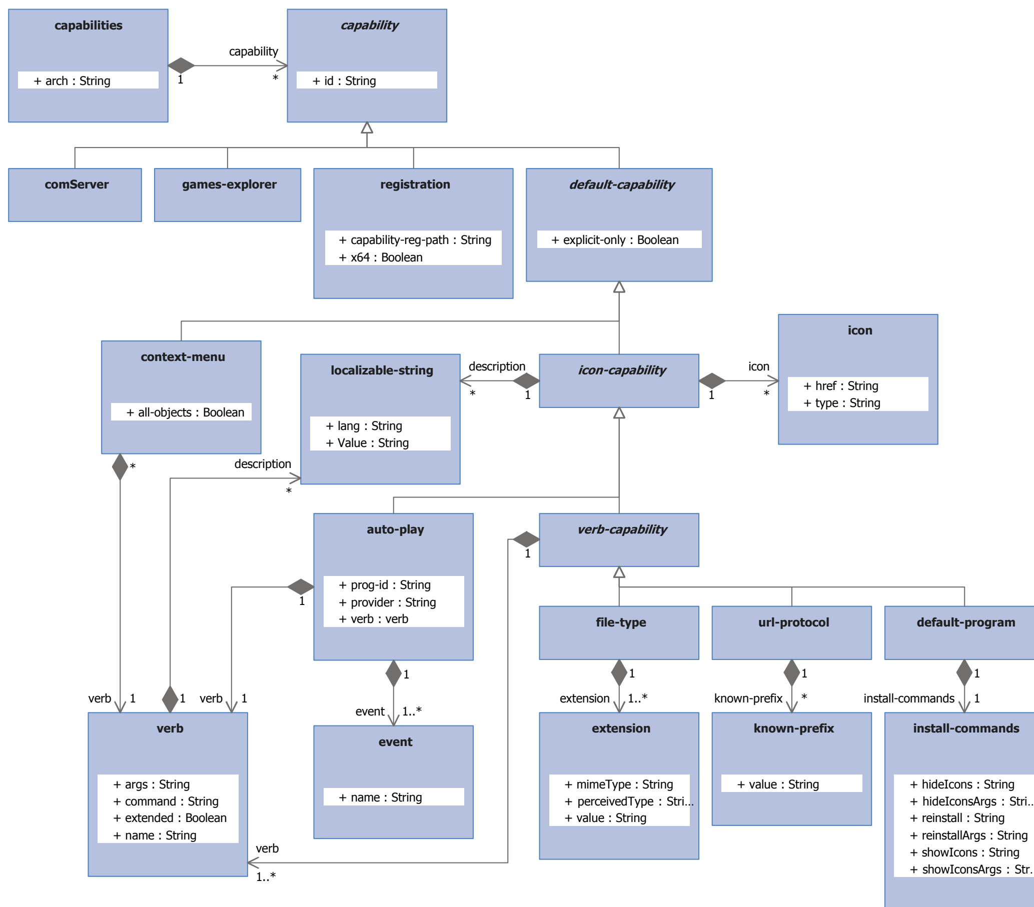

Figure B.2: Capabilities DSL

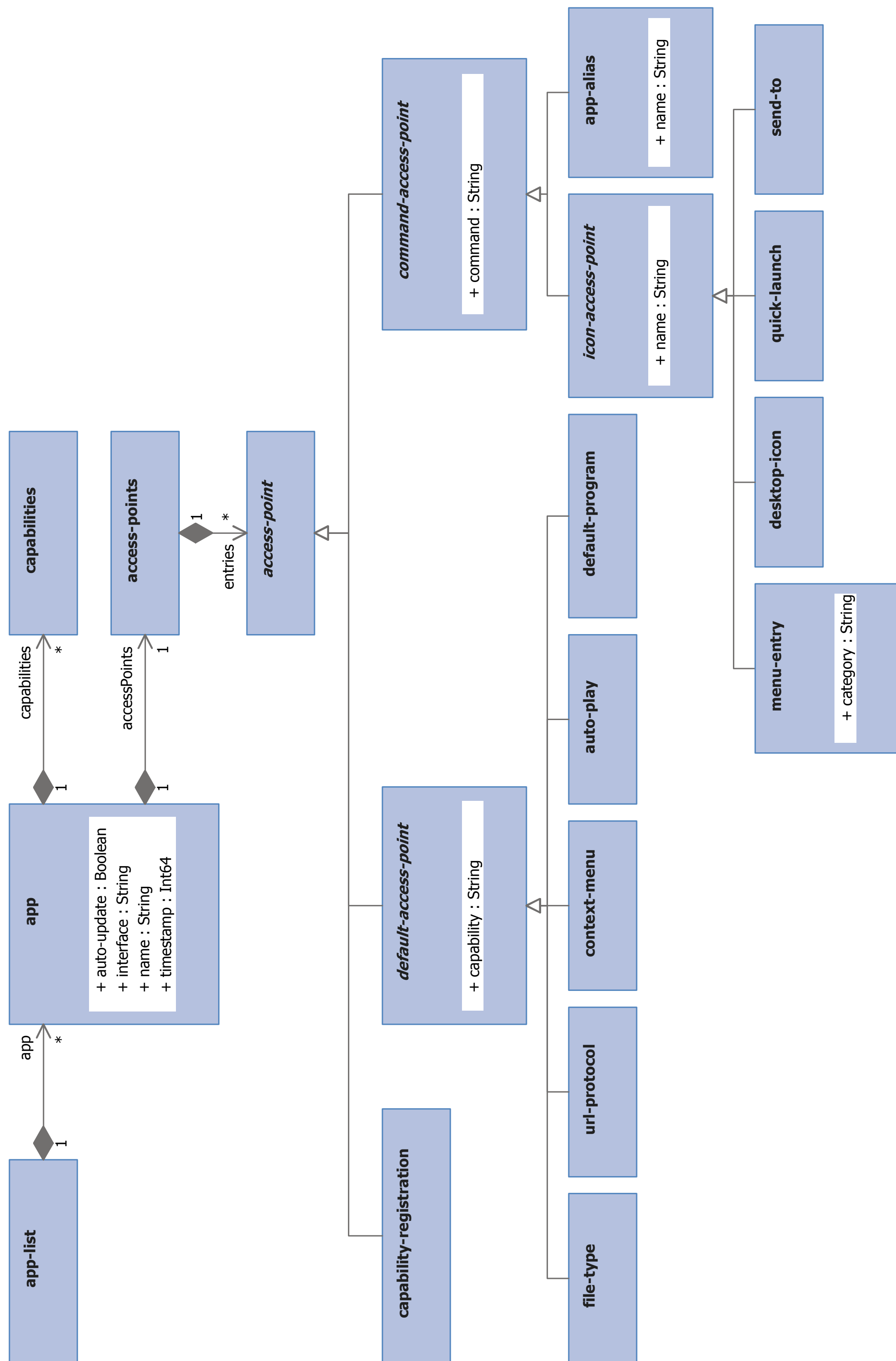

Figure B.3: Application list and access points DSL

Appendix C

API hooking

This appendix contains sequence diagrams depicting the control flow for calls to hooked *Win32* API methods.

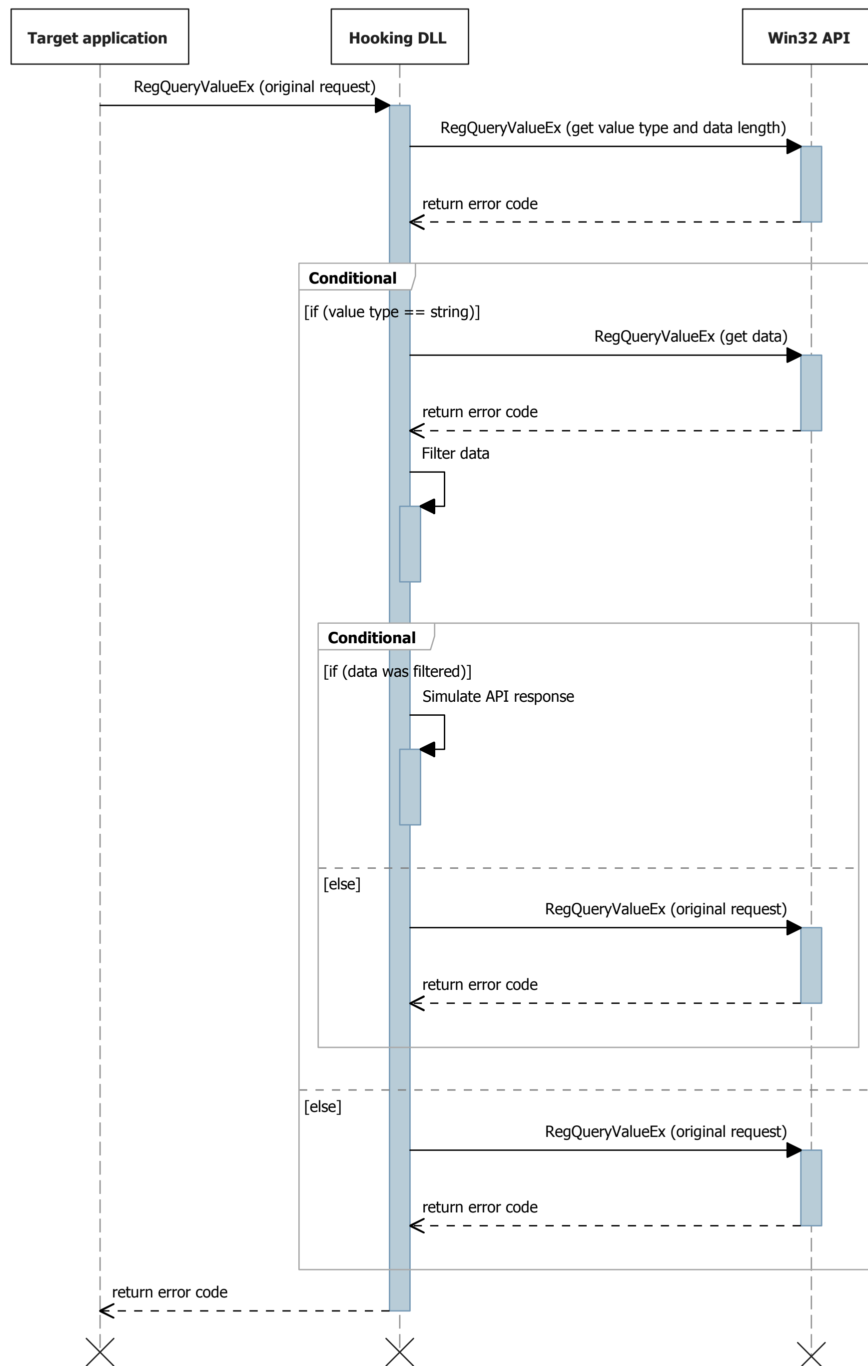

Figure C.1: Calls to the hooked *Win32* API method `RegQueryValuEx()` result in an additional request to the registry by the hooking DLL. This request is used to retrieve the actual data and determine whether any filtering is necessary. If the data is to be filtered a response from the API with the modified content is simulated. Otherwise, the original request is passed through.

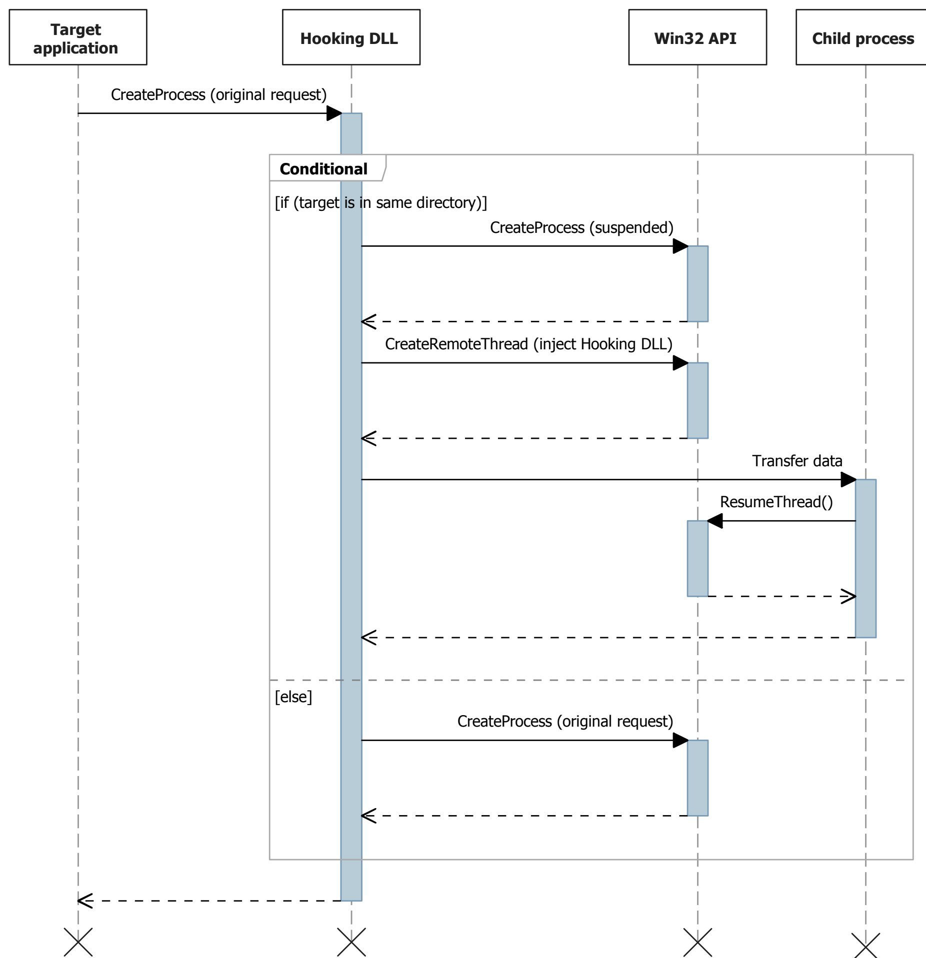

Figure C.2: Calls to the hooked *Win32* API method `CreateProcess()` are modified if the executable file to be loaded is located within the same implementation directory as the one currently running. The new process is spawned in a suspended state in order for the hooking library to be installed there as well.

Acknowledgments

I would like to thank my thesis supervisor Konrad Miller for helping me refine the thesis' thematic priorities and for supporting me during the writing phase.

I am also grateful for Marc Rittinghaus' help with tracking down bugs in the implementation.

This thesis would not have been possible without Thomas Leonard's work on *Zero Install*, off which I based my implementation.

I also wish to express my gratitude to Simon E. Silva Lauinger and Roland Leopold Walking for helping me with the initial *Windows* port of *Zero Install*.

Lastly, I offer my regards to all who supported me in any respect during the completion of this thesis.

Bibliography

[1] Filesystem Hierarchy Standard 2.3:
http://www.pathname.com/fhs/pub/fhs-2.3.pdf
Published: January 28, 2004

[2] MSDN Library - Dynamic-Link Library Search Order:
http://msdn.microsoft.com/en-us/library/ms682586
Accessed: July 2011

[3] MSDN Library - Side-by-side Assemblies:
http://msdn.microsoft.com/en-us/library/aa376307
Accessed: July 2011

[4] LinuxQuestions.org - Linux Wiki - Library-related Commands and Files:
http://wiki.linuxquestions.org/wiki/Library-related_Commands_and_Files
Revision from: 08:34, March 25, 2011

[5] PortableApps.com Launcher documentation:
http://portableapps.com/manuals/PortableApps.comLauncher/intro/overview/
Accessed: July 2011

[6] Zero Install 1.0 release announcement:
http://article.gmane.org/gmane.comp.file-systems.zero-install.devel/4186
Published: May 23, 2011

[7] Zero Install project website: http://0install.net/
Website of the Windows port: http://0install.de/
Accessed: July 2011

[8] Zero Install Feed file format specification:
http://0install.net/interface-spec.html
Accessed: July 24, 2011; website under version control

[9] Zero Install - Sharing downloads between users:
http://0install.net/sharing.html
Accessed: July 24, 2011; website under version control

[10] The Unix tree rethought: an introduction to GoboLinux
Hisham Muhammad, May 09 2003
http://www.gobolinux.org/?page=k5

[11] Eelco Dolstra and Andres Löh and Nicolas Pierron
NixOS: A Purely Functional Linux Distribution
Cambridge University Press, 2010
http://www.st.ewi.tudelft.nl/~dolstra/pubs/nixos-jfp-final.pdf

[12] VMware ThinApp User's Manual (ThinApp 4.0.1):
http://www.vmware.com/pdf/thinapp401_manual.pdf
Accessed: July 2011

[13] W3C - The Open Software Description Format (OSD):
http://www.w3.org/TR/NOTE-OSD
Published: August 11, 1997

[14] Microsoft Application Virtualization 4.6 SP1 Sequencing Guide:
http://download.microsoft.com/download/F/7/8/F784A197-73BE-48FF-83DA-4102C05A6D44/App-V/App-V%204.6%20Service%20Pack%201%20Sequencing%20Guide.docx
Accessed: July 2011

[15] Microsoft Application Virtualization Version 4.6 SP1 Trial Guide:
http://download.microsoft.com/download/F/7/8/F784A197-73BE-48FF-83DA-4102C05A6D44/App-V/App-V%204.6%20SP1%20Trial%20Guide.docx
Published: March 10, 2011

[16] InstallFree Bridge Technical Overview Whitepaper:
Available at http://www.installfree.com/ after registration

[17] MSDN Library - File Types:
http://msdn.microsoft.com/en-us/library/cc144148
Accessed: July 2011

[18] MSDN Library - Registering an Application to a URL Protocol:
http://msdn.microsoft.com/en-us/library/aa767914
Accessed: July 2011

[19] MSDN Library - Association Arrays:
http://msdn.microsoft.com/en-us/library/ee872122
Accessed: July 2011

[20] MSDN Library - Creating Shortcut Menu Handlers:
http://msdn.microsoft.com/en-us/library/cc144171
Accessed: July 2011

[21] MSDN Library - Customizing a Shortcut Menu Using Dynamic Verbs:
http://msdn.microsoft.com/en-us/library/ee453696
Accessed: July 2011

[22] MSDN Library - Set Program Access and Computer Defaults (SPAD)
http://msdn.microsoft.com/en-us/library/cc144162
Accessed: July 2011

[23] MSDN Library - Registering Programs with Client Types:
http://msdn.microsoft.com/en-us/library/cc144109
Accessed: July 2011

[24] MSDN Library - Default Programs:
http://msdn.microsoft.com/en-us/library/cc144154
Accessed: July 2011

[25] MSDN Library - Preparing Hardware and Software for Use with AutoPlay:
http://msdn.microsoft.com/en-us/library/bb776827
Accessed: July 2011

[26] MSDN Library - The Component Object Model:
http://msdn.microsoft.com/en-us/library/ms694363
Accessed: July 2011

[27] MSDN Library - Getting Started with Games Explorer:
http://msdn.microsoft.com/en-us/library/ee417682
Accessed: July 2011

[28] Windows Sysinternals - Process Monitor
http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx
Accessed: July 2011

[29] GNOME Library - Preferred Applications
http://library.gnome.org/users/user-guide/stable/prefs-preferredapps.html.en
Accessed: July 2011

[30] MSDN Library - FileSystemWatcher Class:
http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher
Accessed: July 2011

[31] MSDN Library - RegNotifyChangeKeyValue Function:
http://msdn.microsoft.com/en-us/library/ms724892
Accessed: July 2011

[32] MSDN Library - System.CodeDom.Compiler Namespace:
http://msdn.microsoft.com/en-us/library/system.codedom.compiler
Accessed: July 2011

[33] MSDN Library - Mutex Objects:
http://msdn.microsoft.com/en-us/library/ms684266
Accessed: July 2011

[34] MSDN Library - Application Registration:
http://msdn.microsoft.com/en-us/library/ee872121
Accessed: July 2011

[35] Intercepting System API Calls
by Seung-Woo Kim
http://software.intel.com/file/29382
Published: May 13, 2004

[36] EasyHook Tutorial:
http://easyhook.codeplex.com/releases/view/24401#DownloadId=61179
Published: March 8, 2009

[37] MSDN Library - RegQueryValueEx Function:
http://msdn.microsoft.com/en-us/library/ms724911
Accessed: July 2011

[38] MSDN Library - RegSetValueEx Function:
http://msdn.microsoft.com/en-us/library/ms724923
Accessed: July 2011

[39] MSDN Library - CreateProcess Function:
http://msdn.microsoft.com/en-us/library/ms682425
Accessed: July 2011

[40] MSDN Library - How the Integrity Mechanism Is Implemented in Windows Vista:

http://msdn.microsoft.com/en-us/library/bb625962
Accessed: July 2011

[41] MSDN Library - Taskbar Extensions:
http://msdn.microsoft.com/en-us/library/dd378460
Accessed: July 2011

[42] MSDN Library - System.AppUserModel.RelaunchCommand:
http://msdn.microsoft.com/en-us/library/dd391571
Accessed: July 2011

[43] MSDN Library - CreateWindowEx Function:
http://msdn.microsoft.com/en-us/library/ms632680
Accessed: July 2011

[44] MSDN Library - ICustomDestinationList::AddUserTasks Method:
http://msdn.microsoft.com/en-us/library/dd378395
Accessed: July 2011

[45] PassMark AppTimer product website:
http://www.passmark.com/products/apptimer.htm
Accessed: July 2011

[46] Response time in man-computer conversational transactions
by Robert B. Miller
Published: 1968

[47] freedesktop.org - Desktop Entry Specification 1.1:
http://standards.freedesktop.org/
desktop-entry-spec/desktop-entry-spec-1.1.htm
Accessed: July 2011

[48] freedesktop.org - Shared MIME-info Database 0.18:
http://standards.freedesktop.org/
shared-mime-info-spec/shared-mime-info-spec-0.
18.html
Accessed: July 2011

[49] Mac OS X Developer Library - Bundle Programming Guide:
http://developer.apple.com/library/mac/
documentation/CoreFoundation/Conceptual/CFBundles/
Accessed: July 2011