FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Migration of processes from shared to dedicated systems

Towards maintainable and flexible processes

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Johannes Binder
Matrikelnummer 0727990

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber
Mitwirkung: Dipl.-Ing. Mag. Stephan Strodl

Wien, 17.09.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Migration of processes from shared to dedicated systems

Towards maintainable and flexible processes

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Johannes Binder
Registration Number 0727990

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber
Assistance: Dipl.-Ing. Mag. Stephan Strodl

Vienna, 17.09.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Johannes Binder
Donaufelder Straße 54/1401, 1210 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Abstract

Keeping multiple business processes deployed on one system poses severe threats to maintenance and flexibility of individual processes as well as the hosting system itself. Shared dependencies between processes limit the possibility to update the system for one process, because of the potential side effects on the other processes. The processes are bound to a system, even to a specific state of the system, and it is difficult to transfer them in a minimal way, so that only resources and dependencies of the respective process are considered. If documentation exists it is frequently outdated or complex to transform to an environment that is able to execute the process. Besides maintenance also preservation of the state of processes in a shared system consumes more resources than necessary. This makes it difficult to archive and share processes.

The aim of this work is to provide a process migration framework (PMF) that frees processes from the chains imposed by the shared system where they are deployed. The first step of the PMF is to identify the process environment, so the resources that a process requires during execution, which is done by static and dynamic observation of the process and its environment. This process environment is stored in a model. The model can be adapted to upgrade software or migrate to different data types or other maintenance reasons. Next, a virtual system is created where the process environment is deployed. Finally it is verified that the target system is able to execute the process correctly. The result of the PMF is a documentation of the process environment as well as a virtual system that contains only resources relevant to the process.

The advantages of this approach are that processes can be redeployed independently of any physical machine, processes are able to evolve independently of other processes, and process environments can be shared and archived in a sustainable way, including all dependencies that are required to execute the process.

The evaluation of the PMF on different scenarios shows its ability to create models of the process environments at a level that is sufficient to recreate the process environment for scenarios that use workflows or scripts, which may invoke local tools and web services. The resulting target systems are able to execute the processes.

Kurzfassung

Mehrere Prozesse auf einem System gemeinsam zu hosten ist in mehrerlei Hinsicht kritisch, da Wartbarkeit und Flexibilität unter solch einem Setup leiden. Der Grund dafür ist, dass in solch einer Umgebung mehrere Prozesse oft gemeinsame Abhängigkeiten haben. Solche Abhängigkeiten können dann nicht mehr auf die Anforderungen eines Prozesses angepasst werden, sondern die Auswirkungen auf alle anderen Prozesse, welche diese Abhängigkeit verwenden, muss überprüft werden. Weiters ist es schwierig, die Umgebung des Prozesses zu duplizieren, sodass nur die Abhängigkeiten dieses Prozesses mitberücksichtigt werden. Daher ist das Archivieren und Verteilen solcher Prozessumgebungen schwierig handzuhaben. Vorhandene Dokumentation von Prozessen ist oft nicht aktuell oder unvollständig.

Das Framework für Prozessmigration (PMF) unterstützt die Überführung von Prozessen in gemeinsam genutzten Systemen hin zu dedizierten virtuellen Systemen.

Der erste Schritt in diesem Framework ist die Umgebung des Prozesses zu analysieren. Das geschieht mit einer Kombination aus statischer Analyse der Prozessumgebung und dynamischer Analyse des Laufzeitverhaltens des Prozesses. Vom Prozess verwendete Artefakte und Einstellungen werden in ein Modell abgebildet. Dieses Modell kann im nächsten Schritt angepasst werden, um verschiedene Wartungsarbeiten, wie beispielsweise das Aktualisieren von Software, durchzuführen. Ausgehend von diesem angepassten Modell der Umgebung des Prozesses wird ein neues virtuelles System erstellt, welches nur die Abhängigkeiten des Prozesses beinhaltet.

Die dedizierte Prozessumgebung erleichtert die Wartbarkeit, da der Prozess unababhängig von anderen Prozessen weiterentwickelt werden kann. Eingebettet in einer virtuellen Maschine ist der Prozess zudem unabhabhäng von physikalischen Systemen, und kann nachhaltig aufbewahrt und weitergegeben werden.

Die Evaluation des PMF anhand verschiedener Szenarien zeigt, dass die resultierenden Modelle der Prozessumgebungen einen Detailgrad aufweisen, der ausreichend ist, um für Szenarien welche Workflows oder Skripte verwenden und eventuell auf lokale Tools und Web Services zugreifen, eine Prozessumgebung nachzubilden. Die resultierenden virtuellen Systeme sind in der Lage, die jeweiligen Prozesse auszuführen.

Contents

CHAPTER 1

Introduction

Processes are commonly deployed in shared systems and therefore use common infrastructure such as operating system libraries and installed tools of a system. If no precautions are taken to keep processes separate from each other this causes them to fusion. It becomes difficult to distinguish environments of individual processes, i.e. the infrastructure that a process requires to be able to execute successfully. But process environments evolve with the process, and by adapting one environment another one might be altered. This not only makes processes difficult to understand, but also to monitor, maintain, and migrate. Besides not being able to support the evolution of the process because the environment is not flexible enough also a loss of dynamism, e.g. the possibility to migrate the process to another machine, threatens these processes.

The importance of being able to create a documentation of process environments and create systems which satisfy such models of existing systems with all its direct as well as indirect dependencies can be crucial, as can be seen by the example of `FreeSurfer`[1]. FreeSurfer is a tool that supports analyzing magnetic resonance imaging (MRI) scan data. Running experiments using the same version and the same operating system resulted in the same results, as expected [34]. Different versions of FreeSurfer, and even just running specific versions of FreeSurfer on different operating systems, resulted in divergent results beyond statistical significance. This example shows the importance of carefully selecting software versions, but also the impact of changes to the environment where a process is executed in.

A first step towards sustainable processes is done by documenting the process environment, i.e. the dependencies of a process. This clarifies the border of the process environment. But process environments may overlap, therefore maintenance still is limited as other processes might be affected. So as a second step the process environment needs to be extracted and moved to a dedicated execution environment. Such dedicated environments do not necessarily need to be dedicated physical systems, but also could be virtual systems or execution environments like provided by application packaging systems. Virtual systems are very suitable as target environment because they are highly portable and compatible with most scenarios.

[1]`http://freesurfer.net/`, accessed 2014-02-25

This work focuses on the extraction of process environments from shared systems and redeploying them on dedicated virtual systems. That approach poses several advantages, which are explained in the following in conjunction with basic terminology of this document.

The term *process* in this work refers to a sequence of activities (e.g. tool invocations) executed by a system, like in the context of process mining [12], and not to the runtime instance of an executing application as usually referred to in computer science. *Process migration* in this context is used as synonym/abbreviation for *process environment migration* and denotes the activity of moving a process environment to a different hosting system (*target system*), where the according process can be executed as on the system it has been located originally (*source system*). Process migration involves extraction of the process environment on the source system and redeployment of the process environment on the target system. Automated processes migration support ensures efficient and systematic migration. This is helpful e.g. in optimizing server infrastructure (server consolidation) [19], but also in providing a common environment for a team (e.g. a customized development environment for a team of software engineers), and in sustainably preserving a process (e.g. a process involving legacy tools that are not compatible to current operating systems anymore). More specific scenarios can be found in the e-Science domain, where workflows need to be documented, shared with the scientific community, and preserved for future reference [62]. Such shared workflows are useful to reduce the effort to verify or build on existing experiments [63]. Thereby, trustworthiness and reusability of published results is increased. Process migration also can be used to prepare a test environment for testing new or modified software or a process with specific requirements. Documentation of the process environment that is generated in the context of the migration is an important part of the resulting artifacts as it helps comprehending and subsequently recreating systems. The information that is extracted as part of the migration includes the dependencies that a process requires to operate correctly, i.e. required resources, tools, and services. This information not only supports maintenance of the process, but also redeployment on a dedicated system. The documentation is valuable when allowing the process to evolve and to be optimized.

Virtual systems are eligible to host the target environment because of several reasons. The additional layer of abstraction introduced by software virtualization provides various benefits, especially in terms of flexibility, efficiency and security [32]. Migrating software to virtual systems has gained popularity in recent times, not only because of the rise of Cloud Computing where virtualization plays a major role [87]. Virtual systems are much more portable than those directly deployed to specific physical hosts because they can be moved easily to different providers that support the same virtualization technique. This helps reducing costs by better utilization of hardware, and also supports minimizing infrastructure vendor lock-in by introducing a layer between provider and the hosted system. Virtualization can be used to isolate software from its hardware environment. Advantages that are addressed thereby are, amongst others: availability is improved because of eased migration to different physical hosts (e.g. in case of a hardware failure) and the resulting possibility to quickly respond to changing hardware requirements (e.g. to scale because of an increasing user base or to mitigate access peaks).

Migration of (possibly but not necessarily physical) systems to a virtual environment can be performed by creating a one-to-one clone of a system. While this is rather easy to execute and a sufficient solution for systems that need to be migrated completely, there is the problem that services and applications may be captured that are not used anymore e.g. because they are part of the environment of an obsolete process. As such they are likely to not being maintained, to waste resources of the system, and to pose security risks because of software that is not up to date. There also are migrations where splitting multiple services that are executed on one system to multiple systems is required because of performance or maintenance reasons. The approach followed in this work is to perform a partial and customizable migration. Such migrations can be executed with respect to different scopes, i.e. for specific applications, users, or, as in this work, processes. The result is an environment that only contains resources (e.g. tools and files) that are relevant with respect to the targeted scope. This is valuable for several reasons, a few of them are outlined in the following. The amount of software and data that has to be maintained is reduced. As installed software components, especially those that maintainers are not aware of and therefore are not updated regularly, have the potential to pose security risks, also the attack surface is minimized (c.f. [49] for a correlation of updating behavior and security incidents). Furthermore, resource requirements like storage and processor utilization is optimized by discarding data and services that are not required anymore. This improves the efficiency for the remaining services. Compared to cloning systems, this is more difficult to execute, because the required resources like software, data, configuration files, etc. have to be determined. A main risk of partial migration is that relevant resources are not migrated. Testing the redeployed system is an important step to ensure the completeness of the system. As this work focuses on the migration of processes, i.e. their dependencies like tools and files which are required for correct execution, the term migration is used as abbreviation for process environment migration in this document.

Process migration can be combined with the generation of a process documentation. Because the migration needs to analyze the process environment to be able to rebuild it, the findings that have been generated in this step just need to be stored to also get a documentation of the process environment in form of a model. Such models are used to re-create the process environment on a new system, and therefore need information about the technical requirements of a process, so the dependencies that are required directly or indirectly by the process. Models add value by supporting future maintenance and enhancement of the described process, and therefore also are vital for the long term availability of processes.

Although migration usually is not done on a regular basis, keeping the migration process as easy to execute as possible is valuable. Besides providing a documentation to a specific point in time as shown in this work there are also other uses for process documentation. For example the resulting documentation from subsequent executions of the migration process can be used to track changes of the process environment over time by observing the evolvement of the process model. The main tasks of creating a model of the process environment and redeploying the process in a virtual system are tedious to do manually and automation should lower the effort and hurdles to virtualize the environment of a process. Because of this, and the general advantages of automation (e.g. reduced effort and therefore costs, traceability) an automated approach is proposed in this thesis. The boundaries of automation are limited by the level of automation of the process itself and the scriptability of its required tools. Processes that require user interaction cannot be migrated fully automatically without modification. Also tool installation and configuration is a limiting factor if not possible in an automated way, like graphical installers and configuration tools.

For the problem of extracting processes from a shared environment into new virtual systems there are following main research questions:

- How can the environment of a process that is embedded in a shared system be identified?
  It is aimed to provide an approach that allows the identification of dependencies of processes that are implemented as scripts, using workflow engines, or other techniques. A further goal is to show limits of such an approach, along with possible ways to overcome such limits.

- How can the process be extracted from such a shared system?
  Requirements for transferring the process environment from source to target systems need to be determined.

- How can a process environment be documented?
  An extensible model is required, which is able to represent technical and organizational aspects of a process.

My thesis proposes a framework with the aim to support automated and documented migration of processes into virtual environments, including their software and data with its dependencies. This process migration framework (PMF) also creates a documentation of the environment of the process as process model. It is developed on the basis of current research results and industry best practices. There will be a theoretical design of the framework, but also an implementation with a focused scope in terms of supported operating systems and applications.

In the theoretical part of the work the framework will be designed and described. The input of the framework is the software environment of the process, i.e. the system that contains the process. The system will be analyzed to identify the software components and data of the host system that the process accesses. For this the process is executed and the resource access of the process instance is observed. Multiple executions may be necessary to identify the required resources. The model is refined to improve expressiveness, e.g. by reducing resources irrelevant for the migration, like temporary files. This results in an information system model. The user optionally can make changes in the created model, like using a newer software package or different data files. This may be necessary when moving to another operating system where specific tools are not available and need to be replaced by equivalent tools. It also is useful to support changing requirements without the need to modify the created system or to update to new version of tools. The resulting information system model is used to recreate a virtual machine that corresponds to this model.

So in this thesis process migration refers to the documented migration of processes to a virtual system. This is done by executing following steps:

1. Create a model of the process environment by identification of the software, configuration of thereof, and the data that a process utilizes, via process execution monitoring.

2. Customize the process model by allowing exchanging software or data.

3. Build a virtual system that confirms to the customized process model.

The practical part of the work comprises the prototype implementation of the framework, which is based on the results of the theoretical part of the work. The functionality of the modules of the framework is partly covered by existing tools, which are utilized and if necessary extended. The prototype implementation builds a basic process model (supporting packages, configuration, and files) and creates a virtual environment thereof. The time-consuming tasks of recreating the virtual system and identification of the existing system in the migration process is supported by the framework.

Furthermore, the prototype implementation also aims to point out issues and limitations with respect to the possible automation of implementations of such an approach, i.e. the steps that need to be executed manually and specific cases of the migration that may require special handling, like tools that access their components or data with an encrypted or proprietary protocol or have been installed from a non-public source. The practical applicability of such a framework is tested on different scenarios, including processes that execute workflows or access services. The evaluation also shows limitations of the tools the framework builds on.

This work focuses on the software related aspects of processes. The business layers of processes are excluded from the framework, and so are modeling details about the process itself, like process steps and how they are related to each other. The level of automation that is possible in the migration depends on the level of automation of the process itself. This means that for a process that includes steps that are not scriptable an automated migration cannot be expected, because in the context of the framework the process is executed and the non scriptable tasks need to be executed by the user manually during the execution of the framework. So for an automatic migration the activities of the process need to be specified in one script. Activities that cannot be automated therefore cause manual intervention during the migration process of the framework.

In the implementation of the prototype the scope for modeling and migration are processes that run on single physical or virtual machines which run Debian or derivatives. They will be redeployed on a virtual system with the same hardware architecture also running GNU/Linux. Files and software packages are the main resources that are considered in this framework. Other operating system specific information (e.g. the kernel version) or hardware information (e.g. special graphic devices that are deployed on the system) is not considered. The target system for the migration are virtual machines.

My contributions are the design of the process migration framework, and a prototype implementation that integrates existing tools to test and demonstrate the capabilities of the framework.

The framework is evaluated on different process scenarios, which are implemented as script or workflow. The scenarios for the evaluation are taken from the domain of e-Science and from the domain of civil engineering. The prototype is installed on the host system of each scenario. It is applied on the host and should generate a virtual machines that contains the required software components and data for the respective process. The virtual machines are verified by comparing with a previously generated expected set of resources. From the evaluation the completeness, the level of automation that is possible, but also potential issues are extracted.

The document is structured as follows: Chapter 2 describes existing work that are either used within this thesis or complement the activities of the PMF. The design of the framework is described in Chapter 3. The description of the of the modules of the prototype implementation can be found in Chapter 4, which also contains a description of the used process model, and interfaces of the framework. The framework is evaluated in Chapter 5 on real-world use cases. A conclusion and an outlook on possible future development are given in Chapter 6.

CHAPTER 2

Related Work

This work touches various fields of research, from process monitoring to virtual machine provisioning, process environment modeling and others. This chapter provides an overview of related concepts, which have influenced the direction of this work, could be used to expand the scope towards more specialization, or provide solutions for subtasks of the problem.

2.1 Migration

The overall concept of the framework can be seen as a migration problem, and issues of outsourcing and migration of software infrastructure are an active area of research. A basic approach is the system to system migration, which deals with duplicating (physical) machines. Main issues thereof are described in [52]. As one of the migration methodologies the Butterfly methodology is presented in [109], which describes replacing a legacy system with a target system over time. The PMF has been developed with the focus to migrate processes at once, but it can be used in such methodologies to migrate individual processes that are part of the overall legacy system. Aspects that should be considered when developing a replacement for obsolete software are described in [110]. It is shown how visualization and divide-and-conquer techniques can support understanding the legacy system, which can be helpful when applying the PMF on large-scale systems. Automated service migration in case of malfunctioning or attacked services is described in [116] using a logic based framework. If specified requirements cannot be satisfied, the framework suspends the involved services and resumes them on another system. An implementation of a framework that handles migration of active processes from one to another Linux system is shown in [113]. Although such usage is beyond the scope of the PMF, it shows areas of possible extensions as well as technology that could be utilized in future versions of the framework.

The basic migration approach of capturing the whole system and redeploying it, e.g. on a VM, is covered already by solutions offered by the industry. One open source tool is virt-p2v [9], which allows cloning a complete system over a server into a VM. Proprietary solutions are VMWare® vCenter™Converter™[1], Symantec System Recovery Server Edition[2], NetIQ PlateSpin® Migrate[3], Microsoft Virtual Machine Manager[4] and others. But this approach does not capture only relevant applications and data of a process but also unneeded software, services, and data which may pollute the target system.

Virtual machines are a popular target for migration, also because of the usage in the hyped Cloud Computing [87]. One of the concepts for supporting automated migration to cloud providers that can be found in the literature is CloudMig [114]. CloudMig uses installation templates to specify software configuration (e.g. specifying installation path, prerequisites, etc.) to assist deployment of software. Configuration policies are used to verify the migration. CloudMig is focused on the configuration of the installation and the verification of the migration, wheres this work also covers analysis and is model-based to improve flexibility of the target system's configuration.

CloudMIG [28, 29], another similarly named framework, has a partly shared scope to this work, i.e. the model-based migration of software systems. But in contrast to the framework that is presented in this document, CloudMIG is more fine grained and operates on the level of enterprise system internals. It performs an in-depth analysis of the system by looking at its architecture and runtime utilization. This level of detail is necessary to allow resource-efficient optimization of the software systems for cloud environments. The PMF does not modify system internals in this way, and therefore follows a more generic approach which can be applied without inspecting any source code. A target cloud environment model needs to be provided where available cloud infrastructure properties are specified. This information is used to create a target architecture that can be deployed in cloud infrastructures that provide the features specified in the cloud environment model. The basic activities of CloudMIG, i.e. extraction, selection and generation, are used in a similar way also in the PMF. Further cloud-specific activities like adaption, evaluation and transformation, which deal with modifying the system to better utilize the infrastructure based on observation of the runtime behavior are not considered in the PMF. CloudMIG uses the OMG Knowledge Discovery Meta-Model (KDM)[5] [80] for representing systems. The KDM can be used to describe the internals of enterprise software, from the level of infrastructure to implementation details like packages, source code, data types, etc. In comparison the model of this work needs to be able to describe processes on the level of tools, their interaction and optionally their relation to business activities. While CloudMIG is useful for customizing systems itself, the PMF is useful when alternatives to tools or dependencies should be considered. The frameworks can be used to complement each other, i.e. the PMF can be extended to utilize CloudMIG to handle the migration of complex enterprise software, when a black box view on such software is not sufficient.

[1]`http://www.vmware.com/products/converter/`, accessed 2013-08-22

[2]`http://www.symantec.com/system-recovery-server-edition/`, accessed 2013-08-22

[3]`https://www.netiq.com/products/migrate/`, accessed 2013-08-22

[4]`http://www.microsoft.com/systemcenter/virtualmachinemanager/en/us/default.aspx`, accessed 2013-08-22

[5]`http://www.omg.org/spec/KDM/`, accessed 2013-10-15

2.2 Process discovery

In order to create a model of the system, the system needs to be analyzed. One of the concepts to consider for an automated analysis, is configuration crawling. An approach for this concept is presented in [65], which uses the API of the virtual machine provider to extract attributes, like architecture or which operating system is installed, of the virtual system. Additionally, configuration management agents collect meta data about the virtual appliance. The resulting configuration data model can be used to rebuild a virtual appliance. Contrary to the PMF the scope is on the analysis of virtual machines. Extraction of information about the hardware is discussed in [112].

Analyzing business processes in order to extract models is referred to as process discovery and is part of process mining [101]. An approach for process identification that uses log files as input to analyze the process is presented in [12]. For e-processes this also is referred to as workflow mining [36]. An approach for workflow mining, also based on event logs, is presented in [102]. Such methods focus on the process itself rather than on the dependencies of the process. Compared to workflow mining or process discovery as discussed in the literature the PMF focuses on identifying the process environment instead of the process itself. Nevertheless, workflow mining can be used to extend this approach by adding information of the process to the model that is generated in this work.

An automated extraction of information about the process environment using enterprise topology graphs is described in [6]. A plugin based approach is suggested, where dependent on the detected node in further iterations plugins for this node type are executed to successively refine the graph. In this approach plugins need to be developed for individual node types, which is avoided in the PMF.

Dynamic approaches allow monitoring process execution during runtime. If access to the source code is available, source-to-source transformation can be applied. In [4] an approach is shown that uses source-to-source transformation to insert test probes so that the branch coverage of tests can be determined. Such an approach could be adapted to insert monitoring statements. Some solutions have been developed based on `AspectJ` as described in [46]. Although the general approach is language independent concrete monitoring statements are language dependent. As such methods require access to the source code they are not applicable in this context.

Another way to monitor resource access is to execute the process inside a sandbox and use the monitoring facilities of it (e.g. the Resource Access Monitor of `Sandboxie`[6]). For managed languages also their virtual machine can be utilized (e.g. Java Virtual Machine) to monitor the process. `Robusta` [88] modifies the virtual machine to add hooks such that system calls can be mediated.

Furthermore the operating system functionality for debugging processes can be utilized by allowing to control process execution (e.g. using interrupts), and to read the context of the paused process states [86]. There are also other tools for intercepting system calls. `Ultra` [13] traces system calls by replacing the dynamically-linked standard shared library that provides the system calls. Because of this, statically linked tools would need to be recompiled to allow observation using this tool.

[6]`http://www.sandboxie.com/index.php?ResourceAccessMonitor`, accessed 2014-02-04

Intercepting system calls on the operating system layer is independent of the language and runtime of tools, and does not require modification of the system. Therefore it is used in this implementation. For GNU/Linux there is `strace` [85] which allows monitoring system calls issued by a process. This is possible because of the `ptrace` system call, which strace uses to intercept the communication to the kernel. Because system calls are also used to load resources, access files, spawn processes, and so on, this is helpful to detect dependencies. Strace is useful for identifying which libraries and data has been accessed, but it also can be used to show socket connections, like connections to a database and requests to a web service.

`CDE` [38] is based on strace and also can be used to monitor resource access in its verbose mode. Compared to strace the output is more concise but also less complete. CDE, for instance, omits system calls that are not vital for tracking resources (e.g. `mmap`), or more importantly details of network connections (e.g. arguments of web service requests). The main purpose of CDE is to create portable packages of applications, which is especially useful for processes that use software which is not available as package but e.g. just has been downloaded and extracted or installed using a custom installer. So CDE could be used in future versions to migrate software that is not available as package but also not portable. An example is software that loads resources like configuration files from the user's home directory, but where this user should not be created on the target system. If CDE would be applied to tools that are available in package repositories, these tools could not be managed by the package manager anymore, which makes them significantly more difficult to maintain. Therefore, CDE is only used as complementary tool to strace for observing the process execution in the PMF.

2.3 Provisioning

Techniques that are used to configure systems in an automated manner are helpful for the setup of the VMs. A basic step is building virtual machine images, which is also a topic of e-Science research in terms of reproducibility of experiments. Kameleon [25] is a tool that automates building machines. It uses imperative configuration files for specifying the target operating system and configuration steps. Kameleon targets different virtual machine environments, e.g. VirtualBox[7], XEN[8] and KVM[9].

Configuration management, i.e. the setup and maintenance of (virtual) machines, can be used to provision the target system. Because of the platform independent description that avoids platform specific installation steps, declarative configuration is useful in this context. Furthermore, an abstract description of the system configuration is given by such a configuration file, which could be used to import existing configurations into process models. A prominent declarative configuration management tool is Puppet [8]. Supported types of resources include packages, services, and files. Resources are described by the properties of the desired state instead of a sequence of steps that lead to the desired state (declarative). Puppet ensures that applying resources multiple times does not cause unexpected effects, so to avoid multiple executions of actions (idempotent). Resources that are defined multiple times with conflicting properties are

[7] `http://www.virtualbox.org/`, accessed 2013-10-17

[8] `http://www.xen.org`, accessed 2013-10-17

[9] `http://www.linux-kvm.org`, accessed 2013-10-17

prevented (unique). Besides providing a domain specific language for configuration, Puppet is also able to simulate changes caused by a configuration, apply the configuration, and report differences to the desired configuration [53]. Thereby, Puppet can be used to adapt systems such that they match a specified configuration, and preview the changes such transformation would cause.

The declarative thought has been brought further by Nix. Nix [21] is a package manager that uses a declarative specification to describe dependencies and preconditions besides other information of packages. Packages are managed in a functional way, i.e. there are no destructive updates, which also improves reproducibility. This is implemented by storing packages in immutable paths, which are created using a hash value that is calculated based on the input that is used to build the package. The input consists of sources, build instructions, dependencies of the package, and other information that is used in the build process of the package [23]. Because of this, different versions of packages do not interfere with each other and can coexist. The directory layout used by Puppet for storing packages might not be suitable for some legacy systems, but especially for new projects in the e-Science domain Nix could turn out to be beneficial. Nix runs on Linux, FreeBSD, and Mac OS X[10]. The Nix package manager has been integrated in the NixOS, where the functional concept also is applied on the configuration of the system [23]. Configuration artifacts are handled similar than packages, technical details and an evaluation can be found in [22].

NixOps (formerly called Charon) [24] provides automated provisioning for NixOS based systems. As it also utilizes the functional language of Nix, the configuration is done declaratively. Besides VirtualBox it also targets cloud infrastructure (e.g. EC2 and OpenStack), for which it is able to create new virtual machine instances. Besides the limitation that only NixOS based systems can be managed, the functionality can be compared to those of Puppet or other declarative configuration management tools. So in the context of the PMF NixOps can be considered to handle configuration management for NixOS based systems.

Another tool that provides virtual machine management and its configuration is Vagrant [79]. It is able to instantiate new virtual machines using VirtualBox or other virtual machines infrastructures and provides access like SSH or shared folders to them [40]. For provisioning it uses configuration management tools like Chef [75] or Puppet. It is not limited to a specific target operating system, and therefore can be used in this framework to build a virtual machine and provision it considering resources defined in a process model.

MetaConfig [76] is an approach that aims to cover the complete lifecycle of systems, so additionally to configuration management also virtual machine allocation and bootstrapping are covered by MetaConfig. Because the configuration of software packages is specified by targeting a specific package manager, it is not as platform independent as other frameworks.

Puppet was chosen for the prototype implementation because of the declarative configuration files, the independence to specific operating systems, and its wide acceptance, i.e. the availability in the official Debian package repository[11].

[10] `http://nixos.org/nix/`, accessed 2014-07-15

[11] `http://packages.debian.org/wheezy/puppet-common`, accessed 2014-04-05

2.4 Virtualization

On the application level there are several techniques proposed in the literature which could be used to virtualize applications. One of them is sandboxing, as described, amongst others, in [81]. Sandboxing allows the interception of access to resources of the system, which can be used to detect dependencies. Moving an application into a sandbox makes it portable, so it may also be possible to transfer it to the target system. Sandboxing also is used to mitigate security issues [32]. Examples for available tools are Sandboxie for Windows[12] or Sandbox System Call API[13] for Linux.

Dependencies can be discovered both in a static or dynamic way. The Dependency Checker Tool [5] can be used to statically discover library dependencies in ELF files. Static dependency detection could be a useful addition to the PMF. One main advantage is that the process does not need to be executed. But with static analysis it is easy to overlook resources, as not all paths to resources are necessarily stored in the binary or in a way that is readable by static analysis only.

Apart from static analysis there also is dynamic analysis, which is focused on the analysis of processes during execution. Linux provides an execution tracing technique that can be utilized by using the `ptrace()` system call [10]. Ptrace can be used to attach to a process and intercept its system calls. An approach using `chroot` is described in [55]. On a lower level tools like QEMU can be used to trace program execution [72]. Execution tracing is also used in testing and verification of software [58, 59]. But execution tracing is not limited to Linux, Wetrp [111] is an example for research that targets the Windows platform.

There also have been solutions developed that target the runtime analysis of applications in specific runtime environments. JavaSnoop[14] can be used to analyze the behavior of Java applications by intercepting method calls [93]. One possible approach to allow such monitoring is to utilize the Java Attach API for loading agents into the target virtual machine [30]. The agents can be used to modify loaded classes, in this case to add logging statements that help monitoring the execution of the application. Another example for tracing specific runtime environments is the FXU Tracer [20] for C# in the context of eXecutable UML, which executes the model and visualizes the nodes of the model as they run.

Application packaging goes a step further, and besides monitoring also copies accessed resources to a single location. This allows to provide a portable application by redirecting the resource access to the local copy during execution. Examples for such applications are JauntePE[15] for Windows, and CDE [38, 39] for GNU/Linux. Because of the differences in handling software installations of Windows and GNU/Linux, the differences of this applications are that JauntePE is optimized to create a portable version of the application by installing it using JauntePE, so it can intercept the installer, whereas CDE observes the actual execution of the processes. Also commercial solutions are available, e.g. Spoon.net[16] for Windows.

[12]`http://www.sandboxie.com`, accessed 2013-10-17

[13]`http://sandbox.sourceforge.net/`, accessed 2013-07-24

[14]`https://www.aspectsecurity.com/research/appsec_tools/javasnoop/`, accessed 2013-07-23

[15]`http://jauntepe.sourceforge.net/`, accessed 2013-10-17

[16]`http://spoon.net`, accessed 2013-07-23

The prototype implementation uses the Linux execution tracing technique and CDE, but only for identification of the process environment. For building the virtual target system, VirtualBox is utilized. The main reasons for this are to support maintainability. Resources that have been packaged are decoupled from the package manager, and therefore need to be maintained (i.e. updated, replaced) without the support of a package manager. An additional runtime also adds a layer between the process and the operating system to be able to control resource access, which is avoided by natively deploying the process and its dependencies on the target machine. The packaging component of CDE can be utilized in future versions to allow packaging of applications that have not been installed using a package manager and are not available in package repositories to be migrated.

2.5 Enterprise architecture modeling

Another important aspect of the PMF is how the process environment can be described. Using a model as representation of the process supports tool independence and extensibility of the process environment documentation. For the PMF the model needs to capture the installed software, its configuration files and data. A holistic model would reflects all levels of a business process, from the management perspective to involved physical components. This is beneficial because the dependencies between the different components can be modeled, and therefore also connections of the inspected system to the business process established.

Various metamodels have been developed to describe enterprise software and its environment. As mentioned above KDM is one of them, but its focus lies on the description of enterprise applications rather than the environment and context in which they are used. The Open-system environment reference model (OSE/RM) is a reference model developed by NIST based on the POSIX standard [44]. It focuses on the technical model of enterprise architectures, e.g. applications, their interfaces, external dependencies, hardware and software components. TAFIM [18] has been developed by the United States Department of Defense. It has partly been based on concepts of the OSE/RM but adds aspects about work organization and information management. The Open Group Architecture Framework (TOGAF) [94] focuses on four architecture domains: business architecture, applications architecture, data architecture, and technical architecture. It is in turn based on TAFIM. ArchiMate [95] provides a graphical language for enterprise architecture modeling. Its main concepts are taken from TOGAF, but not all aspects of the TOGAF framework are covered in ArchiMate, such as strategic high level aspects and low level engineering aspects.

The TIMBUS project[17] created a context model to describe business as well as technical aspects of processes [96] using ArchiMate. It is implemented as ontology [91] in the Web Ontology Language (OWL) [35] and has been published online[18] and described in [3]. The context model consists of a domain independent ontology (DIO) that includes the core concepts, which can be refined and extended by domain specific ontologies (DSO's). The DIO includes the core concepts that are needed to describe a process. It is based on the concepts of ArchiMate. Especially the technology and application layer are utilized in the DIO. Aspects of importance to the

[17]`http://timbusproject.net/`, accessed 2013-08-22

[18]`http://timbus.teco.edu/public/ontologies/`, accessed 2013-08-22

TIMBUS project are described in greater detail in the DSOs. Such an approach supports creating extensible process models. For the PMF the CUDF DSO[19] and the Software DSO[20] are used. CUDF (Common Upgradeability Description Format) [99] is used to model software packages and their dependencies. As in [41], in the PMF the term ontology describes the vocabulary that is used in knowledge bases, which themselves represent the actual process environment models.

OWL files can be edited manually, but there is also tool support for editing ontologies available. An example is Protégé[21]. Protégé is open-source, available for different platforms, and allows viewing and editing ontologies, as well as performing queries [92].

2.6 Summary

For many of the research areas that are important to the PMF there is a large amount of research already available. As described in this chapter such areas are the migration of legacy systems, analysis of processes in live systems, provisioning of systems and virtualization on different levels, from systems to applications. The PMF builds on existing tools and frameworks such as Puppet, VirtualBox, and CDE to provide migration and documentation of process environments. The PMF also can be used in the context of the Butterfly methodology and other frameworks with a similar scope to implement or extend their process migration capabilities. The following chapter describes the design of the PMF.

[19]`https://timbus.teco.edu/public/ontologies/DSOs/CUDF.owl`, accessed 2013-11-20
[20]`https://timbus.teco.edu/public/ontologies/DSOs/software.owl`, accessed 2013-11-20
[21]`http://protege.stanford.edu/`, accessed 2013-10-10

CHAPTER 3

Design

In this chapter the design of the process migration framework (PMF) is presented. The PMF describes the process of documenting and migrating process environments to virtual environments. The term *process environment* refers to aspects of a system that a process depends on for successful execution, e.g. the operating system and software packages that the process utilizes. Remote dependencies of the process, like external services (i.e. web services), are not part of the process environment. The process environment is described in a *process environment model*, which is used synonymously to *process model* in the following. *Virtual environments* are provided by virtualization software and describe the container where virtual systems are executed in.

The goals of the PMF are to create a virtual system (*target system*) where a specified process that is embedded in an existing system (*source system*) is able to execute in, as well as creating a documentation of the process environment. To be able to generate this output it requires access to the source system and to a description of the process that is to be migrated, e.g. a script that executes the process (*process execution script*). The process execution script is located on the source system. The configuration that is furthermore part of the input is used by the PMF for the optional adaptions to the system, c.f. Section 3.3. The process model serves as documentation of the process environment. The dashed border around the target system indicates that the target system is virtual and not a dedicated physical system.

The process described by the PMF consists of four main steps. They are executed consecutively and each step returns a result which is used by the next step. The steps correlate with the main components of the framework. The components are introduced briefly in the following, and described in detail in the subsequent sections. Figure 3.1 shows the process including its message flow.

Capture Identification of the process environment. The source system is analyzed and the process environment represented as model.

Adapt Refinement of the model. The model created in the previous step is adapted by e.g. replacing software. This step is optional, but adds the flexibility to handle changing requirements, like in tool versions.

Build Building of the target system. A virtual machine that corresponds to the refined model and where the process can be executed on is created.

Verification Verification of the model and the target system. It is verified that the process on the target system shows the same behavior as the process on the source system. Also the model is verified for correctness and completeness.

In Section 3.1 the process model is described, followed by the process steps Capture in Section 3.2, Adapt in Section 3.3, Build in Section 3.4, and Verification in Section 3.5. A summary of the design is given in Section 3.6. Chapter 4 and 5 subsequently provide a description of the concrete implementation and practical usage examples.

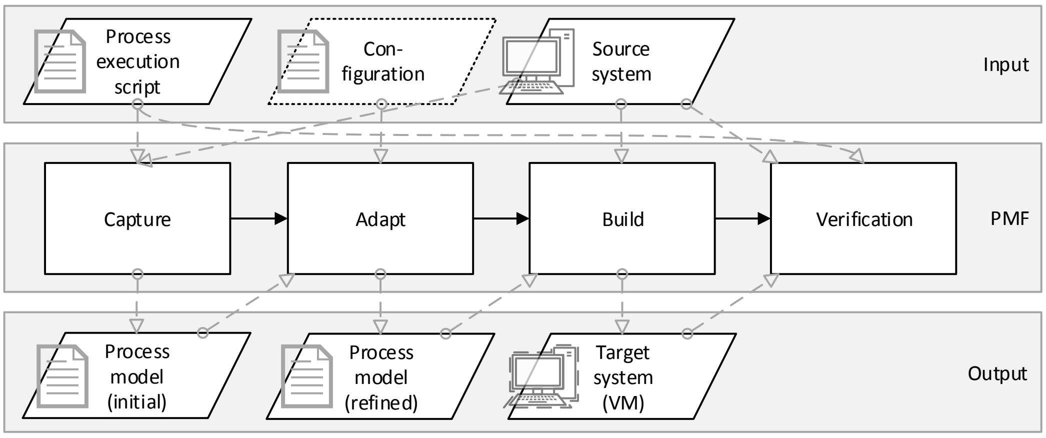


Figure 3.1: High-level representation of the PMF process

3.1 Process environment model

The purpose of the process environment model is to document process environments. It is used both as internal format to share information about the process environment between components, but also as result of the migration process to document the process environment for the user. It should also be possible to generate a system that contains all dependencies and has the same properties as specified in the model in an automated way. This section provides an overview of what information is relevant in a process model in this context.

The process model contains information about the system that is required for successfully executing the process. Successfully in this context means that the process produces the same results as if executed on the source (reference) system. The model needs to be machine interpretable to be able to generate a system from the model. It is a structured description of the dependencies of a process. There is a distinction between resources that are used directly by the process and resources that are used by the process without interacting directly with the process but through other resources. *Direct dependencies* are e.g. tools that are invoked by the process as well as the data these tools access or alter. Such tools in turn also may depend on backing libraries or other tools (*indirect dependencies*). A process consists of a sequence of tool invocations that manipulates data. Tools also can invoke services e.g. web services.

There are several possible ways to represent the model. A representation as ontology is shown in Section 4.2.

The artifacts and their relationships that are relevant to describe process environments are shown in Figure 3.2. The different types of dependencies that are used by a process are described in the following in more detail.

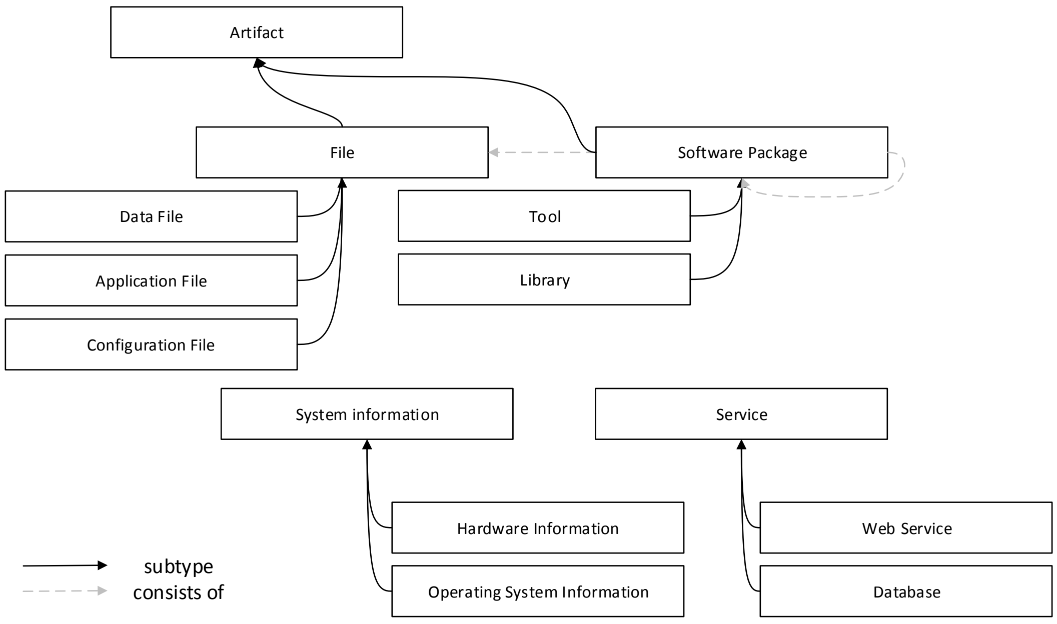


Figure 3.2: Elements of the process environment model

3.1.1 Local artifacts

Local artifacts in this context are software artifacts that are located on a system directly. Such artifacts are files, tools or libraries.

A fundamental type of dependencies are *files*. Files serve different purposes: *data files* contain information that a process uses or modifies, *configuration files* are used to specify settings for tools that are executed as part of the process, *application files* are files that belong to an application but are not part of software packages. An example for the latter are files that belong to *portable applications*, which usually are just downloaded and extracted on the system, thereby frequently located in a single main directory and its subdirectories (e.g. the Java-based Protégé). Portable applications may still use configuration files that are stored in a different location, like the home directory of the user. For data and configuration files the properties of interest for the model include: path, user access permissions, and attributes like flags that mark if files represent symbolic links or if files are executable. This information, together with access to the content of the files, allows migrating the files to the new system.

Tools are software packages that consist of files, i.e. at least one executable and optionally various bundled artifacts like libraries and resource files. But tools also use files, especially for data in- and output, but also to manage configuration. *Libraries* are software packages that provide functionalities for tools. They can be bundled with tools or installed separately to be available for multiple tools on the system. Like tools, libraries also consist of files. In the process model of the PMF tools and libraries are summarized to *software packages*. Packages may depend on other packages. There may exist different versions of one package, which even could be incompatible to each other. Furthermore, packages may reside in different repositories where they can be installed from. This leads to following important properties of packages: name, version, dependencies, and source repository. This information allows reinstalling packages that the process depends on.

3.1.2 System information

General information about the system and its settings is important for migrating processes. To ensure compatibility of the software packages with the operating system, in some cases even the same operating system may be required. Therefore information about the operating system, its architecture and version has to be contained in the model. The user account that executes the process also needs to be available on the target system. This is important to avoid issues with permissions (e.g. file access permissions), and absolute paths (e.g. access to the home directory of a user by absolute name instead of using an environment variable). Environment variables also could impact process execution and therefore have to be modeled.

System information sometimes is represented by local artifacts, like the name of the system which is stored in a dedicated configuration file (`/etc/hostname`) in Debian. Other information sources are environment variables, e.g. to determine the user that executed the process. But there also are properties of interest where it is not that clear which information needs to be extracted, like settings in the registry of Windows. Adding system information to the model can be considered redundant because the information can be extracted from artifacts on the source system also. But to keep the model self contained, and to have the extraction logic in one place, the described relevant system information is explicitly stored in the model. In the model there is no link to the resources where the system information is stored. As an example it is not stored if the hostname has been determined by reading a configuration file, calling a system information tool, or by any other means. This keeps the extraction implementation hidden from the model.

Besides software and data also hardware information can be relevant for specific processes. An example is the processor architecture, which limits the choice when selecting suitable operating systems for the target system. This is not the focus of the initial version of the PMF but may be added in future versions. Other settings of the operating system, like network settings, do not make sense to be transferred without modifications to the target system, because the virtualized hardware on the target system is different then the hardware on the source system. For such settings it makes sense to keep the default values of the operating system. These are therefore not migrated in the PMF.

3.1.3 Services

Processes may access several services during execution. The difference to tools is that services are not instantiated by the process, but exist independently from it. *Local services* include services that are provided by the local system, e.g. local databases or local web services. For local services it can be attempted to analyze the environment of the services. *External services* are remotely accessible services that are hosted on different systems than where the process executes on, e.g. services provided by a third party. For external services the documentation is focused on the name and the type, no information about the service internals is tried to be determined. The analysis of remote service internals is very limited and not part of the PMF. An example for possible identification is fingerprinting, which can be used to determine the web server software that hosts specific services by observing the behavior to specific requests [57]. *Web services* can be realized using SOAP, REST, HTTP or other protocols. They are used to retrieve data, perform calculations, etc. *Databases* can be accessed using standardized protocols, e.g. JDBC or ODBC. Further services are referred to as general services.

It should be noted that not all of this information is relevant for each process. As an example, it is also possible to interpret tools as collection of files and therefore get along without using software packages. Also the model does not reflect actual behavior of the process, so it is not a documentation of the process itself, but rather the environment where it is able to execute in. Depending on the context of the process there are more dependencies to consider, and some aspects can be modeled in more detail. User access permissions to files are not addressed in the PMF, but it can easily be extended to do so. Also the information about which tool processes which resources (i.e. files), is not yet captured by the PMF. In this section the basic elements of the process model and their relations have been described, a concrete process model implementation is shown in Chapter 4. In the following section the design of the components of the PMF is described.

3.2 Capture

The purpose of the capture module is to identify the environment of a specified process on a specific source system and represent it as model. This is done by a combination of observing the process execution, static analysis, and manual analysis of the source system. These three approaches of process environment identification are reflected by the subcomponents `Dynamic Extractor` (c.f. Section 3.2.1), `Static Extractor` (c.f. Section 3.2.2), and `Manual Extractor` (c.f. Section 3.2.3).

The capture component interacts with the source system. The process execution script that specifies the steps of the process serves as input for this component. Details about how this input is processed can be found in the extractor sections later in this chapter.

The module creates a model of the source system which is used as input for the adapt component that executes after the capture component. The refinement in the adapt module is optional, so the output is already a valid process model which can be used in the build component.

The sequence of executing the subcomponents to identify the process environment is described in the following. The dynamic extractor creates the initial model and forwards it to the static extractor. The static extractor adds information that is statically available for the elements in the initial model and passes the resulting model to the refinement component. The refinement component removes information that is not mandatory for running the process and thereby makes the model more readable. The model is then passed to the manual extractor. The manual extractor is used to manually add information that could not be identified by the automated extractors (static and dynamic). Because the subcomponents use information from the model that has been added from previous subcomponents as input, the order of execution is not arbitrary. Using the process environment model as data structure for communication between the subcomponents allows easy addition of new components, e.g. extractors. Each subcomponent returns a valid (but not necessarily complete) process environment model. This process is shown in Figure 3.3.

Compared to a parallel execution of the extractors the sequential execution has the advantage that the step of merging the different models is unnecessary, and extractors can build upon information determined by other extractors. A drawback is the lower performance that derives from sequential execution and the interdependence of extractors.

A prerequisite for the extraction is the process execution script, which executes the process and is used to observe its behavior (c.f. Section 3.2.1). Basics scripts may already be available (e.g. scripts that are invoked to run the process), or are created as manual activity in the PMF. The aim is to provide a script that executes all steps of the process, and for all steps also all process execution paths that the target system is required to be able to execute. The process execution path describes for each decision in the process which path is taken. The script needs to define one or more executions as all possible process execution paths should be executed to ensure complete migration. Only the execution paths that are covered by this script are expected to execute successfully on the target system also. The script furthermore is important for the verification step, as all paths that are covered in this script can be verified by rerunning the process on the target system (c.f. Section 3.5). One possibility for covering multiple paths is to extend the execution script to run the process multiple times with different input. If this is not possible or further resources are anticipated to be used in the future on the target system, the model has to be extended. This can be done manually by providing those resources in the manual extractor, or automated by implementing a custom refiner (see Section 4.3.4).

In the following the three process environment identification methods are described.

3.2.1 Dynamic Extractor

The first process identification approach uses dynamic monitoring of the runtime behavior. The dynamic extractor performs automated process environment identification by monitoring the

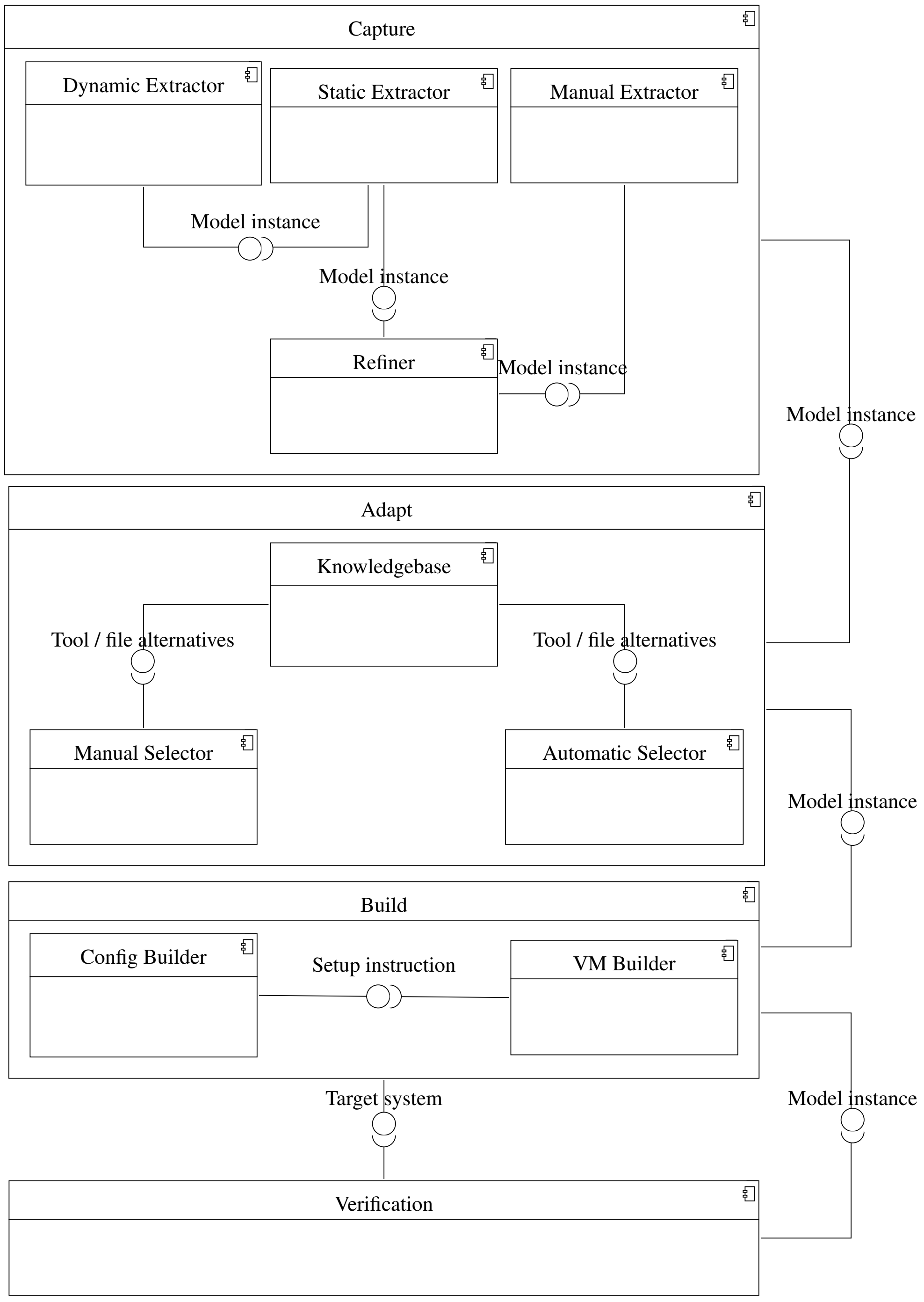


Figure 3.3: The components of the virtualization framework

process execution to extract information about the dependencies of the process. The input of the dynamic extractor is the source system and the process execution script. This extractor runs the script and observes the process during execution. The gained information is stored in a model, which is the output of the dynamic extractor and is further processed in the static extractor.

There are several requirements for the input data. The process execution script is expected to be executable by the system, so e.g. a script which runs the individual steps of the process. If natural language is used to describe the process or if manual intervention is needed the framework is not executable in an automated manner. Because the process is executed also during the verification (c.f. Section 3.5) manual steps should be automated and integrated in the process execution script if possible.

Tools that are invoked independently of the process execution script are not considered by the dynamic extractor and their process environment information has to be added using the manual extractor if the tools are not migration separately using the PMF.

There are different approaches to monitor process execution. The first approach is to use operating system facilities for software process tracing [86]. This approach is independent of the execution environment of the applications but tightly coupled to the operating system.

A higher level approach is to utilize debuggers. Debuggers usually provide the possibility to step through instructions and observe the software process state. This makes it possible to extract information about the process requirement dependencies, like files that are accessed or tools that are invoked during execution. Debuggers are usually limited to a specific execution environment, which means that to debug native applications different debuggers are needed than for applications that are written in managed languages like Java or Python. Although mixed-environment debuggers do exist, implementations like `Blink` [56] are limited in terms of supported execution environments.

Another possibility is to add a layer between the operating system and the process, to then intercept the communication between process and operating system. This is sufficient because only external dependencies like files are provided by the operating system. Sandboxing is an example for such an approach. Furthermore, techniques to rewrite object code have been proposed, which can be used for instance to insert statements that allow tracing the behavior of applications [115]. Also, languages and framework features that provide monitoring may be utilized, as described in [47]. Since such methods are quite specific such approaches are not followed in the PMF. Nevertheless, implementations of the PMF are not limited to use a specific monitoring approach.

In the PMF operating system facilities are utilized to observe processes because of the independence to the implementing technology of the artifacts that are involved in a process. Also no source of the tools and libraries is necessary and the execution environment of the process is altered minimally. The resulting tight coupling to the operating system is mitigated by bundling the analysis scripts as replaceable package, allowing it to be replaced by a package of tailored scripts for the according operating system of the process. Scripts for Debian are described in Chapter 4, extraction scripts for other operating systems are left for future work. A description of a possible implementation can be found in Section 4.3.2. This approach also allows the mapping of accessed resources to the corresponding operating system process. This information, along with the flow graph of operating system processes is added to the model to provide information about temporal aspects of the process execution.

This component extracts the following information about the process environment:

- File names, including data files, configuration files and files that tools consist of
- Service calls, including their address
- Name of the user that executed the process
- Spawned operating system processes and their relations to resources and other operating system processes

Monitoring the execution of a process once with one input in general is not sufficient to ensure completeness of the identified process environment. This is because depending on the input different paths (or branches) may be executed. A possibility to mitigate this issue is to use a test data generator and execute the process multiple times with the generated input data. However, it is possible to execute the process in the target systems with at least that input that has been used during the capturing. The challenge of ensuring high branch coverage is also part of research in other areas. In [70] an approach is presented where taint analysis is performed to monitor the execution flow. Multiple paths are explored by backtracking to decision points and changing the decision so that another branch than in the previous step is taken. In testing branch coverage plays an important part to determine the test coverage.

To summarize, the drawbacks of the dynamic extractor are that the process needs to be executed and that the level of completeness depends on the coverage of executed branches during capturing. The advantages are that this approach is easy to implement and that it allows a generic applicability, i.e. without limitations to a specific programming language or execution environment of components of the process that is analyzed.

The analysis of the dynamic extractor results in a list of files that are accessed by the process, services that are accessed, and the name of the user that executed the process. This information is added to the process model, which is passed to the static extractor. The static extractor adds details to the findings of the dynamic extractor, like which packages the accessed files belong to.

3.2.2 Static Extractor

The static extractor is used to identify the process environment statically, i.e. without executing the process. The input of the static extractor includes the source system and the model generated by the dynamic extractor. The requirements to this input data are the same as for the dynamic extractor (see Section 3.2.1). This component extracts information that is available without executing the process, like the name of the operating system, but also adds information to the resources identified by the dynamic extractor. The resulting model may contain information that is not necessary to rebuild the system, like temporary files, and therefore is then passed to the refinement component (c.f. Section 3.2.5).

This component extracts following information about the process environment:

- Software packages that are used by the process.

- System information like the operating system name and version, or, for future versions of the framework, information about the hardware.

There are various sources of information for static extraction of information about the process environment. Depending on the operating system, environment variables may provide information like the name of the active user or paths to executables and the like. System tools can be used to retrieve information about installed software packages, like determining which files a package belongs to. Furthermore, configuration and other files can be read to retrieve information about tools or the system, e.g. the supported processor architecture of the operating system.

A further possibility is to perform a static analysis of tool executables that are involved in a process to identify their dependencies [105]. Static analysis is used especially in security research, but can be adapted for static extraction of information about tools. Besides executables also other file types can be checked for dependencies. This can make sense for files which contain relevant data, like Taverna workflow files where information like path and location of tools and web services that are used in the workflow is specified. Because the extraction of relevant information from files cannot be done in the same way for all individual file types, this requires customization of the extractor. This approach may improve accuracy for selected processes but is not in the focus of this work and can be added for specific scenarios.

In the PMF operating system utilities are used to extract static information about the system. As the dynamic extractor already identified the resources that are involved in the process, the focus of the static extraction in the PMF does not lie on extracting further dependencies, but to map information that is already available to higher level elements, i.e. the mapping of files that are involved in the execution of the process to a list of software packages. For specific scenarios it may provide useful to extend the static extractor e.g. to be able to parse data files of specific tools and extract dependency information of thereof.

The static extractor summarizes files from the process model for which a corresponding package could be found, and adds these packages to the process model. Libraries and other artifacts that are part of a package are removed and the package is added. This aims to improve the readability of the model. Files that have been modified locally are excluded from this information aggregation, so that local changes are not lost when rebuilding the system. Examples of such files are configuration files, where the default versions of packages are modified so that they reflect user or system specific settings. Depending on the operating system it is possible to detect locally changed files by comparing files against the original files or their hashes (see Section 4.3.2). No required information is lost in this step, because the contents of a package can be looked up by package managers.

Besides adding package information also information about the operating system is added to the process model. Depending on the implementation this may be done using identification scripts that are backed by operating system tools. A description of an implementation can be found in Section 4.3.1. The static extractor can be extended in future versions to extract various other information about the artifacts of specific tools and about the source system itself. Information about data files could include the format and permissions, information about the system could include information about the hardware.

The process model is passed to the refinement component to further improve its comprehensibility. By rebuilding the system on the level of packages it is ensured that the target system is maintainable, so that updates can be performed and dependencies of the process are managed in a standardized way using the package manager.

3.2.3 Manual Extractor

This component is used to manually analyze the source system to add information of the process environment to the model that could not be identified by the dynamic nor the static extractor. In the manual extractor the process of extracting information is delegated to an expert, who uses domain knowledge to complete the process model that has been generated by the automatic extractors. The experts are provided with a user interface that allows manipulating the model, e.g. Protégé. The resulting extended model is passed to the next main component, the adapt component.

Manual inspection is useful for scenarios where it is difficult to automatically identify the process environment. An example for such is the identification of a system in production, where running the process would cause a change of the system state. Manual inspection is also used to provide information that is not captured by the automatic extractors. If the level of detail does not fit the expectations for a specific scenario, the model can be revised, e.g. to remove indirect dependencies. Additional dependencies that could be required for future application, for example to allow usage in a different context, can be added. Organizational aspects of a process can be added to the model also.

The quality of the manual extraction highly depends on the experts involved in manual inspection and how carefully and detailed the manual inspection is executed. The process of manual extraction also does not enforce reproducibility, i.e. documenting the migration process, which is important for future re-execution. Overcoming these issues leads to the development of tools and workflows that support identifying process environments with minimal manual intervention, which eventually can be used to extend the automatic capturing.

The result of the manual extractor is a model of the process environment which is not completely dependent of the automatic extractors, but also includes information derived from the knowledge of experts. The main differences between the extractors are shown in the following section.

3.2.4 Comparison of the extractors

The separation of the extractors is not strictly necessary, but it supports future refinement of individual extractors by being able to adapt them separately. Using multiple sources of information is useful to get information more easily or more complete than by just querying one source. In some cases individual extractors are able to build a complete model of the process environment. But for the manual extractor doing so would be too much effort, e.g. indirect dependencies can easily be missed in manual extraction, so careful inspection is needed. The dynamic extractor could take over functionality of the static extractor by adding instructions to the process execution script that retrieve information which the static extractor usually adds. As example a statement could be added to the process execution script that prints operating system information. This information can then captured by the dynamic extractor from the execution trace of the process. The static extractor would need to be extended to support each binary and data format that is involved in the process and contains information about dependencies, like libraries or tools. So a main disadvantage for a completely static approach is the lack of generality, i.e. that process specific adjustments to the extractors that are necessary to support the involved formats or tools.

Those considerations together with aiming for a modular design have lead to the separation of the extractor components. The extractors aim to minimize the redundant extraction of information as well as to maximize the generality of applicability of the PMF.

3.2.5 Refinement

The refinement component reduces irrelevant information and artifacts that have been added by the automated extraction steps. By *irrelevant* artifacts and information are meant that are not required to rebuild the system and execute the process successfully thereon. The refinement component uses the model passed by the static extractor as input, refines it, and returns a cleaned model. The resulting model is passed to the manual extractor.

This component has two responsibilities: filtering invalid and irrelevant information, and aggregating information. The aim is to create a more compact and therefore better readable model. Filtering resources not only enhances readability, but also is used to ensure that conflicting artifacts are not copied to the target system.

There are several types of artifacts that are not relevant for rebuilding the process environment. Temporary files (i.e. files that are recreated and overwritten by the process), system files (e.g. information about hardware devices and swap files), and tool resources (i.e. configuration files) that would overwrite existing system specific configuration (e.g. IP address settings) are examples for such files. Those files can just be removed from the model for the purpose of rebuilding the system. For documentation purpose it could be considered to keep all files, which also allows easier debugging in the case of issues with the migration. For the sake of keeping the model clean and readable such files are omitted in the PMF. A basic approach to remove irrelevant files from the model is using a blacklist filter, which is described in Section 4.3 including examples of files and directories that need to be filtered for Debian based systems.

The second responsibility of the refinement module is to convert information to a more compact representation. Like the static extractor for software packages, the refinement module groups files that belong to one portable application and adds this portable application to the model instead of the individual files. Another example are web requests. Multiple web requests where the target address differs slightly, like a changed `GET` parameter, are likely to be handled by a single web service only. Such and other heuristics can be used by the refinement component to group elements of the model (c.f. Section 4.3.4).

The refinement module improves maintainability but reduces the level of detail of the model. Knowledge about individual files of packages should not be required in common scenarios, but can be necessary when the information about which exact files are used by a process is needed. The tradeoff selected in the PMF, which is providing a less detailed but more maintainable model, may not be suitable for every scenario, and therefore should be adapted if necessary.

Alterations to the target environment can be applied by the adapt component, which is described in the next section.

3.3 Adapt

The adapt component supports experts in making changes to the process model that has been created by the capture component. The aim of the module is to suggest and use alternatives to tools that are used within the process. This is useful to be able to use the PMF not only to migrate the process into a virtual system, but also to bring the process environment up to date or respond to changed requirements. The module uses a knowledgebase that is described in Section 3.3.1 to determine possible replacements. A configuration is passed to the adapt component, where the artifacts that should be replaced are listed. This module supports experts in selecting appropriate migrations. The techniques described in the following are not sufficient to provide migration alternatives, but help in getting a preselection of possible options. Therefore, there are manual steps included in this component, which are determining for which tool replacements are necessary, and selecting one of the proposed alternatives. Also, arbitrary other changes can be made to the process model in this step, before passing it to the build module.

The process environment model generated by the capture component is used as source model where migration should be performed. Furthermore, a list of resources that should be replaced is needed. This module returns a model where different aspects may have been replaced by alternatives for which a valid migration path was detected. This process is shown in Figure 3.3.

The adapt module focuses on the replacement of tools. *Tool replacement* refers to replacing existing tools with alternative tools that provide the same functionality. This also is useful to be able to use newer versions of tools. Requirements to replacing tools are that they support the same or more file formats in terms of reading and writing, and that they are able to execute equivalent operations to the ones that the source tool provides. So given a unique tool identifier an alternative list of tools is provided that can process a superset of file types of the source tool.

Another approach is to use package managers to determine alternatives. This is possible because in some systems (e.g. various Linux distributions that use Debian software packages [51]) there are relations between packages and virtual packages specified, so that it can be determined which packages can be used for the same purpose. An example for a virtual package is *www-browser*, which is provided by browsers like Chromium, Epiphany, Iceweasel, and Lynx[1]. In this approach it is not ensured that the suggested tools even support the same file formats. Also, such relations are not available for all packages nor for all package managers.

The migration options are limited in the PMF because no information about the high-level process steps of the process is captured, so no information about the relations between different artifacts is available. Therefore, possible conflicts have to be avoided by ensuring that compatibility to any other resource of the process is not broken. If the model would state dependencies between resources, only affected resources would need to be checked for compatibility. As data source information that is freely available on the internet (c.f. Section 3.3.1) is used. Further possible migrations that can be considered for future versions of the PMF are service replacements, amongst others.

[1]`https://packages.debian.org/sid/www-browser`, accessed 2014-04-06

The first step of the adapt component is to determine which resources should be replaced. This is done by consulting the list of resources to replace that is given as input to the module and is referred to as *configuration*. For each resource which is stated in the configuration, and which also is part of the extracted model, alternatives are searched. The data source that is consulted for retrieving alternatives is provided by the `Knowledgebase` component, which is described in Section 3.3.1. As the knowledgebase may return a list of alternatives it is required to select one of the alternatives that will be incorporated in the model. This selection is done either manually in the `Manual Selector` or automatically using the `Automatic Selector` (see Sections 3.3.2 and 3.3.3 respectively).

The following sections describe the subcomponents of the `Adapt` component.

3.3.1 Knowledgebase

The knowledgebase provides reasonable alternatives to specified elements of the model, so it stores information that is required to suggest tool replacements. This description focuses on providing alternatives for tools, but the knowledgebase could be extended to support providing alternatives for file formats or other aspects of the model as well.

The knowledgebase takes a model element as input, e.g. a software package name, or a file name, and returns alternative elements of the same type, which it looks up in a local database. This database contains details about tools, i.e. the name, version, and which file formats they are able to read and write. To limit the effort of populating the knowledgebase, implementations can import data from online databases like `Freebase`[2] and `Pronom`[3]. Freebase provides information about which tools can process which file formats. Pronom provides additional metadata to file formats, such as MIME types. Using this information alternatives are determined as specified previously in this chapter.

The alternatives providing approach is very basic, and can be improved in future versions of the PMF. A main issue that needs to be considered by the knowledgebase is the compatibility and availability of tools and its dependencies. For instance it is not useful if a tool is suggested that is not available on the selected platform or incompatible with other tools or libraries that are part of the system according to the model. The accuracy of this approach depends on the process model and on the data that is available in the knowledgebase. For the former it is important that relations between tools and files are specified, and also that the types of the files are contained in the model. The accuracy can therefore be improved by adding such information to the model if missing. For the latter data of additional sources could be incorporated in the knowledgebase.

The alternatives are passed to either a manual or an automatic selector, which selects the tools that should be used on the target system in replacement for the original tools.

3.3.2 ManualSelection

The manual selection takes the artifact to replace and the list of alternatives that have been determined by the knowledgebase as input. The alternatives are presented to an expert who

[2]`http://www.freebase.com/`, accessed 2014-01-26

[3]`http://www.nationalarchives.gov.uk/PRONOM/`, accessed 2014-01-26

selects an alternative. The advantage is that because of the manual intervention there will be no unexpected or unnecessary replacements. The disadvantage is that a manual step is introduced, which harms efficiency and reproducibility of the migration process.

3.3.3 AutomatedSelection

Like the manual selection this component also gets the artifact to replace, the list of alternatives but additionally also a configuration that contains a set of rules which specify when to select which tool as input. It also returns one selected alternative for the provided input.

Contrary to the manual selection the automated selection evaluates the rules to select one of the alternatives. Compared to the manual selection this provides advantages in terms of efficiency and reproducibility.

3.4 Build

The build component is used to build the target system from the model that has been generated in the previous components. Additionally to the model the source system needs to be available to transfer the files. The output is a virtual system that corresponds to this model and therefore is able to execute the process. The virtual system is embeddable in a virtualization environment and does not require a dedicated physical host.

There are two subcomponents responsible for performing the main activities of this module, which are preparing the install instructions (`ConfigurationBuilder`, see Section 3.4.1) and setting up the system according to these instructions (`VirtualMachineBuilder`, see Section 3.4.2).

3.4.1 ConfigurationBuilder

The ConfigurationBuilder is used to generate an interpretable setup instruction file from which the target system can be built. It requires the model as input and produces a setup instruction, which is interpreted by the VirtualMachineBuilder component to generate the virtual system.

The ConfigurationBuilder analyzes all elements in the model and depending on the types of the elements the ConfigurationBuilder generates according instructions. For each tool it creates an instruction to install the tool on the target system, for each file it creates an instruction to copy the file from the source system to the target system, for each username it creates an instruction to create the user account on the target system, including home directory and so on. The format of the resulting instructions depends on the implementation of the ConfigurationBuilder. An implementation that uses the Puppet language is shown in Section 4.5.1.

3.4.2 VirtualMachineBuilder

The VirtualMachineBuilder uses the setup instruction created by the ConfigurationBuilder to build the target system. Apart from the setup instruction it also needs access to the source system to be able to transfer data to the target system.

The component executes several steps. First it selects a suitable operating system based on the setup instruction. This operating system is installed in a minimal version and configured to match the system information specified in the process model. Tasks of the configuration include adding the user accounts. Then the system is provisioned with tools and libraries (software packages) according to the setup instruction. Finally files (i.e. data files and configuration files) are copied from the source system to the target system. This results in a system which corresponds to the model and which provides an environment where the process is able to execute in.

As stated previously in this chapter the target system is virtual and meant to be executed in a virtual environment. An alternative approach would be to use a physical host as target to build the system. Also cloud services that provide virtual environments like Amazon EC2[4] could serve as target for the system. Changing the target of the system only impacts implementation details and may be adapted according to the requirements of scenarios. Section 4.5.2 shows an implementation of the VirtualMachineBuilder based on VirtualBox.

3.5 Verification

Verification is necessary to be able to confirm that the process on the source system behaves like the process on the target system. It requires access to the source system, the target system, and the model. Using those it decides whether the model and the target system correspond to the source system in respect to the process environment. The aim is to be able to determine if the target system provides an environment for the process that is suitable to generate valid results. The approach described is focused on the final result of process execution and direct dependencies of the process. Details on verifying migrated processes can be found in [67], where the VFramework is described. If the process is not deterministic it is not adequate to compare the results of executing the process on the source and on the target system. In such cases the VFramework needs to be applied.

Aspects that need to be considered when validating the result of the PMF are described in the following.

Valid model The model needs to contain all artifacts and system settings that the process depends on. Exceptions are those elements that have been modified in the adapt step of the framework. As described in Chapter 1, where an example of the possible impact of slight changes in the process environment is shown, it is important that all dependencies are extracted and documented. Even if indirect dependencies may not be required to be stated explicitly in the model to be able to rebuild the system, they are still required for a complete documentation and to ensure that an equivalent process environment can be reconstructed. Validity of the model can be determined by manual inspection of the process model by an expert. Another way is to check if it is possible to run the process in a system that only contains dependencies that are specified in the model.

Valid target system The process environment that is described in the model needs to be reflected by the target system. The target system should contain only those artifacts that are

[4]`http://aws.amazon.com/ec2/`, accessed 2014-01-26

part of the process environment or the underlying operating system. Validity of the target system can be determined by checking if all artifacts that are described in the model are available on the target system and that all settings that are defined in the model match those settings on the target system. Applying the PMF to the target system and comparing the resulting model with the model generated by the application on the source system helps in determining if all resources of the latter model have been transferred to the target system. Manual changes to the model should appear as differences. This check implies the test if it is even possible to run the process on the target system.

Valid result of the process in the target system A simplified version of the VFramework is used to validate the migration. The description of the original environment is captured when applying the PMF on the source system. In the simplest case the only measurement point is the result of the process. The results of the process executions when running the PMF on the source system therefore need to be stored. Redeployment is again performed by the PMF and reflected by the target system. The process is executed on the target system with the same input data as on the source system. Finally the results of executing the process on the source and on the target system is compared. The results should be equal, except if the build module has been used to adapt the model, in which case it has to be determined by an expert if the result is acceptable. Depending on the scenario, the VFramework can be followed more strictly to improve the accuracy of the verification.

To cover those aspects the process of validating results generated by the PMF involves several steps.

- The model is verified against the source system. This step needs to consider adaptions that might cause deviations to the model or the result of the process.
- The target system is verified against the model.
- The process result is verified against the result determined in the first step, also considering deviations because of adaptions to the environment.

If validation fails, this could mean that the model is incomplete. Fixing this issue does not necessarily require complete re-execution of the capture component. It is possible to add missing information to the process environment model manually (manual extraction) and re-execute the build and verification components. The issue might be caused by missing one of the possible execution paths during the execution of the process. This potentially can be fixed by adding another run of the process with different input values to the process execution script. In this case also the automatic extractor components need to be re-executed.

Depending on the implementation the verification is partly executed manually and therefore tedious to run. Future versions of the framework therefore should aim to automate some of the steps that are necessary to verify the target system and its documentation.

3.6 Summary

In this chapter the design of the PMF has been described. The key activities of the framework were shown by giving a description of the corresponding components of the framework. The process environment model is the main data exchange format used inside and between the main components. It contains the information that is necessary to build a system where the process is able to execute in. The first activity is capturing the process environment and representing it as model. This is done in several subcomponents that refine and restructure the model, so that is better comprehensible. This model serves as documentation of the process environment. The second component allows modifying the model by replacing elements of the model for which alternatives should be used in the target system. It allows adapting the process environment because of changed requirements and to keep the environment up to date, which is e.g. important to avoid security issues. The third component builds the target system using this adapted model. For verification a simplified version of the VFramework is utilized. Figure 3.3 shows the components of the framework and their interfaces.

The advantages of the PMF are the consideration of multiple input sources, the replaceable components that communicate through defined interfaces as well as flexibility of the target environment which is provided by the ability to replace components. This design also shows that a process migration framework can be kept very generic and independent of customization for specific dependencies like tools or file formats.

Main drawbacks of the process identification approach selected in the PMF include that the process needs to be executed, which is necessary because of the dynamic extractor as well as the verification components. Yet, the PMF has to be executed on the source system, which usually should not be a real issue but may entail changes in the source system. Necessary changes include preparing the system to be able to run the PMF and depend on the implementation of the PMF.

The design of the PMF has been kept general to be applicable for different scenarios. The challenges for an actual implementation are shown in the following chapter, where a prototype implementation is described. Although the prototype focuses on a restricted scenario, i.e. processes that run on Debian, it shows the main ideas of how an implementation can look like. The applicability of the prototype is shown in Chapter 5.

CHAPTER 4

Implementation

In this chapter a prototype implementation of the framework is presented, which is based on the design introduced in Chapter 3. The implementation covers the automatic components for capturing the process environment and building the target virtual machine. The other components can be executed manually. Both the capture and the build module are independently callable and provide a command-line interface, so they can be run separately and that automation is possible. This allows integrating the manual components refinement and verification. The monitoring functionality has high dependencies on the operating system, other functionality is not bound to a specific platform. The prototype implementation is designed for `Debian` GNU/Linux. GNU/Linux in general was selected because of its openness and the relevance in the evaluation scenarios, `Debian` in particular because according to DistroWatch.com [100] it itself is very popular, but it also is the basis of several other popular Distributions, like Ubuntu [78] and Mint [69].

This chapter is organized as follows. First, the overview of the architecture is shown in Section 4.1. The data model is described in Section 4.2. Then the capture, adapt, build, and verification components are shown in Section 4.3, Section 4.4, Section 4.5, and Section 4.6 respectively. General information about used libraries and other implementation details is described in Section 4.7. A summary is given in Section 4.8.

4.1 Architecture

The components are developed as independent Java console applications. Their structure is based on the components presented in Chapter 3. Subcomponents correspond to Java classes, or packages if the functionality is split in different classes. For the instantiation of subcomponents dependency injection (DI) is used. Therefore components can easily be replaced by implementing the corresponding interface and adapting the DI configuration. The operating system specific process environment identification scripts are implemented in bash. These scripts have a main entry point, which is referred to as *capture script* in the following. The capture script is invoked by the dynamic extractor. A capture script for Debian is provided. As described in Section 4.2 the process model is represented as ontology.

To allow manual components to be executed as part of the process of the PMF, data is not directly passed between components in memory but rather using the filesystem. This allows separate execution of the components. The components can be executed consecutively, so that if no manual extraction nor manual adaption of the model is necessary the process is executed in batch mode.

The implementation includes the console client and communication with the data model.

4.2 Data Model

In this section the data model for the process is discussed. It is used to represent the process environment, and contains the dependencies of the process. This includes files, tools, and other properties that the process accesses during execution. The model is used for documentation purpose and to be able to rebuild the process environment. The elements of the model and their relations are described in Section 3.1.

As model that represents the process environment the ontology from the TIMBUS project, which is described in Section 2.5, is used. A detailed description of the model can be found in [96]. The model is available online[1]. It is designed modularly as ontology and subsets of the ontology can be used to describe details of relevant aspects of the process environment. The model is used both internally as data structure that is shared between and used by individual components, but also serves as resulting documentation of the process. For this framework the CUDF DSO and the Software DSO are sufficient to describe the software and data context of the process. The Security DSO is used to describe the user that executes the process. Future versions of the prototype could include the Hardware DSO to also document hardware configuration.

Software packages are modeled using the CUDF DSO[2] which is depicted in Figure 4.1 and described in [99]. It provides the possibility to store which packages are installed on a system, and which relations they have to other software packages. *Virtual packages* are used to describe abstract functionality. This concept is used to be able to set packages that offer the same functionality in relation. *Versioned packages* are used to specify version information to packages.

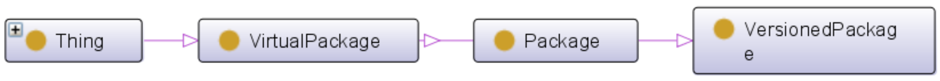


Figure 4.1: Diagram of the CUDF DSO

[1] `http://timbus.teco.edu/public/ontologies/`, accessed 2013-08-22

[2] `http://timbus.teco.edu/public/ontologies/DSOs/CUDF.owl`, accessed 2014-04-15

Files that do not belong to a package are modeled using the Software ontology[3] of the TIMBUS project [3] (c.f. Figure 4.2). The aim is to categorize those files according to their content in the rough categories *LogFile*, *ConfigurationFile*, *DataFile*, and *OperatingSystem*. Because in the PMF the files are transferred directly from source to target system, in this context it is sufficient to model files in a very basic way by just providing the path and the filename. The remaining properties discussed in Section 3.1 are given implicitly by the artifacts, modeling them is left for future development.

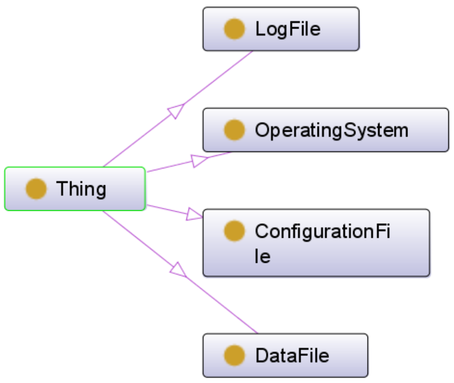


Figure 4.2: Diagram of the Software DSO

From the Security DSO[4] only the *User* class is used. These self-contained ontologies are mapped in the overall process context model, the domain independent object (DIO)[5], using the mapping models CUDFMapping.owl[6], softwareMapping.owl[7], and securityMapping[8] respectively. This is optional, and instead of aggregating the ontologies into the TIMBUS DIO also other ontologies, such as custom ontologies entailed to a specific scenario, may be used.

A mapping or merging of the process model to context specific ontologies is possible. This is useful e.g. for mapping technical aspects about the process to business related aspects. Because the models are represented in OWL and data is represented as statements in OWL, models could just be combined by building a union of the statements of which the models comprise [84]. OWL also provides support to define mappings between ontology elements. So models can be mapped by either establishing a relation from one model to another, or by stating equal elements of the models. Merging of ontologies is described further in [45]. To get reasonable results when

[3]`http://timbus.teco.edu/public/ontologies/DSOs/software.owl`, accessed 2014-04-15

[4]`https://timbus.teco.edu/public/ontologies/DSOs/securityDSO.owl`, accessed 2014-08-05

[5]`http://timbus.teco.edu/public/ontologies/DIO.owl`, accessed 2014-04-15

[6]`http://timbus.teco.edu/public/ontologies/DSOs/CUDFMapping.owl`, accessed 2014-04-15

[7]`http://timbus.teco.edu/public/ontologies/DSOs/softwareMapping.owl`, accessed 2014-04-15

[8]`https://timbus.teco.edu/public/ontologies/DSOs/securityMapping.owl`, accessed 2014-08-05

merging models without manual interference, the models need to be built upon the same schema. The term schema in this context includes all but individuals and their relations, so all classes that specify the type of the artifacts, services, and settings. Individuals represent concrete instances of these classes, i.e. concrete tools, files, and settings. Also equal individuals in different models (e.g. class instances of the schema) need to have the same name in order to be able to merge them without further mapping rules or conventions. Because in general the namespace is part of the name of entities [91], if merging ontologies it has to be agreed on one namespace, to which the models which do not use this namespace are converted to. The approach of mapping instead of merging models is probably better suited for independent models that are expected to evolve separately from each other, where merging is better suited if one model solely exists to provide input for the final model. The context model uses mapping of classes to ensure extensibility and modularity. The process model that results from running the PMF can be integrated into other models by either merging or mapping techniques.

Because of the focus on GNU/Linux systems the main resources that need to be considered are software packages and files. This is reasonable because of the 'everything is a file' philosophy of Unix [61]. Software packages also can be seen as a collection of files but are used for cleaner representation. As described in Section 3.2.2 the files of a software package are not represented separately in the model.

The model is converted to a *Puppet manifest* [53] by the build module of the PMF for actually building the virtual system. The Puppet manifest represents the process environment in a way that is interpretable by `Puppet`, which is used for building and setting up the virtual system. Puppet is described in 2.3. The process of populating the model is described in the following chapter, utilizing the model to build the target system which is described in Section 4.5.

4.3 Capturing

This section describes the implementation of the capture component of the framework, following the design outlined in Section 3.2. The capture component is responsible for extracting the process environment information to a model. This component is executed on the source system and requires access to the artifacts that are used by the process. It takes the process execution script and its arguments as parameter. An existing ontology can optionally be passed as base model. This allows embedding the framework in processes that generate an initial version of the process environment.

A mapping of the design to the implementation is provided as follows. The extractor subcomponents are implemented as classes. The refiner component is implemented as package, where each class of the package is used to define a single refinement strategy. Extractors and refiners both implement a respective interface, so that easy extensibility is ensured. While extractors are rather generic the refiners are more likely to be extended, which is helpful to optimize models for specific scenarios. Additionally there is the `CaptureCLI` which implements a console client for this component. See Figure 4.3 for an overview of the implementation. The monitoring scripts are implemented in bash and independent of the Java project, and therefore not shown in the diagram.

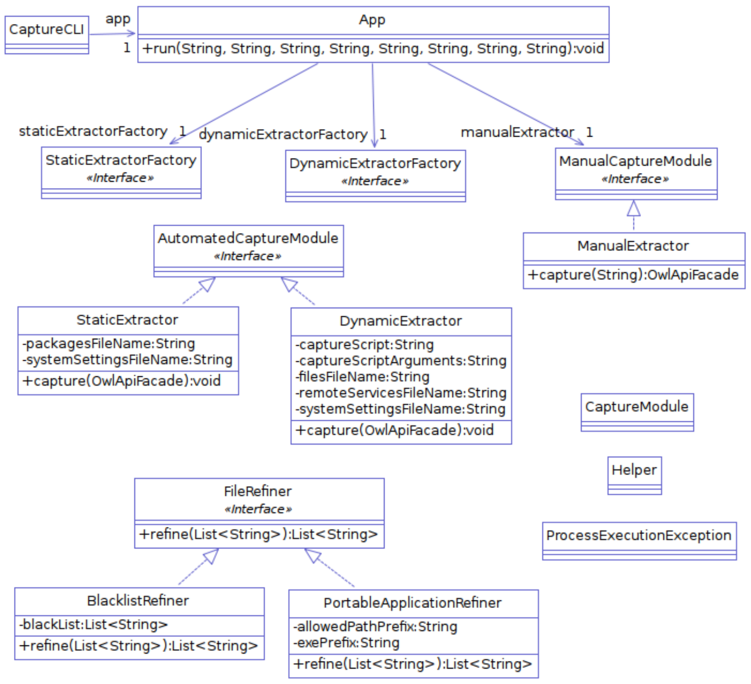

CaptureCLI
app
1
App
+run(String, String, String, String, String, String, String, String):void
staticExtractorFactory 1
dynamicExtractorFactory 1
manualExtractor 1
StaticExtractorFactory «Interface»
DynamicExtractorFactory «Interface»
ManualCaptureModule «Interface»
AutomatedCaptureModule «Interface»
ManualExtractor
+capture(String):OwlApiFacade
StaticExtractor
-packagesFileName:String
-systemSettingsFileName:String
+capture(OwlApiFacade):void
DynamicExtractor
-captureScript:String
-captureScriptArguments:String
-filesFileName:String
-remoteServicesFileName:String
-systemSettingsFileName:String
+capture(OwlApiFacade):void
CaptureModule
Helper
ProcessExecutionException
FileRefiner «Interface»
+refine(List<String>):List<String>
BlacklistRefiner
-blackList:List<String>
+refine(List<String>):List<String>
PortableApplicationRefiner
-allowedPathPrefix:String
-exePrefix:String
+refine(List<String>):List<String>

Figure 4.3: Class diagram of the capture module

As prerequisites for running the capture component it has to be installed on the source system. Dependencies include the JRE and CDE (c.f. 4.3.1). If not already available a process execution script has to be created. The component then can be executed on the source system. A prerequisite for the dynamic extractor is the process execution script, which is described in Section 3.2. The extractors are run sequentially (DynamicExtractor (c.f. Section 4.3.1), StaticExtractor (c.f. Section 4.3.2), and ManualExtractor (c.f. Section 4.3.3)). For optimization reasons the refiners are executed during instead of after the extraction. They run sequentially, first the BlacklistRefiner than the PortableApplicationRefiner (c.f. Section 4.3.4). Extractors and refiners operate in memory to avoid overhead of file I/O, so the intermediate models are not stored as file on the hard disk but passed in memory between the components.

4.3.1 DynamicExtractor

The DynamicExtractor is used to determine the artifacts that are accessed by the process, and described in Section 3.2.1. It expects the manually created process execution script (c.f. Section 3.2) as argument, runs the process and monitors its execution. This allows determining dependencies of the process, like files and services. The main activities of the DynamicExtractor are to execute the process, monitor its resource access, analyze the resources and add them to the model. This section describes the implementation of monitoring file and service access.

Process execution monitoring

Because monitoring in this context is tightly coupled to the operating system and makes use of system tools, the implementation of the monitoring functionality has been decoupled from the Java projecte and uses bash scripts. Those scripts are called from the Java project and provide their results as files that are read by the extractors. This allows inspecting and reusing intermediate results. Also, the scripts are replaceable and can be exchanged to support different operating systems. In this implementation CDE is used to monitor file access only, as the packaging functionality of CDE would convert packages to portable applications, which is not desired because they cannot be maintained using package managers. The results of CDE are combined with those of strace, in case any of the tools has captured more information than the other. Strace is used to log system calls of a process at runtime. System calls not only reflect file access but also access to services. The log files of both tools are merged and processed by the extractor.

The process is likely to run multiple times in order to cover all relevant execution paths, so that, when executing on the target system, no resources are missing (c.f. Section 4.3.3).

The monitoring scripts identify file accesss, local service calls and remote service calls based on the log files of strace and CDE. From this information packages and local tools, as well as information about the services can be extracted, which is shown in the following sections.

As strace maps each system call to the issuing process, it is possible to add information about the relation between operating system processes and the accessed resources. The relation between operating system processes is added in addition, which is necessary to store which process has spawned which other processes. While this information is not necessary to rebuild the process, it shows the chronological sequence of the resource access. An example for an entry for the process in the model is shown in Listing 4.1. It shows that a custom class is used as type for the process, and that besides accessing a tool also another operating system process is spawned.

Listing 4.1: An example for a process artifact as represented in the process model

```
<owl:NamedIndividual rdf:about="http://test#31347">
   <rdf:type rdf:resource=
      "http://localhost/test.owl#Process"/>
   <rdfs:label rdf:datatype=
      "http://www.w3.org/2001/XMLSchema#string">
         31347
   </rdfs:label>
   <DIO:accesses rdf:resource="http://test#%2Fhome%2Ftimbus%2
      FLNEC2%2FClientAppNS%2F"/>
   <DIO:flowTo rdf:resource="http://test#31348"/>
</owl:NamedIndividual>
```

Files

This section describes how files can be extracted from strace execution logs. Strace log files in general contain system calls with their arguments. To get information about system calls the man pages can be consulted using `man 2 [system call]` [106]. Because the system calls `read` and `write` assume an opened file descriptor it is sufficient to filter for calls to `open` and `access`. `Access` can be used to detect if processes check if a file is accessible [48]. The following shows an example strace log entry for opening a file:

```
open("process/run.sh", O_RDONLY)      = 3
```

The concerned file is represented as the first argument of the system call. It can easily be extracted using GNU/Linux tools like `grep` for searching and `awk` or `cut` for text manipulation.

There are some aspects that have to be considered when extracting files from strace logs. To be able to copy files the build component assumes absolute paths. So relative paths, i.e. paths that do not start with `'/'` but e.g. `'..'` or `'.'` are not allowed. In particular also the current directory (`'.'`) is not allowed. Such paths are used in the process for tool execution (e.g. './tool arg1') or for relative references to resources (e.g. `'../res/img.png'`). As the base directory is known from the usage of the `chdir` system call [48], resolving relative paths can be done by switching to the base directory and calling `readlink -e [path]`. This command also resolves symbolic links to their target. Besides the path of files also their type is of interest. Only regular files are considered by the dynamic extractor. Checking if a file is a regular file can be done using the user command `test -f` [11]. This allows avoiding that special files, i.e. device files (`/dev/*`), are added to the model.

Once the files that are used in the process have been identified, they need to be analyzed so that an accurate model of the system can be created. This is described in Section 4.3.2. A similar approach is applied for CDE logs, which uses a slightly different output format where information that is not relevant to the packaging context is omitted (e.g. file access modes). The output of CDE for the above example looks like following:

```
[6732] BEGIN sys_open 'process/run.sh'
```

Files are represented in the process model as shown in Listing 4.2. The listing shows the usage of the Software DSO, which defines basic types of artifacts.

Listing 4.2: A file artifact as represented in the process model

```xml
<owl:NamedIndividual
  rdf:about="http://test#%2Fhome%2Fpmf%2Fprocess%2Frun.sh">
    <rdf:type rdf:resource="http://timbus.teco.edu/
      ontologies/DSOs/software.owl#DataFile"/>
    <rdfs:label rdf:datatype="http://www.w3.org/2001/
      XMLSchema#string">/home/pmf/process/run.sh</rdfs:label>
</owl:NamedIndividual>
```

Tools

This section describes the identification of tool executions during the process. In this context tools are all local binaries that can be executed by the system. In GNU/Linux native executables can be run using the system call `execve` and its derivatives. Executables that are started during process execution are visible in the `strace` log including the arguments the executable has been passed to:

```
execve("process/run.sh", ["process/run.sh"], [/* 50 vars */])=0
```

The first argument is the path to the executable, the second its arguments. The prototype identifies executables that have been called during execution of the process, which is used for the PortableApplicationRefiner (c.f. Section 4.3.4). Besides that, no information about tools is stored in the model, as there is no concept for local tools in the Software DSO at the moment. Tools are represented in the model as regular files instead.

Service calls

The strace logs not only contain file access but also other interaction of the process with systems, like the invocation of services, which is the subject of this section. In particular both local and remote services can be detected by examining the system calls of a process. Services can be hosted locally or on remote machines. Access to the service is done over the same protocol in some cases, like when accessing SOAP web services. Local services are services that are hosted on the local machine, i.e. databases or web servers. Compared to remote services the transport protocol of the interaction mechanism may differ. For MySQL the logs expose sockets for connections to the local Unix systems (*localhost*) and TCP/IP connections for remote services [77]. By checking the target IP address it usually is possible to make a distinction between local and remote service calls. If the process uses external addressing to connect to the service the prototype would not recognize whether if it is a local or a remote service.

MySQL Connections can be established using the system calls `socket` and `connect` from which subsequent `read`/`write` are used for communication. In case of connecting to a local MySQL database the trace looks like following:

```
connect(3, {sa_family=AF_FILE, path="/var/run/mysqld/mysqld.
sock"}, 110) = 0}.
```

So it is possible to filter for the system call `connect` in addition to a specific path that is used by mysql to identify the usage of local MySQL databases.

SOAP The `connect` system call for SOAP requests contains the IP address and port where the service is hosted. The service request (`sendto`) also contains host and port, but additionally gives a hint about the service type, as `application`
`soap+xml` is one of the accepted content types. Furthermore an 'wsdl' POST parameter or the accepted type 'application/soap+xml' in the request gives a hint that the remote service is provided as SOAP service. Such a trace could look like follows:

```
sendto(547, "POST /fexWS/featureExtractor HTTP/1.0\r\n\
Content-Type: text/xml; charset=utf-8\r\n
Accept: application/soap+xml, ...
```

HTTP Another service type for HTTP requests is REST, but REST is a design style and therefore difficult to detect. The accepted type of XML or JSON gives a hint, but this is a rather weak heuristic and therefore is not considered in this prototype. This means that for endpoints of HTTP requests that are not SOAP requests, corresponding HTTP services are added to the model.

Local services that are of a different type than the ones described can be identified by inspecting their executable. As a first step, the service port can be used to determine the process id (pid) of the process that listens on the specific port. This can be done using `lsof` [1]. Having the pid, the executable can be determined by a lookup in the `/proc` file system [60]. The executable is added to the model, and as for all other files tried to be resolved to a package in the StaticExtractor. Another possibility would be to guess the service type by means of checking the standard port mappings [16]. The advantage is that this heuristic is easier to implement. The disadvantage is that services could use alternative ports which are not described by standard port mappings. Furthermore, the executables of local services are not added to the model when only checking the port mappings. The port mappings can be used in the future as complementary approach, but has not been missed during evaluation of the PMF and therefore has not yet been implemented.

Because the prototype only monitors the process and no services that run in independent of the process, it is not able to detect dependencies of the services that the process uses. It would be possible to monitor services by determining the process that listens to a specific port and attaching to it [1]. But because such services may already have been started before executing the process doing so could lead to missed resources. The service would need to be started using strace from the beginning. While this is not in the scope of the PMF, it can be done manually. It is planned to generate a recommendation for complementary migration actions, which includes a list of services that should be migrated separately.

For the case that accessed services are hosted remotely and accessing the hosting system is not possible, techniques like using the port to predict the service or analyzing the requests and responses can be used to identify the services in future versions of the prototype.

Services are represented in the ontology as shown in Listing 4.3. For the distinction of services custom classes have been introduced, that are not part of the TIMBUS context model, SOAP in this example.

Listing 4.3: A service artifact as represented in the process model

```
<owl:NamedIndividual rdf:about="http://test#kronos.ifs.
   tuwien.ac.at%3A8080%2FfexWS%2FfeatureExtraction">
  <rdf:type
   rdf:resource="http://localhost/RemoteServices.owl#SOAP"/>
  <rdfs:label
   rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
     kronos.ifs.tuwien.ac.at:8080/fexWS/featureExtraction
  </rdfs:label>
</owl:NamedIndividual>
```

4.3.2 StaticExtractor

The StaticExtractor uses information of the system that is available independently of the execution state of the process. Tasks of the StaticExtractor are, using the model created by the DynamicExtractor, to determine which packages are involved in executing the process. Also, information about the system is extracted by this component, i.e. information about the operating system. Compared to the DynamicExtractor it does not execute the process or use the process execution script as starting point for analysis but rather the model created by the DynamicExtractor.

Packages

There are several ways to determine which package a specific file belongs to. Because of the fixed directory structure of packages in Unix, it is easy to perform a reverse lookup to resolve a file on the system to a package, if available. Package manager that are used in GNU/Linux for setting up software packages usually also can be used to perform queries about installed packages and information concerning packages. As described in [90] for Debian there are e.g. `apt-file`[9] for a lookup in all packages that are known to the system, `dpkg` which just considers installed packages and `dlocate`[10] which also considers installed packages but is faster than `dpkg`, as it uses a cache [51]. To be sure that also recently installed software is available in the cache of tooldlocate, an update of the cache can manually initiated by using `update-d/locatedb`.

Package managers also can be used to get an overview of the complete system. Retrieving a list of installed packages can be done with `dpkg`[11]. Detailed information can be retrieved using `apt-cache`[12]. In the context of the TIMBUS project[13] a tool has been developed that extracts the packages of a system in a CUDF compliant format.

The prototype does not inspect the whole system, as this would lead to a cloning approach which is not desired in the context of the PMF. Instead, it checks all files using `dlocate` that have been identified as resources of the process for packages where they are contained. Thereby found packages are added to the model instead of the corresponding files. A representation in the process model is shown in Listing 4.4. The listing shows that the type of packages is taken from the CUDF DSO of the TIMBUS context model.

Listing 4.4: A package artifact as represented in the process model

```xml
<owl:NamedIndividual
 rdf:about="http://test#openjdk-7-jre-lib">
   <rdf:type rdf:resource=
     "http://timbus.teco.edu/ontologies/DSOs/CUDF.owl#Package"
   />
   <rdfs:label
     rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
       openjdk-7-jre-lib
   </rdfs:label>
</owl:NamedIndividual>
```

[9] `http://packages.debian.org/sid/apt-file`, accessed 2013-07-16
[10] `http://packages.debian.org/sid/dlocate`, accessed 2013-07-16
[11] `http://packages.debian.org/sid/dpkg`, accessed 2013-07-16
[12] `http://linux.die.net/man/8/apt-cache`, accessed 2013-09-16
[13] `https://opensourceprojects.eu/p/timbus/`, accessed 2014-03-08

There are several aspects about packages that are not covered in the prototype yet. One is that the model does not state the source of the package. Without the source only packages available in the package repositories of both the source and the target operating system are considered. For packages from custom added repositories it would be necessary to state the source (i.e. repository) where packages can be installed from. If a package could not be found in public repositories, the files should be copied and a warning generated to raise awareness about this problem. At least for applications that are just unpacked and in principle portable such attempts could be successful. If not it can be tried to apply application packaging tools like CDE to generate a portable package that can be migrated between different systems.

Future refinements should cope with files that are part of a software package, but which is not recognized by `dlocate` and similar tools, e.g. configuration files in subdirectories of the application directory in the home directory of the user. In several cases a lookup of the parent directory is likely to be sufficient to identify the package of the file. In general, files can be considered as part of the package when it is located inside a directory of a package. But if the owning package of a directory is not unambiguous, i.e. if there are more than one packages where the directory is part of the content, this assumption does not hold anymore.

System information

Also information that is independent of the process execution instances is of relevance. The username should be included in the model because of path and permission issues. This is important because files are transferred to the exact same path in the prototype implementation. Therefore if a file has been located in the user's home directory on the source system, tools may use the home directory to resolve such files. So the home directory on the target system needs to match the one on the source system. This also avoids mismatches of file access permissions between the source and target system. The permissions do not need to be changed on the target system, and therefore also do not need to be stored in the model in the context of the PMF. Furthermore, files may be restricted to specific users, and if the username on the target system differs, access to resources may be restricted. The username can be read without invoking external tools in Java by calling `System.getProperty("user.name")`. Other users on the local system can be retrieved by analyzing `/etc/passwd`, but are not considered in the current version of the prototype, as usually they do not influence the process execution. If the user would be switched during process execution, migrating those users would be necessary, but this has not been the case in any process that was used for evaluating and testing the PMF. Files that are located in any other users directory and accessed by the process are still transferred. Listing 4.5 shows how users are represented in the process model. The class is taken from the Security DSO of the TIMBUS context model.

Listing 4.5: A system information representation in the process model (user)

```
<owl:NamedIndividual rdf:about="http://test#pmf">
  <rdf:type rdf:resource=
   "http://timbus.teco.edu/ontologies/DSOs/
     security.owl#Username"/>
  <rdfs:label rdf:datatype="http://www.w3.org/2001/
```

```
    XMLSchema#string">
    pmf
  </rdfs:label>
</owl:NamedIndividual>
```

Information about the operating system is important to be able to select a target operating system that provides all the required packages, has a matching architecture and is able to run the process. Finding out the distribution name and version can be done using `lsb_release -a`. This information aims to support experts in selecting a suitable target operating system. An example entry of an operating system in the process model is shown in Listing 4.6, which shows the usage of the Software DSO for providing a suitable class.

Listing 4.6: A system information representation in the process model (operating system)

```
<owl:NamedIndividual
  rdf:about="http://test#Linux+Mint+15+Olivia">
    <rdf:type rdf:resource=
      "http://timbus.teco.edu/ontologies/DSOs/
        software.owl#OperatingSystem"
    />
    <rdfs:label rdf:datatype="http://www.w3.org/
      2001/XMLSchema#string">
        Linux Mint 15 Olivia
    </rdfs:label>
</owl:NamedIndividual>
```

Future versions of the prototype also should model environment variables. A reason for this is that if the `$HOME` environment variable has been changed and points to a different directory than the default one, the process on the target system cannot resolve files that are addressed using this variable, because it maps to a different location than on the source system. Therefore, changed environment variables need to be migrated or manually embedded in the beginning of the process execution script as workaround. As not all environment variables should be migrated, because not all of them refer to software that is of interest for the target system, this is not a trivial task. In the evaluation no custom environment variables have been used. Therefore handling environment variables has not yet been implemented.

Files

As stated before it is not necessary to add each file to the model. If the file is part of a package, it can be skipped, except if it has been modified. In this case even if it is part of a package it is added to the model. This is because it is likely that it is used as configuration file, which is part of the package initially but allows modification to reflect configuration specific to the system or user. Although not every configuration file should be migrated to the target system, such files are captured by the StaticExtractor. Examples for configuration files that should not be migrated can be found in Table 4.1. Because most package managers of GNU/Linux maintain a record of hashes of the files in its packages it is possible to determine which files have been modified by querying the package manager. For performance reasons information about modified files is generated by the capture script. It uses `debsums` to check which files have been changed locally [73], which internally recalculates a checksum and compares it to the initial value to determine local modifications.

Another type of files are log files, which are not essential to be available on the target system, but contain information to trace the behavior of the process. The prototype assumes that all files located in a subdirectory of `/var/log` are log files, which is the typical directory for log files in GNU/Linux [74]. Log files are nevertheless included in the model by the prototype as their inclusion causes no harm.

4.3.3 ManualExtractor

The ManualExtractor is used to provide dependencies of the process that are not detected by the dynamic nor the static extractor. The refined model from the dynamic extractors is manually extended by missing dependencies and system information. In the prototype this component is represented by `Protégé` [92] which provides a GUI where the expert updates the process environment model. Of course also any other Ontology modeling tool can be used. Figure 4.4 shows the packages of a process model loaded in Protégé.

The manually altered model needs to conform to the ontology as described in Section 4.2. Manual extraction is necessary to overcome limitations of the automatic extractors, and to be able to migrate processes independent from the completeness of the automatic extractors. Manual extraction is also required when it is not possible to execute the process arbitrary times as required by the dynamic extractor. This could be the case for processes in production environments.

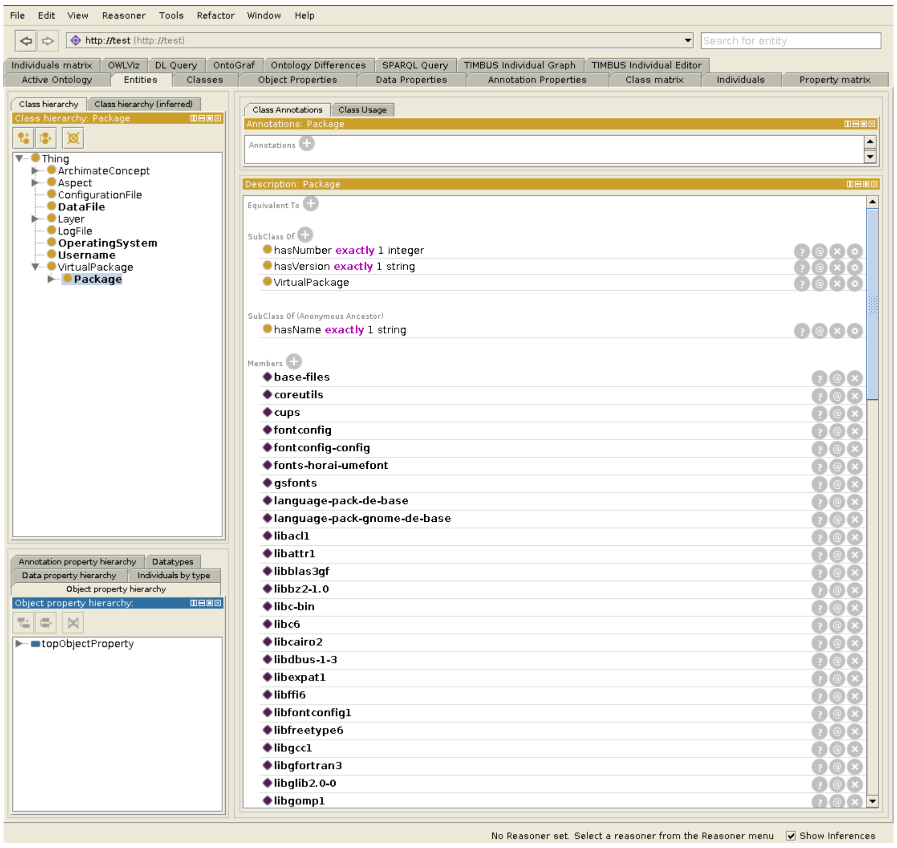
The resulting model is the model that is used to build the target system.

4.3.4 Refiners

To improve the conciseness of the model several strategies are implemented, which are described in the following sections. Refiners are used to remove information that is not needed for the process to execute and irrelevant for describing the process environment, but also for restructuring the model so that readability is improved.

BlacklistRefiner

Some files are specific to a certain system or system state and should not be migrated. If such files are e.g. part of the respective GNU/Linux distribution this is no issue because they are recreated when building the target system. Examples of such files/directories are shown in Table 4.1. This list represents the default of files and directories for the blacklist. All of those files have been found in an `strace` log file during evaluation. Depending on the scenario this list will need some adaptations, which is possible by creating and populating a `blacklist.txt` file one entry per line, which is loaded if available instead of the default blacklist.



File Edit View Reasoner Tools Refactor Window Help
http://test (http://test)
Search for entity
Individuals matrix OWLViz DL Query OntoGraf Ontology Differences SPARQL Query TIMBUS Individual Graph TIMBUS Individual Editor
Active Ontology Entities Classes Object Properties Data Properties Annotation Properties Class matrix Individuals Property matrix
Class hierarchy Class hierarchy (inferred)
Class hierarchy: Package
Annotation property hierarchy Datatypes
Data property hierarchy Individuals by type
Object property hierarchy
Object property hierarchy:
Class Annotations Class Usage
Annotations: Package
Annotations
Description: Package
Equivalent To
SubClass Of
SubClass Of (Anonymous Ancestor)
Members
No Reasoner set. Select a reasoner from the Reasoner menu
Show Inferences

```
Thing
  ArchimateConcept
  Aspect
  ConfigurationFile
  DataFile
  Layer
  LogFile
  OperatingSystem
  Username
  VirtualPackage
    Package
```

```
topObjectProperty
```

SubClass Of
- hasNumber exactly 1 integer
- hasVersion exactly 1 string
- VirtualPackage

SubClass Of (Anonymous Ancestor)
- hasName exactly 1 string

Members
- base-files
- coreutils
- cups
- fontconfig
- fontconfig-config
- fonts-horai-umefont
- gsfonts
- language-pack-de-base
- language-pack-gnome-de-base
- libacl1
- libattr1
- libblas3gf
- libbz2-1.0
- libc-bin
- libc6
- libcairo2
- libdbus-1-3
- libexpat1
- libffi6
- libfontconfig1
- libfreetype6
- libgcc1
- libgfortran3
- libglib2.0-0
- libgomp1

Figure 4.4: A screenshot of a process model loaded in Protégé

PortableApplicationRefiner

Another refinement is to detect portable applications and only add their root directories, omitting all files in their (sub)directories in the model. Applications are considered portable if their data files are placed in one directory and its subdirectories of the filesystem. They are assumed to have been installed manually i.e. by downloading and extracting a compressed file. Portable applications may place configuration files outside the applications root directory, but in general will run on different systems by just copying the root directory. Examples are the Taverna workflow software, the IntelliJ IDEA IDE[14], and CDE amongst many others. As name for the portable application the applications root directory is used.

There is no popular common way of detecting portable applications by just analyzing a file system. A simple approach is to check for executable files and consider the path where those are found as root directories of portable application installations. Only executables that are not part of any package qualify to define a portable application. This method is reflected by the `PortableApplicationRefiner`. Another possibility would be to compare lookup files

[14] `http://www.jetbrains.com/idea/`, accessed 2014-03-08

file/directory	description
`/etc/passwd`	Contains details of user accounts [42]. Migrating this file would overwrite user account information. Because user accounts are coupled to a home directory they are created using system tools and not by copying configuration files.
`/etc/resolv.conf`	Contains DNS specific settings, may be managed by the `Net/workManager` and therefore should not be edited [74].
`/etc/hosts`	Contains network settings, i.e. the identification of the local system in the network [74], which differs in the target system and should not be edited.
`/proc/`	This directory contains information about running processes [74].
`/sys/`	This directory contains information about devices etc. [42].
`/etc/timezone`, `/etc/localtime`	These files are generated automatically and do not need to be migrated [42].
`/etc/ld.so.cache`	Caches the location of shared libraries of a system [106].
`/etc/nsswitch.conf`	Contains information about sources for name resolution [42].

Table 4.1: Examples for files and directories that should not be migrated.

of potential portable applications in a software database, like the NSRL [64]. Because this adds further dependencies it is not implemented in the prototype.

Besides causing better readability of the model, this refinement has the advantage that some files that are part of the application, but have not been used during the reference execution(s) of the process, are not omitted. So it reduces the likelihood that future adaption of the process on the target system fails because of missing resources.

Nevertheless such refinements are not always desired. The listing of files that a portable application consists of is lost, which affects the level of detail of the model and thereby also future maintenance of the process. A compromise would be to add both portable application and its files and also state the relation between those in the model. To be consistent then also packages and its files should be described this way. But the immense count of individual files in this case would affect the readability of the model.

Because information about the software is not available in a standard way for software installed without using the package manager, the model does not contain the exact name and version of the portable application, but rather the name of the directory. Special care has to be taken for directories like `/usr/bin/` which holds multiple binaries and cannot be resolved to one specific portable application. Because of this in the prototype only portable applications installed in the `/home/` directory are taken into account. Besides handling portable applications outside the `/home/` directory, in future versions it should also be considered that the executable of a portable application is not necessarily in the root directory but may also reside in a subdirectory of the application.

Further refinement

Packages that other packages depend on could be omitted as they would be resolved by the package manager. This has not been implemented because the package manager could resolve a different version of the package than the one installed on the source system. Indirect dependencies also are kept in the model to preserve traceability, i.e. depending on which set of indirect dependencies are used, the behavior of the process could change.

As services may get called multiple times during process execution, not each request should be mapped to the model, but only one for each service. As web services may provide different URIs for their services it is not easy to determine externally which web service handles the individual requests. In this prototype the idea that child URIs are likely to be part of one service is utilized. The implementation sorts service requests, and only adds those where the path is not contained in the path of an already added service.

4.3.5 Usage

Listing 4.7 shows the usage of the capture component. The capture script (-s) refers to the replaceable operating system specific extraction script (c.f. Section 4.1). A capture script for Debian is used by default, but can easily be replaced by passing the path to a custom script. The capture script is executed and runs and monitors the process execution script (-a). The capture script creates text files for the extracted information, separated in files, packages, services, and system settings. The filenames are passed to the capture component using the -f, -p, -r, and -y arguments respectively. The default values for those arguments can be taken when using the default capture script. Existing ontologies can be used as base model (-i). Finally, the path to the resulting ontology can be passed using the -o argument.

Listing 4.7: Usage instructions for the Capture component

```
usage: CaptureCLI
 -a <arg> capture script arguments (mandatory), defaults to:
          ''
 -i <arg> input ontology path, defaults to: ''
 -o <arg> resulting ontology path, defaults to:
          'data/process.owl'
 -s <arg> capture script path, defaults to:
          'scripts/main.sh'
 -f <arg> files file path, defaults to:
          'data/files.txt'
 -p <arg> packages file path, defaults to:
          'data/packages.txt'
 -r <arg> services file path, defaults to:
          'data/services.txt'
 -y <arg> system settings file path, defaults to:
          'data/system_settings.txt'
```

4.3.6 Summary

This section showed details about the prototype implementation of the Capture component. While in Linux package manager tools provide valuable information about files and packages there are several limitations in detecting services. For remote services this is because access is limited and only the public interface can be queried. The communication can be observed and thereby a reasonable assumption about the content can be made, e.g. SOAP in the prototype. In future work additional information can be detected. As an example the migration options of web services are limited. Possible approaches could be to record request/replay pairs and setup a mock-up web service that uses the recorded data to replay for known requests [66]. But such issues are not the focus of this work. This also is not necessary for some scenarios that just want to reuse these existing web services. But what can be done is documenting the web service requests, and adding them to the model. To get a precise model also various refinements can be made that strip irrelevant and restructure existing information. The goal of the model is not only to provide a template for rebuilding the environment for a process, but also to serve as documentation.

4.4 Adapt

The Adapt module has not been implemented in the context of the PMF, but a service with a similar scope (preservation identifier[15]) has been implemented within the TIMBUS project. The preservation identifier helps to identify alternative implementations for components that are at risk. It provides a web service, which takes a TIMBUS compliant context model [96] of a process and a list of components at risk that should be replaced by alternative implementations as input. The preservation identifier implements a knowledgebase that specifies artifacts and their relations. This database is used to identify alternatives for original process artifacts. The services returns a list of potential replacements for each risk. Types of artifacts that are processed are packages, file formats, and software tools. In the context of the PMF this service can manually be invoked. Demo instances can be found for both the preservation identifier service[16] as well as for a web client[17]. An exemplary Java client is available in the corresponding repository of the TIMBUS project[18]. The service returns a list of potential alternatives, the selection of the most suitable solution needs to be done manually by experts.

The knowledgebase of the preservation identifier is populated with data from Freebase and Pronom (c.f. Section 3.3.1). It can directly be queried using the web interface[19] that is available online for demonstration purpose. The capabilities of the preservation identifier can quickly be evaluated this way.

[15] `https://opensourceprojects.eu/p/timbus/dpes/preservation-identifier/`, accessed 2014-07-23

[16] `http://kronos.ifs.tuwien.ac.at:8080/pi/preservationIdentifier`, accessed 2014-07-23

[17] `http://kronos.ifs.tuwien.ac.at:8080/pi-client/`, accessed 2014-07-23

[18] `https://opensourceprojects.eu/p/timbus/dpes/preservation-identifier/client/`, accessed 2014-07-23

[19] `http://kronos.ifs.tuwien.ac.at:8080/kbserver/`, accessed 2014-07-23

Other manual steps of the Adapt module are to adapt the process model to remove or add artifacts, so that the process environment can be described in a more suitable way. An example for such modification is shown in Section 5.2, where individual files are removed and the parent directory is added instead.

In future versions of the PMF a connector to the preservation identifier can be implemented to mitigate the manual steps in the adapt module. Such a connector could handle the connection to the preservation identifier in a way so that it allows future extension to support alternative services. It could use expert input or a rule set for batch changes to specify resources at risk. The connector would forward the input data, and present the alternatives to the expert, which can select a suitable alternative.

4.5 Build

This section shows how the Build component of the PMF is implemented. The Build module is used to create a virtual machine from the specification given by a process model instance (c.f. Section 3.4). The implemented is structured in two packages that conform to the subcomponents `ConfigBuilder` and `MachineBuilder`.

One possibility to implement this module would be to issue commands to install packages and transfer files directly. But it would be better to have a platform independent description that can be passed to a tool which handles the provisioning. Platform independence is supported by a declarative description of the configuration, where no system specific instructions need to be specified. An open-source configuration management software tool that supports setup and configuration of systems (provisioning) using declarative configuration is `Puppet`[20]. The advantages of using Puppet are that the format of the configuration file is used in the industry and well understood. It also serves as documentation for a closer look at the system configuration. For creating the virtual machine `Vagrant` is used in the prototype. It serves as wrapper for VirtualBox and supports provisioning of the virtual machine.

Following these considerations the effort to implement this module is reduced to extract the configuration file from the model and use Vagrant to create and provision the virtual machine. Those steps are implemented in the submodules ConfigBuilder and MachineBuilder subsequently. See Figure 4.5 for an overview of the implementation. The `BuildCLI` is the main entry point of the module, which forwards the arguments to the `App`. The submodules `ConfigBuilder` and `MachineBuilder` are invoked by the `App`. The result of the ConfigBuilder describes the system state, and is used by the MachineBuilder for provisioning the target system. The result of the MachineBuilder is a configuration for the virtual machine itself, e.g. amount of memory, and the created virtual machine.

The build component needs to be started with at minimum the same access rights as the process that has been identified by the capture module. This is needed e.g. to ensure that all necessary files can be read and transferred to the target system.

[20]`https://puppetlabs.com/`, accessed 2013-10-04

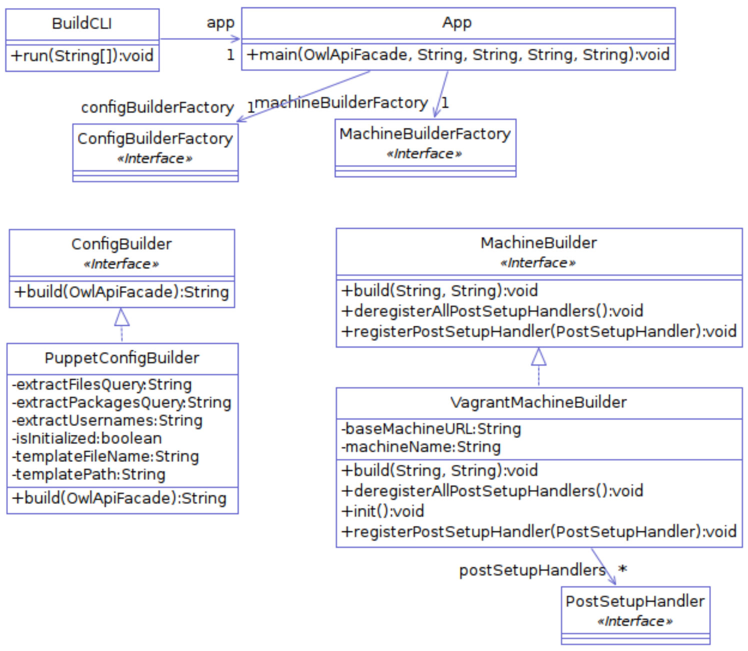


Figure 4.5: Class diagram of the build module

4.5.1 ConfigBuilder

From the process environment model the ConfigBuilder creates a configuration file for Puppet. This configuration file is used by the MachineBuilder to provision the system.

The ConfigBuilder is implemented for Puppet. It uses a configuration file template (Puppet manifest template) which is populated using `Apache Velocity`[21] [2] with packages, files and user accounts. A custom template can be passed to the build component. This allows additional provisioning that is independent from the model, e.g. to satisfy organizational guidelines. Web applications can support creating Puppet manifests, e.g. PuPHPet[22].

Because directories are not automatically created in Puppet [82] a custom script creates directories. This is not platform independent and should be replaced in future versions of Puppet that might allow to create the parent directory automatically. Users and their home directories are also added in the Puppet configuration file. `Apt-update` is called to ensure the latest information about packages of the repositories is available on the target system. Puppet allows to influence the order of execution of the provisioning. This is used to execute the script that creates directories before copying the files inside those directories. It also is used to execute `apt-update` before installing any package.

As permissions are not stored in the process model by the prototype for the time being, the user that executes the process is assigned as owner of all migrated files. This means that this

[21]`http://velocity.apache.org/`, accessed 2013-10-07
[22]`https://puphpet.com/`, accessed 2014-7-24

user is able to execute the process on the target system. Other users that might be able to execute the process are not considered.

The following list shows the tasks that are specified in the Puppet script and its order of execution:

1. Update package information for the configured repositories, i.e. the standard repositories of the distribution. This is done before installing any package to ensure that the latest packages are known to the package manager. This is done by executing a command:

   ```
   exec { 'apt-update':
     command => 'apt-get update'
   }
   ```

2. Create the users, including their home directories. This is done before copying data files, to ensure that the home directory does exist before files are copied into it. The following snippet is used to create a user:

   ```
   user { 'pmf':
     ensure    => 'present',
     groups => ['sudo'],
     managehome => true,
     home => '/home/pmf',
     shell => '/bin/bash',
     password => '$6$Dvq08/xE6.5Op$Ad/SGpPMv8erTA.CmyYyB7/...'
   }
   ```

3. Install the packages. This is done before transferring data files, to ensure that files that are part of packages but have been modified locally are not overwritten during installing the packages. An example to install multiple files using variables is shown in the following:

   ```
   $pkgs = ['coreutils', 'libc6','openjdk-7-jre-headless']
   package { $pkgs: ensure => present }
   ```

4. Transfer the data files from the source system to the target system. Both files and directories can be transferred as shown in the following:

   ```
   file { '/home/pmf/process/run.sh':
     source => 'file:///tmp/host/home/pmf/process/run.sh',
     force => true,
     recurse => true,
     owner => 'pmf'
   }
   ```

4.5.2 MachineBuilder

The MachineBuilder is used to create and configure the virtual machine using the configuration file created by the ConfigBuilder. The MachineBuilder has been implemented for Vagrant, a tool for managing virtual machines (c.f. Section 2.4). This has been done by utilizing the `Vagrant-Binding API`[23] that has been slightly customized[24] to support the package command of Vagrant that is used to export the virtual machine, and also allowing setting additional configuration for Vagrant that is not supported by the API natively. `PostSetupHandlers` can be registered on the MachineBuilder to allow executing commands on the new created and already provisioned machine using SSH. This is useful to execute tasks that are not specified in the Puppet configuration file. It can be used to extend the PMF on the level of source code, but is a step in the direction of passing scripts to the PMF that are executed automatically after provisioning. Vagrant uses *base boxes* as starting point for virtual machines. Using base boxes removes the necessity to setup and configure the basic installation of the operating system. Custom base boxes can be generated from existing virtual machines [79]. There are several base boxes publicly available e.g. from Vagrantbox.es[25]. The prototype uses Debian 7 as default, but can be provided with a specific base box. Boxes are `tar` files that include the hard disk image in the VMDK format, the system configuration of the virtual machine as OVF file, and some Vagrant specific settings [40]. Therefore boxes can also be extracted and used with VirtualBox directly. The MachineBuilder generates the resulting virtual machine as Vagrant box, and also provides the Vagrant configuration file as output. By applying the Puppet manifest which has been created by the ConfigBuilder on the base box, the process environment is built. An exemplary Vagrant configuration file is shown in Listing 4.8. It shows how base boxes are specified, that shared folders are created to be able to transfer files from the source system, and how it is possible to customize the target virtual machine. This example also specifies the usage of Puppet for provisioning, and refers to the manifest created by the ConfigBuilder.

Listing 4.8: A Vagrant configuration file

```
Vagrant::Config.run do |config|
  config.vm.define :vm do |vm_config|
    vm_config.vm.box = "hashicorp/precise64"
    vm_config.vm.box_url = "https://vagrantcloud.com/hashicorp/
        precise64/version/2/provider/virtualbox.box"
    vm_config.vm.provision :puppet do |puppet|
      puppet.manifests_path = "puppet/manifests"
      puppet.manifest_file = "puppet.cnf"
    end
    vm_config.vm.share_folder "host", "/tmp/host", "/",
        :create => true
    vm_config.customize ["modifyvm", :id, "--memory", 1024]
  end
```

[23] `https://github.com/guigarage/vagrant-binding/`, accessed 2013-10-08
[24] `https://github.com/jbinder/vagrant-binding`, accessed 2013-10-08
[25] `http://www.vagrantbox.es/`, accessed 2013-10-28

```
end
```

4.5.3 Usage

The usage of the build component is shown in Listing 4.9. As mandatory argument it requires the process model (-i) as input, which provides all information to build the virtual system. To customize the virtual machine provisioning, optionally an adjusted Puppet manifest template can be passed (-t). This could be used i.e. to include steps during provisioning of the virtual system that are not part of the model. The location and name of the virtual machine can be customized with the -o and -n arguments respectively. Because the output of the build component not only includes the created virtual machine as Vagrant box, but also the Puppet and Vagrant configuration files, a directory needs to be provided as output path. The base box of the new virtual machine is specified using -b.

Listing 4.9: Usage instructions for the Build component

```
usage: BuildCLI
 -b <arg> base machine URL, see http://www.vagrantbox.es/,
          defaults to:
          'http://puppet-vagrant-boxes.puppetlabs.com/ubuntu
              -1310-x64-virtualbox-puppet.box'
 -i <arg> ontology path (mandatory), defaults to: ''
 -n <arg> virtual machine name, defaults to: 'vm'
 -o <arg> output path, for configuration data etc.,
          defaults to: 'vbox'
 -t <arg> Puppet manifest template path,
          usually not required, defaults to: ''
```

4.5.4 Summary

In this section the implementation of the build component has been described. It was discussed that using platform independent and declarative tools for provisioning and virtual machine creation and configuration an implementation is possible that in general is platform independent and is not limited to a specific virtual machine provider. But to be able to migrate all aspects of the model that are considered relevant for the process execution the introduction of operating specific tasks has been necessary.

For this component there also are some aspects that have been left for future development. One is the detection of suitable target base boxes. By analyzing the process environment model, conditions for potential target operating systems need to be established. Examples are that the supported processor architecture matches, and that all packages in the model are available. Based on this conditions target operating systems need to be suggested or one target system automatically selected. Because there are many different distributions of GNU/Linux, a mapping to compatible base boxes that are available online needs to be defined.

For applications that are not part of any package it may be possible to create a portable version of the package that can be migrated to different systems using application packaging tools like CDE. CDE has following requirements [37]:

- The kernel version should have the same major number because of system call interface changes which CDE might not be able to handle.
- Dynamic linked libraries compiled for newer kernel version because the usage of new features that are not available on the older kernel.
- A compatible architecture is required, otherwise the binaries will not be able to execute on the target platform.
- CDE does not capture processes that communicate using IPC, e.g. databases, so such processes need to be captured independently.

But as application packaging is used as fallback (c.f. Section 2.2) only this is acceptable.

4.6 Verification

Verification ensures that the target system is capable of successfully running the process. In doing so it also is checked that the process model accurately describes the process environment. Verification is executed manually in the current version of the PMF. There are several fundamental steps that are applied for a basic verification.

- The generated process model is inspected by an expert. The expert investigates the model with a focus on missing direct dependencies.
- The capture module is executed on the target system. The resulting process model is compared to the process model derived from the target system.
- The process is executed on the target system and on the source system. It is checked if the behavior and the results of the process match.

As described in Section 3.5, refinement of those fundamental steps is possible. If the verification fails, either the process model can be fixed manually, or the process execution script can be extended, i.e. to execute the process with different input, so that more paths are executed. In the latter case both the capture and the build module need to be executed again. When fixing the process model only the build module needs to be executed again.

4.7 Implementation details

This section addresses general information about the implementation and explains the reasoning for decisions regarding the technical stack and the architecture that is used in the prototype.

As the components operate with the ontology as data structure, as specified in Chapter 3, a library that provides common functionality for interacting with a process model has been developed. Out of a variety of frameworks for OWL [43] this library utilizes the Java Ontology API `The OWL API`[26], an open source Java API for OWL licensed under the LGPL. The OWL API is used in `Protégé` (c.f. Section 2.5) which ensures compatibility of the models generated in this prototype and Protégé. This allows manual editing (e.g. adapt step of the PMF) of the model. There also are several different reasoners for the OWL API [43]. For querying the model `Apache Jena`[27] is used, which supports SPARQL queries. Although there are others, Jena has been selected because it supports executing SPARQL queries without invoking a reasoner. This is important because of performance reasons. The common library has been developed as a wrapper on top of the OWL API and Jena. It eases the usage of the OWL API by providing a slightly higher level for interacting with the model.

Java is selected as main language in this prototype, not only because of the OWL API but also because of its platform independence and wide acceptance [7]. Exceptions are platform specific parts of the extractors. According to [15] `bash` is the 'de facto standard for shell scripting', and as such is used for the implementation of the operating system dependent parts of the extractors. Those extractors are meant to be platform specific but are replaceable by extractors suitable for another platform. Extractors are organized as functions which solve elemental tasks. This conforms to the UNIX philosophy [31] and provides advantages like improved reuse- and maintainability. The extractors are also used to encapsulate operating system specific functionality, and called as simple command in a higher level language, such that the extractors can easily be replaced by other scripts or tools.

The implementation is developed following the modular programming principle, supporting libraries are implemented independent from the framework modules. Google Guice [103] is used to assemble (sub)modules using dependency injection [27]. This supports the maintainability of the framework, by allowing to specify alternative implementations.

4.8 Summary

This chapter showed challenges when creating an implementation for the PMF. For Debian it is possible to retrieve the information about the process environment that is necessary to rebuild the system in general. Exceptions are local services which already may load data before the prototype has been executed and therefore are not identified completely. Main features are:

- The prototype uses both static and dynamic analysis to capture the process environment.
- The capture component supports creating a model of the process environment, which uses an open format so that it can be modified and read with third party tools.
- The operating system specific capture script can be replaced without changes to the prototype.
- The process is migrated into a dedicated system where only dependencies are deployed that are required to execute the process.

[26]`http://owlapi.sourceforge.net/`, accessed 2013-10-10
[27]`http://jena.apache.org/`, accessed 2013-10-10

- The prototype can be embedded in third-party scripts using the non-interactive commandline interface.

Except the implementation of the monitoring scripts and partly the Puppet configuration files the prototype is platform independent. The default implementation for Debian GNU/Linux of the monitoring scripts can be replaced by custom scripts for other operating systems.

There are also various limitations, some of them addressed and suggested for improvement of the prototype in future versions, others only affect specific scenarios. Main limitations are:

- Local services are detected but not migrated automatically. They can be migrated separately at the current state of the prototype. Services that can be started within the process execution script also can be included therein, so that separate migration is not necessary.
- The monitoring scripts have been implemented specifically for Debian.
- The operating system needs to be selected manually when building the target system.
- Building the target system is only possible from the source system as no resources are stored temporarily by the prototype.
- The adapt and verification components provide no automated usage.

The usage of the OWL API has introduced some overhead because the API is on a quite low level of abstraction. OWL as format for the model is useful, because there is tool support to view and modify OWL models and the format is suitable for documentation. The model can directly be stored as file, so the export from a database can be omitted.

Future versions could improve some of the mentioned limitations. As described it is possible to implement monitoring scripts for other operating systems. Also, detection of a compatible operating system could be done, e.g. by creating a knowledge base. The capture component could copy all required resources to a data store, so that rebuilding the target system is possible without having access to the source system. Although a complete migration of external dependencies may not be possible without access to the hosting system there exists research about providing a capture/replay mock to partly substitute remote web services [66].

The tool is available online[28], where also usage information for the components and information about necessary dependencies to run the PMF is provided.

The next chapter shows that the prototype eases the migration of processes in shared systems to dedicated virtual systems in various scenarios.

[28]`http://www.ifs.tuwien.ac.at/dp/process/projects/pmf.html`, accessed 2014-09-17

CHAPTER 5

Evaluation

In this chapter the PMF is evaluated based on the prototype implementation described in Chapter 4. The framework is applied on two different use cases in order to check the accuracy and drawbacks of the tool.

By applying the prototype on an existing system and providing it with a command to execute the process, the expected outcome is a description of the environment and a virtual system that satisfies this description. The description contains the packages, files, and web services the process depends on, as well as other information about the environment that is relevant for successful process execution, i.e. the name of the operating system that is used, and the name of the user that executes the process. It is represented as OWL ontology as described in Section 4.2.

To address different challenges, different scenarios are used. The first is a music classification workflow, which is used to check the applicability of the framework to workflow engine based processes (c.f. Section 5.1). The second is a document generation workflow that is based on batch scripts and invokes a Windows tool for fetching data from a remote server (c.f. Section 5.2). For both scenarios local tools as well as web services are accessed.

In the focus of the PMF are processes that are deployed non-exclusively in a system together with other tools and services. The process execution is described in a machine interpretable format, e.g. in a script or as workflow. Possible formats are, for example, bash [15] for scripts and Taverna for workflows [68, 108]. Taverna is an open source workflow management system that is written in Java and licensed under the Lesser General Public License (LGPL) Version 2.1[1]. The two scenarios covered in this evaluation are fully scriptable and deterministic, so fulfill the requirements of the PMF (c.f. Chapter 3).

To be able to run the prototype the source system has to meet several requirements. VirtualBox 4.3 and JRE 1.7 or newer have to be installed. For the Oracle JRE it is sufficient to download the `tar.gz` package[2] extract it and refer to the `java` executable in the `bin` directory when executing the prototype. CDE should be available and executable (e.g. by downloading the

[1]`http://www.taverna.org.uk/`, accessed 2014-02-13

[2]`http://www.oracle.com/technetwork/java/javase/downloads/`, accessed 2014-01-08

standalone binary and adding its containing directory to the `$PATH` environment variable), but this is optional. The binaries of the prototype need to be available. Internet access is required to be able to resolve all ontologies that are referenced by the model and to fetch the base box for the build module. The amount of required available disk space depends on the size of the base image and the number and size of files that are involved in executing the process.

Both the source system of the music genre classification and the LNEC (Laboratório Nacional de Engenharia Civil) scenario satisfy these constraints.

5.1 Musical genre classification

The musical genre classification process represents an e-Science experiment, that is performed by researchers locally on their computers. The aim of the experiment is to validate a music classification method. The process is set in the information retrieval and machine learning domain, where experiments are difficult to re-evaluate and repeat, for reasons that include incomplete documentation and complex setups. In this setting the aim of the PMF is to document the software environment of the experiment, and to create a snapshot of the environment, so that the process can easily be re-evaluated and validated. More details about the scenario can be found in [97].

The musical genre classification process [63] describes the process of validating a method for automated music objects classification. The steps in the musical genre classification process are the following. First, the test data, i.e. MP3 files, is fetched from a web server. The MP3 files are converted to Base64 and passed to a REST service, which performs feature extraction. In the following step the ground truth is fetched, which is a mapping of the music objects to a category label like genre. The features of the music object are combined with the ground truth to WEKA ARFF files using the `SOMToolbox`[3]. The WEKA ARFF files and a set of learning parameters are finally passed to `Weka`[4] which determines the classification accuracy. Further information about the process can be found in [97] and [63]. Figure 5.1 shows the music workflow in Taverna.

The motivation for the migration is to extract the classification process from the source system and deploy it in a virtual system that runs Debian, so it can be shared with stakeholders that want to execute the process on their own machines.

The process has already been modeled as ontology[5] in the context of the TIMBUS project with the scope of performing a holistic preservation of the process, including the documentation of the highlevel steps. The technical layer of this ontology misses information that is necessary to be able to recreate the environment, i.e. the source for libraries and tools that have not been fetched from standard package repositories, and therefore could not be reused in this context.

The process is implemented as Taverna workflow [63]. It is executed using the Taverna Com-

[3]`http://www.ifs.tuwien.ac.at/dm/somtoolbox/`, accessed 2013-12-03

[4]`http://www.cs.waikato.ac.nz/ml/weka/`, accessed 2013-12-03

[5]`http://timbus.teco.edu/ontologies/Scenarios/MusicClassificationDSOs.owl`, accessed 2014-04-08

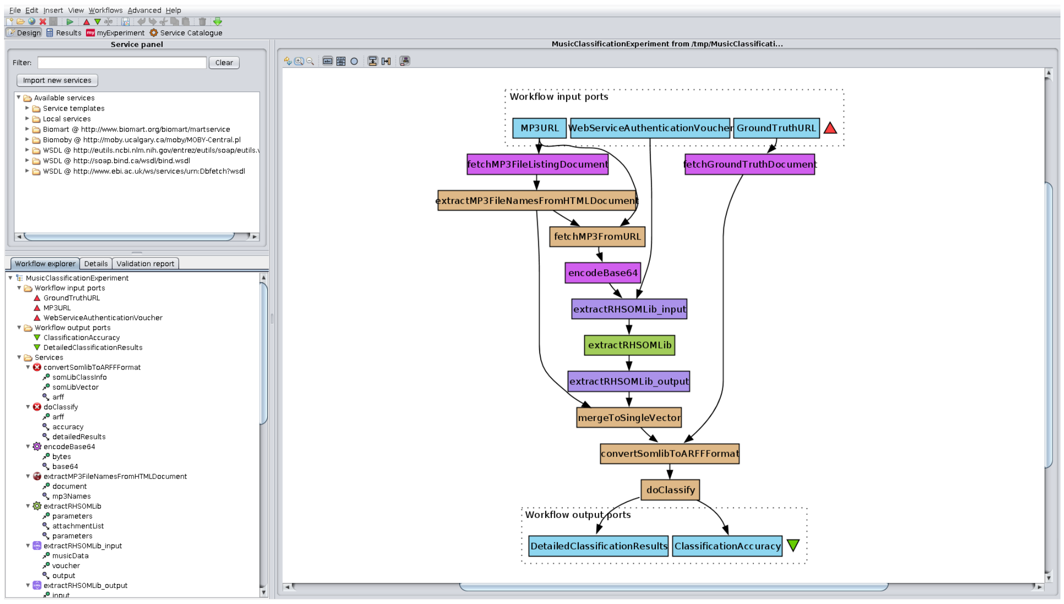


Figure 5.1: The music workflow in Taverna

mand Line Tool[6]. For GNU/Linux Taverna is available as zip package, which requires at least the JRE 1.6 to be installed on the system. The logic of the workflow is implemented in Java, which is embedded in the Taverna workflow description file (`MusicClassification_WSDL.t2flow`). It makes use of SOMToolbox (`somtoolbox_full.jar`) and Weka (`weka-3.6.6.jar`) libraries for format conversion and classification. For the REST web services that is used for audio feature extraction no source is available. Furthermore, the MP3 files and the ground truth document that are used by default are stored on web servers. A custom location for the MP3 files and the ground truth can be specified if required. The source system is Linux Mint 15 (Olivia), which is based on Ubuntu 13.04 (Raring Ringtail), which can be found out by checking the `/etc/*-release` files. Ubuntu 13.03 in turn is based on Debian 7 (wheezy).

As a first step, the expert analyses the target system, to be able to verify the process model, but also to create a suitable process execution script. The minimal, i.e. without any indirect dependencies, expected content of the model generated by the capture module is shown in Table 5.1. All of those dependencies should be documented in the model, not all of them can be migrated to the target system, in this case because they are not located on the system that is migrated (e.g. external services). The process execution script makes Taverna available in the `$PATH` environment variable, so that executing the workflow can be done without absolute path to the Taverna executable. Several arguments are passed to the workflow, which is executed using the command-line tool of Taverna. The script is shown in Listing 5.1.

To be able to run the PMF, it needs to be deployed on the source system. This requires the PMF to be copied on the target system, and packages like openjdk-7-jre, cde, strace, dlocate,

[6]`http://www.taverna.org.uk/documentation/taverna-2-x/command-line-tool/`, accessed 2013-10-28

resource name	resource type	migration to target
openjdk-7-jre	Package	yes
Taverna	Package	yes
somtoolbox_full.jar	DataFile	yes
weka-3.6.6.jar	DataFile	yes
MusicClassification_WSDL.t2flow	DataFile	yes
/usr/bin/executeworkflow.sh	DataFile	yes
mp3 provider	HTTP	no
ground truth provider	HTTP	no
feature extraction	SOAP	no
mcuser	Username	yes
Linux Mint 15	OperatingSystem	-

Table 5.1: The minimal expected results of the prototype.

apt-file, and debsums to be installed.

Invoking the capture module using the capture script argument with the process execution script produces the ontology that is shown in Table 5.2.

Listing 5.1: The script to start the music genre classification process)

```
#!/bin/bash

PATH=$PATH:/home/pmf/apps/taverna-workbench-2.4.0

executeworkflow.sh MusicClassification_WSDL.t2flow
  -inputvalue MP3URL "http://kronos.ifs.tuwien.ac.at/
                      timbus/musicProcess/music-10songs/"
  -inputvalue GroundTruthURL "http://kronos.ifs.tuwien.ac.at/
                               timbus/musicProcess/genres.txt"
  -inputvalue WebServiceAuthenticationVoucher "timbusVoucher"
```

An excerpt of the generated ontolgoy is shown in Listing 5.2. Line 10 to 16 show, how other ontologies can be referenced. Line 19/20 shows the definition of a relation between individuals that is imported from an imported ontology. In line 22/23 a new class is defined. Finally, in line 25 to 33 an example of the individuals is shown. Other types of individuals have been described in Section 4.3.1. This ontology serves as documentation of the process environment, the actual list of artifacts and system settings can be found in Table 5.2.

Listing 5.2: An excerpt of the resulting process model

```
1  <?xml version="1.0"?>
2  <rdf:RDF xmlns="http://test#"
3       xml:base="http://test"
4       xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
```

```
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:DIO="http://timbus.teco.edu/ontologies/DIO.owl#">
    <owl:Ontology rdf:about="http://test">
       <owl:imports rdf:resource=
          "http://timbus.teco.edu/ontologies/DIO.owl"/>
       <owl:imports rdf:resource=
          "http://timbus.teco.edu/ontologies/DSOs/CUDF.owl"/>
       <owl:imports rdf:resource=
          "http://timbus.teco.edu/ontologies/DSOs/software.owl"
       />
...
    </owl:Ontology>
...
    <owl:ObjectProperty rdf:about=
       "http://timbus.teco.edu/ontologies/DIO.owl#association"/>
...
    <owl:Class rdf:about=
       "http://localhost/RemoteServices.owl#HTTP"/>
...
    <owl:NamedIndividual rdf:about="http://test#zlib1g">
       <rdf:type rdf:resource=
          "http://timbus.teco.edu/ontologies/DSOs/
             CUDF.owl#Package"/>
       <rdfs:label rdf:datatype=
          "http://www.w3.org/2001/XMLSchema#string">
             zlib1g
       </rdfs:label>
    </owl:NamedIndividual>
</rdf:RDF>
```

By manually analyzing the generated ontology it can be seen that the remote services have been detected, as well as more than expected packages, data and configuration files. This is because the generated ontology also contains indirect dependencies.

There have been about 30 packages detected, which mostly are generic libraries that have been involved in the execution of the process because they are used by tools of the process or by other libraries (indirect dependencies), i.e. libc6, base-files, libglib2.0-0, and the like. Indirect dependencies would not be needed to be stated explicitly in the model, as they can be resolved by the package manager of the target system. Most of them should be available for different distributions, but such indirect dependencies might be a limiting factor for cross distribution compatibility. For example, it might be the case that packages on one distribution use different dependencies than in another distributions, and indirect dependencies of the source system are not available on the target system. On the other hand, changes in indirect dependencies

resource name	resource type	expected in model	migrated to target
kronos.ifs.tuwien.ac.at/timbus/ musicProcess/genres.txt	HTTP	yes	no
kronos.ifs.tuwien.ac.at/ timbus/musicProcess/music-10songs/	HTTP	yes	no
kronos.ifs.tuwien.ac.at:8080/ fexWS/featureExtraction	SOAP	yes	no
base-files	Package	no	yes
libc6	Package	no	yes
libffi6	Package	no	no
29 more lib packages . . .*	*Package*	*no*	*yes*
openjdk-7-jre-headless	Package	yes	yes
openjdk-7-jre-lib	Package	no	yes
tzdata-java	Package	no	yes
zlib1g	Package	no	yes
mcuser	Username	yes	yes
/etc/issue	ConfigurationFile	no	yes
/etc/issue.net	ConfigurationFile	no	yes
/etc/lsb-release	ConfigurationFile	no	yes
/etc/update-motd.d/10-help-text	ConfigurationFile	no	yes
/home/mcuser/.taverna-2.4.0/lib/somtoolbox_full.jar	DataFile	yes	yes
/home/mcuser/.taverna-2.4.0/lib/weka-3.6.6.jar	DataFile	yes	yes
/home/mcuser/.taverna-2.4.0/logs/derby.log	DataFile	no	yes
7 more files in /home/mcuser/ .taverna-2.4.0/...	*DataFile*	*no*	*yes*
/home/mcuser/apps/taverna-workbench-2.4.0/	DataFile	yes	yes
/home/mcuser/work/ eval_taverna_mc/	DataFile	yes	yes
/usr/bin/executeworkflow.sh	DataFile	yes	yes
/usr/lib/jvm/java-7-openjdk-amd64/jre/lib/ext/h2-1.3.170.jar	DataFile	no	yes
/usr/lib/jvm/java-7-openjdk-amd64/jre/lib/jaxp.properties	DataFile	no	yes
Linux Mint 15 Olivia	OperatingSystem	yes	-

Table 5.2: The actual results of the process virtualization tool.

also could cause a change in the behavior of the process. To exclude incompatible indirect dependencies as potential source of migration issues, indirect dependencies are kept in the model generated by the PMF.

As Java runtime environment openjdk-7-jre-headless has been identified, which is reasonable as there was no GUI involved in the execution of the process.

What has not been detected as package is Taverna, which is because it is not available in Debians package repositories and has been downloaded and extracted manually without involving the package manager. The individual files have been detected though. By applying the PortableApplicationRefiner only the root directory of Taverna has been added to the model. This replacement of the individual Taverna files by its root directory reduced the model by about 1000 individuals.

As for data files, all of the expected files have been detected. Some unexpected files that could not be mapped to a package have been added to the model. Examples for such are files in the `$HOME/.taverna` directory, which is created by Taverna and includes i.e. configuration files, log files, and additional dependencies (JAR files). Because also the configuration files of Taverna are not known to the package manager those have been identified as plain data files instead of configuration files in the model. Like Taverna, also the directory containing the workflow file and the process execution script has been added as portable application because of the PortableApplicationRefiner. This was not necessary and somewhat obfuscates the data files that are read directly by the process, but did not add much of an overhead in terms of additionally migrated data files. On the other hand the files that are located in the same directory as the execution script probably also might be relevant to the process. In future versions the usage information of artifacts should be identified, i.e. the dependencies between resources, which would improve the expressiveness of the model.

It was not expected that any configuration files are to be migrated, yet the capture module detected four files, as can be seen in Table 5.2. None of them contains information that is relevant to the process though, so these files could have been omitted. An example for such is `/etc/issue` which contains description about the distribution. Furthermore two files in the `/usr/lib/` directory have been detected, `h2-1.3.170.jar` a library of the H2 database[7] [26] and `jaxp.properties`, a configuration file of the JAXP API [17]. Table 5.2 also shows that the user and the operating system have successfully been detected.

The process model reveals that several operating system processes have been spawned during execution of the music genre classification process. Some of them perform the communication with web services, for example there are processes that retrieve the individual MP3 files. This can be seen in Figure 5.2, where the relations of a process have been expanded.

The build module was instructed to use Debian 7[8] as base system. It was able to transfer all packages except libffi6 [54], which could not be installed because it is not available for Debian 7 in the official repository, as can be seen e.g. in the database of Debian packages[9]. Libffi is used by the OpenJDK [33], and is not available in this version in the official repositories until Debian 8 (jessie). In earlier versions of Debian earlier versions of libffi are used. As libffi is an

[7]`http://mvnrepository.com/artifact/com.h2database/h2/1.3.170`, accessed 2013-12-17
[8]`https://www.dropbox.com/s/23gupgb0xompvkm/Wheezy64.box?dl=1`, accessed 2014-08-01
[9]`http://packages.debian.org/`, accessed 2013-12-11

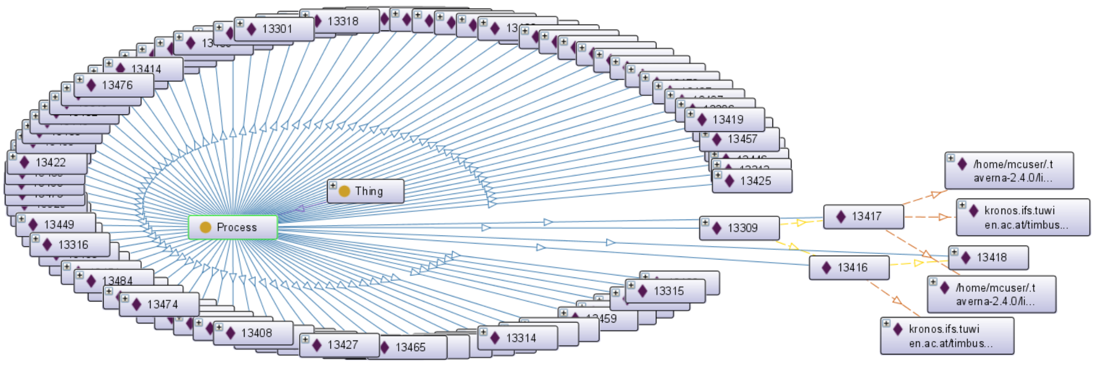


Figure 5.2: An excerpt of the spawned operating system processes and their dependencies

indirect dependency, it does not need to explicitly be installed, but is installed as dependency of another package in the version that is available on the system in any case. To avoid problems with missing packages it would be better to also store the source repository of the packages in the model, and to check during the build step of the target system if these packages are available in the stored source repository or the repositories of the target system. If not, suitable alternatives should be suggested. The other packages could be installed. The user was created and logging in on the target system was possible. Also the files have been transferred to the target system.

As a first step of the verification the process model was analyzed. Although the type of Taverna that was expected by the expert (package) did not match the actual result of the PMF (portable application), this difference was caused by an inaccurate manual inspection of the process environment and not an issue with the capture model. Then the PMF was executed on the target system. For this it was necessary to deploy the capture component of the PMF on the target system. By comparing the resulting process model with the initial process model, it was shown that the target system contains all mandatory elements of the process model. There were operating system specific files missing in the model of the target process environment, namely `/etc/lsb-release` and `/etc/update-motd.d/10-help-text`. As stated above, libffi5 has been installed on the target system instead of libffi6, and this difference is also shown when comparing the process models. Those minor differences did not effect the behavior of the process, and all other elements where the same in both models. As a last step, the process was executed an the target system, and the result compared to the result when running the process on the source system. Running the process on the target system resulted in the same output as running the process on the source system. There were no issues found during process execution on the target system because of the mismatch of libffi versions.

The duration of migrations depends on various parameters, like:

- the familiarity of the expert that performs the manual analyis with the process, i.e. for verifying the process and creating the process execution script
- the technical requirements of the process, the amount of used resources increases the duration of the manual analysis

- the performance of the source system which influences the runtime of the capture and the build module, the latter especially if a large amount of files or large files need to be transferred

- the speed of the Internet connection for downloading packages and the base box

- if the base box already is cached locally and does not need to be downloaded anymore

The time required for the described manual tasks, i.e. manual analysis of the direct dependencies for the verification, creation of the process execution script, and deploying the prototype on the source system, can be expected to take one to two hours, depending on the experience and familiarity of the expert to the process. Running the capture component took about 12 minutes, running the build component took about 13 minutes. The evaluation was performed on a system with an Intel(R) Core(TM) i5-2500 CPU @ 3.30 GHz, 8 GB memory, and a download speed of about 2 Mbit/s. The base box of the target system was already available locally. The size of the hard disk of the target system is about 2.3 GB. The source system had various other processes and data deployed and used about 575 GB. The number of files on the target system is 68756, compared to 3073130 on the source system. On the source system there are 2501 packages installed, while on the target system there are only 483.

It can be seen that cloning approaches would lead to a significant overhead in storage and complexity of the system. Advantages of cloning the system are the easier migration process, and that no resources can be overlooked. For file-based migration this overhead is not given, but for the file-based migration process more manual work is necessary. It is needed to manually determine which packages are involved for executing the process, and which user executes the process. The user has to be created on the target system, and the packages need to be installed thereon. Creating a portable package of a process is very simple with CDE:

```
cde process/run.sh
```

The resulting `cde-package` needs to be copied manually to a new target virtual machine. For executing the process it is necessary to change to the process directory and call the process execution script using CDE:

```
cd cde-package/cde-root/home/pmf/work/eval_taverna_mc
../../../../../cde-exec process/run.sh
```

The CDE package only contains 1203 files in 291 MB. The drawback of CDE is that the packages that have been included in the portable application cannot be maintained by operating system tools. The package manager is not aware of any packages inside the CDE package. The deployment of a CDE package is a matter of copying the package to the target system. Running the process on the target system was successful when executing the process as the same user as on the source system. When using a different user the execution failed, because Taverna had issues locating the home directory of the user, and therefore also the additional libraries that are necessary to successfully run the process. Changing the user in the included list of environment variables of the package (`cde-package/cde.full-environment`) and in the filesystem of CDE (`cde-package/cde-root`) to match the user on the target system resolved this issue.

Future versions of the PMF could aim to migrate remote dependencies to the target system. This would be possible for the MP3 files and ground truth that are hosted on a remote web server in this case, because this data is not generated dynamically and is reflected by static files only.

5.2 LNEC

The second scenario is set in a civil engineering business process of LNEC[10], the National Laboratory for Civil Engineering (Laboratório Nacional de Engenharia Civil). LNEC monitors large civil constructions, like dams, for their structural safety [98]. Any failure could have fatal consequences, i.e. by threatening the environment and human life. The monitoring process, which this scenario is concerned with, has been simplified for this evaluation and is about fetching sensor data, performing transformations on the data, and visualizing the data, thereby producing a report as file. An excerpt of such a report is shown in Figure 5.3.

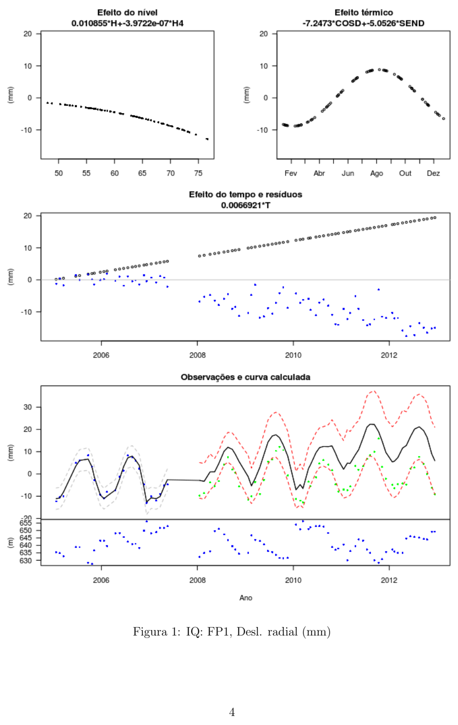


Figure 5.3: An excerpt of the report of the LNEC process

[10] `http://www.lnec.pt/`, accessed 2014-03-09

The LNEC process creates a PDF document that includes graphics generated from dynamic data at compile time (the report). The raw data is fetched from a web service. For getting the raw data a client is provided (LNEC client), which has been developed for the Windows platform only but can be called using `Wine`[11]. Wine can be used to run Windows applications in GNU/Linux [89]. The data is extracted with `unzip`. To avoid encoding issues `iconv` of the `libc6` package is used to encode the raw data to UTF8. From this data graphics are generated using `R`, a tool for statistical calculations and visualizations [50]. Finally, `pdflatex` is invoked (twice to ensure valid references in the document), which generates the PDF file. All tools except the LNEC client are available in Debian repositories.

Like in the previous scenario, a manual analysis of the process environment is described in the following. The execution is specified in a bash script (run script), from which the direct dependencies can be determined by considering the tool invocations. An exception is the web service that provides the raw input, for which no information except the used client can be inferred from the run script. The run script is used as process execution script, as no additional steps are necessary for running the process. It is shown in Listing 5.3.

Listing 5.3: The script that runs the LNEC process (lnec.sh)

```
#!/bin/bash
# lnec.sh

# fetch data
wine ClientAppNS/timbusClientNS.exe AP
mv ClientAppNS/data .
cd data
unzip data.zip

# fix encoding
iconv -f LATIN1 -t UTF-8 iq.r > iq_utf8.r

# generate references
R --vanilla < iq_utf8.r > IQout.txt

# create pdf
pdflatex iq.tex
pdflatex iq.tex
```

The service address of the web service can be found by analyzing the configuration file of the LNEC client, which uses Windows Communication Foundation (WCF) of the .NET Framework to access the service [14]. The result of the process can be determined by analyzing the last commands of the execution script, which call `pdflatex` on `iq.tex`. By consulting `man pdflatex` it can be seen that the process results in producing an `iq.pdf` file. Other data files that are accessed by the process are `data.zip` and `iqAll.zip`, which are downloaded

[11] `https://www.winehq.org`, accessed 2014-04-10

from the web service, `iq.r` which is contained in `data.zip`, as well as the intermediate results `iq_utf8.r` and `IQout.txt`. So other than the execution script (`lnec.sh`) no additional data files seem to be required by the process and are generated as part of the process. Both the `iq.r` and the `iq.tex` can include other resources, the r-file by stating `require` or `library` to include libraries, the tex file by stating `documentclass` or `usepackage` to include classes and packages respectively [83]. Such packages may be bundled with the tool, but also could be installed by users, in which case it would need to be considered also. In this case the `article.cls` is contained in the data directory of the process, which may be a customized version of the article class provided by the official tex packages and therefore needs to be migrated too. This information results in an minimal expected content of the model which is shown in Table 5.3.

resource name	resource type	migration to target
wine	Package	yes
unzip	Package	yes
libc6	Package	yes
R	Package	yes
texlive-latex	Package	yes
ClientAppNS	PortableApplication	yes
article.cls	DataFile	yes
lnec.sh	DataFile	yes
http://lacerta.lnec.pt/gestBarragens/GBServices/timbusService.svc	SOAP	no
timbus	Username	yes
Ubuntu 11.10	OperatingSystem	-

Table 5.3: The minimal expected results of the prototype.

The PMF was deployed on the source system manually preliminary to the migration of the process. Applying the capture module on the system results in a large number of packages and files. As in the previous scenario a lot of packages are indirect dependencies of invoked tools, i.e. there are 54 `lib*` libraries detected that have been involved in executing the process, including the expected `libc6` package. Some other involved general packages are `base-/files`, `coreutils`, `cups`, font and language related packages. `Unzip` has been detected, also `wine`, `texlive-latex-base`, and `R`. For latex several additional dependencies have been detected, which provide tools and libraries. The list of found packages is shown in Table 5.4.

As for portable applications, i.e. applications that have not been installed using the package manager but e.g. just downloaded and extracted, both the client (`/home/timbus/LNEC2/ClientAppNS`) and the execution script (`/home/timbus/LNEC2/`) have been detected. When considering data files, especially a lot of font files have been detected, like truetype fonts. Also, some tools place files that are not known to the package manager into the file system, especially `tex` (e.g. `/var/lib/texmf/*`), `wine` (e.g. `/home/timbus/.wine/*`) and

R (e.g. /etc/R/*). Another example is the directory /home/timbus/.local/share/, which mostly concerns icons for wine. Even though such files are not necessarily user generated and logically belong to certain packages, they are not detected as part of any package. One explanation for this is that files may have been downloaded after installation on demand during usage or initialization of the package, and are not part of the initial set of resources that is packaged and provided by the package repositories. The prototype is able to determine if files have been modified only for files that are tracked by the package manager. All files that are not tracked could have been modified by the user locally, and need to be transferred to the target system. The environment information has been extracted correctly (i.e. user and operating system). Details about the detected configuration and data files can be found in Table 5.5.

resource name	resource type	expected in model	migrated to target
base-files	Package	no	yes
coreutils	Package	no	yes
cups	Package	no	yes
fontconfig	Package	no	yes
fontconfig-config	Package	no	yes
fonts-horai-umefont	Package	no	yes
gsfonts	Package	no	yes
language-pack-de-base	Package	no	yes
language-pack-gnome-de-base	Package	no	yes
54 lib packages ...*	*Package*	*no*	*yes*
locales	Package	no	yes
r-base-core	Package	yes	yes
tex-common	Package	no	yes
texlive-base	Package	yes	yes
texlive-binaries	Package	yes	yes
texlive-common	Package	yes	yes
texlive-generic-recommended	Package	no	yes
texlive-latex-base	Package	no	yes
16 ttf- packages ...*	*Package*	*no*	*yes*
unzip	Package	yes	yes
wine1.5	Package	yes	yes
xfonts-mathml	Package	no	yes
zlib1g	Package	no	yes

Table 5.4: The actual results of the prototype (packages).

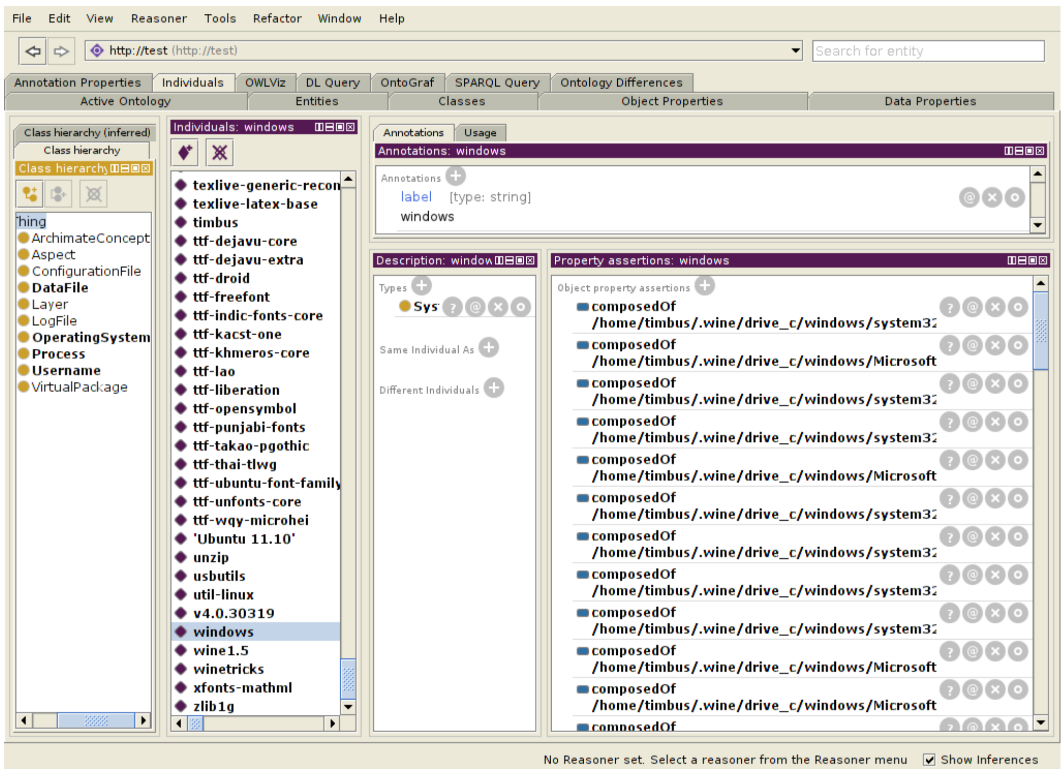
For redeploying the process environment Ubuntu 13.04[12] is used as base system. So instead of using a different distribution, in this scenario only a newer release of the same distribution has been used.

[12]https://dl.dropboxusercontent.com/u/4387941/vagrant-boxes/ubuntu-13.04-mini-i386.box, accessed 2014-08-01

resource name	resource type	expected in model	migrated to target
/etc/group	DataFile	yes	yes
/etc/R/Renviron	DataFile	yes	yes
/etc/texmf/texmf.cnf	DataFile	yes	yes
*26 files in /home/timbus/.local/share/**	*DataFile*	*yes*	*yes*
/home/timbus/.wine/.*	DataFile	no	yes
/home/timbus/.Xauthority	DataFile	no	yes
/home/timbus/LNEC2/	DataFile	yes	yes
/home/timbus/LNEC2/ ClientAppNS/	DataFile	yes	yes
/home/timbus/R/i686-pc-linux-gnu-library/2.13/xtable/.*	DataFile	no	yes
/usr/bin/../lib/libwine.so.1	DataFile	no	yes
/usr/lib/libblas.so.3gf	DataFile	no	yes
/usr/lib/liblapack.so.3gf	DataFile	no	yes
/usr/lib/locale/locale-archive	DataFile	no	yes
/usr/lib/R/etc//Renviron	DataFile	no	yes
/usr/local/share/texmf/ls-R	DataFile	no	yes
56 ttf-files in /usr/share/-fonts/truetype/	*DataFile*	*no*	*yes*
/usr/share/mime/globs	DataFile	no	yes
33 files in /var/cache/fontconfig	*DataFile*	*no*	*yes*
/var/lib/defoma/fontconfig.d/ fonts.conf	DataFile	yes	yes
/var/lib/texmf/fonts/map/pdftex/ updmap/pdftex.map	DataFile	yes	yes
/var/lib/texmf/fonts/map/pdftex/ updmap/pdftex_dl14.map	DataFile	yes	yes
/var/lib/texmf/ls-R	DataFile	yes	yes
/var/lib/texmf/ls-R-TEXLIVE	DataFile	yes	yes
/var/lib/texmf/ls-R-TEXMFMAIN	DataFile	yes	yes
/var/lib/texmf/web2c/pdftex/ pdflatex.fmt	DataFile	yes	yes
Ubuntu 11.10	OperatingSystem	yes	yes
timbus	Username	yes	yes

Table 5.5: The actual results of the prototype (data files, configuration).

The validation showed that the LNEC client initially was not able to execute on the target system, because of missing libraries of `wine`. Therefore, a manual adaption step of the model had been necessary. Observing the behavior of accessing dependencies on the source system revealed that in consecutive executions fewer dependencies are accessed then in the initial run (i.e. `wineboot.exe`). Those dependencies are located in the `/home/timbus/.wine/` directory, so to be sure to include all dependencies for `wine` this directory has been added manually to the process model, while omitting the individual files inside this directory for improved readability. These changes can be performed using Protégé. The model in Protégé before these changes is shown in Figure 5.4. Figure 5.5 shows the model after the changes have been performed. The changes also could have been made using any other editor that supports OWL files or even using standard text editors.



Thing
- ArchimateConcept
- Aspect
- ConfigurationFile
- DataFile
- Layer
- LogFile
- OperatingSystem
- Process
- Username
- VirtualPackage

- texlive-generic-recon
- texlive-latex-base
- timbus
- ttf-dejavu-core
- ttf-dejavu-extra
- ttf-droid
- ttf-freefont
- ttf-indic-fonts-core
- ttf-kacst-one
- ttf-khmeros-core
- ttf-lao
- ttf-liberation
- ttf-opensymbol
- ttf-punjabi-fonts
- ttf-takao-pgothic
- ttf-thai-tlwg
- ttf-ubuntu-font-family
- ttf-unfonts-core
- ttf-wqy-microhei
- 'Ubuntu 11.10'
- unzip
- usbutils
- util-linux
- v4.0.30319
- windows
- wine1.5
- winetricks
- xfonts-mathml
- zlib1g

Annotations
label [type: string]
windows

Types
Sys

Same Individual As

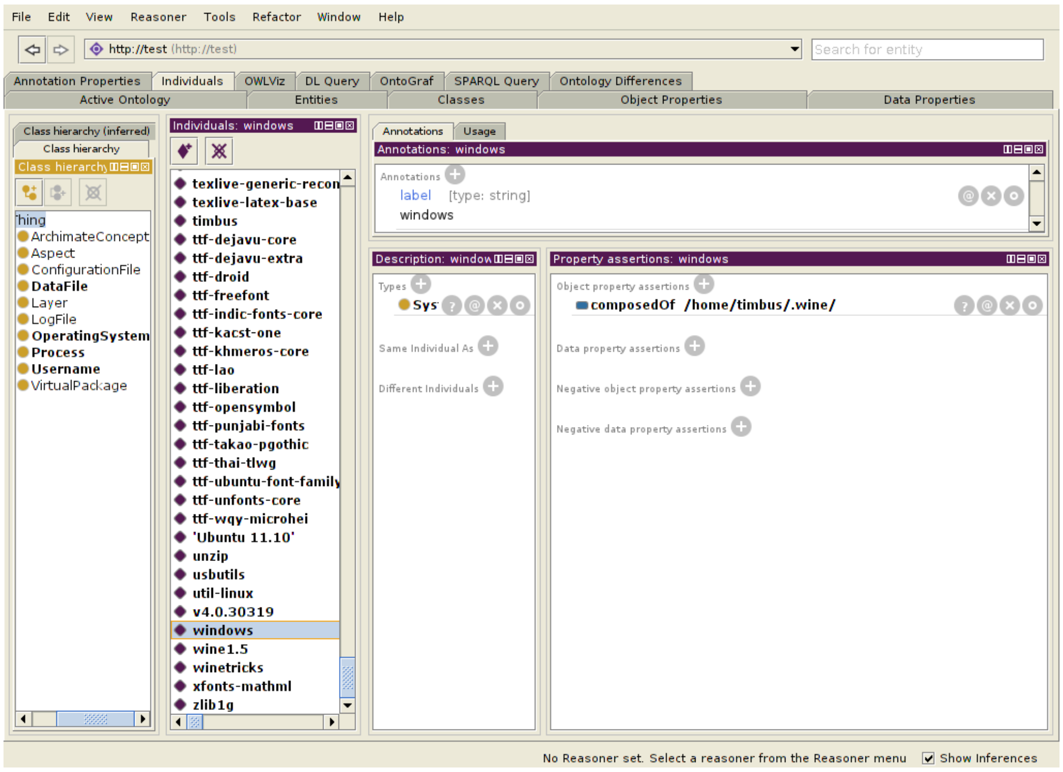
Different Individuals

Object property assertions
composedOf /home/timbus/.wine/drive_c/windows/system32
composedOf /home/timbus/.wine/drive_c/windows/Microsoft
composedOf /home/timbus/.wine/drive_c/windows/system32
composedOf /home/timbus/.wine/drive_c/windows/system32
composedOf /home/timbus/.wine/drive_c/windows/Microsoft
composedOf /home/timbus/.wine/drive_c/windows/system32
composedOf /home/timbus/.wine/drive_c/windows/system32
composedOf /home/timbus/.wine/drive_c/windows/system32
composedOf /home/timbus/.wine/drive_c/windows/system32
composedOf /home/timbus/.wine/drive_c/windows/Microsoft
composedOf /home/timbus/.wine/drive_c/windows/Microsoft
composedOf

Figure 5.4: The `.wine` directory shown in Protégé before manual adaption

Running the build module again and verifying the target system showed that the issue has been resolved by this measure, and the rebuilt system was now able to execute the process. The verification of the process model and the target system has been executed analogous to the verification of the music classification process. The execution of the process also produced the same result as when running on the source system. The model of the LNEC process that has been generated contained all expected direct dependencies.

The runtime behavior of the migration is comparable to the one of the music workflow. The capture component took about 12 minutes for execution, while running the build component took about 38 minutes. The longer execution time of the build component compared to the



Thing
- ArchimateConcept
- Aspect
- ConfigurationFile
- DataFile
- Layer
- LogFile
- OperatingSystem
- Process
- Username
- VirtualPackage

- texlive-generic-recon
- texlive-latex-base
- timbus
- ttf-dejavu-core
- ttf-dejavu-extra
- ttf-droid
- ttf-freefont
- ttf-indic-fonts-core
- ttf-kacst-one
- ttf-khmeros-core
- ttf-lao
- ttf-liberation
- ttf-opensymbol
- ttf-punjabi-fonts
- ttf-takao-pgothic
- ttf-thai-tlwg
- ttf-ubuntu-font-family
- ttf-unfonts-core
- ttf-wqy-microhei
- 'Ubuntu 11.10'
- unzip
- usbutils
- util-linux
- v4.0.30319
- windows
- wine1.5
- winetricks
- xfonts-mathml
- zlib1g

composedOf /home/timbus/.wine/

Figure 5.5: The `.wine` directory shown in Protégé after manual adaption

music classification scenario can be explained by the larger amount of packages in the LNEC scenario. The size of the consumed hard disk storage on the target system is about 2.2 GB, while on the source system about 7.2 GB are consumed. The number of files on the source system is 245256, on the target system there are 116479 files. 1865 packages are installed on the source system and only 1037 on the target system. The evaluation was performed on a system with an Intel(R) Core(TM) i5-2500 CPU @ 3.30 GHz, 8 GB memory, and a download speed of about 2 Mbit/s. The base box of the target system was already available locally.

The reflections of the comparison with other migration approaches from the previous chapter are also valid for this scenario.

5.3 Summary

The evaluation showed how the framework is able to support the migration of processes in shared environments to new dedicated environments. Compared to manual migration it helps improving the completeness of the model and the efficiency of the migration process. Especially indirect dependencies are tedious to detect manually. The limitation to the validation of direct dependencies turned out to be a reasonable trade-off between completeness of the verification and effort of the manual inspection of the source system.

The generalization of the model that results from applying the refiners improves the readability of the model, which eases especially the manual verification. The refiners also have a positive effect on the completeness of the migration, in terms that the portable applications are usable without noticeable restriction, compared to potentially missing libraries when using other functionality than stated in the process execution script on the target system.

For validation of the results of process migrations a feasible way is to run the process in the target system and compare the results.

The prototype can be applied to different scenarios, as long as the requirements stated in the beginning of this chapter are met. Some of the requirements can be relaxed by additional development effort of the prototype, i.e. the limitation of Debian as required platform of the source system. For this only the extraction scripts would need to be replaced, the Java part of the prototype implementation is not affected.

Intermediate and final results are represented using open formats. They can be viewed and adapted with arbitrary tools, but the formats also are human interpretable. The logs of the monitoring scripts are plain text files, the model is provided as OWL document, and the machine provisioning instructions are generated as Puppet manifests. This allows manual customization of the data between the execution of the main components of the PMF.

In the evaluation indirect packages happened to be missing in the official repositories of the target GNU/Linux system. For the current version of the prototype using a derivative or the same target operating system as the source therefore is recommended to avoid such issues.

Although packages and resources are sufficiently detected, there are some aspects that could be improved. For instance the mapping of filenames to packages could be more accurate. Several files that are logically part of some package, but cannot be resolved by the prototype, because they are not part of the initial installation of the package, clutter the model. An example for this is wine, which on demand installs additional files on the system. An additional refiner could try to resolve packages for instance by considering the parent directory or other files that are located in the same directory. The possibility to mark indirect dependencies could improve the readability of the model and help determining the hierarchy of packages. So if a package on a lower level is not available on the target system the package may be skipped, produce a warning, and suggest alternatives. Alternatives could be determined by querying the package manager of the target system. In case that direct dependencies are not available for the target system the source of the package should be added as part of the model, so that the package still might be resolved by adding those additional package sources to the package manager of the target system. This for example is useful if additional package repositories have been added to the source system.

CHAPTER 6

Summary and future work

This chapter describes benefits and drawbacks of the framework and the prototype. Migrating processes using the PMF is compared to migration using other approaches. Furthermore, ways for future extensions of the framework are shown.

6.1 Summary

The evaluation of the framework has shown that using the PMF migrating processes can be simplified significantly in different scenarios. Reasons for this are mainly the reduction of manual operations and the generation of the process environment model. The inclusion of a manual extraction step in the migration process of the PMF ensures that the framework is flexible enough for different kinds of scenarios. The migration can be performed in a way that makes the target system maintainable by using packages instead of individual files only. The refiners help keeping the model readable, and improve the completeness of the migration by including files that are not actively accessed during the process execution that has been observed during the dynamic analysis, but might be required in other process execution paths of the process.

The evaluation of the prototype has shown that processes can be observed at a very detailed level with standard GNU/Linux tools. The information that is extracted can easily be refined by extending the extractors. The model uses OWL, an XML-based and open format, which allows manual interception and comfortable editing using existing tools. Because the prototype has few requirements it is easy to deploy on the source system. VirtualBox is the only requirement that needs to be installed, the other required software can be run portable, i.e. from external devices. Using Puppet for provisioning provides a concrete setup instruction for systems, which can also be used independently from the PMF.

The PMF not only can be applied to extract and relocate a process from a shared existing system, but also to share a specific system environment for teams that need to execute the same process, or to copy the environment e.g. to establish a test setting that allows the execution of the process. The PMF can be used for archiving processes by creating a self contained virtual system and the according documentation. Because the virtual system only contains necessary resources it is more suitable for preservation than systems that result from a clone approach.

Compared to existing solutions especially the scope of the migration, so which parts of the source system should be migrated, but also the thorough inspection of the source system and the resulting open documentation are major differences. For exemplary tools which represent different types of migration strategies, a short comparison is given in the following.

`Rsync` is an example for file-based migration tools, i.e. backup tools that operate on the file system level. Rsync and similar tools are able to migrate processes that rely on data files and portable applications only. This is very flexible but offers no capture support, so determining which files are relevant to the process needs to be done manually. Documentation is implicit available in the optional configuration file of rsync, but also just on the filesystem level, and has to be created manually. Further information to the files could be added as comments. Like the PMF, rsync needs to operate directly in the source system and is able to transfer files to both local and remote locations. Application packaging tools could be combined with file-based migration tools so that the process is converted to a portable application and than transferred, though maintenance is reduced compared to native packages. Rsync is available for different platforms, i.e. GNU/Linux and Windows.

`VMware vCenter Converter` is used to convert physical systems into virtual systems (P2V converter) [104]. It is able to perform hot and cold cloning of the system, so it operates either on a running or a shut down system. Cold cloning is executed by starting the source system from a CD and cloning it without any analysis or documentation. This is useful to migrate systems that are not supported by other tools or the source system should not be altered. Compared to the PMF it has the disadvantages of not creating a model, the missing possibility to adapt the process environment, and the inclusion of more resources than probably would be required. An example for an adaption that is possible when migrating using the PMF has been showed in Section 5.2. Other use cases are the exchange of tools or libraries (e.g. using OpenJDK instead of the Oracle JDK). Hot cloning is done by installing agents on the source system, and the main application on any machine that can access the source system (remotely or locally). Changes to disk partitions, MAC addresses and other hardware properties are possible. But identification of the process environment also is missing, and adaption is limited. Both variants are not focused on processes but rather complete systems. Also, they are not as flexible in respect to making changes to the target system, i.e. allowing to replace software on the target system, and also do not provide a model as documentation. On the other hand migration using the vCenter Converter is not expected to miss any resources, and provides more possibilities on how to extract data from the source system (e.g. locally or remotely).

`CDE` can be used for application packaging. Because the CDE runtime needs to adapt requests to files, which have been relocated into the CDE package, the CDE runtime is required for running executables. So application packaging tools alter the runtime environment of the system, which might be undesired in several scenarios. Also, the target system is not created by such tools, they can be used to provision existing systems instead. So compared to the PMF application packaging software lacks documentation, adaption, and the creation of the initial target system. For packaged tools maintenance is reduced because it is difficult to exchange indirect dependencies, as they are woven into the portable package. For sharing and archiving individual applications this nevertheless is a good solution if documentation is not crucial and the resulting package is not supposed to be adapted.

From this overview of other migration approaches the main advantages of the PMF can be derived.

- The migration scope is on individual processes, not the source system as a whole.
- Documentation of the process environment is generated by the framework.
- The framework operates on high level concepts, which improves the maintainability of both model and target system (i.e. packages vs. bitlevel).
- The resulting target system can be adapted to custom requirements.

6.2 Future work

There is room for improvement in various aspects of the PMF. Some of them are shown in the following. For the framework especially the level of detail in the documentation can be improved, but also the necessary manual intervention should further be minimized. This especially concerns the adapt and verification steps. Another improvement would be the separation of the build module from the source system by storing the artifacts of the process environment in a portable repository. These topics are explained further in this section. Furthermore, the prototype implementation can be extended by providing extraction scripts for more platforms.

There are several ways to improve the level of completeness of the model. Further types of dependencies can be considered in the migration, and for already considered dependencies the level of detail can be improved. Possible methods for both are described in the following. Besides the artifacts that are absolutely necessary to successfully migrate a process (i.e. files, packages), there are many other aspects of a process environment that are relevant to a process and should be contained in a complete model of a business process. A few of them already have been partially implemented, like web service detection, or the name of the user that executes the process. Others are left open, like local database, or local service migration in general. Also, environment variables and additional package repository sources should be transferred to the target system in future versions. Environment variables are important e.g. because there may be additional paths defined in the `$PATH` environment variable that makes executables available to other applications without having to know its absolute path. Executables that have been installed without using a package manager would need to be searched by name, which is slow and not unambiguously. Therefore, the `$PATH` variable needs to be transferred to the source system if it has been modified by a user and there are scripts or tools in the process that rely on this modified data. In respect to the level of detail there are especially the source repositories of packages that should be added to the model in further versions. Missing packages in the official repositories have been an issue in the evaluation of the PMF. The support for handling additional package repository sources is required to be able to install packages that are not available in the default repositories of the distribution. Further information about packages that could be added includes dependencies and other relations to other packages. Security related aspects should also be stated in the process model. Examples include stating read/write/execute permissions for owner, group, and others like in GNU/Linux file systems [85], as well as the owner of files. Furthermore, for users the mapping to user groups should be documented.

For remote services the identification could be improved to retrieve more details about the services. Future work towards including remote services in the migration could go in the direction of recording and playing back web service requests, as shown in [66].

To be able to map organizational aspects of business processes to the technical aspects that is identified by the PMF, some kind of mapping functionality is required. This mapping is a further step towards a maintainable model. Mappings can be done in the lower technical levels by, for instance, mapping resources to their owning software, modeling the flow of the process, i.e. the individual steps of the process, their interaction and so on. On the higher levels it would be beneficial to automate the mapping of technical items to steps and artifacts of the business process. All of the technical information that is extracted should be mapped to a higher level description of the business process in future versions. Process mining frameworks could process the data generated by the PMF further to archive such a mapping.

As for the extractors, static and dynamic extraction can be extended to capture further details. For the static extractor, tailored analysis of binaries or data files of specific tools (e.g. Taverna workflow files), can be applied. This helps in keeping the required executions of the process to a minimum. Information that can be extracted includes dependencies, e.g. libraries and resources for binaries in general, but also dependencies of specific file formats like image resources from HTML files. Like described in [107], also the sources of software contain information about runtime dependencies. An extension of the framework could analyze source code if available and use them as further source of information. One of the challenges of such an approach is the identification of software itself based on their source files only. For software that is not released to the public, like in-house developed software, identification might not be possible.

There also are limitations to the current approach of analyzing the process execution trace. Especially activities that rely on tools or services that are not spawned in the context of the process are difficult to capture. An example of such would be a local database server, which loads settings or data during startup. During process execution such settings may be read from memory, and therefore are not captured by the prototype. A mitigation strategy could be to schedule observation of this services and perform a restart of the system before performing the process environment identification. Also, the incompleteness that arises from hardly being able to cover all paths of possible process executions is an important issue. Resolving files to portable packages helps mitigating this issue. For packages the implementation does not rely on the exact list of detected files, the completeness of the installation of the software is ensured by package managers. Also, extensions of the static extractor help in mitigating this issue. Techniques like automated test data generation [71] can support generating a process execution script for the dynamic extractor with a sufficient high level of branch coverage.

The detection of software that has not been installed using a package manager also is a challenge. Although the prototype uses a simple heuristic to try to detect such software, it does not represent them as software in the model, neither does it know which software this is. It just represents the bunch of files that are supposed to belong to one application as one single directory instead of a large amount of files. This is a very rough information, as it does not include real name or version of the software, and also the source of such software, i.e. a link to a download location or something similar is not provided. Further work could try to identify software by a lookup of its files in a database like the National Software Reference Library (NSRL) [64], which contains software and files that belong to this software, including hash values thereof. Such lookup could be based on either the hash value of the content, or part of the file name.

To reduce the tradeoff between a concise model and an accurate description of resources the refiners could be used to extend the model by e.g. adding additional information, but without removing elements from the model. Instead relations between the new and the original elements could be established. An example would be to add all files of portable applications, but also add an portable application element which has a relation to all of its file elements. Such an approach would be useful to keep the main artifacts of the process visible and still keep all details.

The adapt component is specific to concrete requirements, and many different implementations are possible. Some examples are provided in the following. In one scenario the purpose of the migration of the process is to mitigate risks that arise from using obsolete software. In such a scenario an adapt module can be implemented so that it is configurable with a list of software tools or data formats and a rating of the risk they pose. Using this data the adapt module could try to replace artifacts at risk with other artifacts that are more stable. In another scenario cleanup and reestablishing understandability for a business process is the main purpose of migrating the process. In such a case the adapt module can be used to merge the technical model with a business view, thereby providing a holistic documentation of the business process.

Another aspect that can be optimized in future versions of the PMF is the dependence on the source system. At the current state of the framework it is necessary to execute both the capture and the build module on the source system. This is required because artifacts are directly transferred from source to target system. But this means that for executing the capture and the build module always a snapshot of the source system needs to be available. This is a limitation that can be mitigated by introducing a repository that acts as storage for all artifacts that are required to redeploy the process without having access to the source system. To create a repository, the capture module has to be adapted to place a copy of identified resources in a directory. The build module needs to fetch the artifacts from this repository instead of the source system. Besides data files also packages could be stored in this repository. This can be used if packages are cached locally, can be downloaded from repositories, or if it is possible to create redeployable packages of installed sources on the source system. Depending on the source and the target operating system a conversion of the package might be necessary.

To support unattended automation of the migration process a validation module that executes after the build module could be implemented. The VFramework [67] describes such an approach for verification and validation of preserved business processes.

Bibliography

[1] Victor A. Abell. lsof(8): open files - Linux man page. http://linux.die.net/man/8/lsof. [Online; accessed 9-September-2013].

[2] Jeroen Arnoldus, Mark van den Brand, A. Serebrenik, and J.J. Brunekreef. *Code Generation with Templates*. Atlantis studies in computing. Atlantis Press, 2012.

[3] Marzieh Bakhshandeh, Gonçalo Antunes, Rudolf Mayer, José Borbinha, and Artur Caetano. A modular ontology for the enterprise architecture domain. In *Proceedings of the 8th International Workshop on Vocabularies, Ontologies and Rules for the Enterprise and Beyond (VORTE 2013), in conjunction with the 17th IEEE International EDOC Conference (EDOC 2013)*, Vancouver, British Columbia, Canada, September 9-13 2013.

[4] Ira D. Baxter. Branch coverage for arbitrary languages made easy. http://www.semdesigns.com/Company/Publications/, 2002. [Online; accessed 3-February-2014].

[5] Stew Benedict and Jeff Licquia. Dependency Checker Tool - Overview and Discussion. White Paper, http://www.linuxfoundation.org/publications/compliance. [Online; accessed 16-October-2013].

[6] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Automated Discovery and Maintenance of Enterprise Topology Graphs. In *Proceedings of the 6th IEEE International Conference on Service Oriented Computing & Applications (SOCA 2013)*, pages 126–134. IEEE Computer Society Conference Publishing Services, Dezember 2013.

[7] Tegawendé F. Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillère. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *Proceedings of the IEEE 37th Annual Computer Software and Applications Conference (COMPSAC 2013)*, pages 303–312, July 2013.

[8] Dan Bode and Nan Liu. *Puppet Types and Providers*. O'Reilly & Associates, Inc., 2012.

[9] Matthew Booth. virt-p2v and virt-v2v. http://libguestfs.org/virt-v2v/. [Online; accessed 20-June-2013].

[10] Daniel Pierre Bovet and Marco Cesati. *Understanding the Linux Kernel.* O'Reilly & Associates, Inc., second edition, 2003.

[11] Kevin Braunsdorf and Matthew Bradburn. test(1): check file types/compare values - Linux man page. `http://linux.die.net/man/1/test`. [Online; accessed 09-March-2014].

[12] Joos C. A. M. Buijs, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. Mining configurable process models from collections of event logs. In *Business Process Management*, volume 8094 of *Lecture Notes in Computer Science*, pages 33–48. Springer Berlin Heidelberg, 2013.

[13] Ariel N. Burton and Paul H. J. Kelly. Workload characterization using lightweight system call tracing and reexecution. In *Proceedings of the IEEE International Performance Computing and Communications Conference (IPCCC '98)*, pages 260–266, Feb 1998.

[14] Michele Bustamante. *Learning WCF: A Hands-on Guide.* X Windows Series. O'Reilly Media, 2007.

[15] Mendel Cooper. Advanced bash-scripting guide 6.6 - an in-depth exploration of the art of shell scripting. `http://www.tldp.org/LDP/abs/abs-guide.pdf`. [Online; accessed 22-July-2013].

[16] Michelle Cotton, Lars Eggert, Joe Touch, Magnus Westerlund, and Stuart Cheshire. Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry. RFC 6335, RFC Editor, August 2011.

[17] William Crawford, Jim Farley, and Prakash Malani. *Java Enterprise in a Nutshell.* In a Nutshell (o'Reilly) Series. O'Reilly, 2005.

[18] Defense Information Systems Agency. Department of Defense Technical Architecture Framework for Information Management. Volume 2. Technical Reference Model. Version 3.0., April 1996.

[19] Wei Deng, Fangming Liu, Hai Jin, Xiaofei Liao, Haikun Liu, and Li Chen. Lifetime or energy: Consolidating servers with reliability control in virtualized cloud datacenters. In *Proceedings of the IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom 2012)*, pages 18–25, Dec 2012.

[20] Anna Derezinska and Marian Szczykulski. Tracing of state machine execution in the model-driven development framework. In *Proceedings of the 2nd International Conference on Information Technology (ICIT 2010)*, pages 109–112, 2010.

[21] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. Nix: A safe and policy-free system for software deployment. In *Proceedings of the 18th Conference on Systems Administration (LISA '04), Atlanta, USA, November 14-19, 2004*, pages 79–92, Berkeley, CA, USA, 2004. USENIX Association.

[22] Eelco Dolstra and Armijn Hemel. Purely functional system configuration management. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems (HOTOS'07)*, pages 13:1–13:6, Berkeley, CA, USA, 2007. USENIX Association.

[23] Eelco Dolstra, Andres Löh, and Nicolas Pierron. Nixos: A purely functional linux distribution. *Journal of Functional Programming*, 20(5-6):577–615, November 2010.

[24] Eelco Dolstra, Rob Vermaas, and Shea Levy. Charon: Declarative provisioning and deployment. *1st International Workshop on Release Engineering (RELENG 2013)*, May 2013.

[25] Joseph Emeras, Bruno Bzeznik, Olivier Richard, Yiannis Georgiou, and Cristian Ruiz. Reconstructing the software environment of an experiment with kameleon. In *Proceedings of the 5th ACM COMPUTE Conference: Intelligent & scalable system technologies (COMPUTE '12)*, pages 16:1–16:8, New York, NY, USA, 2012. ACM.

[26] Paul Tepper Fisher and Brian D. Murphy. *Spring Persistence with Hibernate*. Apresspod Series. Apress, 2010.

[27] Martin Fowler. Inversion of control containers and the dependency injection pattern. `http://www.martinfowler.com/articles/injection.html`, January 2004. [Online; accessed 11-December-2013].

[28] Sören Frey and Wilhelm Hasselbring. Model-based migration of legacy software systems into the cloud: The cloudMIG approach. *Softwaretechnik-Trends*, 30(2), 2010.

[29] Sören Frey and Wilhelm Hasselbring. Model-based migration of legacy software systems to scalable and resource-efficient cloud-based applications: The cloudmig approach. In *Proceedings of the First International Conference on Cloud Computing, GRIDs, and Virtualization (Cloud Computing 2010)*, pages 155–158, Lisbon, Portugal, November 2010.

[30] Jeff Friesen. *Beginning Java SE 6 Platform: From Novice to Professional*. Empowering productivity for the Java developer. Apress, 2007.

[31] Mike Gancarz. *The UNIX philosophy*. Digital Press, 1995.

[32] Chris Greamo and Anup Ghosh. Sandboxing and virtualization: Modern tools for combating malware. *Security Privacy, IEEE*, 9(2):79–82, 2011.

[33] Anthony Green. libffi - A Portable Foreign Function Interface Library. `https://sourceware.org/libffi/`. [Online; accessed 11-March-2014].

[34] Ed H. B. M. Gronenschild, Petra Habets, Heidi I. L. Jacobs, Ron Mengelers, Nico Rozendaal, Jim van Os, and Machteld Marcelis. The effects of freesurfer version, workstation type, and macintosh operating system version on anatomical volume and cortical thickness measurements. *PLoS ONE*, 7(6):e38234, 06 2012.

[35] W3C OWL Working Group. OWL 2 web ontology language document overview. Technical report, W3C, October 2009. http://www.w3.org/TR/2009/REC-owl2-overview-20091027/.

[36] Chunqin Gu, Hui you Chang, and Yang Yi. Overview of workflow mining technology. In *Proceedings of the IEEE International Conference on Granular Computing, 2007. (GRC 2007)*, pages 347–347, 2007.

[37] Philip J. Guo. Frequently asked questions - CDE v1.0 documentation. `http://www.pgbovine.net/cde/manual/faq.html`. [Online; accessed 24-June-2013].

[38] Philip J. Guo. CDE: Run any Linux application on-demand without installation. In *Proceedings of the 25th international conference on Large Installation System Administration (LISA'11)*, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[39] Philip J. Guo and Dawson Engler. CDE: Using system call interposition to automatically create portable software packages. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference (USENIXATC'11)*, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[40] Mitchell Hashimoto. *Vagrant: Up and Running*. O'Reilly Media, 2013.

[41] Martin Hepp, Pieter De Leenheer, Aldo de Moor, and York Sure, editors. *Ontology Management, Semantic Web, Semantic Web Services, and Business Applications*, volume 7 of *Semantic Web And Beyond Computing for Human Experience*. Springer, 2008.

[42] Raphaël Hertzog and Roland Mas. *The Debian Administrator's Handbook: Debian Squeeze from Discovery to Mastery*. 2012.

[43] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for OWL ontologies. *Semantic Web*, 2(1):11–21, January 2011.

[44] IEEE. *003.0-1995 IEEE Guide to the POSIX® Open System Environment (OSE) (Identical to ISO/IEC TR 14252)*. 1995.

[45] Yannis Kalfoglou, Bo Hu, Dave Reynolds, and Nigel Shadbolt. Capturing, representing and operationalising semantic integration (CROSI) project - final report. Technical report, October 2005.

[46] Shmuel Katz, Mira Mezini, Christine Schwanninger, and Wouter Joosen. *Transactions on Aspect-Oriented Software Development VIII*. Lecture Notes in Computer Science / Transactions on Aspect-Oriented Software Development. Springer, 2011.

[47] Wu Kehe, Wang Zhuo, Zhao Xing, and Ma Gang. Design and implementation of the monitoring system for EJB applications based on interceptors. In *Proceedings of the 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE 2010)*, volume 4, pages V4–5–V4–9, Aug 2010.

[48] Michael Kerrisk. *The Linux programming interface: A Linux and UNIX system programming handbook.* No Starch Press, 2010.

[49] Moazzam Khan, Zehui Bi, and John A. Copeland. Software updates as a security metric: Passive identification of update trends and effect on machine infection. In *Proceedings of the Military Communications Conference 2012 (MILCOM 2012)*, pages 1–6, 2012.

[50] Robert J Knell. *Introductory R: A Beginner's Guide to Data Visualisation and Analysis using R.* March 2013. [Online; `http://www.introductoryr.co.uk/`; accessed 03-March-2014].

[51] Martin F. Krafft. *The Debian System: Concepts and Techniques.* No Starch Press Series. No Starch Press, 2005.

[52] Henryk Krawczyk and Anna Mizgier. System to system migration for improving interoperability. In *Proceedings of the 2nd International Conference on Information Technology (ICIT 2010)*, pages 125–128, 2010.

[53] Spencer Krum, William Van Hevelingen, Ben Kero, James Turnbull, and Jeffrey McCune. *Pro Puppet.* Apress, 2013.

[54] Jian Kuang, Jie Liu, and Jiali Bian. Implementing java programming language on RTEMS operating system. In *Proceedings of the IEEE Symposium on Electrical Electronics Engineering (EEESYM 2012)*, pages 90–93, 2012.

[55] Björn Könning, Christian Engelmann, Stephen L. Scott, and Al Geist. Virtualized environments for the harness high performance computing workbench. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 133–140, 2008.

[56] Byeongcheol Lee, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. Debug all your code: Portable mixed-environment debugging. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*, pages 207–226, New York, NY, USA, 2009. ACM.

[57] Dustin Lee, Jeff Rowe, Calvin Ko, and Karl Levitt. Detecting and defending against webserver fingerprinting. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC'02)*, pages 321–330, 2002.

[58] Raimondas Lencevicius, Edu Metz, and Alexander Ran. Tracing execution of software for design coverage. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 328–332, 2001.

[59] Raimondas Lencevicius, Alexander Ran, and Rahav Yairi. Third eye - specification-based analysis of software execution traces. In *Proceedings of the 22nd International Conference on on Software Engineering, (ICSE 2000), Limerick Ireland, June 4-11, 2000*, page 772, 2000.

[60] CodeSourcery LLC, Mark L. Mitchell, Alex Samuel, and Jeffrey Oldham. *Advanced Linux Programming*. Landmark Series. Pearson Education, 2001.

[61] Robert Love. *Linux System Programming: Talking Directly to the Kernel and C Library*. O'Reilly Media, 2013.

[62] Rudolf Mayer, Stefan Proell, and Andreas Rauber. On the applicability of workflow management systems for the preservation of business processes. In *Proceedings of the 9th International Conference on Digital Preservation (iPres 2012)*, 10 2012.

[63] Rudolf Mayer and Andreas Rauber. Towards time-resilient MIR processes. In *Proceedings of the 13th International Society for Music Information Retrieval Conference (ISMIR 2012), Porto, Portugal, October 8-12, 2012*, pages 337–342. FEUP Edições, 2012.

[64] Steve Mead. Unique file identification in the national software reference library. *Digital Investigation*, 3(3):138–150, 2006.

[65] Michael Menzel, Markus Klems, Hoang Anh Le, and Stefan Tai. A configuration crawler for virtual appliances in compute clouds. In *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E 2013)*, pages 201–209, 2013.

[66] Tomasz Miksa, Rudolf Mayer, and Andreas Rauber. Ensuring sustainability of web services dependent processes. *International Journal of Computational Science and Engineering (IJCSE)*, 2013. Accepted for publication.

[67] Tomasz Miksa, Stefan Proell, Rudolf Mayer, Stephan Strodl, Ricardo Vieira, José Barateiro, and Andreas Rauber. Framework for verification of preserved and redeployed processes. In *Proceedings of the 10th International Conference on Preservation of Digital Objects (IPRES2013)*, Lisbon, Portugal, September 2–6 2013.

[68] Paolo Missier, Stian Soiland-Reyes, Stuart Owen, Wei Tan, Aleksandra Nenadic, Ian Dunlop, Alan Williams, Thomas Oinn, and Carole Goble. Taverna, reloaded. In *Proceedings of the 22nd International Conference on Scientific and Statistical Database Management (SSDBM 2010), Heidelberg, Germany, June 30 - July 2, 2010*, Heidelberg, Germany, June 2010.

[69] Arturo Fernandez Montoro. *Linux Mint System Administrator's Beginner's Guide*. Packt Publishing, Limited, 2012.

[70] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (SP '07)*, pages 231–245, 2007.

[71] Tafline Murnane and Karl Reed. On the effectiveness of mutation analysis as a black box testing technique. In *Proceedings of the Australian Software Engineering Conference (ASWEC 2001)*, pages 12–20, 2001.

[72] Yukikazu Nakamoto, Tatsunori Osaki, and Issei Abe. Proposing universal execution trace framework for embedded software using QEMU. In *Software Technologies for Future Dependable Distributed Systems 2009*, pages 173–178, 2009.

[73] Christopher Negus. *Ubuntu Linux Toolbox: 1000+ Commands for Power Users*. Wiley, 2013.

[74] Christopher Negus and Christine Bresnahan. *Linux Bible*. Bible. Wiley, 2012.

[75] Stephen Nelson-Smith. *Test-Driven Infrastructure with Chef*. O'Reilly Media, 2011.

[76] Thomas Damgaard Nielsen, Christian Iversen, and Philippe Bonnet. Private cloud configuration with MetaConfig. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD 2011)*, pages 508–515, 2011.

[77] Oracle Corporation. MySQL 5.0 Reference Manual. `http://dev.mysql.com/doc/refman/5.0/en/`. [Online; accessed 5-February-2014].

[78] Jonathan Oxer, Kyle Rankin, and Bill Childers. *Ubuntu Hacks: Tips & Tools for Exploring, Using, and Tuning Linux*. O'Reilly Media, 2009.

[79] Jay Palat. Introducing Vagrant. *Linux Journal*, 2012(220):76–85, August 2012.

[80] Ricardo Pérez-Castillo, Ignacio García Rodríguez de Guzmán, and Mario Piattini. Knowledge discovery metamodel-ISO/IEC 19506: A standard to modernize legacy systems. *Computer Standards & Interfaces*, 33(6):519–532, 2011.

[81] David S. Peterson, Matt Bishop, and Raju Pandey. A flexible containment mechanism for executing untrusted code. In *Proceedings of the 11th USENIX Security Symposium*, pages 207–225, San Francisco, CA, USA, August 2002. The USENIX Association.

[82] Puppet Labs. Directory creation fails if parent directory does not exist. `http://projects.puppetlabs.com/issues/86`. [Online; accessed 9-March-2014].

[83] R Core Team. Writing R Extensions, Version 3.0.3 (2014-03-06). `http://cran.r-project.org/doc/manuals/r-release/R-exts.pdf`. [Online; accessed 9-March-2014].

[84] Dave Reynolds, Carol Thompson, Jishnu Mukerji, and Derek Coleman. An assessment of RDF/OWL modelling. Technical Report HPL-2005-189, Hewlett Packard Laboratories, October 10 2005.

[85] Arnold Robbins. *Unix in a Nutshell*. In a Nutshell. O'Reilly Media, 2008.

[86] Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structure, and Architecture*. Wiley computer publishing. John Wiley, 1996.

[87] Valentina Salapura. Cloud computing: Virtualization and resiliency for data center computing. In *Proceedings of the IEEE 30th International Conference on Computer Design (ICCD 2012)*, pages 1–2, 2012.

[88] Joseph Siefers, Gang Tan, and Greg Morrisett. Robusta: taming the native beast of the JVM. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the ACM Conference on Computer and Communications Security 2010*, pages 201–211. ACM, 2010.

[89] Aleksandar Skendzic, Bozidar Kovacic, and Igor Jugo. Decreasing information technology expenses by using emulators on windows and linux platforms. In *Proceedings of the 34th International Convention of Information Communication Technology, Electronics and Microelectronics (MIPRO 2011)*, pages 1387–1390, 2011.

[90] Software in the Public Interest, Inc. The Debian GNU/Linux FAQ. `http://www.debian.org/doc/manuals/debian-faq/`. [Online; accessed 16-July-2013].

[91] Steffen Staab and Rudi Studer. *Handbook on Ontologies*. Springer Publishing Company, Incorporated, 2nd edition, 2009.

[92] Stanford Center for Biomedical Informatics Research (BMIR). Protégé. `http://protege.stanford.edu/`. [Online; accessed 11-July-2013].

[93] Dafydd Stuttard and Marcus Pinto. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. 2011.

[94] The Open Group. *TOGAF Version 9: A Pocket Guide*. TOGAF Series. Van Haren Publishing, 2009.

[95] The Open Group. *ArchiMate 2. 0 Specification*. The Open Group. Van Haren Publishing, 2012.

[96] TIMBUS Consortium. D4.3: Dependency Models Iter. 2, WP 4 – Processes and Methods for Digitally Preserving Business Processes. `http://timbusproject.net/resources/publications/public-project-deliverables`. [Online; accessed 01-August-2013].

[97] TIMBUS Consortium. D4.5: Business Process Contexts, WP 4 – Processes and Methods for Digitally Preserving Business Processes. `http://timbusproject.net/resources/publications/public-project-deliverables`. [Online; accessed 8-April-2014].

[98] TIMBUS Consortium. D8.1: Use Case Definition and Digital Preservation Requirements, WP 8 – Industrial Project 2: Civil Engineering Infrastructure. `http://timbusproject.net/resources/publications/public-project-deliverables`. [Online; accessed 09-March-2014].

[99] Ralf Treinen and Stefano Zacchiroli. Description of the CUDF Format. *CoRR*, abs/0811.3621, 2008.

[100] Unsigned Integer Limited. Distrowatch.com. `http://distrowatch.com/`. [Online; accessed 3-September-2013].

[101] Wil M. P. van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[102] Wil M. P. van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.

[103] Robbie Vanbrabant. *Google Guice: Agile Lightweight Dependency Injection Framework (Firstpress)*. APress, 2008.

[104] VMware, Inc. VMware vCenter Converter Standalone User's Guide. `http://www.vmware.com/pdf/convsa_50_guide.pdf`. [Online; accessed 25-February-2014].

[105] Tzu-Yen Wang, Chin-Hsiung Wu, and Chu-Cheng Hsieh. A virus prevention model based on static analysis and data mining methods. In *Proceedings of the IEEE 8th International Conference on Computer and Information Technology Workshops (CIT Workshops 2008)*, pages 288–293, July 2008.

[106] Brian Ward. *How Linux Works: What Every Superuser Should Know*. No Starch Press Series. No Starch Press, 2004.

[107] Elisabeth Weigl, Johannes Binder, Stephan Strodl, Daniel Draws, and Andreas Rauber. A framework for automated verification in software escrow. In *Proceedings of the 10th International Conference on Preservation of Digital Objects (IPRES 2013)*, pages 95–103, 2013.

[108] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, Jiten Bhagat, Khalid Belhajjame, Finn Bacall, Alex Hardisty, Abraham Nieva de la Hidalga, Maria P. Balcazar Vargas, Shoaib Sufi, and Carole Goble. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Research*, 2013.

[109] Bing Wu, Deirdre Lawless, Jesus Bisbal, Jane Grimson, Vincent Wade, D O'Sullivan, and Ray Richardson. Legacy systems migration - a method and its tool-kit framework. In *Proceedings of the joint Asia Pacific Software Engineering Conference (APSEC '97) and International Computer Science Conference (ICSC '97)*, pages 312–320, 1997.

[110] Lei Wu, Houari Sahraoui, and Petko Valtchev. Coping with legacy system migration complexity. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2005)*, pages 600–609, 2005.

[111] Xiong Xiaobing, Shu Hui, Chen Jianmin, and He Yongjun. Wetrp: A platform for recording of windows program execution traces. In *Proceedings of the International Symposium on Computer Science and Computational Technology (ISCSCT '08)*, volume 1, pages 621–625, 2008.

[112] Li Yuanyuan, Xiao Peng, and Deng Wu. The method to test linux software performance. In *Proceedings of the International Conference on Computer and Communication Technologies in Agriculture Engineering (CCTAE 2010)*, volume 1, pages 420–423, 2010.

[113] Amirreza Zarrabi, Khairulmizam Samsudin, and Amin Ziaei. Dynamic process migration framework. In *Proceedings of the International Conference of Information and Communication Technology (ICoICT 2013)*, pages 410–415, 2013.

[114] Gong Zhang and Ling Liu. Why do migrations fail and what can we do about it? In *Proceedings of the 25th international conference on Large Installation System Administration (LISA'11)*, pages 25–25, Berkeley, CA, USA, 2011. USENIX Association.

[115] Hongyu Zhang, Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. Using source transformation to test and model check implicit-invocation systems. *Science of Computer Programming*, 62(3):209 – 227, 2006.

[116] Yanjun Zuo. Moving and relocating: A logical framework of service migration for software system survivability. In *Proceedings of the IEEE 7th International Conference on Software Security and Reliability (SERE 2013)*, pages 139–148, 2013.