University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Installing and Using PyPy Standalone Compiler with Parlib Framework

Technology Overview and Installation Manual

Marek Paška

Technical Report No. DCSE/TR-2012-09
August, 2012

Distribution: public

Technical Report No. DCSE/TR-2012-09
August 2012

Installing and Using PyPy Standalone Compiler with Parlib Framework

Marek Paška

Abstract

This technical report describes how to install and use an application framework called Parlib with an experimental compiler based on the PyPy project. This software equipment enables to write programs in the Python programming language and to translate them to efficient machine code and Java byte-code. A development process that takes advantage of these tools is briefly described. The development process also embraces formal methods, literally explicit model checker Java Pathfinder, to improve dependability of the final program. The development process is designed for embedded devices; however, it is not limited to this domain.

Contents

1 Introduction

This technical report describes how an experimental tool-chain based on (also experimental) implementation of the Python programming language can be installed and used. The purpose of the tool-chain is to produce dependable and resource-efficient applications based on Python. The tool-chain and a development process that takes advantage of it were developed as a Ph.D. study project [1].

The resource efficiency is achieved by compilation to the C code. This low-level code is intended for embedded devices, though it is not limited to this domain.

The dependability is addressed by employment of an explicit model checker called Java Pathfinder. This tool can discover awkward bugs such as deadlock or race condition.

This report also provides a brief description of the underlying technologies and basics of our development process.

2 Technology Overview

The tool-chain is based no two established peaces of software: PyPy and Java Pathfinder. These are the giants on whose shoulders we stand.

2.1 The PyPy Interpreter and Compiler

PyPy [2] is an experimental implementation of the Python language developed at ETH Zurich since 2003. The most interesting attribute of the PyPy project is that it is written entirely in Python, i.e., it is self-hosting. The main goal of the project is to bring the recent fruit of research of interpreters and virtual machines to the world of Python.

Python is dynamically and strongly typed object-oriented language. Its main implementation is Python interpreter written in C, usually referred to as CPython. CPython runs on many architectures and operating systems. Apart from PyPy, there exist several other implementations: Jython that runs on the top of JVM and IronPython for .Net.

PyPy consists of several parts. First of all, it is an interpreter. As it is written in Python, it runs on the top of another interpreter; that means, CPython. So the program that runs on top of the PyPy interpreter pays the cost of double interpretation.

Another part of PyPy is a compiler. The primary goal of the compiler is to translate the PyPy interpreter from Python to C source code that can be compiled to native machine code. Apart from C, there are several other target codes supported: Java byte-code, MS Intermediate Language (.Net), SmallTalk, LLVM[1], and JavaScript.

The PyPy compiler is a single-purpose program. It was designed to compile the PyPy interpreter to the more efficient code and thus to avoid the double interpretation. The software stack before and after the translation from Python to C is depicted in figure 1.

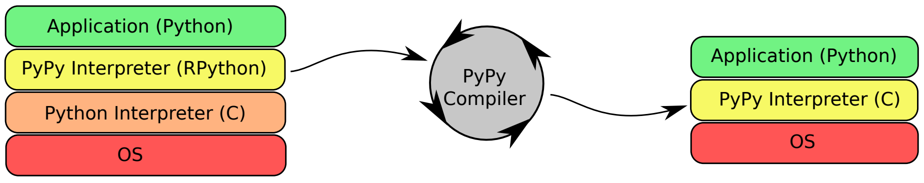

Figure 1: Translation of PyPy Interpreter

Because of the dynamic nature of the Python language, the PyPy compiler only supports more static subset of Python called Restricted Python, or *RPython*.

However, PyPy approach combines dynamic and static code. The main idea is that the dynamic behavior is enabled only up to some fixed point of execution. At first, the program's object space is constructed dynamically; all dynamic features are enabled, including *eval*. At second, dynamic features are suppressed. The dynamics is limited only to the features known from compiled languages, such as dynamic object creation, virtual method call, etc.

The frozen program's object space is used for subsequent transformations:

1. *Type inference.* A static data type is assigned to every data field. These types are abstract, i.e., have only indirect relation with the data types of the target platform.

2. *Low-level typing.* According to the selected backend, e.g., C of JVM, a particular native data type is assigned to each data field.

3. *Backend-specific transformations.* Implementation details such as memory management and exception handling are addressed here.

4. *Code generation.* The final output code is generated, e.g., C source code or Java byte-code.

The main strength of the PyPy compiler is its flexibility and modularity. Every step of the compilation can be customized easily.

[1]Low Level Virtual Machine, http://llvm.org

2.2 Java Pathfinder 2

Java Pathfinder 2 (JPF2) [3] is an explicit model checker for Java developed at NASA. Its predecessor Java Pathfinder 1 [4] attempted to translate Java source code to Promela language though it is now retired.

JPF2 is a special implementation of the Java Virtual Machine (JVM) that has a model-checking facility. The verification is done at the Java byte-code level; JPF2 does not need access to the source code of the investigated program.

Conventional JVM executes Java byte-code sequentially and the state of the running program is constantly altered during the execution. JPF2, on the other hand, has the ability to store every state of the program and restore it later when needed. This approach allows all reachable states of the program to be examined. The JPF2 architecture is pluggable; there is a possibility to use various algorithms for the state space traversal. JPF2 can also use heuristic methods to determine which states should be examined first in order to discover an error.

The model-checker can search for deadlocks, check invariants, user-defined assertions (embedded in the code), and LTL-expressed[2] properties. JPF2 provides techniques for fighting the state space explosion: abstraction, slicing. User can also specify the level of atomicity, the atomic step can be set to one byte-code instruction, to one line of Java code, or to a block of code.

JPF2 also supports non-determinism to be injected into a deterministic Java program. For instance the method *Verify.randomBool()* returns either *true* or *false*, and JPF2 guarantees that both possibilities will be examined.

Java Pathfinder 2 is a mature tool that is practically used at NASA. The main advantage is that it checks real Java programs and can provide a proof of correctness.

3 Development Process Overview

The development process takes advantage of various features of the tool-chain. The first advantage is that the Python code can be directly interpreted (without compilation) by CPython. This allows rapid application prototyping. The second advantage is that we can easily produce Java byte-code that can be checked by Java Pathfinder.

[2]Linear Temporal Logic

3.1 Specifying Application and Target Platform

Let us specify what kind of applications is our development approach suitable for and what software and hardware platform we are targeting.

3.1.1 Architecture

We are interested in multi-threaded programs. First, they are common as many real-world programs have to handle multiple tasks at once. Second, their designing, debugging, and finally proving their correctness is much more complicated then in the case of single-threaded programs.

We are interested in reactive systems; however, we do prefer neither event-driven nor time-triggered systems.

Many reactive systems are real-time. We rely on high level dynamic languages and thus we insist on automatic memory management. Therefore, our approach cannot be used for hard real-time systems due to the fact that real-time garbage collectors are a subject of active research. However, if there were industry-ready hard real-time GC, it can be incorporated. Nowadays, it can be only used for soft real-time systems.

Inability to cope with hard real-time requirements also makes our approach hardly applicable to safety-critical systems.

3.1.2 Software

We assume the final product is interfacing an operating system rather than bare metal. We are not restricted to any particular OS; however, our primary software platform is Linux and thus POSIX interface.

For convenience and to overcome differences of various operating systems, we do not access OS directly but through the standard C library. Our primary target is GNU C Library[3] but other variants such as uClibc[4] should work as well.

Our threading relies on POSIX Threads [5] that are available for many Unix-like operating systems and even for Microsoft Windows.

3.1.3 Hardware

We are not restricted to any particular hardware architecture. Our primary CPU architecture is Intel x86, though. Support for another architectures such as ARM

[3]http://www.gnu.org/software/libc/
[4]http://www.uclibc.org/

can be achieved relatively simply because we use portable assembly code, more known as the C language.

We currently assume only 32-bit architectures.

Other hardware requirements are given by the underlying operating system. As we insist on memory-safe language, it can be safely operated on systems without MMU[5] such as μClinux[6].

3.2 Development Process Based on High Level Dynamic Language

First, the program code is primarily written in a widely used high level dynamic language. The code should have the following properties:

- Relatively short, easy to maintain and debug due to the expressiveness of the high level language.
- Flexible and open for new paradigms due to the dynamic approach.
- Familiar for many developers on the market because it is based on a widespread language.

Second, the final output is in the form of native machine code; it is generated ahead-of-time, not just-in-time. Machine code is fast and compact enough to cope with constrained hardware resources. However, we will embrace automatic memory management.

Third, there is a support for formal verification. There has to be a way how to earn solid *guarantees of correctness* of the final application. Moreover, the formal methods used should be accessible even for wide developer audience.

The development approach is designed to have three steps.

1. The program is written in a high level dynamic language. It is runnable by the standard interpreter of the language.
2. The machine code intended for deployment is generated. The program is runnable on an embedded computer.
3. Various properties of the code are formally verified or at least tested with the help of formal methods.

[5]Memory Management Unit
[6]http://uclinux.org/description/

The main strength of the first step is the easiness of development. Because there is no need of compilation, one can quickly experiment with the code and create rapid prototypes. Debugging of an interpreted code is more straightforward than in the case of machine code that is run on CPU. The behavior of the virtual machine can be easily changed; in contrast, one can not change the behavior of a physical CPU.

The second step takes the debugged high level program as an input and produces low-level machine code as an output. This step is actually challenging and heavily depends on the selected tool-chain, following chapters are dedicated to this topic.

The result of the third step is a set of guarantees of correctness of the final machine code. The guarantees are earned by experiments propelled by formal methods.

The overall scheme of our proposed development approach is depicted in figure 2.

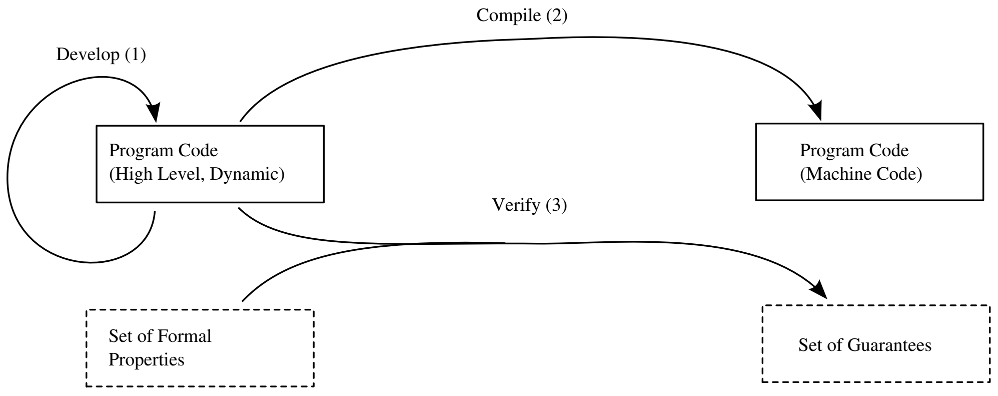

Figure 2: Overall Scheme of Our Approach

3.3 Refining the Development Approach

After we have selected some concrete tools (PyPy and Java Pathfinder), we can describe the development approach in more detail. We propose an iterative development process. There are three types of iteration, every type has a different cost and different purpose, see figure 3.

The first type of iteration exclusively uses the standard Python interpreter. A developer applies a change in a RPython source code of the application, runs the modified code in the Python interpreter and instantly sees how the change work. This type of iteration is very fast as there is no compilation. This approach perfectly fits *test-driven development*.

The second type of iteration deals with a testing based on formal methods. A

Java byte-code is generated by PyPy from RPython sources and the generated byte-code is investigated by Java Pathfinder. This iteration is more expensive than the first one. First, one have to precisely formulate the formal properties the investigated code should meet; second, the investigation itself may consume significant computation time.

The third type of iteration deals with the final code. The C code is generated by PyPy from RPython sources and is subsequently compiled into the machine code. The machine code can be deployed to the intended target embedded device. The final code on the final hardware can be subject of various tests, for instance performance test.

The first type of iteration is very cheap and can be performed with high frequency. The second and the third type tend to be costly. However, their particular cost depends on the nature of the application and conditions. One can have a cheap set of semi-formal tests that can be performed frequently and a very complicated way how to test on the target hardware. And vice versa, one can have a very costly set of formal tests that require hours of computing and an efficient way how to test on the target hardware.

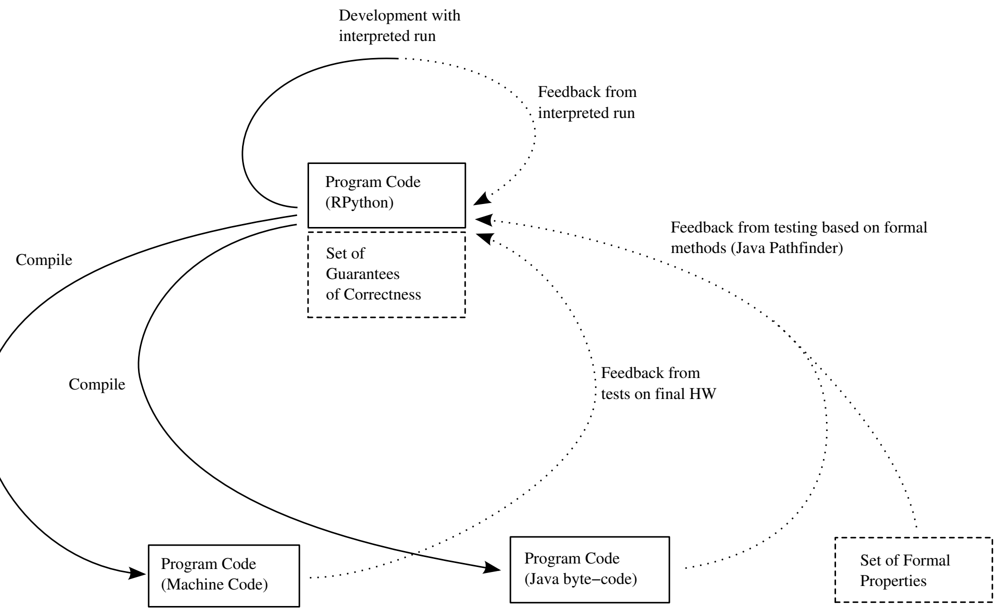
Development with interpreted run
Feedback from interpreted run
Program Code (RPython)
Set of Guarantees of Correctness
Feedback from testing based on formal methods (Java Pathfinder)
Compile
Compile
Feedback from tests on final HW
Program Code (Machine Code)
Program Code (Java byte–code)
Set of Formal Properties

Figure 3: Refined Scheme of the Proposed Development Approach

4 Installation

The PyPy-based tool-chain with *framework-parlib* should run on all contemporary 32bit x86 Linux distributions. The presented installation procedure is created for 32bit release of Debian 6.0 ”Squeeze” and 32bit release of Ubuntu 12.04 LTS ”Precise Pangolin” operating systems.

The installation procedure consists of two steps. The first step installs the necessary development packages from the standard distribution repositories. The second step downloads and builds the source code related to PyPy and *framework-parlib*.

Each of the two steps can be performed by execution of one bash script. This section is merely an explanation of these scripts.

4.1 Installing System Prerequisites

The script that installs all required development packages to Debian or Ubuntu can be obtained from this URL:

`https://github.com/paskma/scripts/raw/master/install_framework_parlib_prerequisites.sh`

It can be downloaded and executed by this sequence of commands:

```
wget https://github.com/paskma/scripts/raw/master/install_framework_parlib_prerequisites.sh
chmod +x install_framework_parlib_prerequisites.sh
./install_framework_parlib_prerequisites.sh
```

Tip: use the `-y` option and you will not be bothered by confirmation questions of the package manager.

After execution of the script, all system-wide tools needed for our tool-chain ought to be installed.

4.1.1 Content of the Script

Now let us explain the content of the script.

```
apt-get install git-core
apt-get install subversion
```

Installs the *Git* and *Subversion* source code management tools. They are need in order to get various source codes from repositories.

```
apt-get install g++
```

Installs C and C++ compiler. Needed in order to generate the machine code from the C code.

```
apt-get install libgc-dev
```

Installs the Bohem garbage collector. This GC is used by the machine code produced by PyPy.

```
apt-get install openjdk-6-jdk
```

Installs the open source Java development kit. This is needed in order to compile *framework-parlib* Java binding and to run Java Pathfinder experiments.

```
apt-get install unzip
```

Some packages are available as zip archives, we need to inflate them.

4.2 Installing PyPy-standalone-compiler, framework-parlib and Java Pathfinder

The second script installs and builds a bunch of source codes from various internet sources. The script can be obtained from this URL:

```
https://github.com/paskma/scripts/raw/master/install_framework_parlib.sh
```

Download the second script into a directory to which you want all the software to be installed. The directory should be somewhere in your home directory, for instance *$HOME/projects* makes the perfect sense. The script can be downloaded and executed by these commands:

```
wget https://github.com/paskma/scripts/raw/master/install_framework_parlib.sh
chmod +x install_framework_parlib.sh
./install_framework_parlib.sh
```

After execution of this script, all tools that we need for creating efficient and dependable applications based on Python ought to be installed.

4.2.1 Content of the Script

```
INSTALL_ROOT=‘pwd‘
```

We need to set this environment variable, the installation root is the same as the current directory.

```
git clone https://github.com/paskma/pypy-sc.git
```

The first command obtains a copy of the PyPy-standalone-compiler. This is our fork of the PyPy experimental implementation of the Python programming language.

```
svn co http://codespeak.net/svn/py/dist/py@60839
ln -s ../py pypy-sc/py
```

This downloads an additional library needed by the PyPy compiler, it is simply called *py*. A symbolic link connects the library with PyPy.

```
git clone https://github.com/paskma/framework-parlib.git
```

This command gets the framework for the intended applications. This code contains mainly an API for unified access to the resources of the underlying platform (Python interpreter, Linux operating system, Java virtual machine). Notable part of the API deals with thread synchronization. The package also includes various example applications and compilation scripts.

The very next thing what we need is an older standard Python implementation (CPython), literally 2.4.4. This Python installation is needed solely for the PyPy compiler. Interpreted tests of application based on *parlib* can be run on the standard (and newer) Python interpreter from the opraráting system.

Moreover, we need to add one patch to the Python interpreter. The default output of the PyPy compiler is randomized: many identifiers contain an integer whose value might change between compilations. Our patch removes this unpleasant feature. It is obvious that the compilation itself works also without this patch.

```
wget http://www.python.org/ftp/python/2.4.4/Python-2.4.4.tgz
tar xf Python-2.4.4.tgz
cd Python-2.4.4
patch -p1 < ../framework-parlib/misc/disable_hash.patch
./configure --prefix=$INSTALL_ROOT/python24
make
make -i install
cd ..
```

This set of commands downloads, unpacks, patches and installs the Python interpreter.

```
wget -O jasmin-2.2.zip \
   http://sourceforge.net/projects/jasmin/files/jasmin/jasmin-2.2/jasmin-2.2.zip/download
unzip jasmin-2.2.zip
```

To produce Java byte-code, PyPy needs a Java assembler called *Jasmin*.

```
svn checkout \
   https://javapathfinder.svn.sourceforge.net/svnroot/javapathfinder/trunk -r 1790 jpf_trunk
cd jpf_trunk
java RunAnt run-tests
cd ..
```

The dependability of the final programs is achieved through the employment of an explicit model checker called Java Pathfinder. These commands download a copy of the tool and compile it. To make sure it works properly we run all the unit tests.

```
echo "PARLIB_FRAMEWORK_ROOT=\"$INSTALL_ROOT/framework-parlib\"" > framework-parlib/environment.sh
echo "PYPY_ROOT=\"$INSTALL_ROOT/pypy-sc\"" >> framework-parlib/environment.sh
echo "PYTHON_BIN=\"$INSTALL_ROOT/python24/bin/python\"" >> framework-parlib/environment.sh
echo "JPF_ROOT=\"$INSTALL_ROOT/jpf_trunk\"" >> framework-parlib/environment.sh
echo "JASMIN_JAR=\"$INSTALL_ROOT/jasmin-2.2/jasmin.jar\"" >> framework-parlib/environment.sh
```

The compilation scripts of the *framework-parlib* need to know the location of the PyPy standalone compiler, Python 2.4, Java Pathfinder and the Jasmin Java assembler. The locations are saved in a file called *environment.sh.* The installation script creates this file.

```
cd framework-parlib/binding/c
./compile.sh
cd ../../..
```

These commands builds a small native library that handles access to the operating system.

```
# We need to make user execute this command in his shell:
#export PATH="$INSTALL_ROOT/framework-parlib/bin:$PATH"

echo "Execute the following command in your shell:"
echo -n 'export PATH="'
echo -n $INSTALL_ROOT
echo '/framework-parlib/bin:$PATH"'
```

The compilation scripts of the *framework-parlib* are located in the directory *framework-parlib/bin.* We recommend to add this directory to your *PATH* environment variable. The installation script can not modify the environment variable, it only prints the recommended command that you can execute manually.

5 Building an Example Program

The *framework-parlib* comes with several example programs that are located in *framework-parlib/experiments.*

Before you start to play with the examples, you need to set your *PATH* environment variable to contain the path where the translation scripts are located, i.e., *framework-parlib/bin.* The exact form of the command that performs this action was printed by the second installation script. On the author's computer, the command looks like this:

```
export PATH=/home/paskma/projects/framework-parlib/bin:$PATH
```

After setting the *PATH*, we can run all the compilation scripts, literally: *par_run_pure.sh*, *par_compile_c.sh*, *par_run_c.sh*, *par_compile_j.sh*, *par_run_j.sh* and *par_check.sh.*

We can try to play with examples. The simplest example is located in *framework-parlib/experiments/hello_world.* After switching to the directory, we can directly run this program by the Python interpreter, this is done by the *par_run_pure.sh* script. A dollar sign states for the command prompt.

```
$ pwd
/home/paskma/projects/framework-parlib/experiments/hello_world
$ par_run_pure.sh
Hello, world!
```

This program simply writes the traditional "Hello, world!" message.

Now, we can translate the program to C and build a native version. This is done by the following command:

```
$ par_compile_c.sh
```

If the compilation succeeds, we can run the native version.

```
$ par_run_c.sh
Hello, world!
```

We can produce the Java version of the program in the very similar way:

```
$ par_compile_j.sh

$ par_run_j.sh
Hello, world!
```

The java version can be checked by Java Pathfinder. This is done by the *par_check.sh* command:

```
$ par_check.sh
Using directory /home/paskma/projects/jpf_trunk/hello_world
JavaPathfinder v4.1 - (C) 1999-2007 RIACS/NASA Ames Research Center

================================== system under test
application: pypy/Main.j

================================== search started: 7/26/12 1:10 PM
Hello, world!

================================== results
no errors detected
```

As we can see, this trial program is really defect-free.

Note that not all of the example programs are directly usable. In some of them (*testcase_deadlock*, *testcase_race*, *testcase_ltl*, *testcase_exception*, *testcase_random*), you have to choose between a correct implementation and an implementation with an injected bug. In that case, there are two scripts in the example's directory: *go_good.sh* and *go_bad.sh.* You have to execute one of them before you compile or run the code.

```
class Resource(Monitor):
    def __init__(self):
        Monitor.__init__(self)
        self.secondLevel = None

    def setSecondLevel(self, secondLevel):
        self.secondLevel = secondLevel

    @synchronized
    def cascadeLock(self):
        print "First level locked"
        sleep(1.0)
        self.secondLevel.lockSecondLevel()

    @synchronized
    def lockSecondLevel(self):
        print "Second level locked"
```

Figure 4: Deadlock Test Case: Class Resource

6 The Deadlock Experiment

This experiment lives in the *framework-parlib/testcase_deadlock* directory. Remember that before you start playing with it, you have to set the *PATH* environment variable correctly and use either *go_good.sh* or *go_bad.sh* script.

Deadlock is a program state in which two or more threads are waiting to each other and thus neither ever finishes. In our case, we have a program that has two worker threads that are locking two resources, *resA* and *resB*. A deadlock is possible if the first thread has locked *resA* and is trying to lock *resB* meanwhile the second thread has locked *resB* and is trying to lock *resA*.

As we do not use plain locks but structured monitors, we connected the resources into a chain so that locking of the first resource causes locking of the second resource and vice versa. See the *Resource* class listing in figure 4: the synchronized method *cascadeLock* calls the synchronized method *lockSecondLevel*. Note that there is a one second sleep between the acquisition of the first and the second resource.

Another component of the testing program is the worker thread. This class is rather simple; it has one resource associated and calls its *cascadeLock* method, see figure 5.

The entry point of our program is a method called *main* of the *Application* class, see figure 6. The method creates the resources, makes the chain of them, creates and starts the worker threads, and waits until they are finished.

```
class Worker(Thread):
    def __init__(self, resource):
        Thread.__init__(self)
        self.resource = resource

    def run(self, *args):
        self.resource.cascadeLock()
        print "Thread finished without deadlocking."
```

Figure 5: Deadlock Test Case: Class Worker

```
class Application:
    def main(self, argv):
        resA = Resource()
        resB = Resource()
        resA.setSecondLevel(resB)
        resB.setSecondLevel(resA)

        w1 = Worker(resA)
        w2 = Worker(resB)

        w1.start()
        w2.start()

        w1.join()
        w2.join()
        return 0
```

Figure 6: Deadlock Test Case: Class Application (with a bug)

```
thread index=0,name=main,status=WAITING,
  this=java.lang.Thread@0,target=null,priority=5,lockCount=1
  waiting on: pypy.worker.Worker_59@728
  call stack:
at pypy.worker.Worker_59.ojoin(Worker_59.j:403)
at pypy.application.Application_55.omain(Application_55.j:138)
at pypy.main_54.invoke(main_54.j:33)
at pypy.entry_point_50.invoke(entry_point_50.j:22)
at pypy.Main.main(Main.j:40)

thread index=1,name=Thread-0,status=BLOCKED,
  this=pypy.worker.Worker_59@728,priority=5,lockCount=0
  owned locks:pypy.resource.Resource_56@711
  blocked on: pypy.resource.Resource_56@715
  call stack:
at pypy.cascadeLock_66.invoke(cascadeLock_66.j:49)
at pypy.resource.Resource_56.ocascadeLock(Resource_56.j:52)
at pypy.worker.Worker_59.orun(Worker_59.j:109)
at pypy.worker.Worker_59.oRUN(Worker_59.j:163)
at parlibutil.ThreadStarter.run(ThreadStarter.java:17)

thread index=2,name=Thread-1,status=BLOCKED,
  this=pypy.worker.Worker_59@755,priority=5,lockCount=0
  owned locks:pypy.resource.Resource_56@715
  blocked on: pypy.resource.Resource_56@711
  call stack:
at pypy.cascadeLock_66.invoke(cascadeLock_66.j:49)
at pypy.resource.Resource_56.ocascadeLock(Resource_56.j:52)
at pypy.worker.Worker_59.orun(Worker_59.j:109)
at pypy.worker.Worker_59.oRUN(Worker_59.j:163)
at parlibutil.ThreadStarter.run(ThreadStarter.java:17)

====================================================== results
error #1: gov.nasa.jpf.jvm.NotDeadlockedProperty
   "deadlock encountered:   thread index=0,name=main,s..."
```

Figure 7: JPF Report: Stack Trace of a Deadlock

Now we can run the program. Because of the *sleep* in the worker thread, the deadlock will occur with very high probability. If we run the program in all three environments, we always see, that the program hangs, the output of the native version is like this:

```
$ par_run_c.sh
First level locked
First level locked
```

Both threads just lock the first resource (from their perspective) and then fail to lock the other. The program never terminates.

If the test program is run by JPF, the bug is quickly discovered and a report is produced, see figure 7. The report shows that the first worker thread owns the lock of the resource object with id 711 and is blocked on the lock of the resource object with id 715. The other worker thread owns resource 715 and is blocked on resource 711.

```
class Application:
    def main(self, argv):
        resA = Resource()
        resB = Resource()
        resA.setSecondLevel(resB)
        resB.setSecondLevel(resA)

        w1 = Worker(resA)
        w2 = Worker(resA)

        w1.start()
        w2.start()

        w1.join()
        w2.join()
        return 0
```

Figure 8: Deadlock Test Case: Class Application (fixed)

```
====================================================== results
no errors detected

====================================================== statistics
elapsed time:        0:00:01
states:              new=191, visited=210, backtracked=400, end=5
search:              maxDepth=18, constraints=0
choice generators:   thread=191, data=0
heap:                gc=484, new=1534, free=119
instructions:        52790
max memory:          9MB
loaded code:         classes=130, methods=1682
```

Figure 9: JPF Report: No Deadlock after the Fix

One of the ways how to avoid a deadlock is to add a global order of the locked objects. In our case, the deadlock can not occur if both threads first lock *resA* and then *resB*. You can see the fixed method *main* in figure 8; both constructors of the worker threads take *resA* as a parameter.

With this modification, the output of the program is as follows:

```
$ par_run_c.sh
First level locked
Second level locked
Thread finished without deadlocking.
First level locked
Second level locked
Thread finished without deadlocking.
```

The program works properly in all three environments. However, to prove that the fix is correct, we utilize JPF again, see report in figure 9.

7 Conclusion

The main purpose of this document is to present the installation procedure. This document contains only the basic information about the development process that leverages the installed tools. You are invited to read the doctoral thesis [1] where we present our motivations, explain why we chose the Python programming language and PyPy, and analyze the internals of the translation process; see also [6] and [7].

References

[1] M. Paska, *Development of Dependable and Efficient Software with Dynamically-typed Languages*, [Submited], Ph.D. thesis, University of West Bohemia, Pilsen, Czech Republic, 2012.

[2] A. Rigo, M. Hudson, S. Pedroni, *Compiling Dynamic Language Implementations*, IST FP6-004779, http://codespeak.net/svn/pypy/extradoc/eu-report/D05.1_Publish_on_translating_a_very-high-level_description.pdf, 2005.

[3] W. Visser, K. Havelund, G. Brat, S. Park, F. Lerda, *Model Checking Programs*, Automated Software Engineering Journal, Volume 10, Number 2, 2003.

[4] K. Havelund, T. Pressburger, *Model Checking Java Programs Using Java PathFinder*, International Journal on Software Tools for Technology Transfer, Vol. 2, No. 4. http://ase.arc.nasa.gov/people/havelund/Publications/jpf-sttt.ps, 2000.

[5] D. R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley. ISBN 0-201-63392-2, 1997.

[6] M. Paška, *Generative Programming with Support for Formal Verification*, 2009 IEEE International Symposium on Industrial Embedded Systems, Ecole Polytechnique Fédérale de Lausanne, Switzerland, July 8 - 10, 2009, IEEE Catalog Number CFP09INB-USB, ISBN 978-1-4244-4110-5, Library of Congress 2009901328, 2009.

[7] M. Paška, *An Approach to Generating C Code with Proven LTL-based Properties*, EUROCON 2011, ISBN: 978-1-4244-7485-1, IEEE Catalog Number CFP11EUR-CDR, Lisbon, Portugal, 2011.