University of West Bohemia
Faculty of Applied Sciences

DEVELOPMENT OF DEPENDABLE AND EFFICIENT SOFTWARE WITH DYNAMICALLY-TYPED LANGUAGES

Ing. Marek Paška

doctoral thesis
in fulfillment of the requirements for the degree of
Doctor of Philosophy
in the specialization of
Computer Science and Engineering

Supervisor: Doc. Ing. Stanislav Racek, CSc.
Department of Computer Science and Engineering

Pilsen 2012

Západočeská univerzita v Plzni
Fakulta aplikovaných věd

VÝVOJ SPOLEHLIVÉHO A EFEKTIVNÍHO SOFTWARU V DYNAMICKY TYPOVANÝCH JAZYCÍCH

Ing. Marek Paška

**disertační práce k získání akademického titulu doktor
v oboru informatika a výpočetní technika**

Školitel: Doc. Ing. Stanislav Racek, CSc.
Katedra informatiky a výpočetní techniky

Plzeň 2012

Department of Computer Science and Engineering,
Faculty of Applied Sciences, University of West Bohemia,
Univerzitní 22, 306 14 Plzeň, Czech Republic
Phone: (+420) 377 63 2401,
Fax: (+420) 377 63 2402
URL: http://www.kiv.zcu.cz/

Declaration of Authenticity

I hereby declare that this doctoral thesis is my own original and sole work. Only the sources listed in the bibliography were used.

Prohlašuji tímto, že tato disertační práce je původní a vypracoval jsem ji samostatně. Použil jsem jen citované zdroje uvedené v přehledu literatury.

....
Marek Paška

Abstract

Computers are ubiquitous; this is especially true in the case of so called embedded devices.

These devices usually have constrained computational resources. Software development for such systems tends to be conservative and use tools such as C programming language. More recently, there is a notable inclination towards Java. Embedded systems have also increased dependability requirements that lead to adoption of formal methods.

In this work, we try to bring the power of dynamically-typed languages to the embedded system development. These languages have higher level of abstraction than Java and due to their flexibility are able to embrace new paradigms such as Aspect Oriented Programming.

We propose a software development process based on the Python programming language and its advanced compiler called PyPy. We enable to create rapid prototypes in Python that are then translated to the efficient machine code.

Last but not least, our development process also presents advanced testing based on formal methods. From the Python code, we also generate the Java byte-code that is then investigated by Java Pathfinder which is an explicit model checker.

Our development approach proved to be viable on a couple of case studies.

Keywords: embedded devices, model checking, Python, PyPy, generative programming, software development process, linear temporal logic

Abstrakt

Počítače jsou dnes všudypřítomné, což platí především pro takzvané vestavěné systémy.

Tato zařízení obvykle mají omezenou výpočetní kapacitu. Vývoj softwaru pro takové systémy je často konzervativní, používá prostředky jako například jazyk C. V posledních letech je pozorovatelný příklon k jazyku Java. U vestavěných systémů je požadována velká spolehlivost, což vede k využívání formálních metod.

V této práci se snažíme přinést sílu dynamicky typovaných jazyků do oblasti vývoje vestavěných systémů. Tyto jazyky mají vyšší míru abstrakce než například Java a díky své flexibilitě jsou schopny absorbovat nová paradigmata jako například aspektově orientované programování.

Navrhujeme vývojový proces založený na programovacím jazyku Python a překladači PyPy. Díky Pythonu můžeme rychle vytvářet prototypy, které se potom přeloží do efektivního strojového kódu.

Náš vývojový proces obsahuje i pokročilé testování založené na formálních metodách. Z kódu v Pythonu můžeme vygenerovat Java bajtkód, který potom zkoumáme nástrojem Java Pathfinder, což je explicitní model checker.

Životaschopnost našeho procesu jsme demonstrovali na několika případových studiích.

Klíčová slova: vestavěná zařízení, model checking, Python, PyPy, generativní programování, vývojový proces, lineární temporální logika

Acknowledgments

I would like to express my gratitude to my supervisor Stanislav Racek who led me through my doctoral studies and shared his knowledge and experiences with me.

I would like to thank my brother Přemysl for improving the language side of this work. While I was working on this thesis, there were wonderful women around me: Anna and Lenka. Thank you!

This work would be impossible without all the free software such as GNU/Linux and Python that I used and learnt from.

Contents

Chapter 1

Introduction

As computer systems become cheaper and more reliable, they are utilized in wider areas of human activity. The computers also become more diverse; the word "computer" no longer describes only the workstation on the table, but also the computer in cell a phone, car, camera and a coffee maker.

These ubiquitous computers are usually called embedded systems. Developing software for such systems involves dealing with a number or specific constrains, mainly computing resources limitations (CPU and memory).

There is a very strong need for dependability of such systems. Embedded systems are sometimes used in safety-critical applications, e.g., in aerospace field. Dependability is emphasized even in common applications where correcting an error via some kind of software update may be very complicated.

Formal methods can substantially contribute to the reliability of embedded systems. However, the state of the practice has been usually behind the state of the art of formal methods [1]. Formal methods are still considered hard by developers as they require special languages and tools. Thus it is important to make use of formal methods as easy as possible.

Mentioned requirements and constraints have crucial impact on the software development process. It tends to be conservative, traditional programming languages are C and assembly, more recently, Java. On the other hand, state-of-the-art approaches are also used, mainly model driven development.

In this work, we propose a novel approach to dependable and efficient software development. It relies on a programming language with very high level of abstraction. We use the code generation as the main principle. The code generation allows us to create very compact machine code from the high level description. Finally, we try to make usage of formal methods, literally explicit model checking, as easy as possible.

1.1 Goals of the Thesis

The overall goal of the thesis is to design a new development approach for dependable software development and prove that it is viable. The new approach should produce efficient and reliable programs with less effort via embracing modern programming paradigms. It is aimed at embedded devices, however not limited to this domain. More precisely, the goals can be summed up into the following points:

- Select a high level programming language that plays well with best software engineering practices, such as Aspect Oriented Programming (AOP) and Domain Specific Languages (DSL). Select also development tools that enable a high level code to be translated to more efficient low level machine code.
- Design a verification procedure that will earn a set of guarantees of correctness of the production code. The verification procedure should be easy to use.
- Show that the generated machine code is suitable for running on an embedded device in terms of memory consumption and computational efficiency.

1.2 Outline of the Thesis

The first chapter ends with definition of some basic terms. Chapter 2 describes embedded devices and properties of embedded software. Chapter 3 was written because we want the whole thesis to be self-contained. It recapitulates programming paradigms, formal verification, and other notable technologies.

Chapter 4, "Towards High Level Dynamic Approach", is one of the most important, it describes why and how dynamic languages should contribute to the dependable software development.

Chapter 5, is core of this thesis. It proposes a development approach based on thoughts from the previous chapter.

A fairly deep analysis of the technologies that we selected for our approach is contained in chapter 6. Chapter 7 describes our customizations of the selected tool-chain.

In chapter 8, we propose a testing procedure based on formal methods that fits our development approach. In chapter 9, we present a set of benchmarks that prove that our code is efficient enough. Chapter 10 contains two case studies, one of them is a real-world program. Chapter 11 concludes the achievements of the thesis.

1.3 Terminology

Through all this work we deal with many terms that are worth clarifying at the very beginning. These definitions are mostly based on [2].

Correctness is a system's ability to perform according to its specification in causes of use within that specification.

Robustness is a system's ability to prevent damage in cases of erroneous use outside of its specification.

Security is a system's ability to prevent damage in cases of hostile use outside of its specification.

Verification is internal assessment of the consistency of the product considered just by itself. Verification is about ascertaining that the product is "doing things right".

Validation is relative assessment of a product vis-à-vis another that defines some of the properties that it should satisfy: code against design, design against specification, specification against requirements, documentation against standards. Validation is about ascertaining that it is "doing the right thing".

Dependability comprises at least three factors: correctness, robustness and security. For our purposes *dependable software* means the same as *reliable software.*

Safety-critical system is a system whose failure or malfunction may result in: death or serious injury to people, or loss or severe damage to equipment or environmental harm.

Chapter 2

Embedded Systems

Our work is not limited to embedded systems; however, software for embedded systems is huge motivation for writing efficient and dependable software.

2.1 Characteristics and Classification

No single characterization applies to the diverse spectrum of embedded systems. Embedded systems are usually special-purpose systems in which the CPU and all the required secondary resources are bundled on a small factor printed circuit board or even on the same chip. The expression "embedded" is used to designate a computer system hidden inside a product other than a computer [4].

Some combination of cost pressure, long life-cycle, real-time requirements, and reliability requirements can make it difficult to be successful applying mainstream computer design methodologies and tools for embedded applications [5]. The reliability requirements are imposed due to the fact that embedded systems typically have to work without human intervention; in fact they are often designed to substitute supervision of a human operator.

For the purpose of this work, we provide a short overview of some embedded system classes [6].

Very small systems (i):

- 4 or 8-bits micro-controllers with no OS-like environment.
- Can be found in many every day devices (from coffee machines to cars).
- Main design constraints are cost, then reliability.
- Development is done mainly in C and assembly.

Micro-controllers (ii):

- 8, 16 or 32-bit micro-controllers possibly with very small OS, still have very limited RAM, ROM and CPU power, and have no MMU[1].
- Development can be done with various tools and languages (C, C++, Java, Basic, assembly). There are also some custom languages.

Small systems with quite standard architectures (iii):

- System built around ARM, Freescale, Geode, etc. CPU acts like a small computer. Can run a complete OS (Linux, VxWorks, QNX, etc.).
- The limitations are small amount of RAM (compared to desktop computers), limited CPU, sometimes power consumption, etc.
- Very common in printers, network devices (routers), PDAs, GPS devices, cars.
- No main programming language, developers usually use standard Unix tools.

In this work, we deal mainly with classes (iii) and (ii).

The presented classification evolves in time as hardware is more and more powerful over time. However, the class of very small systems is not going to "extinct" because we can take advantage of extremely low power consumption to invent new applications, for example based on battery-free devices [3].

2.2 Software in Embedded Devices

An embedded software is a software that runs on an embedded computer. It is the ultimate source of flexibility and controllability of the embedded system [4].

While pieces of embedded software can vary significantly, depending on the purpose they are constructed for, there are some characteristics that are typical. The embedded systems usually does not have graphical user interface (GUI) that we know from personal computers. User interface (if any) is typically very limited. Note that in many desktop programs GUI-related code comprises a vast majority of the program code.

Embedded software development also tends to be conservative. Whereas programmers of desktop applications use new high level object oriented languages (e.g., Java, C#, Python) with features such as garbage collection, embedded

[1] Memory Management Unit

software development relies mostly on legacy tools such as plain C or assembly, despite the fact that performance of embeddable microprocessors grows for decades.

2.2.1 Reactive Systems

Embedded programs usually do not use the traditional operating sequence where input data are supplied to the program when it starts and output data are available when the program finishes its job. Embedded programs have to keep synchronized with external events from the environment where they are deployed; these systems are called *reactive*.

The reactive program usually consists of several tasks that acquire data from the external environment, do a computation, and then emit output data back. The tasks have typically form of (possibly infinite) loop.

According to [7], the reactive systems can be divided into two groups: *event-triggered* and *time-triggered*. Trigger is an event that causes execution of some program code.

Event-Triggered Systems: Events coming to the system at arbitrary time have to be handled properly. Events are connected with a significant change of the state of the environment and thus are asynchronous.

Time-Triggered Systems: The only assumed event is a periodical change of internal clock. When a certain time interval elapses, the state of the environment is acquired and appropriate actions are executed. Note that behavior of such systems is generally more predictable than in the case of the event-triggered counterparts.

2.2.2 Real-Time Systems

Many embedded systems can be also viewed as *real-time*. Correctness of operations of real-time systems depends, in part, on the time at which it is delivered [5].

We distinguish two main classes of real-time systems: *hard* and *soft* real-time systems.

Hard Real-Time Systems: The operation of a hard real-time system is firmly constrained in many ways. First of all, it *guarantees* the response time to be within certain bounded interval, often as tight as several milliseconds; this fact has major consequences:

- *Peak-load scenario* must be well-defined, i.e., the system meets the specified deadlines in extreme situations that may be very rare.
- To guarantee the real-time properties the design phase incorporates special methods such as *worst-case execution time analysis*. To make this analysis feasible, usage of dynamic data structures is limited.
- Safety-critical nature of many hard real-time systems implies that an error-detection must be autonomous and recovery actions must be well-defined.

Soft Real-Time Systems: In the case of soft real-time systems, the temporal properties are weakened and these systems are never safety-critical. The time when a result of a computation is delivered is still important; however, it is not strictly guaranteed. Soft real-time systems use *best effort* approach, i.e., the result is delivered as early as possible. Peak-load performance is not critical, because the system usually can slow-down the external environment, e.g., a human operator.

The data structures in soft real-time systems are less constrained and thus can be more sophisticated; the error-recovery can employ scheme of creating checkpoints and executing roll-back action when necessary.

2.2.3 Program Errors

Errors and Failures

Every software-related failure of a deployed embedded system is caused by a mistake or a bad design decision of a programmer/designer. Overall anatomy of a failure of a system is provided in [8]:

1. *Error:* An omission, a mistake, or a bad design decision of a programmer; may lead to:
2. *Defect (also Fault or Bug):* A Defect (or a bug) in a source code of the system; may lead to:
3. *Run-time Fault:* Invalid run-time state or output; may lead to:
4. *Failure:* Inability of the system to provide a desirable functionality and/or performance.

Failures can be characterized from many aspects. One of the characterizations can be found in [9]:

- *Static* versus *Dynamic*: A static failure (*value failure*) provokes a wrong result. A dynamic failure (*timing failure*) provokes a transient response which is incorrect, either too fast or too slow.
- *Persistent* versus *Temporary*: A persistent failure alters the behavior of a system for a significant portion of mission time. On the contrary, a temporary failure alters the behavior at a certain moment.
- *Consistent* versus *Inconsistent*: A consistent failure is perceived in the same way by all users of a system; the failure is said to be inconsistent in the opposite case.

A failure of an embedded system may have severe consequences or, on the other hand, may have no consequences at all. For example, the aircraft industry is recommending a fault categorization of safety-critical systems according to the following criteria [10].

1. *Catastrophic:* Fault that prevents continued safe operation of the system and can be the cause of an accident.
2. *Hazardous:* Fault that reduces the safety margin of the redundant system to an extent that further operation of the system is considered critical.
3. *Major:* Fault that reduces the safety margin to an extent that immediate maintenance must be performed.
4. *Minor:* Fault that has only a small effect on the safety margin. From the safety point of view, it is sufficient to repair the fault at the next scheduled maintenance.
5. *No Effect:* Fault that has no effect on the safety margin.

Fail-Safe and Fail-Operational Systems

Fail-safeness is a characteristic of the controlled object, not the controlling system. That means, when an error is detected, the controlled object can reach a *safe state* where the failure of the computer system have no consequences. Consider an example of a railway signalling system: the safe state is when all trains are stopped and the state can be easily reached by setting all the signals to red.

On the contrary, the example of a controlled object that cannot reach a safe state easily is a flying plane. The flight control system must always provide some minimal functionality, even under error occurrences.

Chapter 3

Related Work

In this chapter, we present an overview of technologies that provide background for design of our own development approach. The topics that are familiar to reader can be skipped.

3.1 Static Verification

Static verification is a set of processes that analyzes a piece of software without actually executing programs built from that software.

3.1.1 Type Checking

Data types are attributes of pieces of data that determine how the data are interpreted by the computer. They also determine a set of operations that can be done with the data.

The aim of *type checking* is to guarantee that the type structure of a computer program is valid, i.e., all operations performed on data are permitted by the type definitions.

Type Checking Approaches

Type checking is a processes of identifying errors in a program based on explicitly or implicitly stated type information.

In *static type checking*, the type information is associated with variables; the type is usually determined when the variable is declared. As the types are directly apparent in the program source code, type correctness can be checked during compilation. That is, a compiler ensures that operations only occur on operand types that are valid for the operation. This early error detection prevents programmer from significant class of errors. Many wide-spread languages employ static type checking: C, C++, Java, Ada.

In contrast, there is also *dynamic type checking* in which the type information is associated with a particular *value* of a variable rather than with the variable itself. As the variable changes values at run-time, the type of the variable may be changed too. Thus the type correctness can be reliably checked only at run-time.

Programs with static type structure are less flexible, there is a trade-off between early error detection and higher flexibility. Typical languages with dynamic type checking are Python, Ruby, and Smalltalk.

Type system is considered *weak* when distinction between types is weakened by automatic conversions. In a weakly typed language, a programmer can mix variables of different and incompatible types in a single expression, because the types of variables can be automatically converted when needed. For instance, it is possible to 'add' (by operator "+") two objects of different types: an integer of value 10 and a string of value "50". One of the operands have to be converted to the type of the other operand. So the result of the operation can be either an integer of value 60 or a string of value "1050". Automatic conversions are usual in text-processing languages like Perl or PHP.

Type Checking in Ada

Ada is imperative, statically typed, object oriented, general-purpose programming language. It was designed for United States Department of Defense to be universally used for variety of applications at the department [11].

Ada is designed for large, long-live programs, with mission- or safety-critical applications in mind. Ada puts strong emphasis on static checking. The compiler checks whatever is feasible to check at compile time, e.g., type correctness, variable scopes, pointer scopes, and array indices. One of the design goals is memory safety, that means, a direct access to the memory is prohibited. Ada is also known for one of the most advanced type systems: it includes subtypes, integrity checks and operator overloading.

Although Ada programming language is rather complex, the native code produced by its compiler is compact and efficient. This makes Ada very suitable tool for embedded software development even in challenging areas of avionic and space applications [12].

User-defined Types

Programming languages like C, C++, and Java allow to create user-defined types, e.g., structures and classes. However, there is no way how to create prim-

```
  type Apples is new Integer;
  type Oranges is new Integer;
```

Figure 3.1: Elementary Ada Types

```
declare
  Apple_Count : Apples := 10;
  Orange_Count : Oranges;
begin
  Orange_Count := Apple_Count; --yields a compilation error
  Orange_Count := Oranges(Apple_Count);
                              --explicit typecast is OK
end;
```

Figure 3.2: Typecasting in Ada

itive data types like integer or floating-point numbers with user-defined semantics. In Ada, one can create user-defined types that have the same capabilities as built-in types.

Figure 3.1 shows a definition of two new types: *Apples* and *Oranges*. These new types inherit semantics from the built-in type *Integer*. Note that *Apples* and *Oranges* are completely independent types, they only share inner binary representation with the *Integer* type. Compare with C/C++ approach where the keyword *typedef* only creates a new name for an existing type.

Ada compilers guarantee that we cannot accidentally mix apples and oranges anywhere in the program. Explicit typecast is however possible, see figure 3.2.

If we had some physical computation in our program, it would be useful to have a type structure that corresponds with used physical laws. For example, when computing some *area*, suitable types are *Meters* and *Square_Meters*. Ada compiler can then check that *areas* and *lengths* are never confused.

```
type Meters is new Float;
type Square_Meters is new Float;

function "*" (Left, Right : Meters)
  return Square_Meters is
begin
  -- Multiplication is done on Float basis _
  -- to avoid recursive definition.
  return Square_Meters(Float(Left)*Float(Right));
end;

declare
  Width : Meters := 5.2;
  Height : Meters := 7;
  Area : Square_Meters;
  Bad_Area : Meters;
begin
  Area = Width * Height; -- This is OK.
  Bad_Area = Width * Height -- This yields a compilation error.
end
```

Figure 3.3: Physical Computation

```
subtype Angle is new Float range 0.0 .. 2.0 * pi;
```

Figure 3.4: Subtype Example

In Ada, one can also define semantics for user defined types. In our example, when we multiply *Meters* by *Meters*, the type the result is *Square_Meters*. This is done by appropriate overloading of the multiplication operator for *Meters*, see figure 3.3.

Integrity Checks

Ada types also employ value constraints, i.e., a type bears information about range of values it can contain. This is often combined with Ada subtypes. A subtype is derived from an arbitrary type and has a constrained range of values. A variable of a certain subtype can be always assigned to the variable of the type it was derived from. Contrary, when assigning variables from the supertype to the subtype, explicit typecast must be inserted. See figure 3.4 for an example of the *Angle* subtype that can be always assigned to a variable of the *Float* type.

Conclusion

Static type checking is a powerful technique, the main advantage is an early error detection. It is relatively easy to implement and thus it is widely used in mainstream languages such as Java or C++ as well as in languages for safety-critical domain, e.g., Ada.

Although the type correctness of a program is essential, it does not imply that the program run-time behavior is correct as well. Wrong explicit typecast may raise a run-time exception whose correct handling is generally not guaranteed by compile-time check.

3.1.2 Formal Verification Theory

Formal verification is a process where mathematically-based methods are used in order to prove that a certain system, e.g., a computer program, has a desired set of properties. Formal methods are considered "hard" and "expensive" as they require special tools and skills. For our purposes, the most notable formal approach is model-checking.

The Model-Checking Problem

Model-checking is a process of checking whether a given system (e.g., a finite state system) is a model of a given logic formula. The process is done by enumeration (explicit or implicit) of all the states reachable by the system and the behaviors that traverse through them [13].

Input to the model checking-process is:

- A model, e.g., a finite state system M.
- A set of formulae $\phi = \{\varphi_1, ..., \varphi_n\}$ that specify a desired behavior (properties) of the model. The formulae can be expressed for instance by linear temporal logic (LTL).

The model-checking process examines whether M satisfies ϕ, i.e., $M \models \phi$. The result is thus a *yes/no* answer.

A valuable aspect of model-checking is that if M does not satisfy ϕ, the procedure provides a counterexample, i.e., a state sequence from the initial state of the examined system to the state that violates the demanded property.

Kripke Structure

A system that we are going to verify is usually a piece of software (a program). Programs are not directly verifiable by the model-checking because they are typically too complex. For instance, they have infinitely many states. In order to make the model-checking feasible, the examined system has to be represented in a more compact and abstract form. The common approach is to represent the system as the *Kripke structure* [13]. It comprises of states, state transitions, and set of propositions associated with each state. The same propositions are used in formulae that describe properties of the structure. Formal definition follows:

Kripke structure over a set of propositions $P = \{p_1, ..., p_n\}$ is a tuple $M = \langle S, R, L \rangle$ where

- S is a finite set of states,
- $R \subseteq S \times S$ is a set of directed edges,
- $L : S \to 2^P$ is a labeling function which labels each state with a (possibly empty) set of propositions.

For a vertex $s \in S$ with $L(s) = \{p_1, ..., p_m\} \subseteq P$ we say for each $p_i \in L(s)$ that p_i holds in s or shortly: $s \models p_i$.

The unlabeled structure $\langle S, R \rangle$ is a *transition system.* A *pointed Kripke structure* $\langle S, R, L, s_0 \rangle$ is a Kripke structure with a starting state $s_0 \in S$.

Linear Temporal Logic

Linear temporal logic (LTL) is a modal logic with modalities referring to time [14]. It is a subset of richer *Generalized Computational Tree Logic* (CTL*). Its atoms are atomic propositions reflecting the current state of a system.

A *model* for a temporal formula φ is an infinite sequence of states (i.e., a word)

$$\pi = \pi_0\pi_1\pi_2... \tag{3.1}$$

where each state π_i provides an interpretation for the atomic propositions mentioned in φ.

The set of LTL formulae is defined inductively starting from countable set of atomic propositions, Boolean operators, and the temporal operators **X** (Next) and **U** (Until):

$$\varphi := a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \boldsymbol{X}\varphi \mid \varphi\boldsymbol{U}\varphi \tag{3.2}$$

Given a model π, as above, we present an inductive definition for the notion of a temporal formula φ holding at a position $j \geq 0$ in π, denoted by $(\pi, j) \models \varphi$. For a state formula φ,

$$(\pi, j) \models \varphi \Longleftrightarrow \pi_j \models \varphi.$$

That is, we evaluate φ locally, using the interpretation given by π_j.

$(\pi, j) \models \neg\varphi \Longleftrightarrow (\pi, j) \not\models \varphi$
$(\pi, j) \models \varphi \wedge \psi \Longleftrightarrow (\pi, j) \models \varphi$ and $(\pi, j) \models \psi$
$(\pi, j) \models \boldsymbol{X}\varphi \Longleftrightarrow (\pi, j+1) \models \varphi$
$(\pi, j) \models \varphi\boldsymbol{U}\psi \Longleftrightarrow$ for some $k \geq j, (\pi, k) \models \psi$, and for every i such that $j \leq i < k, (\pi, i) \models \varphi$

We adopt standard abbreviations $\vee$, $\Rightarrow$, *true*, and $false$ for Boolean expressions. For convenience, we also define temporal operators **F** (in the future, eventually) and **G** (globally)

$(\pi, j) \models \boldsymbol{F}\varphi \Longleftrightarrow (\pi, j) \models \boldsymbol{true}\ \boldsymbol{U}\varphi$
$(\pi, j) \models \boldsymbol{G}\varphi \Longleftrightarrow (\pi, j) \models \neg\boldsymbol{F}\neg\varphi$

Classification of Temporal Properties

Linear temporal logic is defined over infinite sequences of states that correspond with computations. A *property* is a predicate on such sequences. It determines whether a sequence is acceptable (having the property) or unacceptable (not having the property).

Let a property Π be a set of infinite words. A property Π of the system is defined to be *specifiable by LTL* if there is an LTL formula φ such that $\pi \models \varphi$ if and only if $\pi \in \Pi$.

Consider, for example, a particular program that assigns an integer value to a variable x. Let Π be the property requiring that the value of x is monotonically increasing. Now, it is obvious that the sequence of states

$$\langle x : 0\rangle, \langle x : 1\rangle, \langle x : 2\rangle, \langle x : 3\rangle, ...$$

belongs to Π, whereas the sequence

$$\langle x : 0\rangle, \langle x : 2\rangle, \langle x : 1\rangle, \langle x : 0\rangle, ...$$

does not.

According to [15], temporal properties can be partitioned into two classes: *safety* and *liveness*. The classes can be informally characterized as:

- A *safety* property states that some bad thing *never* happens.
- A *liveness* property states that some good thing *eventually* happens.

Safety properties typically represent requirements that have to be continuously maintained by the system. For example, a safety property should specify the mutual exclusion: a lock is acquired at most by one thread. Liveness properties, on the other hand, represent requirements that need not hold continuously, but have to be *eventually* of *repeatedly* fulfilled. For example, it is guaranteed that one of the threads requiring a lock eventually acquires it.

More sophisticated hierarchical classification of temporal properties was defined in [14] where they are classified into six classes: *guarantee*, *safety*, *obligation*, *persistence*, *recurrence* and *reactivity*. Properties from particular classes can be intuitively viewed as making different claims about occurrences of "good" and "bad" things during the computation. Informal definition follows [16]:

- *guarantee*: something good happens at least once
- *safety*: something good always occurs (nothing bad occurs)
- *obligation*: conditional occurrence of a good thing
- *recurrence*: something good occurs infinitely many times

- *persistence*: something good occurs continuously from a certain point (bad things occur only finitely many times)

- *reactivity*: conditional occurrence of infinitely many good things

For example, suppose that x is a program variable and its value should be positive. Then $\boldsymbol{G}(x > 0)$ is a *safety* property that holds if x is always positive. Similarly, a *guarantee* property $\boldsymbol{F}(x > 0)$ holds if x is positive at least in one state of the computation.

Note that it is decidable whether a given LTL formula belongs to a particular class, though the decision requires exponential time.

Büchi Automaton

In order to decide whether an arbitrary sequence of states (a word) π satisfies a given formula φ, the formula is usually translated to an automaton. Note that the computation (and thus the word) can be *infinite* in general. LTL formulae can be more naturally translated into non-deterministic finite-state automata with a special acceptance condition—Büchi automata.

A *Büchi automaton* is a tuple $B = \langle Q, A, \Delta, q_0, F\rangle$ where:

- Q is a finite set of states,

- A is a finite set of labels,

- $\Delta \subseteq Q \times A \times Q$ is a labeled transition relation,

- $q_0 \in Q$ is the initial state,

- $F \subseteq Q$ is a set of accepting states.

The *execution* of the automaton B on an infinite word $\pi = \pi_0\pi_1\pi_2...$ over alphabet A is an infinite word $\sigma = q_0q_1q_2...$ over alphabet Q, such that: $(q_i, \pi_i, q_{i+1}) \in \Delta, \forall i \in \boldsymbol{N}$. An infinite word π over alphabet A is *accepted* by the automaton B, if there exists an execution of B on π where some element of F occurs infinitely often.

Further information on automata over infinite words can be found in [17]; an efficient algorithm for translation of LTL formulae to Büchi automata was published in [18] and [19].

Model Checking Algorithm

Once we have a system represented as a Kripke structure $K = \langle S, R, L, s_0 \rangle$ and an LTL formula φ specifying desired behavior (both over propositions $P = \{p_1, ..., p_n\}$) then we can check if $K \models \varphi$.

We construct a nondeterministic Büchi automaton $B_{\neg\varphi} = \langle Q, 2^P, \Delta, q_0, F \rangle$ accepting infinite words which are *not* models of formula φ. Then a product automaton $K \times B_{\neg\varphi}$ is constructed in the following way:

$K \times B_{\neg\varphi} = \langle S \times Q, 2^P, (s_0, q_0), \Delta', S \times F \rangle$ where $\Delta'((s, q), a) = \{(s_2, q_2) | a \in L(s), (s, s_2) \in R, q_2 \in \Delta(q, a)\}$.

A word that is accepted by the product automaton is a *counterexample*—a witness of the incorrect behavior of the system. If the language of the product automaton $K \times B_{\neg\varphi}$ is empty, then $K \models \varphi$ holds.

The State-space Explosion Problem

The major drawback of model-checking is that it scales badly. If a model size grows linearly, the state space of the model tends to grow exponentially; the problem is referred to as the *state space explosion.* The nature of the growth is given by the fact that every component added to the model multiplies the number of model states. For example:

- A variable of type 32-bit integer has 2^{32} possible states.
- Threads that can run in parallel are usually modeled by thread interleaving. The state model of the thread interleaving is created by the Cartesian product of the state models of the individual threads.

In spite of the fact that hardware resources (mainly memory) are cheaper and more powerful every day, even trivial models have to employ techniques to reduce the state explosion. The most notable are: *abstraction*, *partial order reduction*, and *slicing.*

Abstraction: Concrete data types, e.g., 32-bit integer, can be abstracted. Instead of storing the exact integer value, only a property of the value is stored, e.g., *Negative, Zero, Positive.* This can be done when a certain specification does not depend on an exact value of some data but instead depends only on the sign of the data.

Partial order reduction: When some state transitions are commutative, i.e., the same state is reached by different order of transitions, one of the equivalent paths can be omitted.

Slicing: A program P is reduced according to some statements of interest $C = \{s_1, ..., s_n\}$ in the following way: all statements of P that do not affect any of the statements in C are removed. If a property Π is affected only by the statements in C and if Π holds for a reduced version of P, it also holds for P.

3.1.3 Formal Verification in Practice

Manual Creation of a Formal Model

In order to perform the model-checking, a formal model (such as Kripke structure) of an examined system must be created.

The most straightforward possibility is to construct the model by-hand. This is usual in an early stage of the development: the model is constructed as a mock-up of the demanded product. The construction is usually done in a special-purpose language of a particular model checker, for instance the SPIN model checker uses Promela as the specification language. The main drawback of this approach is that the production code that is derived from the specification, does not necessarily preserve all the properties of the specification.

Promela (Process Meta Language) is a verification modeling language. It describes possibly large but finite state system that is to be verified by the SPIN model checker. The system can be concurrent; dynamic process creation is also supported. In Promela, inter-process communication can be done via channels that are either synchronous (i.e., randezvous) or asynchronous (i.e., buffered).

An example of a binary Dijkstra semaphore is shown in figure 3.5. The example consists of three user processes and one process that provides mutual exclusion. The communication is done synchronously via channel *semaphore*. Each *user* process has to receive the symbol p before it enters the critical section. When the *user* process is leaving the critical section, the symbol v is sent back to the *dijkstra* process. The *dijkstra* process controls the mutual exclusion by sending p symbol when semaphore is open and accepting v symbol when the semaphore is closed (*open == 0*). Note that a Promela process blocks whenever a non-executable statement is reached, e.g., an attempt to read from an empty channel, an attempt to write to a non-buffered channel nobody is attempting to read from, or a comparison expression that is evaluated to *false*.

Formal Model Extraction

A formal model can be also extracted from a program source code written in a general-purpose language. A major advantage of such an approach is that the

```
#define p      0
#define v      1
chan semaphore = [0] of { bit };

proctype dijkstra() {
        bool open = 1;
        do
        :: (open == 1) -> sema!p;  open = 0
        :: (open == 0) -> sema?v;  open = 1
        od
}

proctype user() {
        do ::
        semaphore?p;
            /*  critical section    */
        semaphore!v;
            /* non-critical section */
        od
}

init {
        run  dijkstra();
        run  user(); run  user(); run  user();
}
```

Figure 3.5: A Semaphore in Promela

properties that are verified in the model are also present in the program source code.

Example of such a tool is Bandera [20]. Bandera is a tool set for extracting a finite-state model from a Java source code. A finite-state model is represented in Bandera Intermediate Representation language that is further used for emitting input of a particular external model-checker, e.g. SPIN or SMV. The result of the external verification is then mapped back to the original program code.

The code translation to the language of a model-checker cannot be performed directly. The state space must be reduced in order to make the verification feasible. Bandera provides several optimizations for state space reduction, mainly abstraction and slicing.

Java Pathfinder 2

Java Pathfinder 2 (JPF2) [21] is an explicit model checker for Java developed at NASA. Its predecessor Java Pathfinder 1 [22] 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 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.

SPIN

SPIN [23] is an explicit model checker developed at Bell Laboratories. The verification is mainly focused on proving the correctness of inter-process communication.

SPIN checks a finite state model specified in the Promela language that was briefly described above. The specification of a valid behavior can be done by built-in Promela assertions as well as by LTL formulae.

The verification process can be done in two modes. In the first mode, simulation is performed: SPIN directly executes the specification. This may or may not discover assertion violation but cannot give a proof of correctness.

The second mode is a formal verification. The Promela model along with the LTL-based properties is translated to C code, i.e., a special-purpose model checker written in C language is generated. The model-checking process itself is performed by the native program that was compiled from the generated C sources. This approach allows SPIN to be very efficient and handle models with relatively large number of states.

3.2 Run-time Verification

Unlike the static verification, the run-time verification analyzes the software by evaluation of the run-time behavior of a program.

3.2.1 Testing

Software testing is a very general term for process of investigation quality of a software product. The very essential approach is to run a tested program with some prepared input data. After the program finishes, we compare the actual output data with the expected output data; when the two data sets are equal then the program passes the test.

Testing is a heuristic method; it should give a good confidence of program correctness but cannot provide a proof of correctness because testing of all combinations of input and preconditions is feasible only for trivial programs. There are plenty of ways how the program can be tested: from the mentioned essential test case to the fault injection techniques.

Important property of a test is *code coverage*, i.e., the portion of the code (measured in statements, paths, conditions, etc.) that is actually examined by the test. Safety-critical applications are often required to demonstrate 100% code coverage.

Basic testing methods:

- *Black-box testing*: the examination of the software functionality is done without any understanding how the internals behave. The only way how to investigate the correctness is to analyze the outputs of the program. The key for successful black-box testing is selecting the input data that has a chance to discover defects; there are many techniques dealing with this issue such as *boundary value analysis* or *model-based testing.*

- *White-box testing*: the examination is done with access to the internal data structures and algorithms of the tested system. The tests can be thus designed to satisfy some code coverage criteria. Apart from analyzing final output data, intermediate results of the computation can be analyzed. Moreover, intermediate data can be also altered (a *fault injection*), which is extremely useful for fault-tolerant systems development.

Testing can be viewed from many aspects. For example *unit tests* investigate the minimal software components (e.g., a class) whereas *integration tests* investigate composition of such components. The aim of *regression tests* is to discover bugs originated by unintended consequences of program changes during the development.

3.2.2 Simulation

Computer simulation is not only useful part of modeling in physics, chemistry and biology; a simulation can be also used for verification of software systems.

Simulation-based Testing

First of all, simulation approach can be used as an advanced testing technique useful for reactive embedded programs. A tested system is run without any modification, but instead of interacting with a real world environment, i.e., some physical process it controls, it interacts with a simulation of the environment. An advantage of this kind of black-box testing is that a simulation process is able to provide far more realistic data than simple hand-written tests.

Simulation-based Checking

Further step is to turn the investigated program into a simulation process as well. Example of such an approach can be found in [24] where a Java concurrent program is checked on top of J-Sim [25]. J-Sim is an object-oriented library for discrete-time process-oriented simulation, it was developed at University of West Bohemia.

J-Sim is capable to simulate a run of Java concurrent programs. In order to perform the simulation, a general Java source code must be transformed to a code that can be handled by J-Sim. Special conversion tool called J-SourceMorph is provided for the task [26]. The transformation is done in the following way: all thread-related interactions with JVM such as new thread creation or synchronizations (e.g., calls to *wait* and *notify* methods) are replaced by J-Sim equivalents. This turns a concurrent Java program into a J-Sim simulation process.

The program is then run in a simulation mode. It interacts with a model of Java threading subsystem and with a model of supposed external environment. The simulation can help to discover thread interaction errors, e.g., deadlocks, because the simulated thread scheduler is able to provide a variety of timing schemes. In comparison, the standard JVM scheduler tends to behave regularly unless it is under a huge stress.

Conclusion

The simulation approach is capable to test a program under fairly realistic conditions. Although it does not provide formal proof of correctness, it provides a reasonable confidence of the correct program behavior. Another strength is that the simulation is able to deal with time-related properties of the tested software system.

3.2.3 Design by Contract

Design by contract (DbC) is a paradigm based on the idea that collaborating parts of a program should explicitly specify conditions, e.g., interface and input data under which they are able to operate. Furthermore, they should also specify what the result of the computation should be and a set of invariants that are maintained during the computation.

The name of the paradigm is taken from the business world where a client and a supplier sign a contract before the business transaction is performed.

Design by contract is native for the Eiffel programming language [27]. Eiffel is statically typed object-oriented imperative language. Another language worth mentioning for built-in DbC support is the D language [28]. The paradigm can be relatively easily used in many common languages.

The Eiffel DbC stands on four constructs:

- *Precondition:* An assertion that must hold *before* a method is executed.
- *Postcondition:* An assertion that must hold *after* a method is finished.
- *Invariant:* An assertion that must hold during a lifetime of an object (*class-invariant*) or during a computation loop (*loop-invariant*).

Whenever an assertion does not hold, an exception is raised. A program should never handle this exception, instead it should "fail hard". In a correct program, the assertions are never violated; this principle allows assertions to be removed after debugging, e.g., for performance reasons.

Example of a factorial computation written in Eiffel is shown in figure 3.6. Te computation requires the input data n to be positive and assures that the result of the computation will be greater or equal to the input.

Conclusion

Adding assertions to a program is in general a good programming practice. DbC is only more precise application of this practice. DbC is a general principle to some extent applicable to any code; however, some languages provide built-in support for it.

```
factorial(n: INTEGER): INTEGER is
require
    n >= 0
do
    if n = 0 then
        Result := 1
    else
        Result := n * factorial(n-1)
    end
ensure
    Result >= n
end
```

Figure 3.6: Factorial with Contracts

3.2.4 LTL Run-time Verification

Run-time verification is a technique between testing and formal verification. Whereas testing relies on ad hoc informal test cases, run-time verification uses formal specification. The specification of correct behavior is typically given by set of linear temporal logic (LTL) formulae.

Unlike model-checking, the verification is not done on a model of the tested piece of software but on the real application. The specification is checked against the running program. The main difference between model-checking and the run-time verification is that the model-checking verifies all possible execution paths while run-time verification investigates only the actual execution path.

Execution paths are finite as every real program earlier or later terminates. Note that Büchi automaton is constructed to recognize *infinite* traces. LTL run-time verification employs *alternating finite automaton* [29] to cope with finite traces.

Java Logical Observer

Java Logical Observer (J-LO) is an implementation of LTL run-time verification for Java programs; exhaustive description can be found in [30]. J-LO introduces a special kind of LTL called *dynamic linear temporal logic* (DLTL). DLTL contains free variables in propositions which can be bound to objects along the execution trace at run-time. An alternating finite automaton is used to match the traces.

DLTL predicates can be embedded into program source code in the form of Java annotations. J-LO views LTL verification as a cross-cutting aspect (see section 3.3.4) and uses AspectJ to inject a verification code into the code of the original program.

Conclusion

LTL run-time verification is a valuable technique; however, it cannot provide a proof of correctness as it investigates only the actual execution path. The major advantage is that it verifies a concrete implementation.

3.3 Tools and Practices for Dependable Software

This section presents a set of approaches that we consider beneficial for effective software development. These techniques help people to express their ideas without obstacles; that should reduce a frequency of design flaws and boost faster development.

3.3.1 Generative Programming

Generative programming is a process of creating a program code by an automated tool, i.e., the code is not directly written by a human.

Every compiler of a programming language such as C or Ada can be viewed as an automated code generator; a programmer writes a human-readable code (the actual source code) and the compiler generates a code runnable by a computer—a low level machine code. Without compilers and automated low-level code generation the creation of large applications would be unfeasible.

3.3.2 Model Driven Development

Generative programming can be used to generate a program code from a model of the intended program. The model of the program is created in order to investigate some properties of the program, for instance, UML models concentrate on design and architecture whereas formal models, e.g., written in Promela, investigate the correctness of algorithms. While it is possible to generate a program code from a model, the process is not straightforward. In order to investigate the selected properties, the model is abstracted, i.e., information not necessary for the purpose of the model is omitted. When generating a program code, we need to add information omitted by the model. When the information is added back, we have no longer a guarantee that the generated program code maintains the properties of the model.

The critical factor of success of the model-centric development is the code generator. For instance, [31] defines the requirements for a code generator that generates RTSJ-compliant[1] [45] Java code from abstracted UML2 model as follows:

1. The code generator should be easy to validate.
    - The generated code should be compact.
    - Model-to-code traceability should be direct.
    - Code generator development tools should allow easy model navigation.
2. Separation of concerns
    - Separating generated from manually-written code should be easy.
    - The separation between functional and non functional semantics should be met.
    - The integration of concurrent and sequential semantics should be easy.
3. The generated code should be high-quality.
4. Expressiveness of target programming language.
    - The concurrency semantics of the target programming language should be expressive and fully encompass the computational model of choice.
    - The target language should support object-oriented semantics.

The model-driven development tends to employ domain-specific models as universal models proved to be more complicated and less practical [32]. These models can be specified by domain-specific languages.

3.3.3 Domain-Specific Languages

Domain-specific languages (DSLs) are languages tailored to a specific application domain [33]. DSLs are basically an opposite for general-purpose programming languages. Whereas general-purpose languages are designed to be as useful as possible in various situations, DSLs trade this generality for better expressiveness in a limited domain.

One of the most widely used DSL is SQL[2] which is a language designed for data definition and manipulation. Another examples of popular DSLs are BNF

[1] Real-Time Specification for Java

[2] Structured Query Language

(Backus–Naur Form) used as notation of context-free grammars, MATLAB for technical computing, VHDL for hardware design, or LaTeX for typesetting.

There are several ways how a new DSL can be implemented. In combination with an application library any general purpose language can act as a DSL. The library's API[3] uses domain-specific vocabulary for the names of classes, methods and functions.

Another possibility is to restrict or extend an existing language. The main advantage of this approach is that users can use syntax or semantics customized for a specific domain without the cost connected with development of a new language.

And finally, one can invent a new DSL. This process is not straightforward and requires both domain knowledge and language development expertise. There are three main ways how the new language can be implemented:

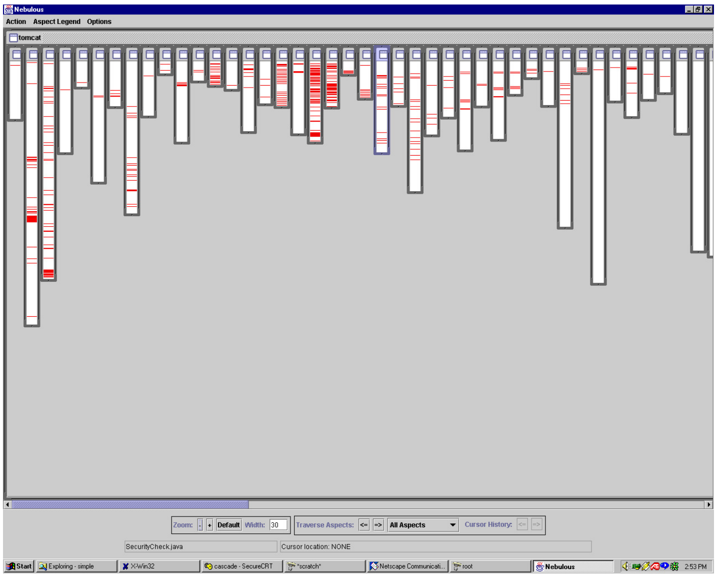
- *Interpreter:* DSL constructs are recognized and interpreted using fetch-decode-execute loop. This is appropriate for languages that have dynamic character.
- *Compiler/application generator:* DSL constructs are translated to some low level language and library calls. The output may be directly executable or interpreted.
- *Preprocessor:* DSL constructs are translated to constructs in an existing language. This can be done by a macro processor as we know from C language or by some templating engine.

In the field of embedded software, the main advantage of DSL utilization is safety and checking of correctness. Usual design goal of a DSL is that user is not able to write inappropriate and unsafe constructs due to the limitations imposed by the DSL designer. It is also much easier to verify a limited DSL code than the equivalent code in general programming language.

For instance [38] presents a language called *Action Language* for specification of behavior of embedded control system components. Developer uses *Action Language* for specification of a state machine; Java or Ada source code is then generated from the specification.

Another example of utilization of generative programming on the field of embedded software can be found in [39] and [40]. They use a DSL for building executable specifications of the demanded program. A specification language is built using attribute grammar; the language can be customized for a particular application. A code generator that employs either a macroprocessor or Prolog then emits an assembly code.

[3] Application Programming Interface



Width: 30

All Aspects

SecurityCheck.java

Cursor location: NONE

Figure 3.7: Logging Aspect in Apache Tomcat [35]

3.3.4 Aspect Oriented Programming

One of the key best practices in software engineering is a *separation of concerns*, that means program code should be divided into parts that overlap in functionality as little as possible. Concerns can be usually viewed as features or behaviors.

In commonly used programming languages, concerns can be separated by breaking program code into program units (in procedural languages) or into classes (in object oriented languages). Unfortunately, there are concerns that cannot be encapsulated easily: so called *cross-cutting* concerns that are scattered across a large portion of the code base. Good examples of such concerns are: logging, security policy, and transactional processing. Note that AOP can be viewed as a superset of DbC, in other words, DbC can be easily implemented with the help of AOP.

Cross-cutting concerns are hard to maintain. Assume we have an application and we want to change the way application logs its activity. Because majority of program modules use logging, the change will affect many unrelated pieces of code.

In figure 3.7 [35], you can see the logging aspect in Apache Tomcat: the white vertical bars represent individual packages, red colour within bars represents code for logging.

Applying Aspects

Aspect Oriented Programming (AOP) attempts to address this problem by allowing a programmer to express the cross-cutting concerns in stand-alone modules called *aspects* [34]. First of all, there are some terms that should be explained; the terms were established by the first widely used AOP implementation for Java, AspectJ [36].

- *Advice* is a piece of code that implements an aspect. For the case of the logging aspect, it should contain a call of some logging routine.
- *Join-point* is a point in the source code where an advice can be applied. (Analogy: a break-point is a point where a program can be stopped for debugging purposes.) In most AOP tools, join-points are defined to be before/after a method call or before/after a particular statement.
- *Cross-cut* is a set of join-points suitable for a particular advice. A cross-cut is usually defined by a matching rule. For instance assume a banking software: an advice for transactional processing should be applied to all methods that transfer money from one account to another. Demanded cross-cut might be defined as follows: method's name contains "transfer" and the method has two parameters of the *Account* class.

Implementation

The AOP paradigm is not directly supported by any of the mainstream programming languages. The code of advices has to be injected into the core application code (the core aspect) by some kind of a preprocessor. The injection process is referred to as *aspect weaving*. The weaver takes the core code written in a particular programming language and the cross-cutting aspects (that are usually written in an aspect-enhanced superset of the used programming language) and produces the final code in the original language. The process is depicted in figure 3.8.

In environments with well-defined binary format, e.g., Java and its bytecode, the weaving can be also performed on the compiled representation of the program. In the case of Java, the transformation can be also done at class load-time. However, the advantage of the source code level weaving is that the final source code (that is actually compiled and deployed) stays human readable and thus suitable for some qualification process [37].

In object oriented environments, some parts of AOP can be implemented by object inheritance. That means a core class is subclassed to be enhanced by aspects.

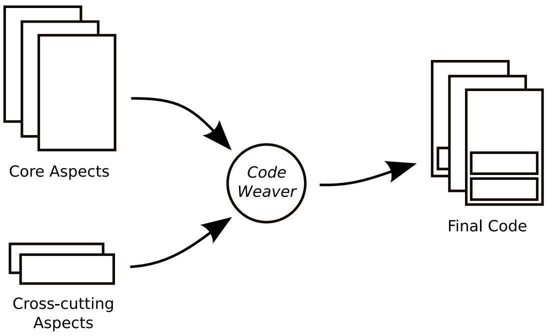


Figure 3.8: Aspect Weaving

Conclusion

AOP is relatively new paradigm that improves software maintainability by encapsulating concerns which, when a conventional design approach is used, are scattered over a large portion code base.

3.4 Java on Embedded Devices

Java is one of the most influential technologies of the last decade. Java programming language is memory-safe, relatively simple (in comparison with C++), and thus easy to use. Java popularized the concept of platform-independent code that is run on a virtual machine. Modern implementations of the Java Virtual Machine (JVM) provide performance comparable to the statically compiled machine code due to the just-in-time (JIT) compilation.

Java was initially intended for embedded devices; however, nowadays it is the mainstream technology for mid- to large-scale enterprise applications.

With the growing popularity of Java, there is naturally a trend to use Java for embedded applications and even for real-time applications, potentially even safety-critical. There are three main obstacles, however:

- Conventional Java implementation is not optimized for devices with constrained memory.
- Conventional Java implementation is not ready for real-time scheduling.
- Real-time garbage collector is still mainly a subject of research.

3.4.1 Issues with Constrained Devices

Conventional Java implementations are primary optimized for enterprise applications; that means, they take an advantage of powerful computers. For example, Sun/Oracle Java Virtual Machine needs at least several megabytes of memory just to start its execution.

Special approaches must be utilized for devices with limited memory and CPU power. There exist several JVMs aimed to this domain. For instance, Sun Java Micro Edition provides a subset of Java Standard Edition API and is available in two configurations: Connected Limited Device Configuration (CLDC) targeted at small mobile devices and Connected Device Profile (CDC) for network appliances. Another examples of small memory foot-print JVMs are Java In The Small (JITS)[41] or LeJOS [42].

Typically, every special JVM implementation brings its own limitation of features and its own standard library. These incompatibilities go against one of the most important factor of Java's success which is portability, i.e., the "write once, run everywhere" motto.

Slightly different concept is presented in [43] where a custom JVM is built to meet the requirements of a particular application that uses the standard Java API. The idea is that an application usually does not utilize all features provided by JVM, for instance, some applications do not use threading, floating-point arithmetic, etc. By removing support for byte-code instructions of these unused features, one can get pretty compact JVM. This approach scales well: the most simple applications get the most compact JVM.

Java code is compiled to platform independent byte-code. This byte-code can be either interpreted or further compiled to a native machine code. Conventional JVMs perform the translation at run-time. This just-in-time (JIT) compilation can cause some temporal and unpredictable slowdown when an application starts and also requires additional memory. So this approach does not fit well for embedded applications. Pure byte-code interpretation runs predictably but also brings significant performance overhead that is often unbearable for hardware with limited computational power.

For these reasons, ahead-of-time (AOT) compilation is often used [44]. That means, Java code (or byte-code) is translated to the native machine code before the application is run. Possible disadvantage of this approach is that the machine code is not as portable as the Java byte-code. On the other hand, embedded applications are often designed for a specific hardware.

3.4.2 Real-time Issues

Conventional JVMs implementations are not suitable for developing real-time embedded systems. The Java garbage collector (GC) can cause unbounded lags of threads execution. Java thread scheduling is out of control of the application. To address these issues, the Real-time Java Experts Group created the Real-Time Specification for Java (RTSJ) [45].

The guiding principles followed by the expert group who created the RTSJ specification included [47]:

- Backward compatibility with the Java 2 platform.
- No syntactic extension to the Java language, i.e., no new keywords
- Write once carefully, run anywhere conditionally.
- Enable predictable execution.
- Balance between current practice and advanced features.

The RTSJ extends Java memory model by providing *scoped memory areas*. These areas guarantees bounded allocation time. A real-time thread can explicitly enter a memory area; all subsequent object allocations are then performed within the current memory area. The memory area is destroyed when it is left by the thread. The concept of *scoped memory areas* is real-time friendly; however, this explicit memory management requires fundamentally different programming style than is usual in conventional Java applications.

The RTSJ defines universal API for real-time threading that is usable for many real-time scheduling algorithms. The RTSJ also defines asynchronous event handling and fine-grained timers.

There are several RTSJ compliant Java implementations nowadays.

As mentioned above, the concept of *scoped memory areas* is a special approach, unknown to the standard Java. Therefore, there is a lot of research of GCs suitable for real-time Java.

There is not a winning solution; every proposal has its own pros and cons. There are two major flaws in many approaches: either the GC is not provably real-time, or it imposes large space overheads to meet the real-time bounds.

For instance, Bacon et al. present a real-time GC that guarantees even utilization of worker threads while keeps low space overhead [46]. It also beats memory fragmentation by dividing memory into movable pages. However, this GC works only for uniprocessor systems.

3.5 Compilation of Dynamic Languages

Dynamic language such as Python cannot be fully compiled ahead-of-time. The main reason is dynamic code generation, literally the *eval* function.

3.5.1 PHP Compiler

PHP Compiler's goal is to speed up PHP based applications [48], it uses C as the output code. PHP is a dynamically and weakly typed scripting language with support for object-oriented programs. It is primarily used for web applications development. The major implementation of PHP is Zend Engine which is a byte-code interpreter.

The PHP Compiler is tightly interconnected with the original interpreter. It uses Zend Engine for parsing of some parts of the PHP source code. The generated C code also calls some functions from the interpreter, mainly for operator evaluation that is rather complicated due to weak typing. Dynamically generated code (via *eval*) can not be translated; however, it can be forwarded to the original interpreter.

3.5.2 The PyPy Interpreter and Compiler

PyPy [49] 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[4], 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 3.9.

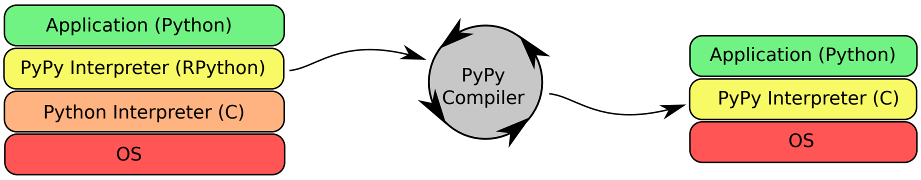


Figure 3.9: Translation of PyPy Interpreter

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.

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

3.6 Two Main Development Approaches in a Nutshell

In this section we describe two basic approaches for development of embedded software. We know that in practice there are many approaches, some are more advanced than the others, every of them having their own pros and cons. We deal with these two basic types to emphasize the features of our own dynamic-language-driven approach proposed in section 5.3.

3.6.1 The Traditional Approach

Despite the progress in software development tools, nonnegligible amount of software for embedded devices is done in a way that is several decades old.

The traditional way of developing embedded software relies mainly on statically typed relatively low level languages like C or C++. These languages offer error detection based on the compile-time type analysis.

Thanks to the system nature of the mentioned languages, programs can be very resource efficient. In more recent languages such as Java or C#, the resource efficiency is reached at cost of utilization of special methods mentioned in section 3.4.

The only correctness that is guaranteed by the tools is the type correctness of the program. The type correctness is really useful as it detects errors early; however, it can handle only limited class of errors.

The most traditional way of delivering guarantees of correctness is testing. There is a whole universe of testing methods. The naive ones only demonstrate functionality under certain conditions; advanced testing methods can, however, provide solid guarantees. On the other hand, they can never provide a formal proof.

Low level languages are not friendly to formal methods, mainly because their code is polluted by implementation details. It is much easier to extract a formal model from Java than from a C code.

Low level languages also demand more effort to finish the software simply because developers have to deal more with various implementation details. Static nature of the languages also prevent embracing less traditional programming paradigms.

3.6.2 Approach with Formal Methods

Generally the highest possible guarantee of correctness, i.e., a proof of correctness, can be achieved by formal methods. First of all, there are plenty of ways how one can employ formal methods; this section tries to describe the most general approach.

Model Driven Development

Due to the high complexity of real world software, the formal methods are usually applied to a simplified model of the intended piece of software. The model is usually used for the most critical part of the intended application, e.g., the part that coordinates the work of multiple threads.

Both the model and the requirements are expressed by some language specific for this domain, e.g., Promela. The model is investigated whether it fulfills the desired requirements with help of powerful techniques such as model-checking.

If the model is successfully verified, it is transformed to the real program implemented in some general purpose language. It is important to say that this transformation is the major weakness of the approach. If it is done by hand, an error can be introduced due to a developer's mistake or omission.

Even if the transformation is done by an automated tool, there is no real guarantee that the final program has exactly the same properties that were proven for the formal model. The problem is that we add information during the transformation. The final program has to deal with implementation details. The added behavior can, in principle, break the proven properties.

Verifying Real Software

The opposite approach is to try to formally investigate the real software. Again, there are many possibilities.

First, the formal model can be extracted from a piece of the real software. Generally, it is a nontrivial task. One has to simplify certain aspects of the program to keep the model verifiable but not to introduce errors due to an inappropriate simplification.

Some programming languages are friendlier to this approach than the others. For example, code in high level functional languages, e.g., Haskell, can be formally investigated more easily than code written in low-level system languages such as C.

According to the risk connected with utilization of a rare technology (the *technology risk aversion* will be discussed further in section 4.1.1), the formal model extraction tool is more useful if it can extract model from widely used programming language. The most notable tool in this domain is Bandera (see section 3.1.3) that extracts models from Java programs.

Second, there are tools that formally check real programs for defects without the need of an explicit formal model. Major strength of this technique is that the tool verifies real behavior of the program. The confidence of the anticipated behavior of the program is almost absolute if the verification tool investigates the program in the form of machine code (or byte-code), i.e., the code that is then really deployed.

Java Pathfinder is the most known tool from this domain. It is an explicit model checker that verifies programs in a form of Java byte-code. More on this tool in section 3.1.3.

In spite of the fact that tools like Java Pathfinder have the ability to investigate real software, in practice, it is usual to investigate a simplified version of the program for better results. But the translation between simplified and unsimplified version of a Java program is probably less error-prone than the translation between an abstract program model in Promela and the real implementation in C.

Chapter 4

Towards High Level Dynamic Approach

This chapter describes the very nature of dynamically typed languages, e.g., dynamic creation of objects space, compilation vs. interpretation, etc. Also, the role of these languages in the software development process is discussed.

4.1 Introduction

With the help of the analyses of various tools and technologies useful for embedded software development that were provided in chapter 3, we have identified three main characteristics of embedded software development:

- *Technology risk aversion.* Tendency to use general and widespread programming languages as much as possible.

- *Abstraction.* According to the particular task, the language with the highest level of abstraction is selected.

- *Avoidance of hard tools.* Required quality standards are met with utilization of standard tools rather than with special-purpose tools that require special knowledge. The friendlier the standard tools are for testing and formal verification the better.

We will analyze these characteristics and confront them with properties of high-level dynamically typed languages.

4.1.1 Technology Risk Aversion

General and widely used programming languages are preferred for several reasons. These languages are mature and numerously proved their qualities in practice. Because there is a large code-base already existing, there are also many support tools such as IDEs and profilers. And, last but not least, there are many sufficiently experienced developers. Affinity to proven languages causes a slowdown of adoption of new programming languages.

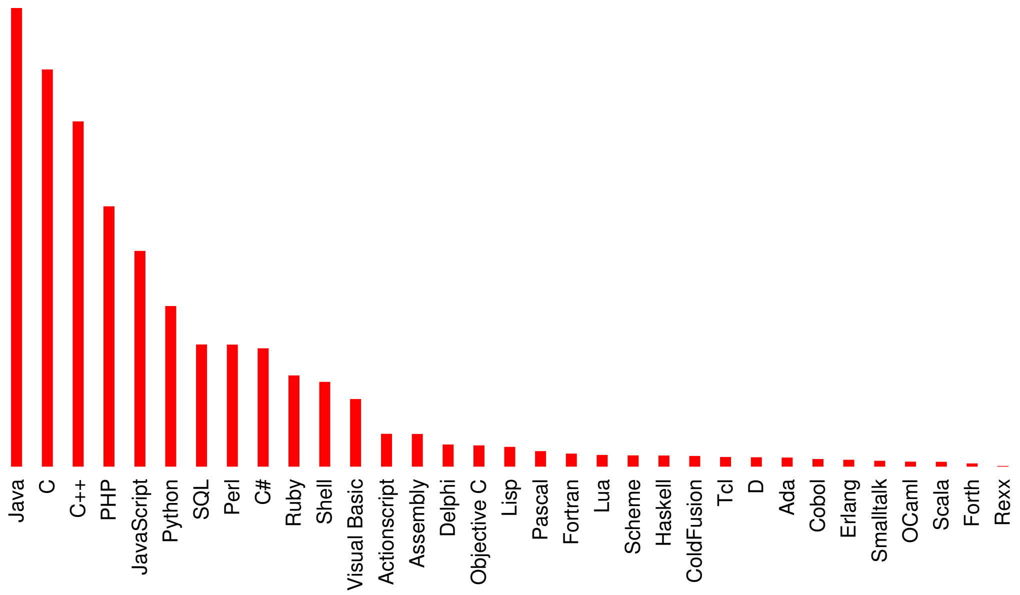


Figure 4.1: Language Popularity [50]

There is no precise way how to measure the popularity of programming languages. Estimates are usually built with the help of web search engines, one of them is depicted in figure 4.1 [50].

For the last three decades the most generally used programming language was C. Many younger and more modern languages are built on top of C: C++, Java, C#. There is a noticeable move from C to Java in the last decade in the world of embedded software development. Java is now a proven technology with huge knowledge-base and many engineers at the market. It is also often taught at universities.

Java owe its popularity to the growth of network applications. It is popular choice for mid- to large-scale server-side applications for its reliability and security. Thanks to its portability it is also used at the client side: as Java applets in Internet browsers and also in cell phones.

Smaller Internet applications, tend to use dynamically typed languages such as PHP, Ruby or Python. However, Python is established for big web sites as well, it is one of the languages officially used at Google, Inc., for instance YouTube[1] web site is mainly powered by Python [51].

There is also an apparent growth of JavaScript a the client side. This dynamically and weakly typed object oriented language started as an auxiliary technology for web browsers. Nowadays, there are complex applications developed in JavaScript with the help of mature libraries and frameworks. Since 2007, there is also a major competition among JavaScript interpreter implementations that results in adoption of the state-of-the-art optimization techniques [52].

Note that there were similar competitions between various C and Fortran implementations when these technologies emerged in the very mainstream.

Internet applications stimulate the growth of dynamically typed languages. The rise of dynamical paradigm is depicted in figure 4.2 [53]. As a consequence, there are more and more developers that are familiar with these languages. If these become mainstream, we anticipate there will be a pressure for adoption of dynamic typing languages for embedded software development in the same way as there is a pressure for Java adoption today.

4.1.2 Abstraction

Selecting the right tool for a particular task is a crucial factor of success. Programming tools are evolving in time and newer tools usually outperform the older ones. In the history of computer programming, we have seen that the tools improvement and consequential productivity boost is often achieved by adding a new level of abstraction.

We always pay a price with every level of abstraction. First, we usually need more computing power; second, we give up some control in favor of the tool itself.

This can be showed on the following examples:

- When we use the C compiler instead of writing directly an assembly code, we give up the possibility to use some specific features of an underlying CPU. Thus, every operating system written in C actually contains some assembly code.

[1] http://www.youtube.com

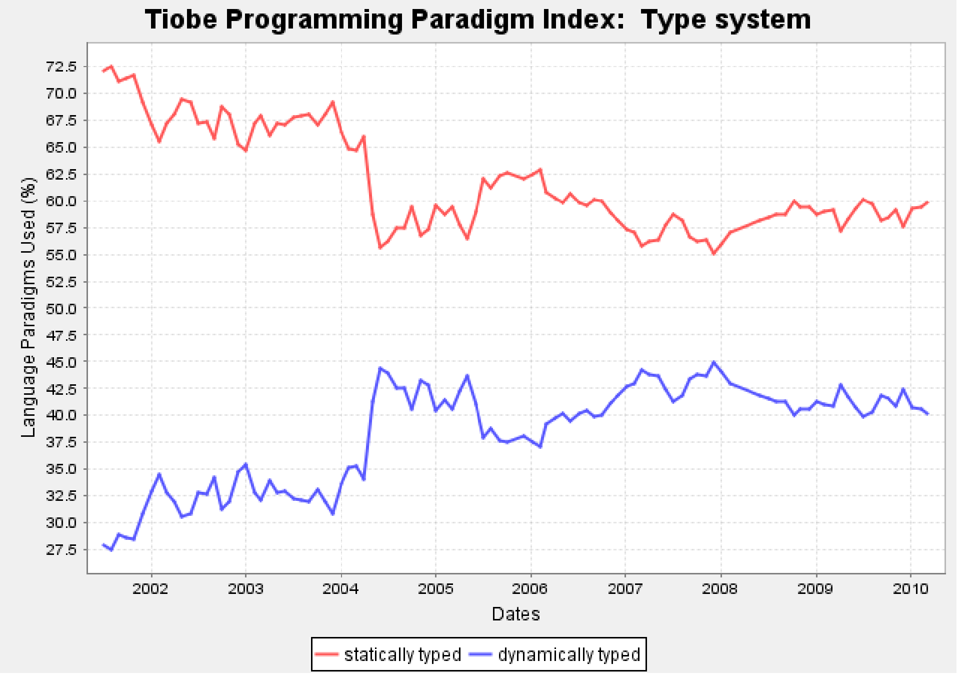


Figure 4.2: Statical/Dynamical Type System Popularity [53]

- Automatic memory management (GC) allows us to write simpler programs but we usually sacrifice deterministic behavior.

- With virtual machines such as JVM, we can deploy unchanged programs on many platforms at the cost of interpretation slowdown or complicated optimization techniques, e.g., just-in-time compilation.

- Dynamically typed languages offer higher flexibility but sacrifice static type checking and make optimization techniques harder.

- Domain specific languages or visual design tools help us to better describe the problem of some application domain but these tools are by definition not universal.

Mainstream development embraces these new layers as computers are more and more powerful. The world of embedded software usually adopts new layers with a delay or with some customizations.

The most notable example of the special approach is Java. For embedded devices one needs special virtual machines and optimizations such as ahead-of-time compilation. The default Java garbage collector also cannot be used for real-time applications. Pitfalls of Java on embedded devices were described in section 3.4.

Flexibility of dynamically typed languages can be also employed by embedded software development. Optimization of dynamic languages is more difficult than optimization of the statically typed counterparts. It is obvious, that special approaches have to be used here.

4.1.3 Avoidance of Hard Tools

Special tools are inevitable when there are strong dependability requirements; safety-critical systems are the best example. Formal methods and tools are considered hard and they require very well qualified developers.

Ideally, the general tools used for the development would be capable of doing all required testing and verification. Unfortunately, it is rarely true as general tools are designed only to meet the average quality standards.

Therefore it is desirable that the general tools play smoothly with the specialized ones. The level of abstraction comes into account again. It is hard to think formally about low level assembly or C code. Higher level languages are friendlier; for instance there exists a tool named Bandera that extracts formal models from a Java programs, see section 3.1.3. When a program is written in a high level purely functional language such as Haskell, one can directly formally investigate a part of the program because Haskell guarantees that the investigated part has no side-effects.

High level dynamically typed languages are generally not as formally oriented as Haskell; however, they provide sufficient level of abstraction.

4.2 What Is High Level Language?

Originally, the term *high level programming language* denoted a programming language that abstracts from instructions of a processor (CPU), e.g., C language. Nowadays, C language is not viewed as a high level language for two reasons. First, assembly code (i.e., human readable notation of CPU instructions) is now rarely written by hand. Second, there are widely used languages with much higher level of abstraction than C.

For the purposes of our work, a high level programming language is the language that abstracts computer resources in far more general way than C language. Literally, it should have the following properties:

Memory safety. Program cannot reference invalid memory area; this usually implies absence of pointer arithmetic.

Automatic memory management. Unused memory allocated objects are automatically reclaimed; this is usually achieved by garbage collection algorithms.

Data type abstraction Powerful data types such as variable-size arrays, associative arrays (dictionaries) or sets are incorporated into the language. Language should have the ability to express literals of these data types, e.g., *[1,2,4]* and *{'a':1, 'b':2}* are list and dictionary literals in Python. If the powerful data types are not part of the language, they can be provided in a form of tightly integrated standard library. Language construct should play smoothly with these data types, i.e., it should be easy to iterate over container's items and to perform certain operation with every item.

Exceptions. Language support for exceptions enables to isolate regular and error-handling code. This leads to cleaner, shorter and more readable programs.

OOP Although the language does not necessarily have to be object-oriented, some support for object-oriented programming is desirable as OOP is widely used paradigm. Good support for mainstream paradigms reduces the technology risk for the users of the language.

The characterization given is not very precise. The point is that a programmer is able to express his or her ideas in natural, readable, and non-verbose manner without dealing much with the implementation details.

4.3 Dynamic Programming Languages

Some programming languages are more *static* while others are more *dynamic*. Static languages tend to do more work at compile-time whereas dynamic do more work at run-time.

4.3.1 Being Static

Programs written in static programming languages cannot alter their structure at run-time. One of the biggest benefits of this design decision is that type-checking can be reliably performed prior of the program run, i.e., at compile time.

Although the compile-time type-checking is very helpful early error detection, it can detect only a limited class of errors. Most statically typed languages allow to work with types dynamically and thus the type checking is deferred to run-time in some situations to gain more flexibility. For instance Java has operator *instanceof* that is useful for object polymorphism. Last but not least, even a program written in a language with advanced static type checking such as Ada may fail at run-time due to an incorrect type usage as in the case of Ariane 5 rocket failure [55]. See also section 3.1.1.

The fact that all programming interfaces are unchangeable at run-time opens the possibility of deep static analysis and subsequent aggressive optimizations during compilation. Perhaps the best example of language with static compile-time optimization is C++.

Modern object oriented statically typed languages like Java or C# have an introspection ability, usually called *reflection*. By reflection, one can examine an unknown object at run-time, for instance, acquire a list of methods it implements. It is also possible to call any of the methods from the acquired list.

Reflection effectively bypasses static type checking which prevents compiler from aggressive optimizations. Therefore this feature is rather available only for languages that run on the top of a virtual machine and rely on its run-time optimizations. Reflection us usually used only for special purposes because it is not very convenient for a programmer, at least in Java and C#; and it also brings some performance penalty.

4.3.2 Being Dynamic

There is no real distinction between compile- and run-time in dynamic programming languages. The main reason is that everything what is specified by a programmer prior to the program is run, can be later altered at run-time.

The most obvious consequence is that types of variables may change over time. That means, reliable static type checking is not feasible; type correctness can be checked only at run-time.

Moreover, every programming interface can be changed at any time. For instance, methods of an object can be added or removed. One can also take an existing method and completely redefine its implementation. This feature gives dynamic languages outstanding flexibility.

In fact, the structure of a dynamic program is *exclusively* built at run-time. When the program is run, it starts with an empty object-space. As the object definitions are reached during the interpretation, these objects are constructed and added to the object-space. Take a class definition (that is a special kind of object) as an example. Every class definition is first instantiated as an empty class, i.e., without any methods and data fields. Later, when a method definition is reached, it is instantiated (it is just another kind of object) and added to the

```
class C:
    def returnOne(self):
        return 1

def f(self):
    return 1

class D:
    pass

D.returnOne = f

c = C()
x = c.returnOne()
d = D()
y = d.returnOne()
assert x == y
```

Figure 4.3: Dynamic Class Definition

class. Every method and data field is constructed at run-time. This also implies that the class definition is never closed.

While programs in static languages usually have "data definition part" and "computation part", programs in dynamic languages consist only from sequence of statements. Some statements create data structures while the others do the computation. It is possible to arbitrarily mix these two types of statements; however, it is natural that data structures are defined first and computations are based on them. Let us call it the *two-stage design*.

Despite this run-time initialization of the object-space, actual language syntax may resemble static initialization as we know it from, say, Java. This is illustrated in Python code snippet in figure 4.3 where the *C* class is defined in a way as it is usual in static languages. The *D* class is defined without any "syntantic sugar". An empty class is defined first and the method *returnOne* is added later by referencing an independently created function *f*. The classes *C* and *D* are identical.

Note that the statement *D.returnOne* = *f* is the last statement from the "first stage" that defines the data structures.

Method calls have also more dynamic nature. In static languages a compiler checks whether methods are called correctly according to the predefined type rules. In dynamic languages, however, method calling can be more naturally viewed as *message passing*, i.e., the method call is actually a message with a given name (identifier) and arguments. The callee handles the incoming message at run-time. The standard reaction on the incoming message is an execution of the code of the method with the corresponding name. Thanks to this rather loose coupling, objects are also able to handle unknown messages. The predefined handler for an unknown message usually raises an exception but this behavior can be redefined. For instance, an unknown message can be forwarded to another object.

Dynamic systems are much harder to optimize than the static counterparts. When everything can be changed at run-time, there is no room for any assumptions about the code flow. Optimizations have to be done also dynamically at run-time and have to evolve as the program structure may evolve.

4.3.3 Compilation vs. Interpretation

Compilation is a process that transforms a source code written in a programming language (source language) to another computer language (target language). The source code is typically human-readable and has a higher level of abstraction than the target code which is usually machine-readable. When the target language is a machine code, it can be interpreted directly by a computer, i.e., *run* on hardware.

Apart from interpretation by a computer hardware, it is also possible to reimplement the hardware functionality in software and build a *virtual machine*.

As another step, we can build a virtual machine that does not interpret a machine code of a particular physical computer but interprets some kind of artificial hardware-independent machine code. The most notable example of such a technology is Java Virtual Machine that interprets portable Java byte-code.

For the purpose of this work, we define *machine code* as the code that can be directly run on a contemporary hardware; code that runs only on virtual machine will be referred to as *byte-code*.

A program can be run on different levels of interpretation. Programming languages that enable their code to be compiled to machine code are usually referred to as *compiled* while others are referred to as *interpreted*. It is important to note that each level of interpretation brings an overhead; compiled languages run order of magnitude faster than interpreted. On the other hand, a software-implemented virtual machine provides a greater amount of flexibility.

Static and Dynamic Compilation

Static languages such as C or C++ are designed with compilation to the machine code in mind. Because of their very static nature, the compiler is able to perform aggressive ahead-of-time optimizations. Note that there also exists C and C++ interpreters [54] but they are not widely used.

More recent static languages such as Java an C# are compiled to byte-code rather than machine code. The main advantage of this approach is portability. Because the byte-code interpretation is relatively slow, virtual machines usually perform just-in-time compilation to machine code. Optimizations during the just-in-time compilation can take advantage of the knowledge of the run-time program behavior. This approach is, however, optimized mainly for large, enterprise-grade applications.

Dynamic languages are, on the other hand, designed with interpretation in mind. They also usually depend on the possibility to create, parse and run a piece of source code at run-time; the function that takes an arbitrary string containing a piece of source code and runs it is usually called *eval*.

The most straightforward and naive way of interpretation is to directly interpret the abstract syntax tree that results from the source code parsing. For instance, the initial implementation of Ruby used this approach. Parsing the source code prior to every program start is rather inefficient; moreover, the AST interpretation is not very fast. Therefore, many dynamic languages are compiled ahead-of-time to byte-code that is then interpreted. Python is a typical example of this approach. It is important to note, that this compilation can not be done in all cases ahead-of-time because a new code can be created at arbitrary time via *eval*. On the other hand, there is only a limited use of *eval* in an average program.

It is of course possible to compile at least some limited parts of the dynamical code to machine code; some recent JavaScript implementations do that [52].

Dynamic languages implementations naturally tend to perform optimizations at run-time. This approach can generally lead to rather fast execution.

Note on Constrained Devices: Run-time optimization for embedded devices is challenging if not infeasible. First, emitting optimized version of the code just-in-time naturally requires additional memory; the more aggressive optimizations with code variants for different data types, the more memory is needed. Generating the optimized code also needs CPU time; in the case of the reactive system, the just-in-time compiler has to be very careful in the sense of scheduling its work. Last but not least, the result of the dynamic optimization is uncertain because it relies on heuristics. It can lead to a very fast execution in general; however, there might be some corner cases in which the execution path remains unoptimized and thus slow.

4.4 Open for New Paradigms

There are many programming paradigms; some of them are widespread while others are adopted rarely. Most languages support several paradigms; the support can be either explicit or implicit. Explicit support means that the language constructs are designed with a particular programming paradigm in mind. Implicit support only allows a particular paradigm to be used in the language; however, usually with less convenience.

4.4.1 Objects, Aspect, Contracts, ...

It is possible to write an object oriented program in plain C but it generally requires more effort than implementing the same program in C++ or Objective-C. *Design by Contract* (DbC) is explicitly supported in Eiffel and D programming language (note that none of these languages are mainstream); however, this paradigm can be easily embraced by many languages. Se also section 3.2.3 for more information about DbC.

Some paradigms, such as *Aspect Oriented Programming* (AOP, see section 3.3.4), cannot be easily embraced in a programming language that does not have explicit support for it.

The key process of AOP is called *aspect weaving*. If the language has an explicit support for AOP, the aspect weaver is a part of the compiler or interpreter. One can also create aspect weaver as an external tool that works as a source code preprocessor or byte-code postprocessor.

There is a better implicit support for new paradigms in dynamic languages than in static languages. The reason is that the structure of the program, the program's object space, can be manipulated at run-time. This manipulation is done within the program itself. Utilization of self-modifying constructs in a dynamic language is much easier than customization of a compiler of a static language or a virtual machine.

In order to demonstrate this fact, we have implemented a simple DbC support in Python, see figure 4.4. Our implementation assumes that for a particular method, there may exist an auxiliary method that checks precondition: the method *factorial* has an auxiliary precondition check in method *factorial___precond*. The DbC weaver transforms a class, in our example the *Math* class. The transformation injects call of the precondition check method before every call of the method that actually implements the desired functionality, i.e., the *factorial___precond* method is always called before the *factorial* method. Postcondition checks can be implemented in the same way.

The injection is done by *DbcWeaver* that is activated during the construction

```
def make_wrapped_call(func, precond):
    def wrapped(*args):
        # This wrapper first calls the precondition check,
        # then the original method.
        precond(*args)
        return func(*args)
    return wrapped

class DbcWeaver(type):
    def __init__(cls, clsname, bases, dict):
        super(DbcWeaver, cls).__init__(clsname, bases, dict)

        for name in dict.keys():
            # This method will be wrapped.
            func = dict[name]
            if not callable(func): continue

            try:
                # Try to obtain the method that will be used for
                # precondition checking.
                # It is identified by naming convention.
                precond = dict[name + '__precond']
            except KeyError:
                # There is not a precondition
                # check for method "func".
                continue

            # Set wrapped method instead of the original one.
            setattr(cls, name, make_wrapped_call(func, precond))

class Math:
    # Weaving will be applied to this class.
    __metaclass__ = DbcWeaver

    def factorial__precond(self, n):
        # After weaving, this method is called
        # before "factorial" method.
        assert n >= 0

    def factorial(self, n):
        result = 1
        for i in xrange(2, n+1):
            result *= i
        return result
```

Figure 4.4: Design by Contract in Python

of the *Math* class.

Note that our example relies on Python's support for metaprogramming (see the ___metaclass___ attribute); however, this support is only a sort of syntactic sugar and an equivalent functionality can be achieved without it. The weaver traverses through all attributes of a newly constructed class and suitable methods are replaced by methods that first do the precondition check and then call the original algorithm implementation.

Our example shows one way how a new paradigm can be incorporated into a dynamic language. We also use slightly different approach to DbC in our FTP client case study, see section 10.2.2.

4.4.2 Domain Specific Languages

Dynamically typed languages also provide viable base for building domain specific languages, because they provide useful features such as named (keyword) arguments, closures and operator overloading (in the case of Ruby). These features enable the classes and objects of some domain-specific library to be accessed in a way natural for the domain.

Good examples are *Rake*[2] and *SCons*[3] that are software construction tools like *GNU Make* and *Apache Ant*. Rake is based on Ruby while SCons is Python-based. See simple examples in figures 4.5, 4.6, and 4.7 for illustration of how Ruby and Python can be used to build a makefile-like DSL.

See figure 4.8 for an example of relation data definition and querying in Python, literally Django[4] web framework. These constructs are used for generating appropriate SQL commands.

4.5 Conclusion

Focusing on high level dynamic languages makes sense. Dynamic languages are and will be part of the main stream technologies, so there will be engineers and support tools available in the future. There is a huge opportunity for utilization of these flexible technologies for development for constrained embedded devices.

The strong advantages of dynamic approach are paid by several disadvantages. It is obvious that this may work for constrained embedded devices only with the help of new methods and methodologies. Code performance, correctness, and memory management are the main issues that have to be analyzed and solved.

[2] http://rake.rubyforge.org/

[3] http://www.scons.org

[4] http://www.djangoproject.com

```
require 'rake/clean'

CLEAN.include('*.o')
CLOBBER.include('hello')

task :default => ["hello"]

SRC = FileList['*.c']
OBJ = SRC.ext('o')

rule '.o' => '.c' do |t|
  sh "cc -c -o #{t.name} #{t.source}"
end

file "hello" => OBJ do
  sh "cc -o hello #{OBJ}"
end

# File dependencies go here ...
file 'main.o' => ['main.c', 'greet.h']
file 'greet.o' => ['greet.c']
```

Figure 4.5: Rake Script That Builds a C Application

We will research and propose a development approach aimed to dynamic languages on embedded devices in the subsequent chapters.

```
namespace :cake do
  desc 'make pancakes'
  task :pancake => [:flour,:milk,:egg,:baking_powder] do
     puts "sizzle"
  end
  task :butter do
    puts "cut 3 tablespoons of butter into tiny squares"
  end
  task :flour => :butter do
    puts "use hands to knead " +
         "butter squares into 1{{frac|1|2}} cup flour"
  end
  task :milk do
    puts "add 1{{frac|1|4}} cup milk"
  end
  task :egg do
   puts "add 1 egg"
  end
  task :baking_powder do
   puts "add 3{{frac|1|2}} teaspoons baking powder"
  end
end
```

Figure 4.6: Abstract Rake Script Describing Pancake Cooking [56]

```
env = Environment()   # Create an environmnet

lib_target  = "hello"
lib_sources = ["libhello.c"]

libhello = env.SharedLibrary(
  target = lib_target, source = lib_sources)
hello = env.Program(
  source = ["helloworld.c"], target = "helloworld")
myhello = env.Program(
  source = ["myhello.c","libhello.so"], target = "myhello")

env.Install(dir = "Build", source = hello)
env.Install(dir = "Build", source = myhello)
env.Install(dir = "Build", source = libhello)
env.Alias('install', ['Build'])
```

Figure 4.7: SCons Script That Builds a C Application with a Library

```
class User(Model):
    login = CharField(max_length=20, unique=True)
    first_name = CharField(max_length=30)
    last_name = CharField(max_length=30)

my_users = User.objects.filter(
            first_name='Andrew').order_by('last_name')
```

Figure 4.8: Data Definition and Querying in Python (Django)

Chapter 5

The Proposed Development Approach

This chapter contains an overall design of our development approach. First we determine the properties of the intended applications, i.e., the hardware and software platform. Then we select the tool-chain for our approach: the language, the compiler and the formal verifier.

5.1 Introduction

There are three main objectives of software development for embedded devices. Their nature forms the development process.

- Fit for purpose
- Dependability
- Reasonable development cost

Software is *fit for purpose* if it does the desired job. That means, it has all the functional features that were previously specified. Moreover, it has some non-functional properties foremost its hardware requirements fit the intended computer.

The key aspect of *dependability* is *correctness*, i.e., behavior according to the specification. It is practically impossible to prove the ultimate correctness of a piece of real-world software. We can only earn a set of guarantees of the right functionality in practice. The most common and relatively mild guarantees are based on testing. One can earn more solid guarantees by a formal proof of some functionality. Thus the development process should be friendly to formal methods.

Without any doubts, the cost of development is always important. The development process should embrace methods that enable to build software in a reasonable time and thus with a reasonable cost.

In this chapter, we propose a development approach for embedded devices that relies on high level dynamic languages and profits from their properties.

5.2 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.

5.2.1 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.

5.2.2 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[1] but other variants such as uClibc[2] should work as

[1] http://www.gnu.org/software/libc/

[2] http://www.uclibc.org/

well.

Our threading relies on POSIX Threads [57] that are available for many Unix-like operating systems and even for Microsoft Windows.

5.2.3 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 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[3] such as μClinux[4].

5.3 Dynamic-Language-Driven Approach

We believe our development approach has some significant strengths in comparison with common approaches briefly described in section 3.6. This section points out key features of our proposed approach.

5.3.1 Basic Principles

First, the program code is primarily written in a widely used high level dynamic language. According to previous chapters, 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.

[3] Memory Management Unit

[4] http://uclinux.org/description/

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.

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 5.1.

5.3.2 The Tool-Chain Selection

As we have defined our requirements, it is time to choose the language and the tools we will rely on. Note that our requirements are also partially in contradiction; we embrace high level dynamic approach but we require full ahead-of-time compilation. So it is obvious that dynamic behavior has to be suppressed to some reasonable level at some point of translation.

We want to rely on mainstream language to conform *technology risk aversion* so there are only a few possibilities: PHP[5], JavaScript[6], Python[7], Perl[8], and

[5] http://www.php.net

[6] http://www.ecma-international.org/publications/standards/Ecma-262.htm

[7] http://www.python.org

[8] http://www.perl.org

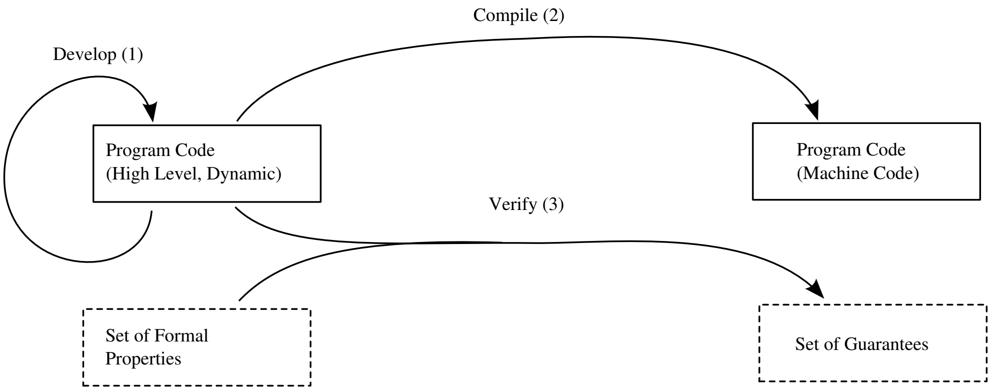


Figure 5.1: Overall Scheme of Our Approach

Ruby[9].

All these languages are really high level and dynamic. Level of support of object-oriented programming and exception handling, however, differs. Perl has only implicit support for OOP and exception handling. There are also less rational criteria such "cleanness" of the language design; we state that JavaScript, Python and Ruby are cleaner and more elegant than Perl and PHP.

The main criteria is, however, availability of an ahead-of-time compiler that emits machine code with no additional dependencies. The compiler should not completely sacrifice dynamic features. The only tool that meets our requirements is the Python compiler called PyPy, see section 3.5.2.

The main useful features of the PyPy compiler for ours intentions are:

- The output code is not emitted directly from the source code but from a dynamically constructed object space and thus really embraces dynamics. In contrast, the PHP compiler, see section 3.5.1, works directly with the source code and thus completely avoids the dynamic features.
- There are several output codes available; though, we employ only C and Java byte-code.
- The translation process is fully modular and easy to tweak. PyPy itself is written in Python.

There are also many features of the Python language that we consider appealing:

- It has strong type system. Implicit conversion in weakly typed languages (PHP, JavaScript, Perl) are considered error-prone.

[9] http://www.ruby-lang.org/

- It is object-oriented. Despite multiple inheritance the object model is simple and will be familiar for average Java developers.
- It has explicit support for exception handling.
- Powerful syntax that allows short but readable code, for instance *list comprehensions.*

There are naturally some risks. The PyPy compiler is a single-purpose program. Its mission is to translate the PyPy interpreter from Python to more efficient code, usually C. The compiler has to be modified in order to translate a general multi-threaded embedded program. The PyPy compiler also introduces some limitations to the language, or, more precisely, to the structure of the compiled program's object space. The set of these limitations is called *Restricted Python*, or *RPython.* RPython properties have to be evaluated in order to determine impact of the restrictions to our development approach. And finally, PyPy is relatively mature project with more than seven years of history; however, it is a research project that does not necessarily meet industry quality standards.

PyPy is still in active development. This work is based on PyPy development snapshot from August 2008.

5.3.3 Generating Target Code

We have defined that the final program has to be in the form of native machine code. There are many machine codes, each for every CPU family; we do not want to limit our development approach to any particular platform.

An obvious solution of this issue is not to directly generate the platform specific assembly code but to generate some platform independent intermediate code. Natural candidate for this *portable assembly code* is C language. It is cross platform but sufficiently low-level to provide precise control over the final machine code.

With standard C as an intermediate code, we also gain entire ecosystem of tools developed for this language. The most important is that we can utilize optimizations that are provided by contemporary C compilers.

Standard C is the primary output code of the PyPy compiler so we get the support for various platforms for free. It is, however, important to note that the generated C code is hardly human readable. It is really just a platform independent version of the machine code: bodies of functions are unstructured, blocks of code are interconnected by labels and jumps (gotos).

Notable part of the generated C code is reimplementation of some features of Python interpreter. These features fill the gap between the high-level RPython code and the low-level C code.

Python has exceptions, C does not. PyPy compiler has to generate equivalent code by utilizing labels and jumps.

Python is object oriented, C is not. Instead of classes, PyPy compiler generates structures with explicit virtual method tables.

Python interpreter provides garbage collection; C code manages memory manually. PyPy compiler addresses this issue by custom garbage collectors for C code.

We will describe and analyze the PyPy compilation process to specify the exact properties of the generated C code. Chapter 6 is dedicated for such an analysis.

5.3.4 Support for Formal Verification

Only a negligible portion of real software is suitable for direct formal verification. Also specification of requirements is rarely given as a set of formulae. The mathematical proof of correctness of the whole program is not the only goal that is worth reaching.

We view formal methods as an advanced testing tool. It is really hard to prove that a program is ultimately correct; it is, however, not so hard to prove that some particular thread interaction scheme is deadlock free. Moreover, the effort to formally prove even some trivial properties leads to a better structure of the code as the developers have to separate individual aspects of the application more precisely in order to make the verification feasible. The positive influence of the formal methods should work in the same way as in the case of *test-driven development* [58].

Examining Possibilities

There are plenty of ways how to support formal methods and every way has its own pros and cons; there is not any universal verification process. Selection of the verification process depends on the set of properties that needs to be verified.

Thus we can do nothing better than to evaluate some possibilities and choose the most promising one. We are especially interested in explicit model checking because it can be applied to real programs.

First, it does not seem promising to verify the final output code, i.e., C or machine code. This code is full of low-level implementation details. The more abstract the code is, the better.

The process of translation from RPython to C is about adding implementation details. The input high level code is rather abstract, it relies on a garbage collector, assumes objects and exception handling as native features. Contrary, the generated C code explicitly implements these features on the lower level of abstraction. Thus verification of the RPython source code is easier than verification of the C code.

The second possibility is to generate some special version of the final C code. Instead of sequential run, the generated program would enumerate all possible states. This is patterned after the SPIN model checker. SPIN translates a formal model specified in Promela language to C code. The C code is a one-purpose model checker that solves the model checking problem for the specified formal model. So this approach is feasible; however, it would require a tremendous amount of work to implement it.

The third possibility is to add some verification facility to the PyPy compiler itself. During the compilation, PyPy uses a technique called *abstract interpretation* in order to create a data structure called flow-graph from the Python byte-code. The abstract interpreter can be possibly tweaked into a model-checker that enumerates the program's state space. The result would be an explicit model checker patterned after Java Pathfinder. We believe that this approach is also possible; however, also requires amount of of work we can not afford.

The fourth possibility relies on a simpler PyPy modification. Instead of generating C code, we would generate an input code for some verification tool, such as the SPIN model checker. The main disadvantage is that Promela is not a general purpose programming language. It lacks some features such as floating point computations and synchronization primitives based on channels differ significantly from common locks and semaphores.

The Selected Tool: Java Pathfinder

We finally decided for the fifth way. We let the PyPy compiler to generate Java byte-code, then we verify the code by explicit model checker called Java Pathfinder.

Java Pathfinder is one of the most known verification tools, see also section 3.1.3. Its main advantage is that it does not verify any special purpose modeling language but the real program code, literally, Java byte-code.

Java byte-code can be generated by the PyPy compiler more straightforwardly than C code because RPython and Java are similar in many ways. Both languages are object-oriented, garbage collected, have exception handling.

We use the Java byte-code as a modeling language. It is slightly more abstract than the C code we intend to deploy. On the other hand, the abstraction gap between C and Java-byte-code is not so wide as in the case of, say, Promela, because Java byte-code is a binary representation of general-purpose programming language.

To make the verification process meaningful, we have to guarantee that the C code and the Java byte-code are equivalent. It is obvious that the two codes that are run on different platforms can not behave exactly equally. However, there are also many properties that are the same in both codes. For instance, the deadlock found by Java Pathfinder in the Java byte-code means that there is certainly a deadlock in the C code—under the condition that the semantics of synchronization objects is equal.

Our verification process also can take advantage of dynamic approach. The Java Pathfinder verifies a real program code. However, in practice, one often have to build a simplified version of the program to make the verification feasible, e.g., extract the critical part. The dynamic nature of Python—the dynamic creation of program's object space—may help here. As object interconnections in dynamic languages are more loose, it is easier to generate a version of the program that contains some mockup objects instead of full-featured objects.

5.3.5 Refining the Development Approach

After we have selected some concrete tools, 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 5.2.

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.

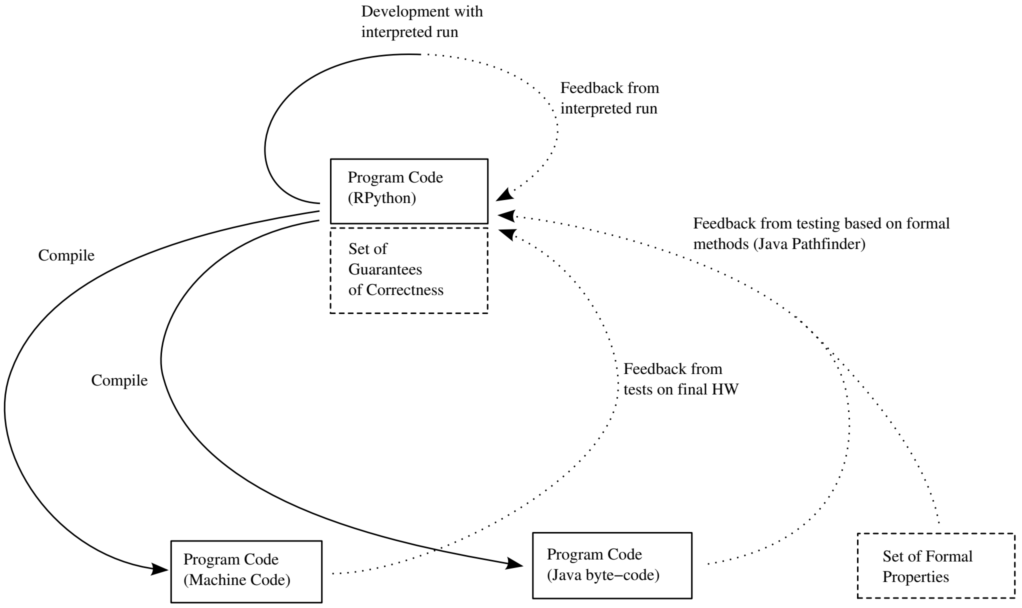


Figure 5.2: Refined Scheme of the Proposed Development Approach

5.4 Conclusion

We have specified a domain for our development approach: the software and hardware platform and the range of applications it is suitable for. We have also selected the verification tool.

Real results of our development approach heavily depend on the PyPy compiler. PyPy defines a set of limitations of the source code (RPython), it is responsible for efficiency of the output code and finally, it generates code that is an input for the model checker. That is why we have to seriously analyze this tool and the compilation process. Chapter 6 describes the relation between generated C and Java byte-code, chapter 8 describes the verification procedure in details.

Chapter 6

Analysis of the PyPy Compilation Process

This chapter is based mostly on two sources, the first is a technical report *Compiling Dynamic Language Implementations* [49], the second is an analysis of the PyPy source codes. Properties of the PyPy compilation process have crucial impact on usefulness of the proposed development process.

The compilation process is a bit unusual so we first provide a short overview. Then we describe the subset of Python that makes the compilation feasible. Then we dive deeper into the technical details of the compilation process: abstract interpretation, flow graph, type inference, flow graph transformations.

The main achievement of the chapter is a formal definition of the flow graph and formally described flow graph transformation.

Last, but not least, we deal with generation of the output code (C or Java byte-code).

6.1 Translation Process Overview

Input of the translation process is a source code in the form of Python language. The source code has to meet some guidelines that are called *Restricted Python*, or, RPython. The output of the translation process is, for our purposes, C code Java byte-code.

In section 4.3.2, we mentioned the *two-stage design.* The PyPy compiler not only embraces this idea but actually enforces it. The first stage that contains definitions of data structures, classes, methods, and standalone functions is executed by the PyPy interpreter as a standard Python program. The result of this step is an initialized program's object space, i.e., all classes and their methods are defined. Object construction is not limited by RPython constrains; one can build classes and methods with the help of all dynamic features, including *eval*, which is crucial for adoption of paradigms such as AOP and DbC.

Execution of the "second stage" would follow after the object space initialization. If the Python program is run by an ordinary interpreter (CPython), the code after the initialization part is simply executed.

As the PyPy compiler tries to compile the program, not to run it, the "second stage" processing is slightly different. Instead of ordinary interpretation, the *abstract interpretation* is used. The PyPy abstract interpreter does not execute Python byte-code instructions but instead records them into a data structure called *flow graph*. A set of flow graphs—one for each function—can be viewed an alternative representation of the program's object space.

To make the abstract interpretation feasible (finite), it has to run only over frozen program's object space. When the abstract interpretation is in progress, no new classes or methods can be introduced. From this point on, RPython guarantees that the program code is as static as for instance in Java.

When the abstract interpretation is finished, then several flow graph transformations are performed. The first transformation adds a type annotation: static data types are inferred and type information is added to every data field in the flow graph. From this point on, the program is completely statically typed. To make the type inference feasible, RPython imposes another set of constrains to the code, for instance a single variable can not change types over time.

Up to this point, the compilation process is independent of any target platform; the ongoing step depends on the selected target platform. The data types inferred are rather abstract, i.e., do not define the binary representation. In order to generate some real target code, the abstract types have to be mapped to the real data types of the target platform: for instance, an abstract integer becomes C's or Java's *int*.

Subsequent transformations deal with limitations of target platforms. In the case that the target platform lacks an explicit exception handling or garbage collection, these aspects are added to the flow graph. For instance, exceptions can be implemented by additional jumps and labels; garbage collection can be implemented by adding of reference counters.

When the flow graph is at the level of abstraction of the target platform, the final code is generated.

Overall compilation scheme is depicted in figure 6.1

6.2 Restricted Python

Restricted Python (RPython) is a subset of the Python programming language. Every RPython program is also a valid Python program. Goal of the restrictions

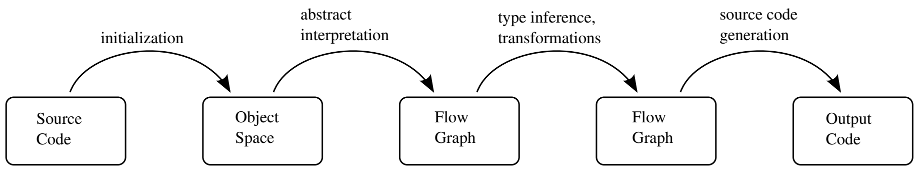


Figure 6.1: Overall Compilation Scheme

defined by RPython is to enable translation of RPython to a lower level static code.

The restrictions are not defined on the syntax level. The restriction is rather *temporal*: during the construction of the program's object space (that we call "the first stage") the usage of the Python constructs is unlimited. Code of the "second stage" can only use the classes and methods that were build in the "the first stage".

The main difference between RPython and general Python is that the first and second stages are strictly separated in RPython. In general Python, bits of the "first stage code" and the "second stage code" can interleave, i.e., the object space can be altered at any point during the execution.

The structure of the program's object space has to comply with another set of restrictions because we have to assign a static data type to every variable. Also some more exotic language constructs such as *iterators* or *generators* are either limited or not allowed during "the second stage" because the authors of the PyPy compiler did not feel the need to support them.

This work is based on the PyPy interpreter that implements the version 2.4 [59] of the Python language. So RPython is subset of Python 2.4.

The following sections describe the limitations of the code that is the result of the first stage.

6.2.1 Primitive Data Types

In dynamic languages, the type information is connected with a *value* rather than with a *variable*. Therefore, a single variable can contain values of different types over time. Contrary, RPython requires that the type of a *variable* has to be static. So the following code is valid Python code but not valid RPython code:

```
a = 42
a = "a string"
```

It is important to note that the type of a variable is never explicitly defined by a programmer in RPython, he or she only defines it implicitly by values that are assigned to the variable. Static data types are inferred by the PyPy compiler.

Numeric Data Representation

Integers

Python has two integer data types: *int* and *long*. The size of the first type equals the size of the machine integer; that is 32 bits if we assume 32-bit target platform. This type is straightforwardly rendered as a 32-bit *long* in C and as 32-bit *int* in Java byte-code.

Python's *long* is an infinite size integer and this type is unsupported by RPython. In the CPython interpreter every *int* can be silently converted to *long* if it would overflow. Contrary, *int* in a RPython program translated to C or Java byte-code overflows silently. So this is a point in which the compiled and the interpreted RPython programs may differ in behavior and programmers have to be careful. Integer arithmetic error handling is consistent on all platforms; when dividing an integer by zero, *ZeroDivisionError* exception is always raised.

Floats

Python has one floating point type called *float* that is implemented by the double precision format as specified by IEEE 754 and therefore by *double* data type as specified by C and Java. RPython also supports computations with special values such as *NaN* and *INF*; however, there is one notable difference between interpreted and compiled RPython. When dividing a float by zero, CPython interpreter raises *ZeroDivisionError* whereas after translation to C or Java byte-code the expression is evaluated as *INF* or *NaN*. So this corner case have to be carefully handled by programmers.

Booleans

Python has a type called *bool* with the domain {*True*, *False*}. In Java byte-code, this type is represented by Java's *boolean* data type whose bit size depends on JVM implementation (usually 8 bits). In the generated C code, this type is declared as *bool_t* whose bit size usually equals the bit size of *int*.

Strings

Python strings are internally represented as arrays of bytes. There is no special data type for a single character; a character is simply a string of length one.

RPython does not impose any limitation on strings; however, the internal representation on target platforms may differ. In C, the string is represented as an array of 8-bit chars; in the case of Java, native *String* is used, i.e., every character is converted to 16-bit Java's *char*.

Python also provides Unicode strings in UCS-2 encoding, i.e., every character is represented by a 16-bit value. If translated to Java byte-code, the native *String* can still be used[1]. For C, the PyPy translator uses an array of 32-bit integers, i.e., it encodes strings by UCS-4/UTF-32.

6.2.2 Compound Data Types

The standard Python list—more precisely resizable arrays—can contain items of various types. RPython requires that all items of a particular list have to be of one type. Statement *a = [1,2,"three"]* is valid in Python but invalid in RPython.

Tuples—immutable lists—are not type restricted, thus *b = (1,2,"three")* is valid in RPython. There is, however, one limitation of tuples in RPython: tuples can be indexed only by a constant value and therefore elements of a tuple can not be iterated by a for-cycle.

Dictionaries—associative arrays—are also restricted. All keys have to be of one type and also all values have to be of one type. Again, the following code is valid in Python but invalid in RPython:

```
d = {}
d[1] = 42
d["two"] = "a string"
```

6.2.3 Classes

In standard Python, a class definition is never closed, i.e., it is possible to arbitrary add or remove data fields and methods at any time. RPython class definitions are closed as, for instance, in Java.

Class hierarchy in standard Python can use multiple inheritance. RPython allows only simple inheritance.

However, RPyhon supports one object-oriented feature that goes beyond standard single-inheritance: *mixins*. *Mixins* are, for instance, used in Ruby. The basic principle is that a class can import, i.e., *mix-in*, a set of methods that are syntactically defined in some other class.

[1] Java uses UTF-16 which is a superset of UCS-2.

```
class ProvidesInfo:
    _mixin_ = True
    def getInfo(self):
        return self.info

class Shape:
    pass

class Circle(Shape, ProvidesInfo):
    def __init__(self, r):
        self.r = r
        self.info = "A Circle"

class Constant4(ProvidesInfo):
    def __init__(self):
        self.value = 4
        self.info = 4

circle = Circle()
constant4 = Constant4()
print circle.getInfo()
print constant4.getInfo()
```

Figure 6.2: Two Unrelated Classes Use Mixin

Unlike methods, class data fields are not mixed-in. However, a mixed-in method can access data fields of the class it was mixed into.

It is important to note that mixins are fundamentally different from Java-style interfaces. When two unrelated classes mix-in methods from a third class, the two classes will remain without any relation. In the contrary, when two unrelated Java classes implement one interface, they gain a relationship. Usage of mixins is equivalent of pasting the source code of the mixed-in methods into the target class's source code. At run-time, a class does not have a notion about which methods were defined directly and which were mixed-in. See example in figure 6.2.

If the RPython program runs in standard Python interpreter, mixing-in is implemented through multiple inheritance. The example piece of code is valid in RPython and thus also in Python.

6.2.4 Memory Model

In Python, all data fields are accessed through references. The consequence of this fact is that it is possible to use special object *None* as a value for every variable. It is similar to Java's *null*.

In PyPy, however, some types do not use references, i.e., they are always passed by value. Literally, they are: integers, floats and tuples.

If we use Java terminology: while in Python integers behave as instances of the *Integer* class, in PyPy, integers behave as the primitive type *int*.

Thus the following code is not valid RPython code:

```
def inc(n):
    if n is None:
        return 1
    else:
        return n + 1

assert 1 == inc(0)
assert 1 == inc(None) # Type inference fails.
```

The reason for this change of the memory model is code efficiency. Optimization also stands behind Java's *Integer/int* ambivalence.

6.2.5 Functions

Default Argument Values

Python has the ability to provide default values of function arguments; these arguments can be omitted when the function is called. RPython supports this feature. Thus the following code is valid RPython code.

```
def compute(n, addition=0, substraction=0):
    return n + addition - substraction

assert 10 == compute(10)
assert 15 == compute(10, 5)
assert 14 == compute(10, 5, 1)
assert  6 == compute(10, substraction=4)
```

Variable Number of Arguments

Python functions can have variable number of arguments; inside the function, the arguments are accessed via a sequence, i.e., list or tuple, or via a dictionary. RPython limits this feature, the arguments can be stored only in tuple. Recall that RPython tuples can be indexed only by a constant which is at this point really limiting.

The code in figure 6.3 demonstrates variable number of arguments in RPython.

```
def sum(*args):
    l = len(args)
    if l == 1:
        return args[0]
    elif l == 2:
        return args[0] + args[1]
    else:
        raise Exception();

assert 2 == sum(2)
assert 5 == sum(2,3)
a_tuple = (2,3)
assert 5 == sum(*a_tuple)
```

Figure 6.3: Variable Number of Function's Arguments

6.2.6 Advanced Language Constructs

Python provides some language constructs that are only for convenience, i.e., it is only a "syntactic sugar"; equivalent functionality can be expressed by a longer code.

List Comprehensions

List comprehensions are supported by RPython. List comprehensions enable a list to be build by a *for-cycle*. For instance, list of squares of even numbers up to 10 can be denoted as.

```
numbers = [i*i for i in range(0, 10) if i % 2 == 0]
# The result is [0, 4, 16, 36, 64].
```

Iterators

Objects that support *iterator protocol* [60], i.e., implement certain methods, are *iterable*. Iterable objects can be directly used as a data source in for-cycles. All Python compound data types are iterable.

In RPython, only built-in compound data types, e.g., a list, can be directly used in for-cycles. For user-defined iterable objects, one have to use less elegant code.

Generators

Generators [61] are a convenient way how to implement iterable objects. Generators are not supported in RPython.

Decorators

Decorator [62] is a construct for wrapping-up a code of a function into another function. They are supported in RPython.

It can be viewed as a kind of support for aspect oriented programming. The following code defines a decorator for tracing and *decorates* function *foo*.

```
def trace_decorator(func):
    def body():
        print "before"
        func()
        print "after"
    return body

@trace_decorator
def foo():
    print "foo"
```

6.2.7 Evaluation of Restrictions

The "second stage" code that must obey RPython rules does not lose much of its expressiveness. During our experiments regarding this work, we ported some programs from general Python to RPython without troubles. Properly designed programs seldom need data containers that store items of unrelated data types; true multiple class inheritance is also often considered as a bad practice. Occurrence of more advanced features such as generators is rare in real programs and it can be always easily avoided.

6.3 Abstract Interpretation

Abstract interpretation is a method that PyPy utilizes to convert an initialized program's object space to a structure called *flow graph*. Program's object space is a Python-specific in-memory data structure that represents the entire program. It is de-facto defined by CPython and reimplemented by the PyPy interpreter. It consists mainly of hash maps that map identifiers (symbols) to pieces of code and other objects. The code itself is in the form of Python byte-code.

For better illustration, see the conversation in Python interactive console in figure 6.4. The conversation starts with a definition of a function *myfunc*, then the function is used with an argument of value five. Subsequently, we investigate two self-descriptive attributes of the function: *__name__* and *func_code*. The byte-code is listed with the help of the disassembler from the Python standard library. The last statement (*locals*) demonstrates that the defined function itself

```
>>> def myfunc(n):
...     return n+1
...
>>> myfunc(5)
6
>>> myfunc.__name__
’myfunc’
>>> myfunc.func_code
<code object myfunc at 0x8623650, file "<stdin>", line 1>
>>> import dis # Import the module for disassembling.
>>> dis.disassemble(myfunc.func_code) # Print the byte-code.
  2           0 LOAD_FAST                0 (n)
              3 LOAD_CONST               1 (1)
              6 BINARY_ADD
              7 RETURN_VALUE
>>> locals()
{’myfunc’: <function myfunc at 0x862f3e4>,
 ’__builtins__’: <module ’__builtin__’ (built-in)>,
 ’__name__’: ’__main__’,
 ’__doc__’: None,
 ’dis’: <module ’dis’ from ’/usr/lib/python2.5/dis.pyc’>}
```

Figure 6.4: Conversation in Python Interactive Console

is contained in the hash-map that maps local symbols to objects. The object space that is a result of the definition of one function (*my_func*) and import of one module (*dis*) is depicted in figure 6.5.

Another example: an object space that is the result of the code from section 4.3.2 is depicted in figure 6.6. The *first stage* part of the code builds classes *C* and *D*.

The abstract interpretation starts from a selected entry point; it is an equivalent of the *main* function known from C. In RPython, the entry point is a function that is returned by the function *target*. The code in figure 6.7 is a *Hello world* program directly compilable by the PyPy compiler.

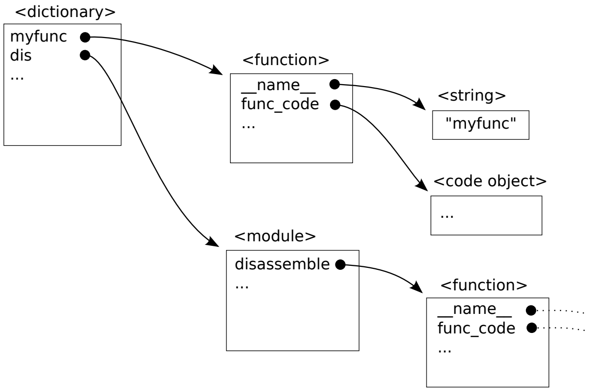


Figure 6.5: Objects Space Built in Interactive Console

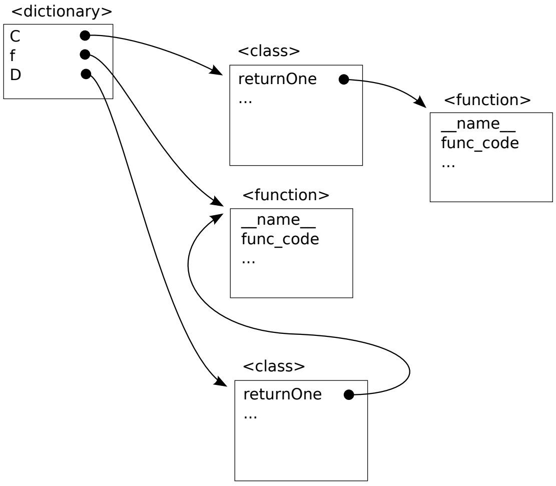


Figure 6.6: Object Space of Example from Section 4.3.2

```
def my_entry_point(argv):
    print "Hello, world!"
    return 0

def target(*args):
    return my_entry_point, None
```

Figure 6.7: A Program Compilable by PyPy

The start of execution of the entry point function is exactly the point from which the dynamic features are restricted; it is the start of the "second stage" that goes after the "first", i.e., initialization, stage.

The PyPy abstract interpreter starts to interpret the entry point function. It fetches a byte-code instruction but instead of executing it, it records an equivalent action to the *flow graph*. The abstract interpreter also maintains the state of the current stack frame. Thus not every abstractly interpreted byte-code instruction results in a new record in the flow graph: if the abstract interpreter sees that the state was already seen, it only adds appropriate link to the graph. This procedure is used mainly for loop detection, so a loop in the source code is naturally represented by a loop in the flow graph.

There is another important point in which the abstract interpretation differs from the ordinary interpretation: program branching, or, more technically, conditional jumps. The ordinary interpreter always selects only one code path according to the actual result of the condition test; the abstract interpreter has to go through both code paths. Thus every time the abstract interpreter reaches a conditional jump, both possible code paths are scheduled for subsequent interpretation.

We stated that the program's objects space has to be frozen when it is abstractly interpreted. It is not exactly true, in fact, the object space can be altered during the "second" stage in some cases. However, this alteration can occur only bounded number of times. For our purposes, we assume program's object space to be always frozen after its initialization.

The PyPy compilation process fails if

- the first, i.e., initialization, stage does not terminate, or
- abstract interpretation itself results in a new code in the object space that needs to be abstractly interpreted, unbounded number of times.

These two pathological cases can be reached only intentionally.

6.4 Flow Graph

In the previous sections, we mentioned the term *flow graph* many times. It is the key data structure of the translation process and thus worth a precise definition. We extracted the formal definition by precise analysis of the PyPy source code.

Flow graphs are built by the PyPy abstract interpreter. There is one flow graph per function. Note that methods of objects are handled as plain functions; methods are just functions whose first argument is an object instance. Thus a set of flow graphs *GRAPHS* is an alternative representation of initialized program's object space.

$$GRAPHS = \{G_1, G_2, ..., G_N\}$$

When the flow graph is created, it does not depend on the selected target code—C or Java byte-code. Later we will introduce a set of flow graph transformations. The information that is added by a transformation may depend on the selected target back-end. There is also one flow graph transformation that changes the graph topology.

6.4.1 Syntax

Graph

Flow graph is an oriented graph with some additional attributes. Formally, we define it as

$$G = \langle V, E, start, return, EXC, CONST \rangle,$$

where:

- V is a set of vertexes; a vertex denotes a block of code,
- E is a set of edges; an edge denotes a jump from one block to another,
- $start \in V$ is a start block, i.e., an entry point of the function,
- $return \in V$ is a return block, i.e., an ordinary exit point of the function,
- EXC is either an empty set or $\{v_{exc}\}$, where $v_{exc} \in V$ is an exception block, i.e., the exit block of the function in which en exception is thrown out of the function,
- $CONST$ is a set all constants used in the graph.

Every flow graph has exactly one start block and exactly one return block. If the function may raise an exception, there is one exception block. If the function can not raise an exception, EXC is empty. Constants are just literals of any type, variables are also untyped.

Blocks

A block of code $v \in V$ is defined in the following way:

$$v = \langle VAR, last\_exception, IN, OP, SWITCH \rangle,$$

where:

- VAR is a set of block's variables,
- $last\_exception \in VAR$ is a variable dedicated for storing a raised exception object instance,
- $IN = \langle in_1, in_2, ...in_n \rangle, in_i \in VAR$ is a n-tuple of block arguments,
- OP is an n-tuple of block's operations,
- $SWITCH$ is a set containing an optional exit switch variable. $SWITCH$ is $\{var\}$, $var \in VAR$ or an empty set if the exit switch is not defined. Exit switch variable serves as the condition in the conditional jump. According to the value of this set, an outgoing block's edge is selected. For blocks that end with unconditional jump the set is empty.

Blocks can use every constant defined in the graph. Variables are used for storing intermediate results. They are defined locally and are not shared among multiple blocks. Variables may contain either a direct value or a reference to an object. The variables that contain references may also have special value *None* that denotes an undefined reference; it is an equivalent of *null* from Java.

Then, there is an n-tuple of block's operations

$$OP = \langle op_1, op_2, ..., op_m \rangle,$$

$$op_i = \langle name, ARG, result, last\_exception \rangle,$$

where

- $name$ is the name of the operation,
- $ARG = \langle arg_1, arg_2, ..., arg_k \rangle$, $arg_i \in CONST \cup VAR$ is a n-tuple of operation's arguments,
- $result \in VAR$ for storing ordinary result of the operation
- $last\_exception \in VAR$ for storing exceptional result of the operation.

Edges

An edge $e \in E$ is defined as a n-tuple

$$e = \langle v_1, v_2, CASE, EARG \rangle,$$

where

- $v_1, v_2 \in V$ are the source and the target block,
- $CASE$ is either $\{const\}$, $const \in CONST$, or an empty set,
- $EARGS = \langle val_1, ..., val_k \rangle$, $val_i \in CONST \cup VAR(v_1)$ is a tuple of edge's arguments. These arguments are used for the data transfer from one block to another.

6.4.2 Semantics

We described the syntax of flow graph; to make the definition complete, we provide a description of its semantics.

A flow graph represents a single function in a program written in RPython. The semantics of the flow graph is defined by the rules of its interpretation.

First, we describe the interpretation in a situation without occurrence of exceptions. Then we describe how exceptions are raised and handled.

Ordinary Interpretation

The interpretation starts in block $start \in V$. The arguments of the function that the flow graph represents are assigned to the tuple of block's input variables IN. Then the block's operations are interpreted.

Every operation takes a n-tuple of arguments. An argument is either a constant or a variable. An operation saves the result to the variable *result*. Every variable can be assigned only once within a block, i.e., every variable from VAR can be used at most once as the result of an operation. The list of all possible operations that abstract interpreter can emit is in tables 6.1 and 6.2.

When the block's code is finished, an outgoing edge has to be selected in order to perform a jump to another block. The selection is based on the $SWITCH$ set defined by the block and the $CASE$ set defined for every outgoing edge. The selection is defined by two rules:

1. if $SWITCH = \emptyset$ then there is only one edge, an edge with empty $CASE$; so this edge is followed,

Operation Name	*Description*	*Special method*
abs	Absolute value	__abs__
add	+ operator (adding, concatenating)	__add__
and	& operator (bit and)	__and__
call_args	call a function with default arguments	
contains	search an object in a container	__contains__
delitem	removes an object from a container	__delitem__
floordiv	// operator (dividing with floor)	__floordiv__
div	/ operator (dividing)	__div__
eq	== operator (equivalence of objects)	__eq__
ge	>= operator	__ge__
gatattr	get attribute of an object	__getattribute__
getitem	get item from a container	__getitem__
gt	> operator	__gt__
hash	object hash code	__hash__
hex	hexadecimal representation of integer	__hex__
inplace_add	+= operator	__iadd__
inplace_and	&= operator	__iand__
inplace_div	/= operator	__idiv__
inplace_floordiv	//= operator (inplace division, result floored)	__ifloordiv__
inplace_lshift	<<= operator	__ilshift__
inplace_mod	%= operator	__imod__
inplace_mul	*= operator	__imul__
inplace_or	|= operator	__ior__
inplace_pow	**= operator	__ipow__
inplace_rshift	>>= operator	__irshift__
inplace_sub	-= operator	__isub__
inplace_xor	^= operator	__ixor__
invert	~ operator (bitwise invert)	__invert__
*is_*	object identity	
is_subtype	inheritance test	
is_true	boolean representation of an object	
iter	get object's iterator	__iter__
le	<= operator	__le__
len	get object's length	__len__
lshift	<< operator	__lshift__
lt	< operator	__lt__

Table 6.1: Flow Graph Operations, Part 1

Operation Name	*Description*	*Python special method*
mod	% operator (modulo)	__mod__
mul	* operator (multiplication)	__mul__
ne	!= operator	__ne__
neg	- unary operator	__neg__
newdict	create a new dictionary	
newlist	create a new list	
newslice	create a new slice	
newtuple	create a new tuple	
next	moves iterator forward	
oct	octal representation of integer	__oct__
or	| operator (bitwise or)	__or__
ord	integer ordinal of a character	
pos	+ unary operator	__pos__
pow	** operator (power)	__pow__
rshift	>> operator	__rshift__
setattr	set attribute of an object	__setattr__
setitem	set item of a container	__setitem__
simple_call	call a function	
str	string representation of an object	__str__
sub	- operator (substracting)	__sub__
type	object's type	
xor	^ operator (bitwise xor)	__xor__

Table 6.2: Flow Graph Operations, Part 2

2. else $SWITCH = \{var_{switch}\}$. Then all edges have nonempty $CASE_e = \{const_e\}$. An edge is selected if the values of var_{switch} and $const_e$ equal.

When an outgoing edge is selected, execution moves to the target block of the selected edge. The target block's arguments IN are initialized by values of edge's arguments *EARG*. Both *IN* of the target block and *EARG* that leads to the target block are n-tuples of the same size.

There is always one block that has no outgoing edge, it is block $return \in V$. The return block does not have any operations. Return block's argument *IN* is the return value of the function that the flow graph represents.

Exception Raising

Exception can be explicitly raised by reaching an exception block $v_{exc} \in EXC$. There is zero or one exception block per flow graph. The exception block does not have any operations. Input arguments of the exception block $IN(v_{exc})$ are exception's data, i.e., exception instance and exception class. After executing the exception block, the exception is thrown out of the function. Note that explicit exception are instantiated by the Python keyword *raise*.

The second situation in which an exception is thrown out of the function is *unhandled exception*. Some operations can end with an exception. If so, then the exception instance is stored in *last_exception* $\in$ *VAR*. If the exception is not handled, then execution of the function is immediately terminated and control is returned to the caller function.

Exception Handling

The exception is *handled* if the variable that contains the exception object instance, i.e., *last_exception*, is immediately used as a block exit switch, i.e., *last_exception* $\in SWITCH$. This also means, that only an exception of the last block's operation can be *handled*.

For exception handling, the outgoing edges are arranged in the following way:

- The first edge is for the situation when no exception occurred, i.e., $CASE(e_1) = None$.

- Every other edge handles a specific class of exceptions, i.e., $CASE(e_i), i = 2, 3, ..., N$ contain an exception class object such as *IndexError* or *AritmeticError*. The order of the edges is the same as the order of *except* blocks in the source code.

```
def factorial(n):
    if n < 0:
        raise ArithmeticError

    result = 1

    while n > 0:
        result *= n
        n -= 1

    return result
```

Figure 6.8: Factorial, Exception Raising

Formally, we define a function *edgeorder* that assigns a natural number to each edge, that is:

$$edgeorder : E \longrightarrow \mathbb{N}.$$

For all edges leading from a particular block $v \in V$ the *edgeorder* defines a sequence $1, 2, ..., N$ where N is the number of outgoing edges from v while maintaining the ordering form source code as mentioned above.

The outgoing edge is then selected by the following rules:

1. if the *last_exception* is *None* then the first edge is selected,

2. else *last_exception* is an exception instance. Edges $e_i, i = 2, 3, ..., N$ are examined in the order given by *edgeorder*. En edge e_i is selected if the type of *last_exception* is *subclass*[2] of $CASE(e_i)$.

Example: Factorial Flow Graph

For better understanding, see figure 6.9 in which the flow-graph of a function that calculates factorial is depicted. The functions raises *ArithmeticError* if its argument is illegal. The computation itself is exception free because RPython does not check integer overflows—thus *last_exception* is never assigned or checked. Therefore, instead of ⟨*resutl, last_exception*⟩ = *operation(arg1, ...)*, we write only *result = operation(arg1, ...)*.

The exact RPython code of the function is in figure 6.8.

The while-loop is represented by a loop in the graph. Interpretation of the flow graph starts in the start block and ends either in the return block or in the

[2] There is a class hierarchy of exception types.

exception block. Note that we named all variables of the graph by hand; these variables are generated by the compiler.

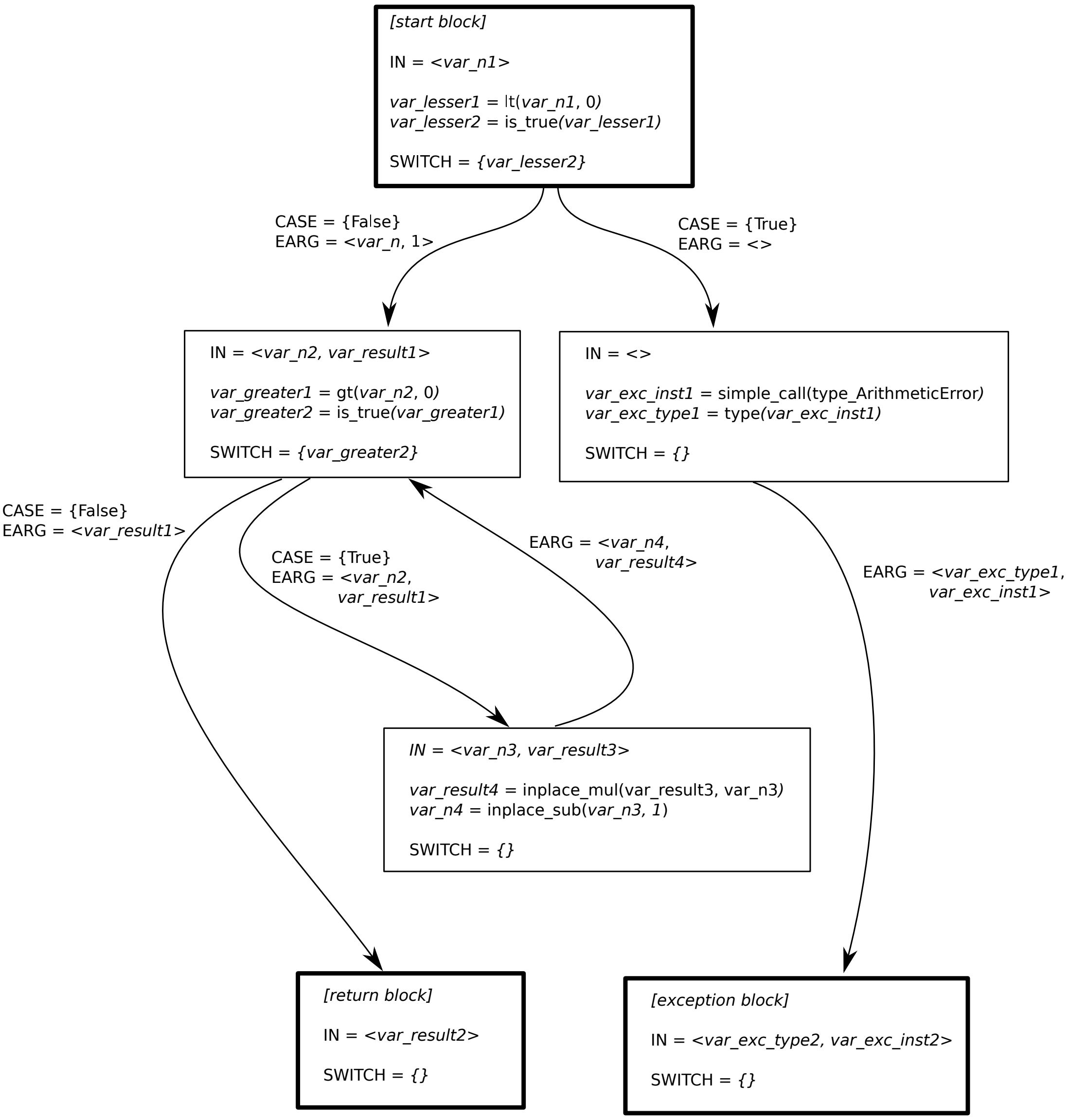

[start block]
IN = <var_n1>
var_lesser1 = lt(var_n1, 0)
var_lesser2 = is_true(var_lesser1)
SWITCH = {var_lesser2}
CASE = {False}
EARG = <var_n, 1>
CASE = {True}
EARG = <>
IN = <var_n2, var_result1>
var_greater1 = gt(var_n2, 0)
var_greater2 = is_true(var_greater1)
SWITCH = {var_greater2}
IN = <>
var_exc_inst1 = simple_call(type_ArithmeticError)
var_exc_type1 = type(var_exc_inst1)
SWITCH = {}
CASE = {False}
EARG = <var_result1>
CASE = {True}
EARG = <var_n2, var_result1>
EARG = <var_n4, var_result4>
EARG = <var_exc_type1, var_exc_inst1>
IN = <var_n3, var_result3>
var_result4 = inplace_mul(var_result3, var_n3)
var_n4 = inplace_sub(var_n3, 1)
SWITCH = {}
[return block]
IN = <var_result2>
SWITCH = {}
[exception block]
IN = <var_exc_type2, var_exc_inst2>
SWITCH = {}

Figure 6.9: Flowg Graph of Factorial

6.5 Type Inference

When the flow graphs are created, they have no notion of the types of the data fields. In order to generate statically typed target code, static types have to be inferred.

PyPy defines a set of abstract, i.e., platform independent, types. These types are called *annotations*. We will use this term in order to distinguish the abstract types from the native types of a particular target platform. The component that performs type inference is internally called *annotator*.

The annotator takes a set of flow graphs representing the entire program as an input and assigns a type annotation to every data field. Formally, we define a function that assigns annotations to variables and constants of all flow graphs:

$$VAL = \bigcup_{G_i \in GRAPHS} CONST_{G_i} \cup VAR_{G_i}$$

$$annotation : VAL \rightarrow ANNOTATION$$

A type annotation can be viewed as a set of values that a data field can contain.

Basic hierarchy of annotations is depicted in figure 6.10.

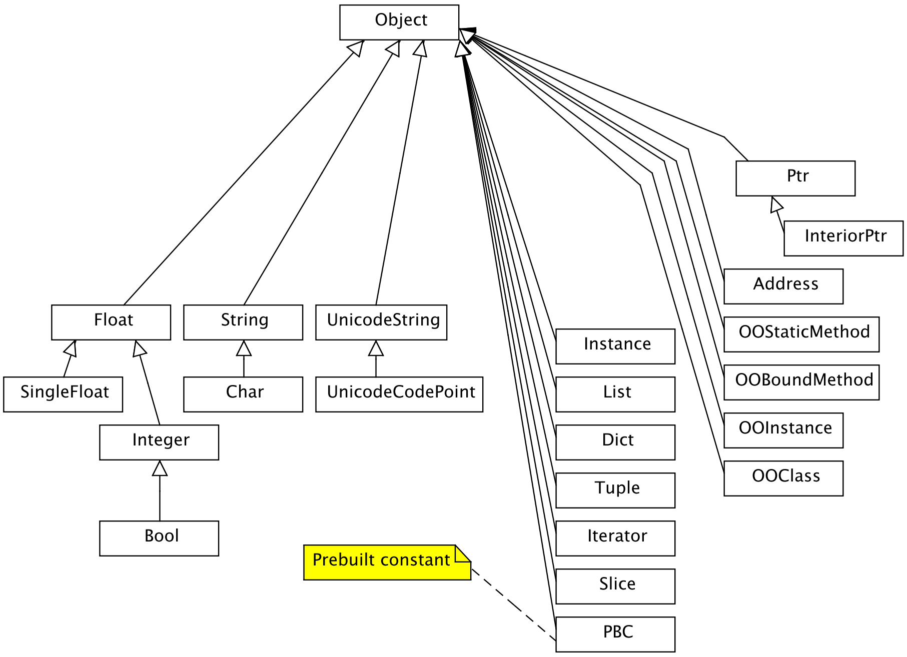


Figure 6.10: Type Annotations Hierarchy

The basic hierarchy is simplified; in fact there is some additional information for every "basic" annotation. For instance, the *integer* annotation also bears the information whether it can contain negative values. Object instance and string annotation have the variant that can contain *None* and the variant that can not.

Lists can be either fixed- or variable-sized. There is a partial ordering defined for annotations. The ordering denotes generality, for instance:

- $Bool \leq NonNegInteger \leq Integer$
- $Char \leq String$
- $Instance(subclass) \leq Instance(class)$, for any class and subclass
- $a \leq b$, for any annotation a with a nullable twin b

The annotation process works as follows. First, the straightforward type annotations of constants are determined. Then the annotator traverses the flow graphs and transitively determines annotations of values created from the values with already known annotation. Because of loops, annotations can not be determined in one pass. The type annotation algorithm is actually a *fix-point search.* In [49], there is provided a proof of *termination* and *soundness* of the annotation algorithm.

The annotator always tries to assign the most precise type annotation; however, if it later learns that a particular annotation is too restrictive, it assigns a more general annotation.

More precise annotations enable better optimizations of the output code. The most general annotation is *Object*; if some data field obtains this annotation, the program cannot be translated to the output static code, the compilation fails.

6.6 Native Operations and Types

Up to this point, the flow graph was completely independent of any properties of the selected output code. Now comes the time for transformations that brings the level of abstraction closer to a particular output code.

6.6.1 RTyper

PyPy has a component called *RTyper* that takes an annotated flow graph as the input and produces a flow graph with native data types and operations of the selected platform (for our purposes, C or Java byte-code). Thus there are two PyPy "lower level" type systems we are interested in; one is simply called *Low Level Type System* and the second is *Object-Oriented Type System.*

RTyper never changes a flow graph's topology, i.e., the number and the structure of vertexes and edges remain always the same. RTyper works only with constants, variables and block's operations.

```
def main(self, argv):
  res1 = Resource()
  w1 = Worker(res1)
  w1.start()
  w1.join()
  return 0
```

Figure 6.11: RTyping - Method Source Code

RTyper assigns a low-level type to every constant or variable. Formally, we define a function

$$lltype : VAL \rightarrow LLTYPE.$$

Some annotations have a straightforward low level representation. For instance the *Integer* annotation is naturally represented by C or Java *int* type.

More complicated annotations such as *List* can have multiple low-level representations. The selection of a particular low level type is based on additional information that the annotation bears. The annotator can possibly determine whether a list is modified at run-time. Another special case are lists that are the result of the Python's *range* function. These lists contain an ordered sequence of integers—*range* is often used by for-cycles.

The ordinary low level representation of *List* is an array. However, for an unmodified list created by *range*, the low level representation can be a special *range object* that contains only the lower and upper bound of the sequence and values of list's items are computed at run-time. It is obvious that this representation saves memory.

RTyper also replaces all high level operations by equivalent low level operations. More precisely, every single high level operation is replaced by one or more low level operations. As every flow graph's variable can be assigned only once, RTyper can also add block's variables to store intermediate results of the low level operations.

Let us illustrate the translation from high level operations to low level operations on an example. In figure 6.11, you can see a method that is first translated by the abstract interpreter to a block containing operations that you can see in figure 6.12. These high level operations can be translated by RTyper to either low level operations for the C backend (see figure 6.13) or for the Java byte-code backend (figure 6.14).

```
v14 = simple_call((classobj Resource))
v15 = simple_call((classobj Worker), v14)
v16 = getattr(v15, ('start'))
v17 = simple_call(v16)
v18 = getattr(v15, ('join'))
v19 = simple_call(v18)
```

Figure 6.12: RTyping - High Level Operations

```
v20 = malloc((GcStruct resource.Resource), ({'flavor': 'gc'}))
v21 = cast_pointer(v20)
v22 = setfield(v21, ('typeptr'),
         (<* struct object_vtable { subclassrange_mi...=... }>))
v23 = direct_call((<* fn Resource.__init__>), v20)
v24 = malloc((GcStruct worker.Worker), ({'flavor': 'gc'}))
v25 = cast_pointer(v24)
v26 = setfield(v25, ('typeptr'),
         (<* struct object_vtable { subclassrange_mi...=... }>))
v27 = direct_call((<* fn Worker.__init__>), v24, v14)
v16 = same_as(v15)
v28 = cast_pointer(v16)
v29 = getfield(v28, ('typeptr'))
v30 = cast_pointer(v29)
v31 = getfield(v30, ('cls_start'))
v32 = direct_call((<* fn Thread.start>), v16)
v18 = same_as(v15)
v33 = cast_pointer(v18)
v34 = getfield(v33, ('typeptr'))
v35 = cast_pointer(v34)
v36 = getfield(v35, ('cls_join'))
v37 = direct_call((<* fn Thread.join>), v18)
```

Figure 6.13: RTyping - Low Level Operations for C

```
v6 = new((<Instance(resource.Resource)>))
v7 = direct_call((sm Resource.__init__), v6)
v8 = new((<Instance(worker.Worker)>))
v9 = direct_call((sm Worker.__init__), v8, v0)
v2 = same_as(v1)
v10 = oosend(('ostart'), v2)
v4 = same_as(v1)
v11 = oosend(('ojoin'), v4)
```

Figure 6.14: RTyping - Low Level Operations for Java Byte-code

6.6.2 Low Level Type System

The type system that is used for the C backend is called *Low Level Type System*. The type system hierarchy depicted in figure 6.15 is richer than the hierarchy of annotations—figure 6.10—because it deals with some implementation details.

Types *Array* and *Struct* have twins called *GcArray* and *GcStruct*. The types with *Gc*- prefix are allocated on the heap and managed by the garbage collector; they may have some additional data fields such as reference counters. Contrary, types without the support for garbage collection can be allocated only on the stack.

RPython is an object oriented language; however, the Low Level Type System does not have explicit support for objects and classes. All the OOP related stuff have to be expressed by some low level primitives. For instance a class is just a structure with a virtual method table stored as an array of pointers to functions.

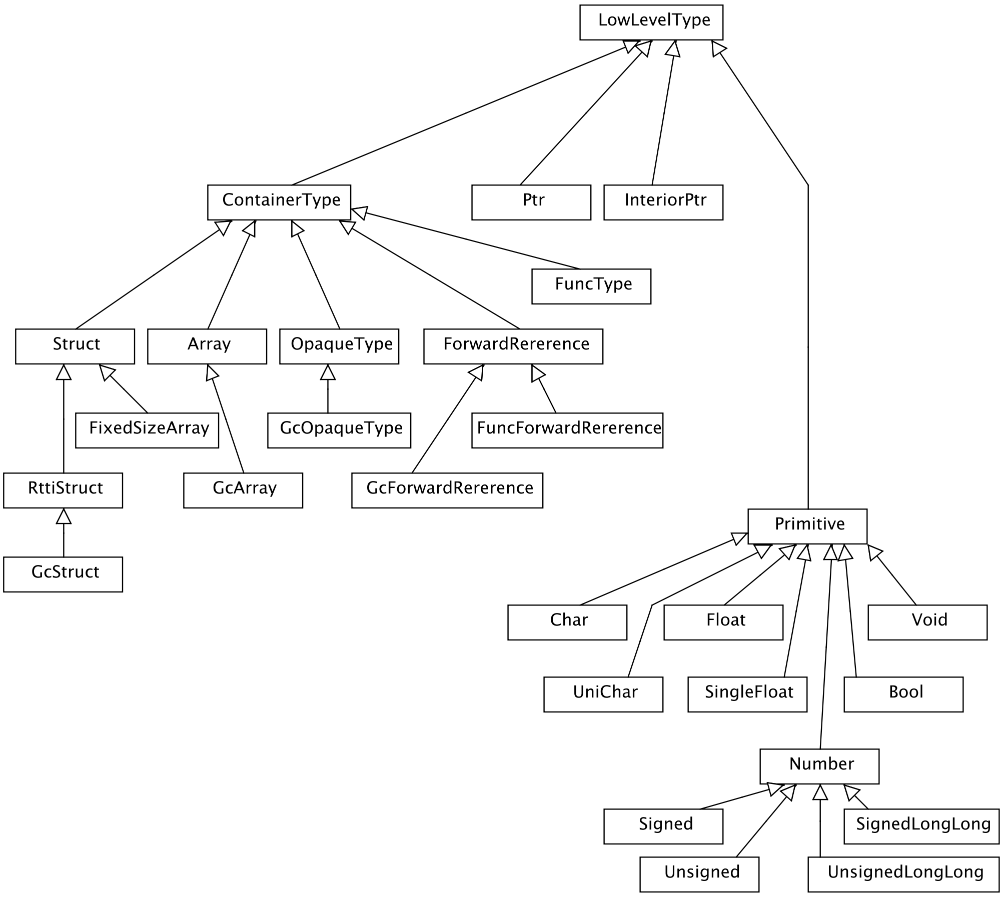


Figure 6.15: Hierarchy of Low Level Types

6.6.3 Object-Oriented Type System

For object-oriented output code—such as Java byte-code, PyPy uses *Object-Oriented Type System*. It is not a replacement for Low Level Type System—it is an addition. Primitive types such as integers and floating-point numbers are reused from Low Level Type System.

What Object-Oriented Type System brings is direct mapping of RPython's compound types to compound types of the selected object backedn, i.e., JVM. For instance the *List* annotation can be most straightforwardly implemented by Java's *ArrayList* container. More or less direct mapping from Python to Java is also available for strings, classes, dictionaries, exceptions.

The hierarchy of the type system is depicted in figure 6.16.

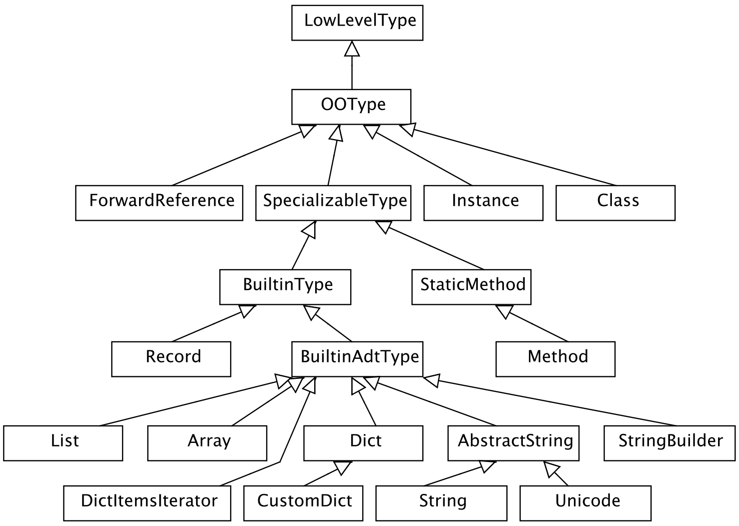


Figure 6.16: Hierarchy of Object Types

6.7 Flow Graph Transformations for C Code

After "RTyping" the flow graphs contain native operations and data types of the selected target code.

However, there can still be some aspects of flow graphs that can not be easily expressed in low level code. In order to eliminate these too high level aspects, there are two flow graph transformations needed: *Exception transformation* and *Garbage collection transformation*. Java byte-code back-end does not need these transformations because it has native exception handling and built-in garbage collector.

We devote extraordinary attention to these transformations because they define the difference between output C and Java byte-code. We need to be sure that the differences will not make usage of our formal verification approach built on Java Pathfinder infeasible.

6.7.1 Exception Transformation

After the flow graph operations and types were transformed to the lower level, there are still two situations connected with exceptions in which the flow graph interpretation relies on some high level features that are not available at the lower level, i.e., the level of C language.

The first is *stack unwinding*. Even if a function neither raises nor handles an exception, it uses exceptions implicitly when an exception is raised by some function deeper on the stack. When such a situation occurs, an exception handler is searched through the stack. This process is called *stack unwinding*. During this process, for each function on the stack, it has to be decided whether the function handles the exception or just passes it away (which is exactly what we mean by the *implicit use of en exception*). Stack unwinding is supported by JVM, but there is not a direct support at neither C nor machine code and thus it have to be explicitly implemented in software.

The second situation is exception handling. As was described in section 6.4.2, the exception handling relies on examining whether a particular instance of exception is subclass of some exception class. Again, this process is supported only by JVM and have to be implemented in the case of C code.

Implemetation of Exceptions in C

In order to understand the goal of *exception transformation*, we have to describe what is the actual PyPy's approach to exceptions in C.

In the flow graph, every block has a dedicated variable for storing a raised exception, usually called *last_exception*. On the C level, instead of these individual variables, there is one global variable, let us call it *c_last_exception*[3]. This variable is *NULL* when the program runs without an exception. If an exception is raised, pointer to the exception instance is assigned to *c_last_exception*. When the exception is handled, *c_last_exception* is set back to *NULL*.

Stack unwinding is done by checking *c_last_exception* after every operation that may raise an exception and exiting the function—jumping into the return block—when it is not *NULL*. These checks and jumps are created by the *exception transformation*.

[3] Note that when Java byte-code is generated, instead of variables that hold exception instance, native JVM exception implementation is used

Exception handling is done by setting *c_last_exception* back to *NULL*. The exception transformation has to create some new blocks that explicitly perform the *is subclass* examination.

Exception Transformation Algorithm

Input of the algorithm is a flow graph G whose operations use *Low Level Type System*. The graph contains all implementation details needed in order to generate C output code except two aspects:

- Stack unwinding
- Exception handling

Output of the *Exception transformation algorithm* is a modified flow graph G that:

- Explicitly implements *stack unwinding*. It is done by splitting some code blocks, adding low-level operations that check presence of an exception, and adding edges that exit the function when an exception is raised. See also figure 6.17 where the transformation for a block that uses exceptions implicitly is depicted.
- Explicitly implements exception handling. It is done by adding new code blocks (an appropriate edges) that examine exceptions' classes.

The algorithm is formally defined by three procedures. *Top level procedure* traverses blocks of the graph and generates exception checks. The *top level procedure* calls *exception matching procedure* whenever it needs to handle exception matching (the *is subclass* operation). Both procedures call *split block procedure* for block splitting.

Top Level Procedure: performs entire exception transformation.

- *Input*: Flow graph G with unimplemented stack unwinding an exception handling.
- *Output*: Flow graph G with explicit implementation of stack unwinding and exception handling.

1. *Prepare*

   (a) Create a copy of $V(G)$, that is $V_{copy} = V(G)$.

2. *Transform ordinary blocks:*

   - For each block $v \in V_{copy}, v \notin EXC(G), v \neq return(G)$ do:

     (a) Let m_0 be the number of operations in v. Let $V_{matching} = \{\}$ be a set that contains zero or one block that needs *exception matching* transformation. Let $m = m_0$

     (b) If $SWITCH(v) = \{last\_exception\}$ then:

        i. Let $V_{matching} = \{v\}$.

        ii. Let $m = m_0 - 1$

     (c) For each $op_i \in OP(v), i = m, m-1, ..., 1$

        – If op_i can raise an exception[4] then:

          i. Perform the *split block* procedure on graph G: split the v block at position op_i. Let the newly created block be v_{new} and the newly created edge that leads from v to v_{new} be e_{new}. Recall that after *split block* procedure the operations are divided between v and v_{new}, that is $OP(v) = \langle op_1, op_2, ..., op_i \rangle$ and $OP(v_{new}) = \langle op_{i+1}, op_{i+2}, ..., op_{m_0} \rangle$.

          ii. Remove the newly created edge $e_{new} = \langle v, v_{new}, \emptyset, EARG \rangle$ from $E(G)$.

          iii. Add k low-level operations into the block v. These operations check whether op_i ended with an exception. So that $OP(v) = \langle op_1, op_2, ..., op_i, op_{check_1}, op_{check_2}, ..., op_{check_k} \rangle$. Insert into $VAR(v)$ new variables that are used by $op_{check_j}, j = 1, 2, ..., k$. The result of the check is stored in a variable var_{check} that is used as exit switch $SWITCH(v) = \{var_{check}\}$. The value of the var_{check} is `False` if the exception did not occur, `True` otherwise.

          iv. Construct an ordinary link
          $e_{ordinary} = \langle v, v_{new}, \{False\}, EARG(e_{new}) \rangle$, add $e_{ordinary}$ into $E(G)$.

          v. Construct an exception link
          $e_{exc} = \langle v, return, \{True\}, \langle\rangle \rangle$, add e_{exc} into $E(G)$.

          vi. If $i = m$ and $V_{matching} \neq \emptyset$ then $V_{matching} = \{v_{new}\}$.

     (d) If $V_{mathing}$ is not empty then perform *exception matching* transformation on $w \in V_{matching}$.

3. *Transform exception block:*

[4] Most operations can potentially raise an exception; however, there are some that are safe such as comparison of two numbers.

- If there is an exception block $v_{exc} \in EXC(G)$ then:
  (a) Add low-level operations that explicitely raises an exception.
  (b) Construct a new edge from exception block to return block, that is $e_{ret} = \langle v_{exc}, return(G), \{\}, \langle\rangle\rangle$, add e_{ret} into E

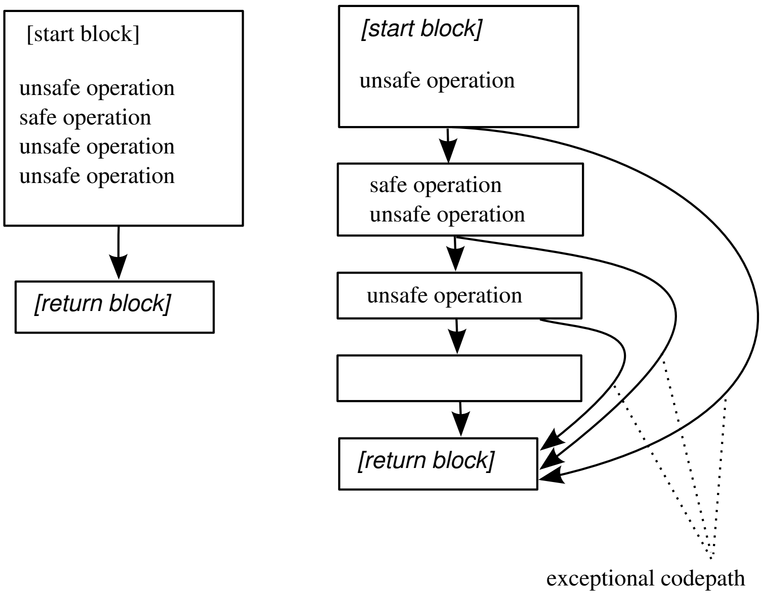


Figure 6.17: Flow Graph before and after Exception Transformation, Implicit Use of Exceptions

Exception Matching Procedure: generates low level exception handling for a single block that ends with an instruction that can raise an exception. If the operation can raise multiple exception types (there are multiple outgoing edges), the exception matching is implemented by inserting new blocks that contain only low level operations.

- *Input*: Flow graph G, $v_s \in V(G)$ that has an operation op_f on the last position. Operation op_f may raise an exception and there is no explicit low level exception handling implemented.

- *Output*: Flow graph G with explicit low level implementation of handling of exceptions that may be raised by operation f called in block v_s.

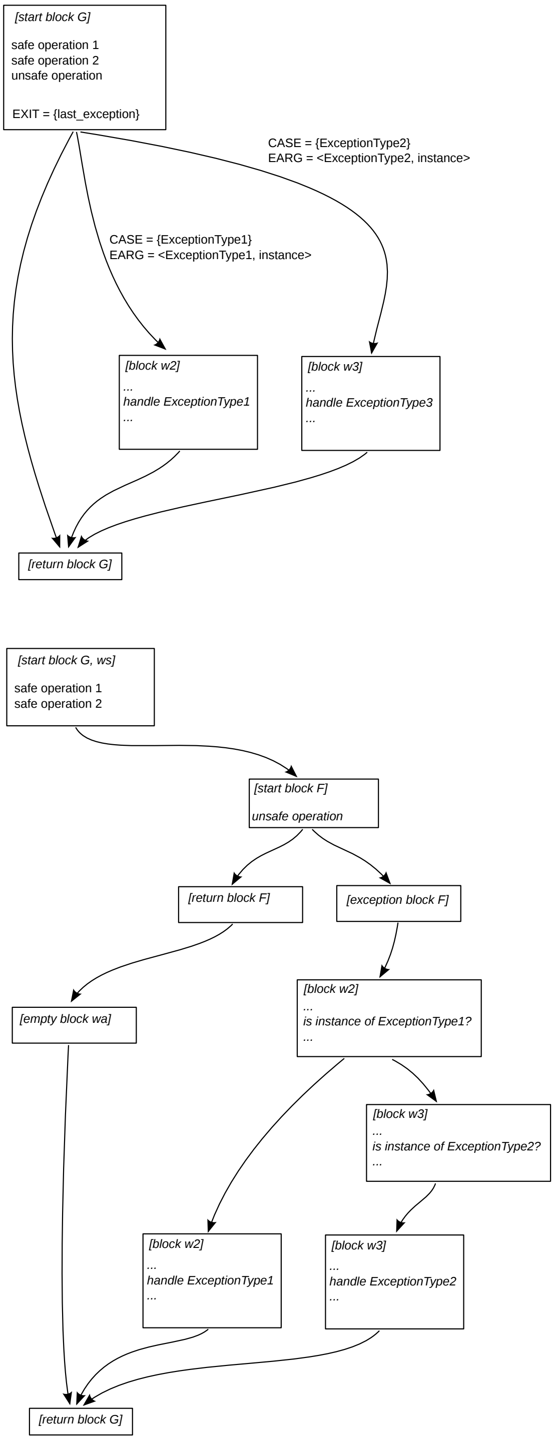


Figure 6.18: Flow Graph before and after Exception Matching Procedure, Explicit Exception Handling

1. Create a new flow graph F that only performs the operation op_f. If op_f does not raise an exception, the result of op_f is returned; otherwise, the exception is reraised. Graph F has only three blocks, $V(F) = \{start, return, except\}$, $except \in EXC(F)$ and two links so that:

   - $IN(start)$ contains arguments of operation op_f,
   - $OP(start)$ contains the operation op_f, and low level operations that check whether an exception occurred. The result of the check is stored in a variable var_{check} that is used as an exit switch $SWITCH(start) = \{var_{check}\}$. The value of the var_{check} is *False* if the exception did not occur, *True* otherwise.
   - An ordinary link $e_{ordinary} = \langle start, return, \{False\}, \{result(op_f)\}\rangle$, add $e_{ordinary}$ into $E(F)$.
   - An exception link $e_{exc} = \langle start, except, \{True\}, \langle\rangle\rangle$, add e_{exc} into $E(F)$.

2. Let $V(G) = V(G) \cup V(F)$.

3. Let $E(G) = E(G) \cup E(F)$.

4. Perform *split block* transformation on G splitting block v_s at position op_{f-1}. Let the newly created block be v_a and the newly created edge that leads from v_s to v_a be e_{sa}.

5. Block v_a now contains only one operation, op_f. It is because *exception matching* is performed for the last operation of the block. Remove this operation so that $OP(v_a) = \langle\rangle$.

6. Having an edge $e_{sa} = \langle v_s, v_a, CASE_{sa}, EARG_{sa}\rangle$. Construct an edge $e_{sf} = \langle v_s, start(F), CASE_{sa}, EARG_{sa}\rangle$. Add e_{sf} into $E(G)$ and remove e_{sa} from $E(G)$.

7. Construct edge $e_{fs} = \langle return(F), v_a, \emptyset, \langle result(op_f)\rangle\rangle$, add e_{fs} into $E(G)$.

8. For edges $e_i \in E(G)$, $e_i = \langle v_a, w_i, CASE_i, EARG_i\rangle, w_i \in V(G)$ that goes from v_a excet the first one, i.e., $i = 2, ..., N$ where N is the number of outgoing edges of ${v_a}$[5]:

   (a) Create a new block v_i, so that

[5] The ordering is defined by *edgeorder* function, see 6.4.2. Recall that the first edge is for the case without an exception.

- $IN(v_i) = EARG(e_i)$,
- $OP(v_i)$ contains a test whether an exception instance that is stored in IN is a subclass of class stored in $CASE(e_i)$. The result is stored in the variable *is_subclass*.
- $SWITCH = \{is\_subclass\}$,
- $VAR(v_i)$ contains all items from $IN(v_i)$ together with variables needed by the *is subclass* check.

(b) If $i = 2$:

- Create an edge $\langle exc_F, v_2, \emptyset, IN(exc_F)\rangle, exc_F \in EXC(F)$, add this edge into $E(G)$.

(c) If $i > 2$:

- Construct an edge $\langle v_{i-1}, v_i, \{False\}, IN(v_{i-1})\rangle$, add this edge into $E(G)$.

(d) If $i \neq n$:

- Construct an edge $\langle v_i, w_i, \{True\}\rangle, EARG(e_i)\rangle$, add this edge into $E(G)$.

(e) If $i = n$:

i. Let $OP(v_i) = \langle\rangle$.

ii. Let $SWITCH(v_i) = \emptyset$.

iii. Construct an edge $\langle v_i, w_i, \{\}\rangle, EARG(e_i)\rangle$, add this edge into $E(G)$.

(f) Remove e_i from $E(G)$.

Split Block Procedure: splits given flow graph block at given operation.

- *Input*: Flow graph G, block $v \in V(G)$, an operation $op_s \in OP(v)$.
- *Output*: A modified G with one additional block, the block $v \in V(G)$ is divided into two blocks after the operation op_s. The semantics of the graph is unchanged.

1. Create a new block v_{new} and add it into $V(G)$. The v_{new} is defined as follows:

   - $OP(v_{new}) = \langle op_{s+1}, op_{s+2}, ..., op_m\rangle$, where $op_i \in OP(v)$ and m is the number of operations in $OP(v)$.
   - $last\_exception(v_{new}) = last\_exception(v)$,

- $SWITCH(v_{new}) = SWITCH(v)$.
- $VAR(v_{new}) \subset VAR(v)$, $VAR(v_{new})$ contains only variables used by $OP(v_{new})$,
- $IN(v_{new})$ is arbitrarily ordered $VAR(v_{new})$,

2. Alter block v so that:
   - $OP(v) = \langle op_1, op_2, ..., op_s\rangle$, i.e., remove all operations that were moved to v_{new}.
   - $SWITCH(v) = \emptyset$.
3. For each edge that leads from v, i.e., $e = \langle v, u, CASE, EARG\rangle, u \in V$, construct a new edge $e_{moved} = \langle v_{new}, u, CASE, EARG\rangle$. Add e_{moved} into $E(G)$ and remove e from $E(G)$.
4. Construct a new edge $e_{new} = \langle v, v_{new}, \emptyset, IN(v_{new})\rangle$, Add e_{new} into $E(G)$.

6.7.2 Garbage Collection Transformations

Python—and also RPython—requires automatic memory management. JVM has its own garbage collector that is used by PyPy generated Java byte-code without any problem. However, automatic memory reclamation must be addressed in PyPy generated C code.

Basic Garbage Collection Approaches

One of the most simple GC is based on counting active references for every object; an object can be reclaimed if the counter reaches zero. Just for curiosity, standard Python interpreter (CPython) uses this algorithm. There are two issues connected with this approach. First, this algorithm never frees an object that has a (transitive) reference to itself; CPython solves the problem by additional cycle detector. Second, it has poor performance in multi-threaded environment since every operation with the reference counter of every object has to be guarded by mutual exclusion; CPython contains *global interpreter lock* (GIL) that causes that only one thread is executed at once, therefore additional mutual exclusion is not needed. An advantage of the reference counting GC is that it releases the resources deterministically.

The most widely used family of GCs is based on the mark-and-sweep algorithm. In short, it works as follows: when there is not enough free memory, the program is stopped for a while and all reachable objects are marked. The marking goes transitively from so called GC roots that are the data in CPU registers, variables on the stack(s), and global variables. Then all objects that were not marked, i.e., are inaccessible, are reclaimed (swept). Then the program is resumed.

Mark-and-sweep algorithms can work in multi-threaded environment; however, they produce (possibly unbounded) program pauses and from the point of view of the program are nondeterministic.

The third popular GC approach is based on copying. Briefly, the heap is divided into two halves. All new allocations are performed in the first half; individual objects are never deallocated. If the first half is full, all live (accessible from GC roots) objects are copied into the second half of the heap and the first half is then declared as "free". Then the two halves are switched. This family of GCs eliminates memory fragmentation; on the other hand, it needs at least twice as much memory as the sum of the memory occupied by all live objects.

In practice, a combination of mark-and-sweep and copying is quite common.

Garbage Collectors in PyPy

The PyPy compiler has a framework that eases implementation of various GC algorithms. There is a conventional reference-counting GC as well as several mark-and-sweep and copying (and also hybrid) GCs implemented with the help of the framework.

Unfortunately, none of the framework-based GC implementations can run in multi-threaded environment as they are. Adaptation for multi-threaded applications should be possible; nevertheless, it is beyond the scope of this work.

There is one GC that fits our purposes, though. The PyPy-generated C code can employ Boehm GC[6] that is available as a third-party library. Boehm GC is a conservative mark-and-sweep algorithm for programs written in C.

Boehm GC is fast and its functionality is proven by practice (GNU Java Compiler[7], Mono[8], etc.). The therm *conservative* means that it does not precisely know which peace of data is a pointer and which is not; therefore, a piece of data can be misinterpreted as a pointer. This may lead to the situation in which an object is considered as accessible even if it is not. Real programs can routinely live with this behavior, deeper analysis of conservative GCs with bounding of space usage can be found in [63].

Note that all other PyPy GCs are so called *type-accurate*; that means, they always know which peace of data is an actual pointer.

A standard C program can use Boehm GC by just redirecting calls of *malloc* from standard C library to Boehm GC library.

[6] http://www.hpl.hp.com/personal/Hans_Boehm/gc/

[7] http://gcc.gnu.org/java/

[8] http://www.mono-project.com/Main_Page

We would conclude the section about GC transformations as follows. For the JVM backend, we get memory management completely for free. For subsequent experiments with generated C code, we will employ the Boehm GC library. Therefore, the GC transformation is trivial and we do not need to document it.

However, for the sake of determination of lower bound of memory consumption, we will make also some experiments with reference counting GC.

We also note, that adapting some of the type-accurate GCs provided by the framework for multi-threaded applications would be excellent choice for extending this work.

6.8 Generating the Output Source Code

6.8.1 Generating the C code

A flow-graph that contains all necessary low-level details is then used by the PyPy code generator to emit the final C source code. The structure of the generated code is pretty flat. There is a single C function for every flow-graph and all functions are serialized into one huge file called *implement.c*. Classes are represented by C structures.

The names of the generated functions are derived from fully qualified method names in the original RPython source code. For instance a method *myMethod* of a class *MyClass* that is located in a module *pkg_a.pkg_b.my_filename* is represented by a function with the following prototype:

```
long pypy_g_MyClass_myMethod(
        struct pypy_pkg_a_pkg_b_my_filename_MyClass0 *l_self_5,
        long l_x_2)
```

The name of the function does not take a package name into account. If there is a class and a method of the same name in a different package, a unique integer is added as a filename suffix. For instance, a method *MyClass.myMethod* from a package *pkg_a.pkg_b2.my_filename* is rendered as:

```
long pypy_g_MyClass_myMethod_1(
        struct pypy_pkg_a_pkg_b2_my_filename_MyClass0 *l_self_5,
        long l_x_2)
```

The bodies of the generated functions are directly derived from the flow-graphs; there is a block of statements for every graph vertex and *goto* statements that represent graph edges.

6.8.2 Generating the Java byte-code

Generation of Java byte-code is more straightforward because Python is far closer to Java than to C.

The PyPy translator does not directly generate the binary representation of the byte-code, i.e., the *.class* files. It generates a human readable assembly code for JVM. These assembly files (with *.j* extension) are converted do the binary form of *.class* files by a JVM assembler called *jasmin*[9].

Both Java and Python support source code organization based on hierarchical entities. These entities are called *modules* in Python and *packages* in Java. RPython modules can be translated to Java packages without any problem; however, there is one difference worth mentioning. Java packages are filesystem directories and every (public) Java class is a standalone file placed in a particular directory. In Python, module hierarchy is expressed through directories as well; however, bottom level modules are ordinary files that may contain several publicly accessible classes (and other entities).

For instance, a Python class with a fully qualified name *pkg_a.pkg_b.my_filename.MyClass* is saved in a file *pkg_a/pkg_b/my_filename.py.* The PyPy code generator has to create a package structure that conforms to Java standards; therefore, the Java assembly code is saved in file *pypy/pkg_a/pkg_b/my_filename/MyClass_22.j* (PyPy adds a unique integer as a suffix to every class name). Note that all the generated classes are placed in a top level package called *pypy.* When the Java assembly files are compiled to Java class files, we obtain a program runnable on the top of JVM.

6.9 Conclusion

We went deep into the PyPy compiler design; we described its main data structure called *flow graph* in detail. We also precisely described the transformations of the flow graph. These transformations are used to overcome the abstraction gap between Java byte-code and C code.

For the sake of our testing based on formal methods, we now can say what is the relation between C and Java byte-code generated by the PyPy compiler. Both back-ends use the same initial flow graph; the type inference (also called annotation) is also the same. The first point in which the flow-graph becomes back-end specific is the assignment of a platform-specific data type to every type annotation and replacement of every high-level operation by one or more platform-specific operations. This is just a substitution and therefore it is verifiable. For C code, two additional transformations have to be performed: to

[9] http://jasmin.sourceforge.net/

implement exceptions and automatic memory management. The first one was precisely described in this chapter and the second one is trivial in our case.

We did not deal with multi-threaded programs yet because the original PyPy tool-chain does not deal with it either. See the next chapter for our approach to multi-threaded programs.

Chapter 7

Customization of the PyPy Compiler

As mentioned in section 3.5.2, the PyPy compiler was originally created in order to compile one program: the PyPy interpreter. Features of the PyPy Python interpreter, e.g., Python thread semantics, have some impact on the capabilities of the PyPy compiler. To use PyPy compiler for general multi-threaded embedded programs as specified in section 5.2, we have to customize the PyPy compiler.

7.1 Threading and Locking Models

In order to make our development approach feasible, we have to run an intended embedded application in three different modes:

- Interpreted by the Python interpreter.
- C code translated to machine code run by bare CPU.
- Translated to Java byte-code run by JVM.

Every of the environments has its own approach to threads and synchronization.

7.1.1 Python Interpreter Threads

Python standard library offers synchronization objects patterned after POSIX threads. However, there are some special rules how the threads run.

Standard Python interpreter, CPython, guarantees that byte-code instructions are serialized, i.e., only one instruction is executed at once. Python programs can run multiple native operating system threads; however, the consequence of this guarantee is that only one thread is executed at once, even on hardware with multiple CPU cores. In CPython, it is implemented via so called *global interpreter lock* (GIL) that has to be acquired by every thread prior a byte-code instruction is executed. The GIL is usually released back after several instructions. Because many Python programs depend on the instruction serialization, the PyPy interpreter embraces GIL in the same way as CPython.

7.1.2 POSIX Threads

For threading in C code, we use POSIX threads. POSIX threads (or just pthreads) is the standard interface for multi-threaded programming in Unix-like operating systems. There is also a pthread implementation for contemporary Windows operating system available. Interface is defined in the form of a set of C functions.

POSIX threads provide functions for thread starting and synchronization. There are two main synchronization objects: the *mutex* (possibly recursive) and the *condition*.

Our primary target platform is Linux 2.6.x and 3.x. POSIX threads are implemented by library called NPTL (Native POSIX Thread Library) which provides 1:1 mapping to the kernel-level threads.

7.1.3 Java Threads

Java has its own platform independent abstraction for threading. New threads are started by a special predefined class called *Thread*. The mapping from the Java threads to the system threads is defined by JVM; the Java threads are mapped to the kernel-level threads in standard Oracle JVM version 6.

Java's native synchronization object is the *monitor* as defined by C. A. R. Hoare [64]. In fact, every single object instance may act as a monitor in Java. There is a lock associated with every object instance. Object's methods can acquire and release the associated lock. A method or a block of code can be marked by the *synchronized* keyword. Java then guarantees that the lock is acquired prior the execution of the code.

The monitor's capability of waiting and being signaled is implemented by methods *wait* and *notify* that are inherited by all user-defined classes.

7.2 Unified Threading and Locking Model

We need to have one unified threading and locking semantics in order to guarantee that the program's behavior is as similar as possible in all three environments. The behavior should be identical in the C and JVM environments because otherwise our verification process would be weakened. On the other hand, we have to accept that the Python interpreter defines thread serialization and make sure that we never rely on this behavior. Recall that we use Python interpreter mode in order to create "unsimplified model" of the final application.

First of all, we have to create a unified threading and locking model for C and Java byte-code that we then try to use in the Python interpreter. We have two choices; the first one is to create POSIX-compatible locks on the top of Java's monitors. The second one is to create monitors built on the POSIX locking objects.

For several reasons, we decided for creation of monitors that will be used in all three environments. Although monitors and simple locks have equal expressing power, monitors fit better with object-oriented programming. Monitors are seamlessly incorporated not only into Java but also into C#, there is no reason why monitors could not be elegantly used in RPython.

The second reason why we decided for Java's native synchronization is Java Pahtfinder and its state-space optimizations. The optimization algorithms are designed for Java threading semantics. If we implemented POSIX-like locks as a layer over native monitors, it it would complicate the Java Pathfinder's fight with state-space explosion. In [65], we show that using POSIX-like locks in Java leads to unbearable growth of time and memory needed for checking by Pathfinder. We also show that Java byte-code generated from RPython that uses native Java synchronization brings no significant overhead in comparison with hand written program in pure Java.

It is important to note that native Java's monitors are built on the top of recursive locks. Once a thread acquires the lock, i.e., enters the synchronized method or block, it also succeeds in all subsequent tries to acquire the lock again, until the lock is released. Therefore our monitors for C will be built on the top recursive POSIX mutexes.

7.2.1 Usage in RPython

Python has no direct support for synchronized methods or blocks. So we have to create such a block by manually acquiring a lock at the beginning of the block and manually releasing it when we are done.

We developed a library with synchronization objects for all three environments. The library is internally called *parlib*, see figure 7.1 for the position of parlib in the code generation scheme. There is a class *Monitor* defined in the library.

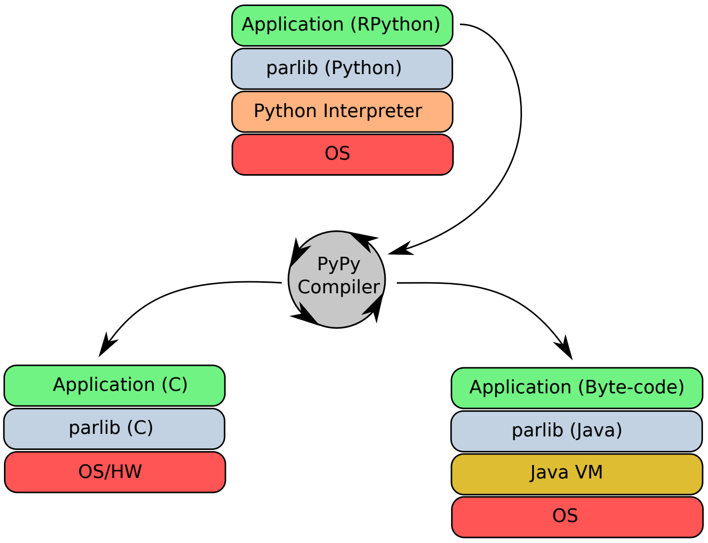


Figure 7.1: Detailed Compilation Scheme with Parlib

If the program runs in the standard Python interpreter, the *Monitor* class encapsulates a reentrant (recursive) lock from the standard Python library. The class has two methods, *MONITOR_ENTER* and *MONITOR_EXIT*, that encapsulate the calls to *acquire* and *release* methods of the reentrant lock.

To achieve the compatibility with Java monitors, the method *MONITOR_ENTER* is supposed to be called at the very beginning of the method that is aspires to be *synchronized*. Similarly, *MONITOR_EXIT* is supposed to be called as the last command of the method.

The class *Monitor* also provides methods for waiting and notification: *WAIT*, *NOTIFY*, and *NOTIFYALL* that behaves like Java methods *wait*, *notify*, and *notifyAll*.

The internal implementation of the wait/notify mechanism is patterned after the *Condition*[1] class from the Python standard library. There is a list of locks containing one lock for each blocked thread. If a thread wants to wait for a notification, it is blocked on a newly created lock. The lock is then stored in the list. If the notification occurs, the lock is released (releasing the thread) and removed from the list.

The code in figure 7.2 demonstrates a usage of our monitor. It is a thread-safe box that contains zero or one item inside. It provides a classical solution

[1] http://docs.python.org/release/2.4.4/lib/condition-objects.html

```
from parlibutil.locking import Monitor

class Box(Monitor):
    def __init__(self):
        Monitor.__init__(self)
        self.item = None;

    def getItem(self):
        try:
            self.MONITOR_ENTER()
            while self.item is None:
                self.WAIT()

            result = self.item
            self.item = None
            self.NOTIFY()
            return result
        finally:
            self.MONITOR_EXIT()

    def putItem(self, item):
        self.MONITOR_ENTER()
        while not (self.item is None):
            self.WAIT()

        self.item = item
        self.NOTIFY()
        self.MONITOR_EXIT()
```

Figure 7.2: Thread-safe Box

for the producer-consumer problem. Method *putItem* inserts the item into the box and blocks if it is already full; method *getItem* removes the item from the box and blocks if it is empty. Both methods notify each other if the state of the box changes.

One can see that calling *MONITOR_ENTER* and *MONITOR_EXIT* at the right places is not very comfortable and even error-prone in comparison with use of the Java's *synchronized* keyword. Fortunately, in Python we can gain similar elegance by creating a decorator; see figure 7.3.

After utilizing this decorator, the code of the *Box* class looks much cleaner, see figure 7.4.

```
def synchronized(meth):
    def synchronized_meth(self, *args):
        try:
            self.MONITOR_ENTER()
            return meth(self, *args)
        finally:
            self.MONITOR_EXIT()
    return synchronized_meth
```

Figure 7.3: Definition of decorator @synchronized

```
from parlibutil.locking import Monitor, synchronized

class Box(Monitor):
    def __init__(self):
        Monitor.__init__(self)
        self.item = None;

    @synchronized
    def getItem(self):
        while self.item is None:
            self.WAIT()

        result = self.item
        self.item = None
        self.NOTIFY()
        return result

    @synchronized
    def putItem(self, item):
        while not (self.item is None):
            self.WAIT()

        self.item = item
        self.NOTIFY()
```

Figure 7.4: Thread-safe Box with decorator @synchronized

7.2.2 Implementation in C

The *parlib* variant suitable for the C backend is built on the POSIX synchronization objects. The *Monitor* class contains one *mutex* that is configured as *recursive* because Java monitors have also recursive semantics. Call of *MONITOR_ENTER* results in acquiring the mutex; the method *MONITOR_EXIT* releases the mutex.

The wait/notify mechanism works the same way as described above. Note that the list contains non-recursive POSIX mutexes.

7.2.3 Implementation in Java Byte-code

The implementation of *parlib*'s monitor for C the backend is based on encapsulating the POSIX synchronization objects. Implementation for Python interpreter is done in the same way, with the difference that it encapsulates the synchronization objects defined in the Python standard library.

In the case of Java byte-code backend, we use a different approach. Instead of encapsulating JVM monitors by *parlib*'s monitors, we use the Java native monitors directly. The absence of the encapsulating layer will help Java Pathfinder to perform state-space optimizations such as partial order reduction.

We customized the translation to the Java byte-code in the following way. If a particular object's method contains call of *MONITOR_ENTER*, the method is marked as *synchronized* in the generated Java byte-code. The actual implementation of the *MONITOR_ENTER* method is empty. The implementation of the *MONITOR_EXIT* is empty as well.

Calls to *WAIT*, *NOTIFY*, and *NOTIFYALL* are rewritten as calls to *wait*, *notify*, and *notifyAll* methods of the Java's *Object* class.

The only way how to create a new Java thread is to use *java.lang.Thread* class. If a user class defines a thread, it inherits from *Thread* class defined in *parlib*. The RPython classes do not have direct access to the standard Java library. In order to access native Java API, we can change (rewrite) the inheritance chain during the translation so that the user class inherits from our custom *ThreadStarter* class implemented in pure Java. The *ThreadStarter* inherits directly from *java.lang.Thread* class and thus can start a new thread.

7.3 Multi-threaded C

As mentioned in section 6.7.1, exception object instances are stored in a global variable. This approach is insufficient in multi-threaded program because we have to distinguish between exceptions in various threads.

We solved the issue by making the global variable, i.e., *c_last_exception*, thread specific. We use the thread local storage (TLS) so every thread has its own instance of the variable. In GCC, this can be achieved by adding the __*thread* modifier to the variable definition in the C source code.

Another potential issue is multi-threaded memory management. We utilize Boehm Garbage Collector that is able to work in such an environment; see also section 6.7.2.

7.4 Conclusion

The achievement of this chapter is the design of primitives for thread synchronization. These primitives behave exactly the same in pure Python, C and Java byte-code instances of a program. This design enables utilization of Java Pathfinder; in our previous publication [65], we have already shown it is the best solution for the model checker's performance.

Last but not least, our synchronization primitives are very elegant to use thanks to Python decorators.

Chapter 8

Testing Based on Formal Methods

Our development process benefits from the strengths of high level dynamic languages and generative programming. The development process should also benefit from formal methods as much as possible.

In this chapter, we present how to employ a tool for formal verification for discovering bugs in a program code. We first define a class of bugs that might be efficiently discovered by the tool then we show how to trace tool's findings (reports) back to RPython source code of the program.

Finally, we demonstrate this approach on a set of test cases: a deadlock, a race condition, an uncaught exception, an LTL formula violation and testing with random data. In every test case, we present a simple program with a bug injected. The bug is then discovered by the tool for formal verification and fixed.

See [70] in order to get the source codes of the experiments.

8.1 Tools

8.1.1 Java Pathfinder

As we mentioned earlier in this work (sections 3.1.3 and 5.3.4), we use Java Pathfinder model checker as an advanced tool for discovering bugs. Recall that Java Pathfinder (JPF) is actually a special-purpose reimplementation of the Java virtual machine, see figure 8.1 for the placement of JPF in the software stack; compare with detailed compilation scheme in figure 7.1.

JPF has a plug-in architecture; the properties of the program are verified by independent modules. Among basic modules that we utilize are deadlock detector and condition race detector.

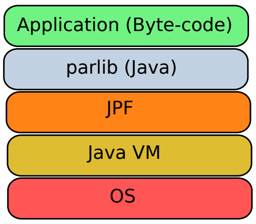


Figure 8.1: Sofware Stack with Java Pathfinder

Operator	*Meaning*
`G`	Temporal operator G
`F`	Temporal operator F
`X`	Temporal operator X
`U`	Temporal operator U
`v`	Logical or
`^`	Logical and
`~`	Logical negation
`->`	Implication

Table 8.1: Operators in LTL Module

8.1.2 Module for Linear Temporal Logic

JPF does not deal with LTL on its own; however, there exist LTL verifiers as third party modules. We use the LTL verifier developed by Nguyen and Khoo [66].

This verifier uses method calls as atomic propositions. A particular proposition is *true* if the method mentioned in the proposition is called.

The module also defines a notation for LTL formulae specification; it uses only ASCII characters and thus is slightly different from the standard notation described in section 3.1.2. Atomic propositions start by the keyword *method:* that is followed by a class name and a method name, separated by a dot; a proposition is also enclosed by curly braces. For instance: *{method:Worker.run}* is an atomic proposition that denotes a call of the method *run* of the class *Worker*.

The language also defines how temporal and logical operators are expressed, see table 8.1.

For instance, a popular LTL pattern that ensures that every call to the *Server.request* method is inevitably followed by an execution of the *Server.response* method looks as follows:

```
G{{method:Server.request}->{X{F{method:Server.response}}}}
```

For comparison, the formula in the standard notation would look like this:

$$G((method\_Server\_request) \Rightarrow (X(F(method\_Server\_response))))$$

8.2 Aiming the Tests

There are plenty of testing methods used. Now, we have to decide what is the main goal of our testing procedure.

Java Pathfinder analyzes the generated Java byte-code; however, the code intended for deployment is the generated C. Recall that according to our analysis in chapter 6 and our threading and locking model designed in chapter 7 we can consider these codes as identical.

Threading Issues

The main advantage of the formal approach is that it can reliably discover threading issues. Simple testing methods such as unit tests are not appropriate for this class of issues; unit tests by definition do not deal with component interaction. Tests that employ simulation can deal with threading; however, this kind of testing is probabilistic.

Linear Temporal Logic

Apart from threading issues, checking of scenarios defined as LTL formulae is a promising approach. The used LTL verifier uses method calls as atomic propositions. Since every method call on RPython level is translated as a function call in C or a method call in Java byte-code, the validity of an LTL formula is never hurt by the translation process.

8.3 Traceability

One of the key advantages of model checking is that it can provide counterexamples, i.e., a program trace that leads to a violation of a formal property. As we work with several codes (Python, C, Java byte-code), we have to translate these counterexamples from one code to another.

```
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 8.2: JPF Report: Stack Trace of a Deadlock

8.3.1 Reports of Java Pathfinder

Every module of Java Pathfinder can provide its own way how to report a property violation. For instance, a report of the module that checks the deadlock-free property consists of a set of deadlocked threads; for every thread there is a stack-trace, list of locks it has acquired and the lock it is blocked on. See figure 8.2 for an example of such a report.

For another example, a report from the module that checks an LTL formula violation provides full log of executed methods, see figure 8.3. You can see that together with bare method names, there are also links to some sort of source code. These *.j* files are a human readable representation of *.class* files. See section 6.8.2 for more information about these Java assembly files.

```
STEP invokestatic pypy.entry_point_50.invoke(Ljava/util/ArrayList;)I
  --pypy/Main.j:40--
STEP invokestatic pypy.main_54.invoke(Ljava/util/ArrayList;)V
  --pypy/entry_point_50.j:22--
STEP invokespecial pypy.application.Application_55.<init>()V
  --pypy/main_54.j:24--
STEP invokespecial java.lang.Object.<init>()V
  --pypy/application/Application_55.j:8--
STEP invokevirtual
 pypy.application.Application_55.omain(Ljava/util/ArrayList;)V
  --pypy/main_54.j:33--
STEP invokespecial pypy.files.File_56.<init>()V
  --pypy/application/Application_55.j:56--
STEP invokespecial java.lang.Object.<init>()V
  --pypy/files/File_56.j:8--
STEP invokevirtual pypy.files.File_56.oopen()V
  --pypy/application/Application_55.j:63--
STEP invokevirtual pypy.files.File_56.owrite()V
  --pypy/application/Application_55.j:68--

======================================================= results
error #1: res.min.verifier.LTLVerifier
   "The property G{{method:File_56.oopen}->{X{F{method..."
```

Figure 8.3: JPF Report: A Method-call Trace of a LTL Formula Violation

8.3.2 Mapping of Identifiers

As we perform a testing driven by formal methods on a generated Java byte-code, the discovered bugs are always described in the terms of this generated code. But the only place where a bug can be fixed is the original source code written in RPython from which the Java byte-code was generated. Therefore, we have to be able to map the identifiers from JPF reports and traces back to the original RPython source code.

The C and Java byte-code codes are generated from the flow graphs and the flow graphs are the result of abstract interpretation of the RPython source code. Therefore, the mapping is rather limited; however, it is till feasible.

Every flow graph bears an RPython method's name it was created from. This name is also kept in the generated C or Java byte-code. So there is in principal a straightforward relation between the methods in RPython source and the methods in Java byte-code/functions in C code. The generated Java classes also bear the name of corresponding RPython classes; the C structures that implement OOP support for the low level code are also named appropriately.

In contrast, bodies of the generated methods/functions are only loosely coupled with the original code. The bodies are serialized flow graphs: the code blocks can be mapped to the flow graph vertexes. However, the flow graph vertexes can not be straightforwardly mapped back to the RPython code constructs. Moreover, there are many auxiliary variables without meaningful names.

So for more effective work, we recommend to write rather shorter methods which is generally considered as a good practice.

As mentioned above, a flow graph can be in principle always mapped to a corresponding entity in the generated source code (and vice versa). However, we currently provide only limited tools for this task, development of comfortable developer tools is not the aim of this work. The following paragraphs show how one can achieve such a mapping with the help of elementary tools.

Mapping from Java byte-code to RPython

This mapping is needed in order to apply the information from the Java Pahtfinder reports and traces to the original RPython source code.

The structure of the generated Java byte-code is described in section 6.8.2. The generated Java *packages* can be easily mapped to the RPython *modules.*

Because of some internal reasons (the names of the flow graphs are note necessarily unique because they do not take module names into account), PyPy adds a unique integer as a suffix to every class name. To avoid collisions with methods from the standard Java library, PyPy also adds prefix *o* to every method name.

For instance, the fully qualified method name *pypy.files.File_56.owrite* mentioned on the second last line in figure 8.3 can be found as method *write* of the class *File* in the module (file) *files.py.*

Mapping from RPython to Java byte-code

We also need to map RPython identifiers to the generated Java byte-code identifiers in order to construct LTL formulae to be used by JPF.

As mentioned earlier, PyPy translator adds a unique integer as a suffix to the RPython class names. The only way how to get this integer is to search through the generated Java byte-code assembly files. We have developed a set of scripts for this task. The most useful script is called *findclassandmethod.sh*; it takes two parameters: a class name (either with or without a module name) and a method name.

For example, if we want to find class *MyClass* with method *myMethod* assuming that the class is in module *pkg_a.pkg_b.my_filename*, we use the script as follows (a dollar sign states for the command prompt):

```
$ findclassandmethod.sh pkg_a.pkg_b.my_filename.MyClass myMethod

pypy/pkg_a/pkg_b/my_filename/MyClass_56.j:.method public omyMethod(I)I
```

For this example, the result of the script is the file name from which we can see that the actual generated class name is *MyClass_56*. After the semicolon there is the method name that was actually generated (*omyMethod*) and also the method signature (it is public, takes one integer as an argument and returns an integer as a result).

The script can be used by a preprocessor that translates LTL formulae from the RPython identifiers to the Java byte-code identifiers.

Mapping from C Code to RPython

If we are debugging a machine code compiled form a PyPy generated C code by a low level system debugger such as *GDB*[1], we need to find an appropriate RPython code eventually.

As described in section 6.8.1, entities of the generated C code bear identifiers from the original RPython code. We do not have a reliable tool that would map the C functions back to the RPython methods; however, it is usually not a problem for a human to do so. The main issue is that C code is not organized into any packages or modules.

A generated C function that represents *MyClass.myMethod* method looks like this:

```
long pypy_g_MyClass_myMethod(
        struct pypy_pkg_a_pkg_b_my_filename_MyClass0 *l_self_5,
        long l_x_2)
```

The function name leads clearly to the appropriate RPython method name; though, it does not contain a package name. However, the type of the pointer to the *self* object is of type *struct pypy_pkg_a_pkg_b_my_filename_MyClass0* from which we can infer that the appropriate module is *pkg_a.pkg_b.my_filename.*

In the future, we might improve the C code generator in order to provide an automatic mapping from the low level functions to the RPython entities.

Mapping from RPython to C Code

This mapping is useful when we want to make sure that the generated C code is according to our expectations.

Currently the easiest way how to find the appropriate function for a particular RPython method is to search through the generated C files. We have a small script for this task called *findfunc.sh* that takes a class name and a method name as arguments. For instance, if we are looking for the *MyClass.myMethod*, we use it as follows:

[1] The GNU Debugger Project

```
$ findfunc.sh MyClass myMethod

testing_1/forwarddecl.h:104:extern long pypy_g_MyClass_myMethod(
    struct pypy_pkg_a_pkg_b_my_filename_MyClass0 *l_self_2, long l_x_0);

testing_1/implement.c:734:long pypy_g_MyClass_myMethod(
    struct pypy_pkg_a_pkg_b_my_filename_MyClass0 *l_self_2, long l_x_0) {
```

The result states that the appropriate function is declared at line 104 of file *forwarddecl.h* and its implementation starts at line 734 of file *implement.c*.

8.4 Test Cases

We will demonstrate the testing driven by formal methods on several test cases. We first write a simple program with an intentional bug. Then we show how the bug is affecting all three representations of the program (interpreted RPython, C code, Java byte-code) and how Java Pathfinder discovers the bug in the Java byte-code version. Then we fix the bug and ensure that all three representations of the program work correctly and that JPF considers the program bug-free.

8.4.1 Deadlock

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 8.4: 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 8.5.

The entry point of our program is a method called *main* of the *Application* class, see figure 8.6. The method creates the resources, makes the chain of them, creates and starts the worker threads, and waits until they are finished.

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:

```
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 8.4: Deadlock Test Case: Class Resource

```
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 8.5: 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 8.6: Deadlock Test Case: Class Application (with a bug)

```
$ 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 8.2. 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.

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 8.7; 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 8.8.

```
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 8.7: 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 8.8: JPF Report: No Deadlock after the Fix

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

    def run(self, *args):
        for i in range(self.n):
            self.counter.inc()

class Counter(Monitor):
    def __init__(self):
        Monitor.__init__(self)
        self.value = 0

    # no synchronized here
    def inc(self):
        self.value += 1

    @synchronized
    def getValue(self):
        return self.value
```

Figure 8.9: Race Condition Test Case: Classes Worker and Counter (with a bug)

8.4.2 Race Condition

A race condition is a situation in which two threads or processes alter some shared data at the same time and the result of the alteration depends on the timing or ordering of particular operations on the data. The usual way how to avoid this undesired behavior is to add some synchronization mechanism that enforces mutual exclusion of operations coming from different threads.

To demonstrate discovering of such a bug, we have written a test program that consists of two worker threads that increment a shared counter. The thread and the counter is depicted in figure 8.9. The *Counter* class inherits from *Monitor* class; however, the only synchronized method is *getValue*. Therefore, calls to non-synchronized method *inc* from different threads may lead to unexpected values of the counter.

As mentioned above, the application consists from two instances of the worker thread and one counter that is shared among them, see figure 8.10. The program accepts one argument from the command line that is the number of incrementations of the shared counter performed by each thread. When the worker threads are finished, the final value of the counter is printed.

```
class Application:
    def main(self, argv):
        n = int(argv[1])

        counter = Counter()

        w1 = Worker(counter, n)
        w2 = Worker(counter, n)

        w1.start()
        w2.start()

        w1.join()
        w2.join()
        print counter.getValue()
        return 0
```

Figure 8.10: Race Condition Test Case: Class Application

The program, however, seems to work correctly if every thread increments the counter only several times. For instance, the native version compiled from the generated C files behaves as follows:

```
$ par_run_c.sh 100
200
```

The bug comes to the light only if the execution time is long enough for thread preemption. See the incorrect result 1881421 in the following listing:

```
$ par_run_c.sh 1000000
1881421
```

Note that also Java byte-code and CPython interpreted versions of the program produce wrong results.

Now we will use JPF to discover and locate the bug. For this task, we activate the *precise race detector* module. In order to make the investigation fast enough, we analyze the behavior of the program for 5 increments. The produced report is very clear: the race is for the *value* field of the *Counter* class and the race is accomplished by the method *inc*. See figure 8.11.

The obvious fix of the bug is to add synchronization to the *inc* method, see listing 8.12. After this correction, the program really appears to work:

```
$ par_run_c.sh 1000000
1000000
```

To prove that the fix is really correct, we run the JPF with the race detector again. The result is as expected: no errors detected. See figure 8.13.

```
======================================================= snapshot #1
thread index=0,name=main,status=WAITING,
  this=java.lang.Thread@0,target=null,priority=5,lockCount=1
  waiting on: pypy.worker.Worker_60@730
  call stack:
at pypy.worker.Worker_60.ojoin(Worker_60.j:465)
at pypy.application.Application_55.omain(Application_55.j:139)
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=RUNNING,
  this=pypy.worker.Worker_60@730,priority=5,lockCount=0
  call stack:
at pypy.counter.Counter_57.oinc(Counter_57.j:44)
at pypy.worker.Worker_60.orun(Worker_60.j:245)
at pypy.worker.Worker_60.oRUN(Worker_60.j:115)
at parlibutil.ThreadStarter.run(ThreadStarter.java:17)

thread index=2,name=Thread-1,status=RUNNING,
  this=pypy.worker.Worker_60@757,priority=5,lockCount=0
  call stack:
at pypy.counter.Counter_57.oinc(Counter_57.j:29)
at pypy.worker.Worker_60.orun(Worker_60.j:245)
at pypy.worker.Worker_60.oRUN(Worker_60.j:115)
at parlibutil.ThreadStarter.run(ThreadStarter.java:17)


======================================================= results
error #1: gov.nasa.jpf.tools.PreciseRaceDetector
  "race for: "int pypy.counter.Counter_57.ovalue"   T..."
```

Figure 8.11: JPF Report: Stack Trace of a Race Condition

```python
class Counter(Monitor):
    def __init__(self):
        Monitor.__init__(self)
        self.value = 0

    @synchronized
    def inc(self):
        self.value += 1

    @synchronized
    def getValue(self):
        return self.value
```

Figure 8.12: Race Condition Test Case: Class Counter (fixed)

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

====================================================== statistics
elapsed time:        0:00:03
states:              new=1315, visited=1906, backtracked=3220, end=5
search:              maxDepth=46, constraints=0
choice generators:   thread=1315, data=0
heap:                gc=3453, new=1361, free=476
instructions:        97629
max memory:          9MB
loaded code:         classes=137, methods=1699
```

Figure 8.13: JPF report: Fixed Race Condition

```
class Result(Monitor):
    def __init__(self):
        Monitor.__init__(self)
        self.value = 0

    @synchronized
    def put(self, value):
        if value <= self.value:
            raise ValueError
        self.value = value

    @synchronized
    def get(self):
        return self.value
```

Figure 8.14: Uncaught Exception Test Case: Class Result

8.4.3 Uncaught Exception

Java Pathfinder also checks that none of the program threads ends by an uncaught exception. We demonstrate this unwanted behavior by a simple program that relies on thread execution speed, which is considered as a bad practice.

The heart of the program is a class named *Result*. The result has some inner value that is updated by a method named *put*. This method guarantees, that the value can be only increased; if it is called with a parameter with lesser or equal value than the value already stored in the instance, the *ValueError* exception is raised. See the listing in figure 8.14.

The *Result* class is used by two worker threads. Every thread sleeps for a given amount of time, then puts a new value into the result and finishes. The worker class listing is in figure 8.15.

An application object creates one instance of *Result* and two instances of

```
class Worker(Thread):

    def __init__(self, sleep_period, value, result):
        Thread.__init__(self)
        self.sleep_period = sleep_period
        self.value = value
        self.result = result

    def run(self, *args):
        sleep(self.sleep_period)
        self.result.put(self.value)
```

Figure 8.15: Uncaught Exception Test Case: Class Worker

```
class Application:
    def main(self, argv):
        result = Result()

        w1 = Worker(0.1, 100, result)
        w1.start()

        w2 = Worker(0.2, 200, result)
        w2.start()

        w1.join()
        w2.join()
        print result.get()
        return 0
```

Figure 8.16: Uncaught Exception Test Case: Class Application (with a bug)

Worker. The first worker sleeps for 0.1 seconds and then puts 100 into the result, the second worker sleeps for 0.2 seconds and then puts 200 into the result; see listing in figure 8.16.

The program seems to work in all our environments; here is the output of the C (native) version:

```
$ par_run_c.sh
200
Done.
```

The reason is that the sleep intervals are in order of magnitude longer than time intervals of execution of other parts of the program. However, it is not guaranteed, that the first thread is executed before the second one. JPF discovers this bug; see the report in figure 8.17.

```
====================================================== snapshot #1
thread index=0,name=main,status=UNBLOCKED,
  this=java.lang.Thread@0,target=null,priority=5,lockCount=1
  blocked on: pypy.worker.Worker_59@727
  call stack:
at pypy.worker.Worker_59.ojoin(Worker_59.j:464)
at pypy.application.Application_55.omain(Application_55.j:118)
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=2,name=Thread-1,status=RUNNING,
  this=pypy.worker.Worker_59@757,priority=5,lockCount=0
  blocked on: pypy.worker.Worker_59@757
  call stack:
at pypy.worker.Worker_59.oRUN(Worker_59.j:143)
at parlibutil.ThreadStarter.run(ThreadStarter.java:17)


====================================================== results
error #1: gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
    "pypy.exceptions.ValueError_28  at pypy.ConstantIni..."
```

Figure 8.17: JPF Report: Uncaught Exception

The report shows stack traces of the active threads and the type of the unexpected exception. In this case, JPF can not provide precise location of the creation of the exception because PyPy exceptions do not carry valid stack trace (unlike in pure Java).

This bug can be fixed by making sure that the first thread finishes before the second thread starts. See the fixed listing in figure 8.18.

The JPF report in figure 8.19 proves that the fix is correct.

8.4.4 LTL Formula Violation

In this section, we check a simple linear temporal logic property on a trivial demo program. We will check whether the program works correctly with a file, i.e., whether a file is correctly closed after it was opened.

See the listing of the application that does not correctly close the file in figure 8.20.

We investigate the program by JPF with LTL module. We check whether it does or does not violate the following LTL formula:

```
G{{method:File_56.oopen}->{X{F{method:File_56.oclose}}}}
```

The bug is discovered and a report with a method call trace is produced, see the listing in figure 8.3 earlier in this chapter (page 121).

If we add the call that closes the file, see listing 8.21, JPF reports that the program satisfies the LTL formula, see listing 8.22.

```
class Application:
    def main(self, argv):
        result = Result()

        w1 = Worker(0.1, 100, result)
        w1.start()

        w2 = Worker(0.2, 200, result)
        w1.join()
        w2.start()

        w2.join()
        print result.get()
        return 0
```

Figure 8.18: Uncaught Exception Test Case: Class Application (fixed)

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

====================================================== statistics
elapsed time:       0:00:01
states:             new=61, visited=37, backtracked=97, end=2
search:             maxDepth=14, constraints=0
choice generators:  thread=61, data=0
heap:               gc=129, new=870, free=69
instructions:       14162
max memory:         9MB
loaded code:        classes=132, methods=1685
```

Figure 8.19: JPF report: Fixed Uncaught Exception

```
class Application:
    def main(self, argv):
        f = File()
        f.open()
        f.write()
        # missing f.close()
        return 0
```

Figure 8.20: LTL Violation Test Case: Class Application (with a bug)

```
class Application:
    def main(self, argv):
        f = File()
        f.open()
        f.write()
        f.close()
        return 0
```

Figure 8.21: LTL Violation Test Case: Class Application (fixed)

```
====================================================== LTL Report2:
trace 0
all ok

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

====================================================== statistics
elapsed time:       0:00:01
states:             new=1, visited=0, backtracked=0, end=1
search:             maxDepth=0, constraints=0
choice generators:  thread=1, data=0
heap:               gc=1, new=707, free=18
instructions:       10127
max memory:         6MB
loaded code:        classes=115, methods=1489
```

Figure 8.22: JPF Report: Fixed LTL Violation

```
class Application:
    def main(self, argv):
        low_bit = random(1)
        high_bit = random(1)

        result = (high_bit << 1) | low_bit

        assert result != 0 #wrong assumption

        print result
        return 0
```

Figure 8.23: Random Test Case: Class Application (with a bug)

```
======================================================= results
error #1: gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
    "pypy.exceptions.AssertionError_57  at pypy.Constan..."
```

Figure 8.24: JPF Report: Wrong Assumption About Random Data

Another example of utilization of LTL formulae can be found in [67] or in section 10.1.

8.4.5 Testing with Random Data

There are two sources that make programs behave differently upon each run. The first one is the thread scheduling that is a competence of an underlying operating system. The second one, even more obvious and common, is the input data.

Java Pathfinder has a tool for simulating "nondeterministic" input data. JPF defines a set of functions that return a "random" piece of data from a predefined data set. The subsequent formal verification guarantees that all possible results of, i.e., all possible choices from the set, are examined. In the *parlib* library implementation for Java, we can utilize the JPF interface of this feature.

Our test program uses two random bits that are combined into an integer value. The program, however, wrongly assumes that the resulting integer never equals zero. See listing 8.23. The program in one of four cases ends with an assertion error.

Java Pathfinder deterministically discovers the bug, see listing 8.24.

If we remove the *assert*, JPF systematically prints all possible program outputs and then states that the program is without errors; see listing 8.25.

```
0
2
1
3

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

====================================================== statistics
elapsed time:       0:00:01
states:             new=4, visited=3, backtracked=6, end=4
search:             maxDepth=2, constraints=0
choice generators:  thread=1, data=3
heap:               gc=7, new=757, free=61
instructions:       10921
max memory:         6MB
loaded code:        classes=116, methods=1490
```

Figure 8.25: JPF report: removed wrond assumption about random data

8.5 Conclusion

In this chapter, we have demonstrated our testing approach based on formal methods, more precisely on Java Pathfinder explicit model checker. We used our approach to found a typical set of defects that a multi-threaded program may contain and that are very hard to discover by more conventional testing. We also dealt with traceability, i.e., how the Java Pathfinder report can be used to fix the defect in the source code. We also showed that using of our approach is easy.

In chapter 10, we will use the approach for realistic programs.

Chapter 9

Benchmarks

Every time you start to talk about a new programming language or a compiler, one of the first questions of the audience is about the performance. One might think that with exponential growth of computer performance the efficiency of programs is no longer important, but that is not true for several reasons. First, we demand more from contemporary programs: HD video, pattern recognition, semantic analysis, etc. Second, we can build performance-constrained embedded devices to be employed in new areas; and these constrained devices require efficient programs. See the battery-free devices in [3].

The natural answer for the performance questions are benchmarks. Our benchmarks are synthetic and rather trivial. We compare programs that resulted from our development approach to programs that resulted from more conventional technologies, such as plain C. Fair benchmarking of different computational environments is always difficult as one can always design a test cases that favors one of the environment. Our set of trivial benchmarks have only one goal: to find out whether the performance of RPython and PyPy is reasonable, i.e., whether it is comparable to more conventional approaches.

All benchmarks are run on Intel Core Duo T2300 processor at 1.66 GHz with 1 GiB RAM. The system runs on Ubuntu Linux 8.10 with kernel 2.6.35. GCC is in version 4.3.2, Python is 2.5.2, Sun Java SE is 1.6.

9.1 Variants of Benchmarked Programs

For every benchmark, we typically test several variants of the executable form. From RPython source, we can generate two kinds of output code (C and Java byte-code). In the case of generated C, we can use various switches of GCC and use couple of implementations of the standard C library.

The conventional approach is represented by a program in handwritten C or C++.

Here is the list of the variants:

- *PyPy-C, BoehmGC* is written in RPython, compiled by PyPy to C, uses Boehm garbage collector (mark-and-sweep) and *glibc*[1] implementation of the standard C library. This is the most universal and promising variant and this form is intended for deployment.
- *PyPy-C, refGC* is written in RPython, compiled by PyPy to C, uses PyPy reference counting garbage collector and *glibc*.
- *PyPy-C, refGC uClibc* is written in RPython, compiled by PyPy to C, uses PyPy reference counting garbage collector. Linked with *uClibc*[2]. This library is optimized for embedded applications, it is much smaller than *glibc*. Note that *uClibc* is not compatible with Boehm GC.
- *C* is a handwritten implementation in C, manual memory management, linked with *glibc*.
- *C, uClibc* is a handwritten implementation in C, manual memory management, linked with *uClibc*.
- *C, BoehmGC* is a handwritten implementation in C, linked with *glibc*. It uses Boehm GC for memory management.
- *CPP* is a handwritten implementation in C++, manual memory management, linked with standard C++ libraries.
- *PyPy-JVM* is the same RPython code as in *PyPy-C* cases, compiled by PyPy to Java byte-code.
- *CPython* is the same RPython code as in *PyPy-C* and *PyPy-JVM*, interpreted by the standard Python interpreter.

For the computational benchmarks, we recognize several additional attributes of the tested programs. Instances denoted as *backendopt* were compiled with aggressive optimizations of PyPy compiler (function inlining, some memory operations are moved from the heap to the stack). We do not recommend this optimizations for our development approach for two reasons. First, we did not investigate them in chapter 6 and therefore we are not sure whether they are safe

[1] http://www.gnu.org/s/libc/

[2] http://uclibc.org

enough. Second, they affect traceability needed for formal testing described in chapter 8. However, the reader should be informed that this kind of optimization is possible and is able to improve the performance.

Another additional attribute is the level of GCC optimization; we usually use *-O6* switch that enables the most aggressive optimization, though GCC currently implements only 3 levels of optimization. We consider this level of optimization as safe.

The third case are JVM parameters. The *-Xms200M* parameter sets the initial heap space to 200 MiB which may substantially affect the behavior of the garbage collector.

9.2 Memory Footprint

Reasonable memory consumption is a critical factor of success for embedded software because it determines a class of hardware that can run the software. In comparison, unreasonable consumption of CPU cycles does not prevent the program to run on a given piece of hardware; of course, the run might be unbearably slow.

Measurement of memory consumption is not straightforward in contemporary operating systems with virtual memory and shared libraries. As our primary target platform is Linux, let us provide a short introduction to memory measurement on this platform.

The source of information on memory consumption of every process is the *status* file that is stored in the */proc* directory. In this file, there are memory areas named as follows.

- *VmSize* is the total amount of virtual memory allocated for the process. It is just an address space that is not necessarily used for actual data. Address space is usually not a scarce resource; on 32-bit x86 systems there are about 3 GiB of virtual memory available per process.

- *VmExe* is the size of the segment that is used for mapping of the program code from persistent storage into the operation memory. If the program is run by several processes, this segment is shared among them.

- *VmLib* is very similar to *VmExe*, it contains the code of shared dynamically loaded libraries (DLLs). If a certain DLL is used by several processes, the appropriate part of the memory is shared among them.

- *VmStk* is the size of the system stack of the process. Size of this segment is relatively small for most programs; however, it may grow when a program contains deep recursion.

- *VmData* is the size of the virtual memory segment dedicated for dynamically allocated memory, usually called *heap*. Size of the segment grows as program allocates new memory, for instance via *malloc* from the standard C library. The size also may grow due to the heap fragmentation.

- *VmRSS* is the real amount of physical memory, i.e., not just address space, that a process actually uses. It is called *resident set*. Memory in this set is on addresses from the VmExe, VmLib, VmData and VmStk segments.

- *VmSwap* is the amount of memory that was removed from *VmRSS* and was written to the external memory (disk) because the operating system had a shortage of physical memory.

A simple memory map of a process is depicted in figure 9.1.

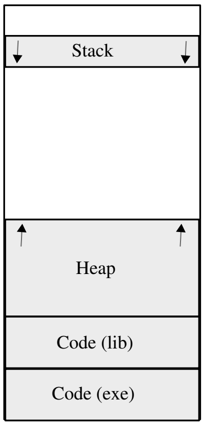


Figure 9.1: Memory Map

In the memory benchmarks, we measure the *VmRSS* value as we consider it as the most relevant. We also make sure that *VmSwap* is always zero.

9.2.1 Static Memory Allocation

In this test, we try to estimate the static memory allocation overhead of run-time environments in order to find out whether our C code generated from RPython is able to compete with handwritten C/C++.

For every programming language/programming environment, there is some infrastructure that allows the program to run. For instance, for a C program, the standard C library has to be loaded in the memory and some data structures instantiated. The standard Java environment does not contain only a huge and powerful standard library, but also the byte-code interpreter and JIT compiler.

Program\List Length	*100*	*1000*	*10000*	*100000*
PyPy-C, BoehmGC	663	663	881	3043
PyPy-C, refGC, uClibc	82	82	348	1696
C	279	279	541	1892
C, uClibc	74	74	336	1688
C, BoehmGC	598	598	602	2208
CPP	807	811	811	2159
PyPy-JVM	11694	11690	11989	13476
CPython	2781	2748	4579	21586

Table 9.1: VmRSS Memory Consumed by a Process with a Linked List, in KiB.

There may also be a significant language-dependent overhead connected with the dynamic objects allocation, i.e., allocation on the heap. Overhead of plain C consists only of data structures needed for the heap management itself. The support for object-oriented programming needs an additional infrastructure such as virtual method tables and data for run-time type identification. The objects in fully dynamic languages such as Python or JavaScript are usually implemented as hash-tables that allow flexibility but are much more memory-hungry than C *struct*s.

The PyPy compiler generates C code; however, it also adds a significant amount of infrastructure: the support for OOP, exceptions, garbage collection, and functions from the standard Python library.

In this benchmark, we have a test program whose data structures consist merely of one dynamically allocated linked-list. Items of the list are "native" objects of the particular environment and contain only one integer as a payload, see figure 9.2.

We let the program allocate a single list of certain length and observe the memory consumption. The lengths of the lists are 100, 1000, 10000 and 100000. The memory consumed by the shortest list is rather negligible so in this case the overall memory consumption goes on the account of mandatory data structures of the particular programming environment.

After the program allocates the list, it starts to infinitely iterate through the list. This kind of infinite loop prevents GC from releasing the data structure; moreover, this steady state allows us to reliably measure the amount of consumed memory.

The amount of really consumed memory (VmRss) for various number of allocated objects and various execution environments can be seen in table 9.1, figures 9.3 and 9.4.

```
# Python
class Item:
        def __init__(self, n, next):
                self.n = n
                self.next = next
// C
struct Item {
        int n;
        struct Item* next;
};

struct Item* Item_init(int n, struct Item* next) {
        struct Item* result = XMALLOC(sizeof(struct Item));
        result->n = n;
        result->next = next;
        return result;
}

// C++
class Item {
        public:
                int n;
                Item* next;
                Item(int n, Item* next);
};

Item::Item(int n, Item* next) {
        this->n = n;
        this->next = next;
}
```

Figure 9.2: Item of Linked-list in Various Environments

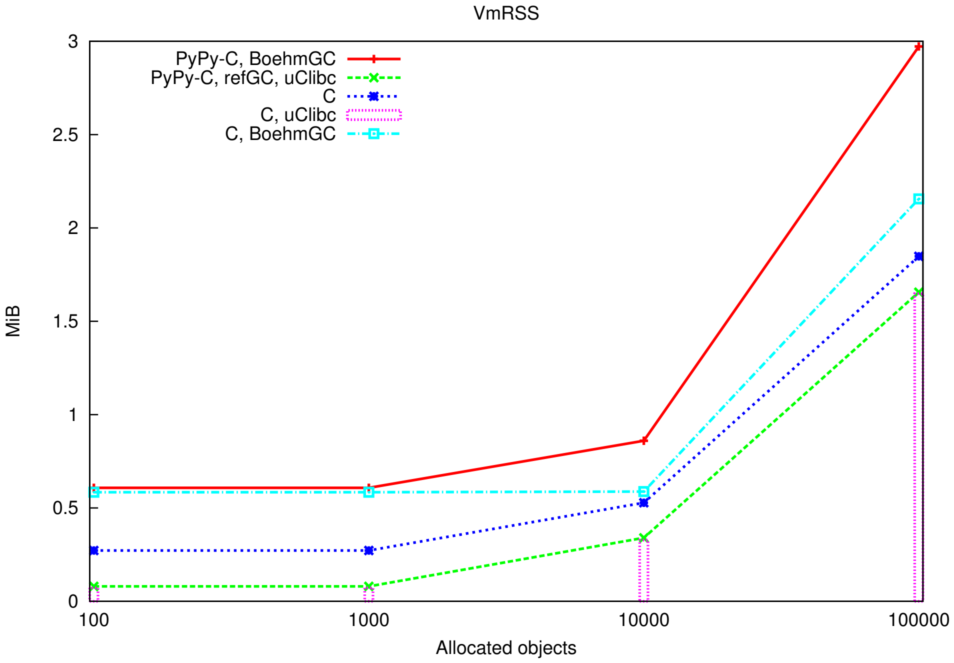


Figure 9.3: Memory Consumption

Analysis of the Results

First see the graph in figure 9.3. The main variant for our intentions is *PyPy-C-BoehmGC*. For the lists of lengths 100 and 1000 the memory consumption is almost the same as for *C-BoehmGC* so we can say that the *PyPy-C-BoehmGC* run-time environment is quite compact. However, for the lengths of 10000 and 100000 there is obvious that allocation of a single object is more expensive in *PyPy-C-BoehmGC*.

Variant denoted as *C* that manages memory manually consumes less memory than an equivalent program that utilizes garbage collector: *C-BoehmGC*. There is a static overhead caused by the Boehm GC library code; on the other hand, we can not surely say whether the allocation of a single object is more expensive when Boehm GC is used.

Apart from *libgc* that implements Boehm GC, there is another library with significant memory requirements, the standard C library implementation: GNU C Library (*glibc*). The variant denoted as *C-uClibc* that uses a more compact implementation of the standard library consumes less memory. The saving is only in the terms of the code size; the heap object allocation expense is the same as in the case of *glibc*.

We can utilize uClibc also for the C code generated by PyPy. The pit-

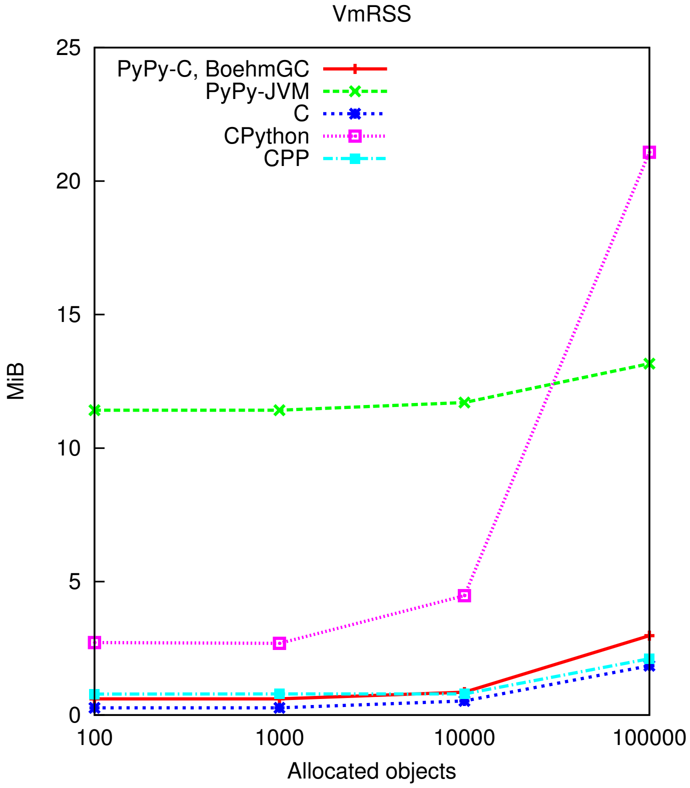


Figure 9.4: Memory Consumption (with Java and CPython)

fall is that Boehm GC is not compatible with this library so we have to use reference-counting GC that can be generated by PyPy itself. It is sufficient for our single-threaded test program. The memory consumption of *PyPy-C-refGC-uClibc* variant is almost identical as the memory requirements of *C-uClibc*. It seems that the overhead of the object-oriented support is negligible which is not surprising as we do not really employ OOP features in this program. In contrast, there was a measurable difference between *C-BoehmGC* and *PyPy-C-BoehmGC* which means that Boehm GC is less efficient for C code generated by PyPy.

Now see the graph in figure 9.4. We have also C++ implementation of the test. We just see that standard C++ library is slightly bigger than standard C library.

The more interesting is the *PyPy-JVM* variant. It is run by the standard Sun/Oracle Java 6. We can see that the static overhead is enormous as the Java run-time environment is powerful and apart from a huge standard library also contains an interpreter and a JIT compiler. On the other hand, the memory consumed by the list itself is comparable to C environment.

Program	*Consumed Memory [MiB]*
PyPy-C, BoehmGC	22.2
PyPy-C, refGC	16.0
C	15.6
C, BoehmGC	31.0

Table 9.2: VmRSS memory consumed by repeated allocation cycles, in MiB.

The last variant is the RPython source run by the standard Python interpreter. The interpreter itself is much more compact than JVM but also much more memory consuming than compiled C. Allocation of objects is extremely expensive. It is due to the fact that every object instance contains a hash table in order to provide fully dynamic behavior.

9.2.2 Repeated Memory Allocation

The memory allocation pattern of most real world programs consists of repeated allocation and release of variable-sized blocks. This pattern may cause additional and inappropriate memory requirements due to the heap fragmentation.

To make sure that programs produced by our development tools behave reasonably, we have written another benchmarking program. The program works as follows. It has an array of length 100 called *history*. Every element of *history* is an array of length 1, 1000 or 100000 that contains integers. Since the distribution of the lengths is regular, the lower bound of program working set is $(1 + 1000 + 100000) \cdot 100/3 \cdot 4 \doteq 12$ MiB.

The program repeatedly iterates through *history* and reallocates each field. Note that 3 and 100 are coprime numbers; therefore, a particular element of *history* is always reallocated to a different size. Every allocated array of integers is filled by a sequence in order to make sure that the memory is really committed.

We iterate the *history* array 1000 times which means we make 100000 individual reallocations, allocating at least 12 GiB in total. Results are in table 9.2 and in figure 9.5.

Analysis of the Results

It is obvious that our simple allocation pattern did not cause any visible heap fragmentation. *PyPy-C-refGC* and plain C versions are only 4 megabytes above the lower bound. The reference-counting garbage collector and manual memory management release memory immediately after an array is destroyed.

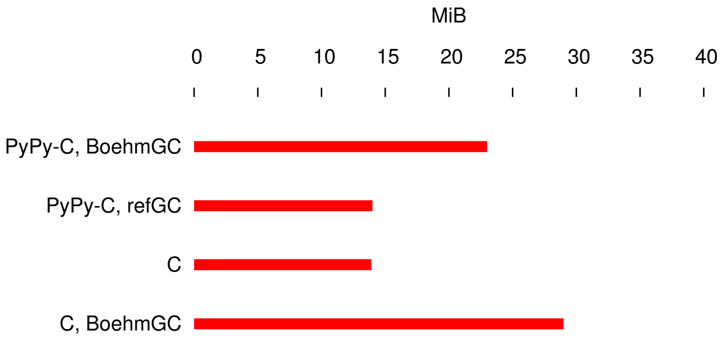


Figure 9.5: Memory Consumption After Allocation Cycles

However, there is a significant overhead caused by the mark-and-sweep Boehm garbage collector. It is because this kind of collecting algorithm waits some time and then releases the unreferenced objects in a batch mode. This overhead is at least 40 %. As this benchmark enormously stresses the GC, we consider this overhead reasonable.

9.3 Speed of Execution

Apart from memory consumption that determines the set of target devices, there is also execution speed. In spite of the fact that there is a non-negligible class of applications that can live with 20 times slowdown caused by naive byte-code interpretation (for instance), computational performance usually matters a lot. Therefore we need to investigate whether the C code generated by PyPy is at least in order of magnitude as fast as handwritten C code.

We measure the consumed time by the standard unix *time* utility. We measure the total run time of a program which may handicap environments that perform JIT compilation, in our case, JVM. However, our test programs are very small and their compilation should be just a fraction of total run time.

We always repeat each experiment at least five times and take median as the result. The dispersion was in all cases negligible.

9.3.1 Computation of a Polynomial

The task of the program is as simple as follows. To numerically compute

$$\int_0^1 \frac{x^2 + 7x + 1}{3}\,\mathrm{d}x$$

Program	*Consumed Time [s]*
PyPy-C	32.24
PyPy-C backendopt	30.20
PyPy-C -O6	24.26
PyPy-C backendopt -O6	24.23
PyPy-JVM	20.55
C	22.40
C -O6	24.20
CPython	1857.40

Table 9.3: Numeric Integration of a Polynomial

using a naive algorithm with step 10^{-9}. See table 9.3 (and figure 9.6 that excludes *CPython*) for the consumed times.

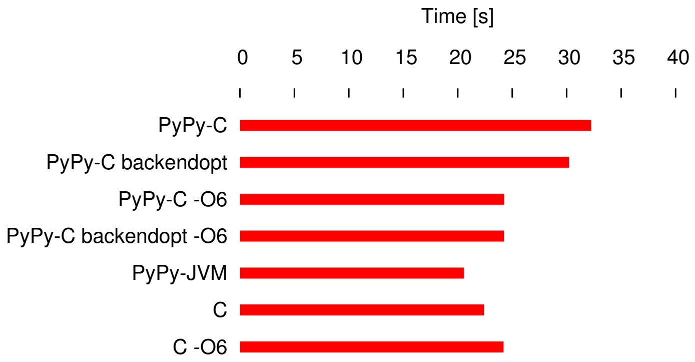


Figure 9.6: Numeric Integration of a Polynomial

Analysis of the Results

The fastest variant in this test is the Java byte-code one. It is a bit surprising because the measured time also includes the time spent by JIT itself; on the other hand, it is known that contemporary JVM JIT can produce very efficient code.

It is also obvious that the default *PyPy-C* variant is as much as one third slower than handwritten C; however, if we switch on optimizations, we can match the pure C results. It is interesting that GCC-level optimizations for PyPy generated C are very successful regardless whether the PyPy level optimizations (*backendopt*) were activated.

Program	*Consumed Time [s]*
PyPy-C	9.55
PyPy-C backendopt	4.99
PyPy-C -O6	1.25
PyPy-C backendopt -O6	1.23
PyPy-JVM	2.21
C	1.75
C -O6	0.57
CPython	71.28

Table 9.4: Fannkuch Benchmark

Just for curiosity we measured also *CPython* interpretation which is naturally in two orders of magnitude slower. It is also interesting that in this particular case the *-O6* option actually hurt the performance of handwritten C.

9.3.2 Fannkuch

This benchmark intensively works with arrays/lists. The implementations were taken from *The Computer Language Benchmarks Game* [68]. The algorithm itself was initially published in [69]; *fannkuch* is an abbreviation for the German word Pfannkuchen, or pancakes, in analogy to flipping pancakes. It works as follows:

1. Take a permutation of $\langle 1, ..., n \rangle$, for example: $\langle 4, 2, 1, 5, 3 \rangle$.
2. Take the first element, here 4, and reverse the order of the first 4 elements: $\langle 5, 1, 2, 4, 3 \rangle$.
3. Repeat this until the first element is a 1, so flipping won't change anything more: $\langle 3, 4, 2, 1, 5 \rangle$, $\langle 2, 4, 3, 1, 5 \rangle$, $\langle 4, 2, 3, 1, 5 \rangle$, $\langle 1, 3, 2, 4, 5 \rangle$.
4. Count the number of flips, here 5.
5. Do this for all $n!$ permutations, and record the maximum number of flips needed for any permutation.

In table 9.4 and figure 9.7 you can see the time consumed by fannkuch algorithm for $n = 10$.

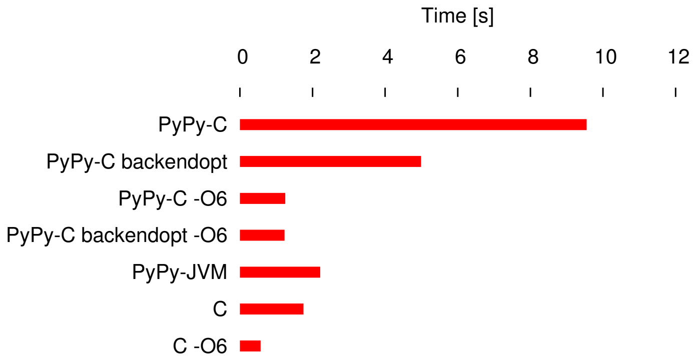


Figure 9.7: Fannkuch Benchmark

Analysis of the Results

What we can see in the results is that *PyPy-C* variant without any additional optimizations is at least 5 times slower than pure C without optimizations. With GCC-level optimizations, the C code generated by PyPy is as fast as handwritten unoptimized C.

9.3.3 Repeated Memory Allocation

In section 9.2.2, we have investigated the memory consumption of a program that cyclically allocates and releases three types of memory blocks.

In this section, we investigate the execution speed of the same program. The time consumed by the 10000 reallocations is in table 9.5 and figure 9.8.

Analysis of the Results

The handwritten C version with GCC optimizations enabled is much faster than the rest. Without the GCC-level optimizations, the result is about 3 times slower. The handwritten C with Boehm GC is about two times slower than unoptimized C. The PyPy-C runs with GCC-level optimizations are about the same speed as unoptimized handwritten C. The *backendopt* switch without GCC-level optimizations have only moderate impact; however, the GCC-level optimizations help PyPy-C a lot.

Performance of Java byte-code version can be significantly improved if the initial heap size is 200 MiB; that probably enables the GC to run in more efficient "batch" mode.

The version that runs on the top of Python interpreter is not much slower than the compiled counterparts. It is due to the fact that memory management

Program	*Consumed Time [s]*
PyPy-C	8.37
PyPy-C backendopt	7.43
PyPy-C -O6	3.59
PyPy-C backendopt -O6	3.60
PyPy-JVM	7.90
PyPy-JVM -Xms200M	3.40
C	3.12
C -O6	1.08
C, BoehmGC	6.08
CPython	11.04

Table 9.5: Time Consumed for Allocation Cycles

as well as the way in which we initialize the array (by the built-in *range()* function) is efficiently implemented in the interpreter itself, i.e., it is actually written in C.

9.4 Conclusion

We have proven that the C code generated by PyPy from the RPython source code can be generally compared with handwritten C code in both memory consumption and computational performance. However, we admit that the code generated by PyPy is usually bit slower and consumes more memory.

The memory consumption of the programs that build small data structures depends also on the size of the standard C library implementation. For programs with larger data structures the overall memory consumption is significantly affected by the way how the memory is managed: manually, by reference-counting GC, or by mark-and-sweep GC.

Boehm GC brings some overhead that is in our opinion bearable. The reference counting GC works without significant memory overhead, i.e., it has the same efficiency as the manual management.

The computational performance depends on the tool-level optimizations used. The optimizations that are provided by the C compiler (GCC) can significantly improve the performance; the benefit for C code generated by PyPy is more significant than in the case of handwritten C. We have seen that GCC-optimized code from PyPy is about the same speed as handwritten C without such optimizations.

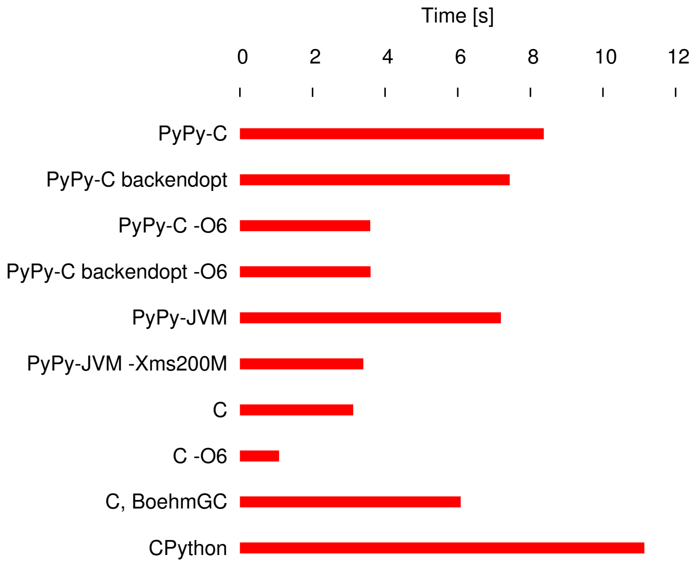


Figure 9.8: Time Consumed for Allocation Cycles

Another information about the performance of the code generated by PyPy can be found in [65].

Chapter 10

Case Studies

Let us apply our approach to two examples. The first one is inspired by a real-world application; however, it is simplified in order to demonstrate formal-based testing with LTL formulae.

The second example is a real application that can be immediately used by users, or can be incorporated as a module into a larger software. It was implemented as a part of the project "Methods of development and verification of component-based applications using natural language specifications"[1] sponsored by Grant Agency of the Czech Republic (GAČR).

10.1 Program for Logging Events

This example is a model of software embeddable into a class of devices called NVR (*network video recorder*). NVR is an embedded computer system that manages IP cameras over a computer network.

10.1.1 Description of the Program

The main goal of the software is to store records from the associated cameras together with metadata such as whether a camera detected a motion, how many frames-per-second are in the video record, temperature, etc.

The metadata are stored in a database that acts as a data warehouse. The metadata are not stored in the raw form but summarized into time intervals. There is a predefined hierarchy of lengths of intervals: 10 seconds, 3 minutes, 1 hour and 1 day. In the case of the motion detection and similar events, the value of every interval is the total number of the events that happened within

[1] GAČR P103/11/1489

the interval. For physical quantities such as temperature, the value of the longer interval is computed as an average from the subintervals that are lower in the hierarchy.

The data warehouse is built on-the-fly by a NVR component called *Logger*. It handles incoming metadata from camera drivers. The camera driver runs in a separate thread and reads a video stream through a network socket; raw video is saved to the disk and extracted metadata are pushed to the *Logger*.

The key class of *Logger* is called *Summarizer*. It maintains the state of the current intervals in the memory, i.e., updates the values according to the events incoming from drivers, and writes the final value into the database when a particular interval elapses. After the interval is successfully written, its value in the memory is cleared, i.e., a new interval is started. The process of flushing interval values from the memory to the database is performed by a thread called *IntervalWriter*. It sleeps most of the time, wakes up only when an interval ends to perform the write & clear operation.

It is important to note that the camera driver runs in soft real-time mode, i.e., it has to be guaranteed that it is never blocked when interacting with *Summarizer*. This proposition holds, because the possibly slow operation of inserting an interval value into the database is performed in *IntervalWriter* thread.

Apart from standard the events that are summarized, there are special events called *alarms* with slightly different policy. Whenever an alarm occurs it has to be written into a persistent alarm log as soon as possible. For this task, there is a thread called *AlarmWriter* that waits for an alarm; when one arrives, the thread is waken up and writes the alarm into the log. Unlike intervals, the alarm data are stored in operating memory as shortly as possible.

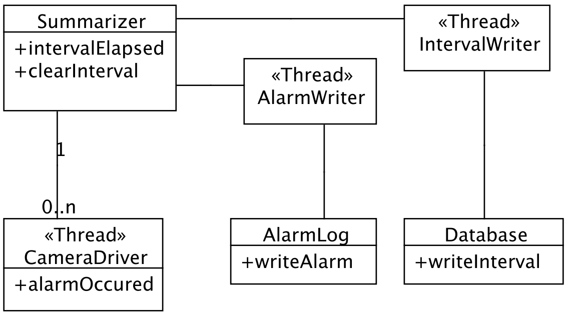


Figure 10.1: Logger UML Scheme

10.1.2 Experiments

In order to check the program by Java Pathfinder, we created a simplified version; simplification is done by replacing the interaction with the real environment by an interaction with some model of the environment. We have a virtual camera driver that does not communicate with a real device but only semi-randomly generates events. The database is replaced by a mockup that simply does nothing. It is important to emphasize that *Summarizer* itself and the thread interaction stays unsimplified. With this setup, the model checker can enumerate all possible states of *Logger*.

There are plenty of properties that we can possibly specify by LTL formulae. For our purpose, we investigate the reaction of the system when an interval elapses. The following formula denotes that whenever an interval elapses, it has to be written to the database (see section 8.1.2 for the specification of the syntax of the formula).

```
G((method:Summarizer.intervalElapsed)
   ->(X(F(method:Database.writeInterval)))
 )
```

Java Pathfinder can check that our implementation has this property. If we inject a bug into *Summarizer*, i.e., remove the *writeInterval* method call from the program, JPF finds this bug and provides a program trace as a proof.

There is also a requirement that when an interval elapses, the intermediate value for the interval has to be cleared in order to start a new interval. This can be expressed by a simple modification of the formula mentioned above, that is:

```
G((method:Summarizer.intervalElapsed)
   ->(X(F(method:Database.clearInterval)))
 )
```

The order of *writeInterval* and *clearInterval* operations has to be always valid, i.e., the interval is cleared only after it is written. The following formula denotes that whenever an interval elapses, it is not cleared until it is written to the database.

```
G((method:Summarizer.intervalElapsed)
   ->(X(
       (~(method:Summarizer.clearInterval))
       U(method:Database.writeInterval)
      )
     )
 )
```

We can ensure the correctness of the formula by injecting a bug into *Summarizer*. If we swap the *clearInterval* and *writeInterval* method calls in the program, the model checker will discover this misbehavior. Note that all these three methods are executed by one thread, the *IntervalWriter* thread.

To demonstrate the ability to find bugs in multi-threaded programs, we will investigate the part of the program dealing with alarms. As mentioned above, alarms are detected by the camera driver (in the driver thread) and always have to be written to the alarm log (by *AlarmWriter* thread). The following LTL formula holds if every occurrence of an alarm is eventually followed by the write operation.

```
G((method:Driver.alarmOccurred)
   ->(X(F(method:AlarmLog.writeAlarm))))
```

After we run the verification procedure, we are sure that *Logger* has this property. Again, if we inject a bug, i.e., remove a thread notification that wakes up *AlarmWriter* from *alarmOccurred* method, JPF discovers the bug.

10.1.3 Conclusion

The presented program is rather a model of a thread interaction. However, this executable model can be evolved to the form of real application with the same scheme of the thread interaction. We earned a set of guarantees that the design is correct.

Apart from the LTL formulae, we checked that the program code has the following properties:

- no deadlock occurs,
- none of the threads ends by an uncaught exception,
- there is no unsafe data access among the threads, i.e., there is no race condition.

10.2 FTP Client

We will develop a simple but fully functional FTP client. It is designed as a library, that can be easily incorporated into programs that need access to data via this protocol.

We have two programs utilizing this library: a simple command-line utility, and a Java OSGi based file manager.

Last but not least, with the help of Java Pathfinder, we provide a set of guarantees of correctness of our implementation.

10.2.1 File Transfer Protocol

The File Transfer Protocol (FTP) was defined by Internet Engineering Task Force's (EITF) standard RFC 959.

The protocol defines two channels: one for commands and one for data. The communication in the command channel is text-based. Every client's command is a single line of text; server responses consist of one or more lines of text.

A client negotiates the conditions of a data transfer via the command channel: which file to transfer, direction of the transfer, which side opens the data connection, etc. When the contract is negotiated, the data channel is opened and the file content is sent, then the data channel is closed. Subsequently, a negotiation of another transfer may begin.

Directories are treated as ordinary files, the content of a directory-file is a list of files present in the directory. Format of the listing is not defined by the protocol; the most common format is the output of the *"ls -l"* Unix shell command.

The protocol is still actively used; however, some parts of the standard are obsolete. Nowadays, the only relevant code set is ASCII (the standard also deals with EBCDIC) and the only relevant transfer mode is *stream*, in which the file content is sent through the network without any interpretation or transformation; RFC 959 also defines structured files and compression.

10.2.2 Design of the FTP Library

Our FTP client library implements only a minimal subset of the FTP protocol necessary for filesystem browsing and file downloading. With these features, the implementation is still useful because file downloading is the most common use case.

The selected feature set is fully covered by these FTP commands:

- USER - The user asks for login with a user name.
- PASS - Send user's password to the server.
- PASV - Sets the server into *passive mode*; the server starts to listen on a port whose number is sent back to the client with response to this command. The client can then open the data channel by connecting to the port.
- LIST - Obtain directory listing of the working (current) directory.
- CWD - Change the working directory.

- RETR - Obtain file content of a specified file.
- QUIT - Let the server close the command channel.

The heart of the library is a state machine depicted in figure 10.2. Transitions from between states generally deal with some network operations such as sending a command and evaluating the server's answer; the transitions labeled as *error* allow a user to cancel an operation that consists of several transitions.

Every state transition can be also viewed as a transaction: it is either fully accomplished or not performed at all. The transaction is accomplished if the associated network interaction is successfully performed: the command is sent, the response is received and the response is evaluated as a desired one.

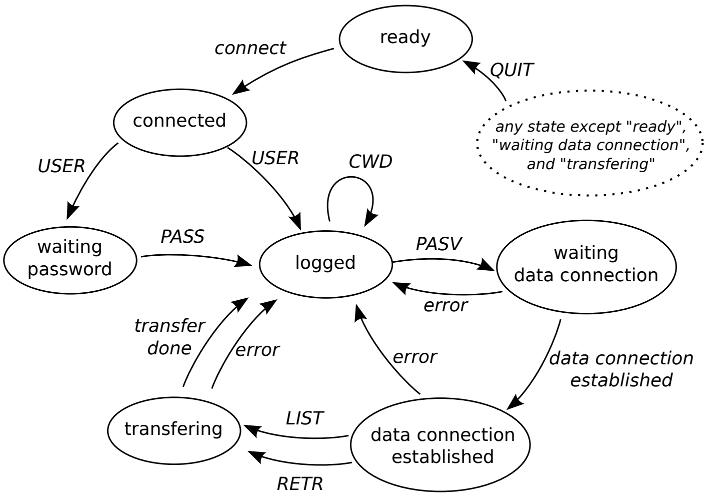


Figure 10.2: FTP Client State Diagram

Particular transitions are relevant only for certain states. When a client application tries to perform illegal transition a *StateException* is raised. Such a situation corresponds with an attempt to violate the protocol; for instance, an attempt to change a working directory (CWD) before the log in procedure (USER/PASS) is performed.

The FTP client library is not used directly through the state machine because it is too low-level; there is a *Client* class that has a convenient interface and encapsulates the interaction with the state machine. For instance, it handles user's login that consists of one or two state transitions (depending on whether a password is required by the server). On the other hand, this facade does not add additional checks; a user can still generate an incorrect chain of commands that is then refused by the state machine. See figure 10.3 for a UML class diagram.

The network interaction is encapsulated into two classes: *Network* and *NetworkReader*. Implementation of this part depends on the selected back-end, therefore there are three implementations: C, Java and pure Python. The implementations are actually part of the *parlib* library mentioned in chapter 7 because it is a general infrastructure.

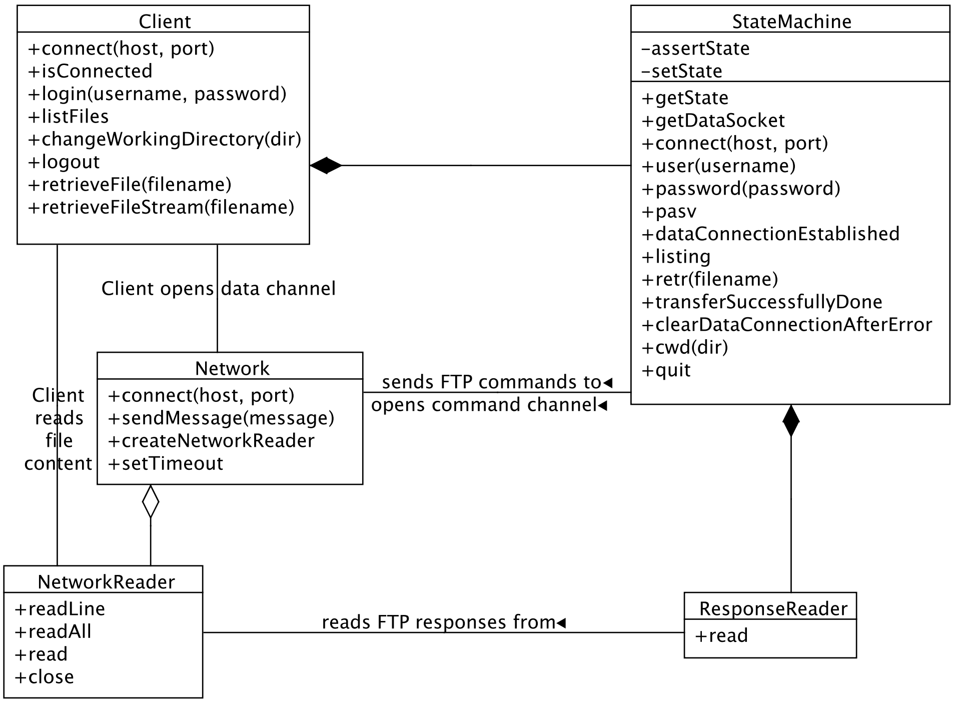


Figure 10.3: FTP Client Classes UML Scheme

Implementation Note: DbC and AOP

We argued that dynamic languages are open for new paradigm in section 4.4. It is now the right time to show how one can leverage this advantage in practice.

See figure 10.4 for a listing of the *password* method of the *StateMachine* class. The method sends the *PASS* FTP command and evaluates the response.

First of all, the *StateMachine* is thread-safe, which means that several methods must be protected against re-entrance by being implemented as critical sections. This is done by the *@synchronized* decorator introduced in section 7.2.1. The decorator has the same semantics as the Java keyword of the same name.

Every method of the *StateMachine* also checks whether it can be executed in the current state. This repetitive *aspect* can be also implemented as Python decorator: *@precond_state* injects a statement that raises *StateException* if the machine state is not the expected one.

```
@synchronized
@precond_state(STATE_WAITING_PASSWORD)
@postcond_states(STATE_LOGGED, STATE_WAITING_PASSWORD)
def password(self, password_str):
    command = Command.PASS(password_str)
    sent = self.net.sendMessage(command.toString())
    if not sent:
        return False

    response = self.reader.read()
    if response is not None and response.isPositiveCompletion():
        self._setState(STATE_LOGGED)
        return True

    return False
```

Figure 10.4: Listing of the *StateMachine.password* Method.

Additionally, we can provide explicit *contract* of the method regarding the machine state after the method is executed. This is done by the *@postcond_state* decorator that checks the machine state after the method body is executed.

The explicit contract given by the pre- and post-condition checks makes the method more readable. For our particular case, it is obvious that the method is valid only in the state *WAITING_PASSWORD* and pushes the machine to the state *LOGGED* if successful; if unsuccessful, the machine state remains unchanged.

10.2.3 Testing

In order to earn a set of *guarantees of correctness*, we need to construct a set of tests. We demand two behaviors:

- The client library behaves according to the protocol. That means that if the server also behaves correctly and there is not a network malfunction and the order of commands is valid, then all the commands are successful.

- The client library reacts reasonably in the case of server or network malfunction. That means, communication may be interrupted at any point and server may violate the protocol (send incorrect response). Under these conditions, some commands may fail, however, fail gracefully, i.e., without unexpected exception or unexpected state transition.

```
server = Server() # mockup server
# mockup network interfaces:
#   (command and data channels
#   distinguished by a boolean)
commandNet = TestNetwork(server , False)
dataNet = TestNetwork(server , True)
client = Client(commandNet, dataNet)
suc = client.connect("foo", 21)
assert suc
client.login("anonymous", "osgiftp@kiv.zcu.cz")
files = client.listFiles()
assert files is not None
for i in files:
        print i.toString()
f = client.retrieveFile("mockup-file")
assert f is not None
```

Figure 10.5: Example of a Test Script

Mockup Server Experiment

In order to provide a very basic guarantee of proper functionality, we built a mockup FTP server. The server is not connected to the FTP client library via TCP/IP network connection but directly interfaces the *Network* class, see figure 10.3. This configuration makes our tests self-contained and repeatable; on the other hand, we do not test the layer of network sockets.

The server contains a simple hardcoded filesystem, i.e., the server can list the root directory and serve several files.

The test scenarios are "scripts" that contain typical use-cases. For instance: to connect, to log in, to obtain the root directory listing, and to obtain a file; see figure 10.5.

The script is correct (contains valid sequence of commands) so it is always executed successfully (assuming both client and server conform the protocol).

We also can write a script containing an incorrect scenario, for instance a one that omits log-in, and see whether an exception is raised.

The experiments with a mockup server give us a basic guarantee that the library implementation works well if the server behaves as expected ("a happy day scenario").

Failing Mockup Server Experiment

We want our FTP client library to behave reasonably even if the server is experiencing failures that may result in violation of the FTP protocol. For this section,

Code	*Meaning*
1xy	Positive Preliminary reply
2xy	Positive Completion reply
3xy	Positive Intermediate reply
4xy	Transient Negative Completion reply
5xy	Permanent Negative Completion reply

Table 10.1: FTP Reply Codes

we assume that the server produces incorrect replies, though still syntactically correct.

With the help of Java Pathfinder we can easily simulate these situations and make sure that our library handles them correctly.

The communication in the FTP command channel consists of commands and replies. Every reply starts with a three-digit code and a human readable message follows. The most significant digit of the code has a meaning defined by the protocol, see table 10.1.

Now, we can create a failing FTP server that responds by a random code to every client's request. That means a particular command either succeeds or fails. Then we run several test scenarios as mentioned above and evaluate the behavior of our code.

Recall that with Java Pathfinder we actually execute all possible runs of the scenarios. A *run* is defined by actual reply code of every command in the scenario. Among all the runs, there is one that completely succeeds, i.e., it is a "happy day scenario" with all reply codes according to client's expectations. But wast majority of runs contain one or more unexpected reply code. If an unexpected reply code is encountered, the particular scenario's operation (e.g. a directory listing) is considered unsuccessful; the scenario continues with the next operation if the failure is not fatal. For instance, log in failure is fatal for all the subsequent operations.

If none of the runs ends with an unexpected exception, then we have a guarantee that our FTP client library behaves reasonable even if the server violates the protocol.

Failing Network Experiment

The FTP client library has to deal with network failures; we have to earn a guarantee that a network malfunction is handled correctly. Recall the transactional design of the library; a state transition succeeds only if both request and response are performed without errors.

```
def sendMessage(self, message):

    # Is tampering on?
    if failure_config.WRITE_LEVEL == 1:
        # This random() method is defined by Java Pathfinder,
        # so both alternatives are eventually examined.
        if random(1) == 0:
            return False # sendMessage signals failure!

    # Normally, this method directly delivers the message
    # to the server mockup.
    self.mockupServer.setCommand(message)
    return True
```

Figure 10.6: Listing of *Network.sendMessage* with Injected Unreliability

Simulation of network unreliability can be achieved by tampering the methods of *Network* and *NetworkReader* classes in a such way that these methods randomly fail to perform a desired action. For some methods, for instance *Network.connect* the failure is straightforward (the method simply returns *False*); on the other hand, *NetworkReader.readLine* may fail in two different ways:

- Returning *None*; that represents a network timeout.
- Returning a truncated data.

For better illustration how the error injection is done, see figure 10.6. The injection utilizes Java Pathfinder's randomization.

If we inject a potential failure to every network operation, we have a perfectly unreliable network. With this unreliable network, we run the test scenarios. Again, if there is no unexpected exception, we have a guarantee that FTP client library deals well with network errors.

Testing Real Network Layer

The real network layer of the FTP client library can not be tested with the help of our mockup server. The layer is very thin, it only encapsulates the network interface of the underlaying platform (Linux operating system, JVM, or Python standard library).

The guarantee of correctness can be earned by code inspection and by testing a real network interaction with real FTP servers. We have several test scenarios that interact with public FTP server installations such as *ftp.zcu.cz* or *ftp.gnu.org*.

```
client: PASV
server: 227 Entering Passive Mode (140,186,70,20,99,15).
client: RETR readme.txt
server: 150 Opening BINARY mode data connection for readme.txt (1765 bytes).
(client opens data channel, reads file content, and closes data channel)
client: 226 Transfer complete.
```

Figure 10.7: FTP Communication of a File Download

We inspected the network code to ensure that it can handle acts of a hostile FTP server such as infinite-length reply. Such an act can not cause a harm because we limited the size of the input buffer.

Multithreaded Experiment

The FTP protocol is not designed to do tasks in parallel; the control channel can not manage multiple data channels. However, FTP client library can be potentially used by several threads. For instance, the application might have a dedicated thread for reading a file content, other threads can check the FTP client state at the same time.

In order to make the client library thread-safe, we only need to add *@synchronized* decorator to every public method of the *StateMachine* class, see also figure 10.4. To earn a guarantee that this solution is correct, we have created a test scenario that utilizes two threads and ran this scenario in Java Pathfinder with race-condition detector.

Discovered Bugs

We found and fixed many bugs in our FTP client library with the help of our testing approach. Let us present two examples.

In the figure 10.7, you can see a listing of FTP communication of a download of one file. The client first sets the server to the passive mode; the server responds by an IP address and a port that can be used for future data connection. Then the client makes the file request; the server responds that it is ready to accept the data connection. Then the file content is transfered and finally, success of the operation is confirmed by the server.

The first discovered bug was incorrect handling of the response to the *PASV* command. Response to this command is expected to be *positive completion* (see table 10.1) with an IP address and a port attached to the message. However, even if the reply code is positive as expected, it does not guarantee that the IP address can be really obtained from the message. If the message is truncated and the IP address and the port are unreadable, the client has to signal the

PASV operation as unsuccessful. The original version of our FTP client library raised a null-pointer exception in this corner case. This bug was discovered by the failing network experiment.

The second discovered bug was in handling of the final confirmation of the transfer operation. In our original version, the client signaled success of a file download even if the final confirmation was not successful. Moreover, the *StateMachine* stuck in the state *transfering.* After the fix, the client signals the whole file download operation as unsuccessful and the state is reset back to *logged.* This bug was found by the failing mockup server experiment.

With the help of testing driven by formal methods, we tested a wide range of situations in which the correct function is guaranteed. Some of the found bugs occur in situations that are highly improbable, however, not impossible; this makes a discovery by conventional testing unlikely. Some more advanced methods such as randomized simulation have potential to discover these corner-case bugs; however, one have only probabilistic (therefore imprecise) definition of the range of situations that were actually tested.

10.2.4 Programs

We have two programs based on our FTP client library. The first is a simple utility that is operated via the command prompt; it is similar to the standard *ftp* utility from Unix systems. As our development approach suggests, the program can be run in three ways: interpreted by the standard Python interpreter, as a native application compiled by C compiler, and on the top of JVM.

The second program that utilizes the FTP client library is an experimental file manager. The file manager is based on a Java component model called OSGi. As mentioned before, the file manager is developed as a part the project sponsored by Grant Agency of the Czech Republic. For this purpose, the generated class files are packed into an OSGi bundle, which is a Java archive (*jar*) with a special manifest. The FTP client library's interface is patterned after the FTP client provided by *Apache Commons Net* library[2]; therefore, we can interchange these two implementations and evaluate functional and non-functional properties for the sake of the project.

10.3 Conclusion

We have proven that the proposed development approach is perfectly usable and useful for developing real-world programs. We have written a clean, elegant and

[2] http://commons.apache.org/net/

high level code, leveraging dynamic languages' strengths by easily embracing paradigms such as AOP.

The FTP client library example also showed how versatile our approach is: we can generate both very compact native command-line utility and an "enterprise" OSGi component from the same source code.

Last but not least, we managed to earn a solid set of guarantees of correctness that can be in our opinion hardly earned by more conventional testing procedures.

Chapter 11

Conclusion

The motivation of this work is to ease the work of developers that can not afford to do things easily: they have to cope with constrained computation power, increased requirements of dependability, etc. This is mainly the case of embedded devices where a significant amount of code tends to be written in traditional low level languages such as C. In contrast, mainstream development of desktop and server applications embraces languages that are memory-safe, provide automatic memory management and are separated from hardware by virtual machines. Embedded software development naturally can adopt these new techniques; however, there are obstacles to be overcome: for instance, it is able to use Java programming language but you have to compile it ahead-of-time or use a custom virtual machine.

In this work, we presented a novel development approach that profits from flexibility of high level dynamically-typed programming languages while allowing generation of compact and efficient machine code and plays well with formal methods. It uses mainstream techniques whenever possible to minimize the technical risks. The approach is aimed at embedded devices; however, it is not limited to this domain. The main overview of the development approach is given in chapter 5.

In section 1.1, we set down three main goals, in short they are: selection of high level programming language and tools for our approach, design a verification procedure that fits the tools and show that the final code is suitable for embedded devices.

Selection of language and tools: To fulfill the first objection of this work, we analyzed the recent trends of both embedded and general software development. In the case of embedded development there is a significant effort to embrace Java as it is modern, portable and widespread language. On the field of general desktop/server applications, there is a notable growth of very high level dynamically-typed languages stimulated by contemporary internet applications. This is analyzed in chapter 4.

For our development approach, we selected a Python based tool-chain called PyPy; it is flexible to embrace a broad set of modern paradigms: aspect oriented programming, design by contract, creation of domain-specific languages. Part of the tool-chain is also a compiler that allows generation of low-level code for various platforms, mainly the machine code with C as an intermediate language, and Java byte-code. The development process also takes advantage of purely interpreted run, i.e., without compilation; that enables rapid application prototyping.

We state that the selected tool-chain is more than promising for the future needs and PyPy can play the role of the "Java+1" for embedded software development.

Verification procedure: The second goal of this thesis was to employ formal methods as dependability of embedded software is crucial. We presented a testing procedure based on explicit model checker called Java Pathfinder. We use the generated Java byte-code as a model of an intended application. We also deal with traceability: if Java Pathfinder finds a bug in Java byte-code instance of the application and provides a report, we are able to translate the program entities mentioned in the report to the appropriate entities in the original program source code, i.e., the Python code.

To assure that the generated C and Java byte-code are equivalent (for our purposes) we investigated and documented the PyPy compilation process, see chapter 6. We also designed thread synchronization objects that work the same for C, Java byte-code, and for interpretation by Python interpreter, see chapter 7.

We demonstrated the testing procedure based on the formal methods by finding a set of typical defects in artificial examples, e.g., a deadlock and a race condition. We also demonstrated the verification against formulae specified in linear temporal logic. Chapter 8 is dedicated to this topic.

Feasibility: The third goal of the thesis was to prove that the generated C code is able to run on a constrained embedded device. We made a set of benchmarks that prove that both memory consumption end computation efficiency is satisfactory. However, we admit that in comparison with hand-written C, there is some memory consumption overhead connected with the utilization of automatic garbage collection. The benchmarks are presented in chapter 9.

Last but not least, we developed a couple of case studies to show that our development approach is viable for real software development. One of the examples is a fully operable FTP client that was developed as a part of the project sponsored by Grant Agency of the Czech Republic.

We state that the code of the case studies is clean and elegant, embraces modern paradigms. We were also able to earn a fair set of guarantees of correctness with the help of our testing procedure based on Java Pathfinder. Case studies are presented in chapter 10.

11.1 Future Work

Improvements of automatic memory management, i.e., the garbage collection, should be the main goal of the future work. We use Boehm GC for the production machine code. This garbage collector is fast; however, it does not have any real-time guarantees and it is *type-inaccurate*.

PyPy has a framework for implementation of custom GCs, it also has several *type-accurate* GC implementations that can only work in single-threaded program.

The first future task should be to port one of these GCs to the multi-threaded environment; in our opinion, that should be straightforward. The second and more challenging task is to implement one of the state-of-the-art real-time garbage collectors on the top of the PyPy GC framework.

The second field with huge potential for improvements is the user friendliness of our customized PyPy-based tool-chain. As PyPy itself is a research project, it was used only by the people that know how the tools work inside and therefore are able to comprehend rather cryptic error messages. For wider adoption of this tool-chain, the user experience should be better.

List of Figures

Bibliography

[1] D. S. Rosenblum, *Formal Method and Testing: Why State-of-the Art is Not the State-of-the Practice*, ISSTA '96/FMSP '96 Panel Summary, 1996.

[2] B. Meyer, *Dependable Software*, to appear in *Dependable Systems: Software, Computing, Networks*, eds. Jürg Kohlas, Bertrand Meyer, André Schiper, Lecture Notes in Computer Science, Springer-Verlag, ISBN 978-3-540-36821-2, 2006.

[3] R. D. Fernandes, N. B. Carvalho, J. N. Matos, *Design of a Battery-free Wireless Sensor Node*, Eurocon 2011 IEEE conference, Lisbon, 2011.

[4] A. Pasetti, *Software Frameworks and Embedded Control Systems*, LNCS Vol. 2231, Springer-Verlag, 2002.

[5] P. J. Koopman, *Embedded System Design Issues*, Proceedings of the International Conference on Computer Design, 1996.

[6] L. Aubry, D. Douard, A. Fayolle, *Case Study On Using PyPy For Embedded Devices*, IST FP6-004779, 2007.

[7] H. Kopetz, *Real-Time Systems, Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, Netherlands, 1997.

[8] P. Herout, *A Proposal of Reliable Embeded Microcomputer*, PhD Thesis, University of West Bohemia, Pilsen, 1999.

[9] J. C. Geffroy, G. Motet, *Design of Dependable Computing Systems*, Kluwer Academic Publishers, ISBN 1-4020-0437-0, Netherlands 2002.

[10] *Software Considerations in Airbone Systems and Equipment Certification*, Document RTCA/DO-178B. ARINC, Annapolis, Maryland, 1992.

[11] J. Barnes, *Programming in Ada 95*, Addison-Wesley Professional; 2 edition, ISBN 978-0201342932, 1998.

[12] GNAT Pro High-Integrity Family, [Online], http://www.adacore.com/home/gnatpro/safety-critical, 2010.

[13] E. M. Clarke, O. Grumberg, D. A. Peled, *Model Checking*, The MIT Press, Cambridge, Massachusetts, 1999.

[14] Z. Manna, A. Pnueli, *A hierarchy of temporal properties*, Proc. ACM Symposium on Principles of Distributed Computing. ACM Press, 1990.

[15] L. Lamport, *Proving the correctness of multiprocess programs*, IEEE Trans. Software Engin. 3, 1977.

[16] R. Pelanek, *LTL Model Checking, Faculty of Informatics*, Masaryk University, Brno, Master's Thesis, 2003.

[17] M. Mukund, *Finite-state Automata on Infinite Inputs*, Tutorial talk, Sixth National Seminar on Theoretical Computer Science, Banasthali Vidyapith, Banasthali, Rajasthan, August 1996.

[18] M. Daniele, F. Giunchiglia, M. Y. Vardi, *Improved Automata Generation for Linear Temporal Logic*, Computer Aided Verification, 1999.

[19] D. Giannakopoulou, F. Lerda, *From States to Transitions: Improving translation of LTL formulae to Büchi automata*, in Proc. of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2002). Houston, Texas, 2002.

[20] J. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, H. Zheng, *Bandera: Extracting Finite-state Models from Java Source Code*, ICSE '00 Proceedings of the 22nd international conference on Software engineering, Pages 439 - 448, Limerick, Ireland, 2000.

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

[22] 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.

[23] G. J. Holzmann, *The Model Checker SPIN*, IEEE Transactions on Software Engineering, Vol. 23(5) pp. 279-295, May 1997.

[24] J. Kačer, *Simulation-Based Checking of Java Concurrent Programs*, University of West Bohemia, 2005.

[25] J. Kačer, *J-Sim – A Java-based Tool for Discrete Simulations*, University of West Bohemia, Faculty of Applied Sciences, Department of Computer Science and Engineering, 2001.

[26] J-SourceMorph, [Online], http://www.j-sourcemorph.zcu.cz/, 2009.

[27] B. Meyer: *Object-oriented Software Construction*, University Press, Cambridge, 1988.

[28] D Programming Language, [Online], http://www.digitalmars.com/d/, 2011.

[29] P. Gastin, D. Oddoux, *Fast LTL to Büchi Automata Translation*, In CAV, 2001.

[30] Eric Bodden, *J-LO A tool for runtime-checking temporal assertions*, Master Thesis, RWTH Aachen University, 2005.

[31] M. Bordin, T. Vardanega, *Real-Time Java from Automated Code Generation Perspective*, Proceedings of the 5th international workshop on Java technologies for real-time and embedded system, ACM, 2007.

[32] D. C. Schmidt, *Model-Driven Engineering*, IEEE Computer, Vol. 39, No. 2, pp. 41-47, February 2006.

[33] M. Mernik, J. Heering, A. M. Sloane, *When and how to develop domain-specific languages*, ACM Computing Surveys, 37(4):316–344. doi:10.1145/1118890.1118892, 2005.

[34] K. Czarnecki, U. Eisenecker, *Generative Programming– Methods, Tools, and Applications*, Addison-Wesley, 2000.

[35] M. Kersten, A. Colyer, *aspectj tools*, [Online], http://kerstens.org/mik/publications/aspectj-eclipse-oopsla2002.ppt, 2007.

[36] R. Laddad, *AspectJ in Action*, Manning Publications Co., ISBN 1-930110-93-6, 2003.

[37] O. Rohlik, I. Birrer, P. Chevalley, *Adapting Control Software Systems Through Aspect-Oriented Programming*, Proceedings of the IFAC World Congress 2005, Prague, Czech Republic, 2005.

[38] V. Cechticky, M. Egli, A. Pasetti, O. Rohlik, T. Vardanega, *A UML2 Profile for Reusable and Verifiable Software Components for Real-Time Applications*, in: M. Morisio(ed), Reuse of Off-The-Shelf Components (ICSR), LNCS Series, Vol. 4039, Springer-Verlag, 2006.

[39] M. Sveda, R. Vrba, *Executable Specifications for Distributed Embedded Systems*, IEEE Computer, IEEE Computer Society Press, 2001.

[40] M. Sveda, *Rapid Prototyping of Networked Embedded Systems*, Proceedings of the IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, 2003.

[41] *JITS - Java in the Small*, [Online], http://jits.gforge.inria.fr/, 2010.

[42] *LeJOS*, [Online], http://lejos.sourceforge.net/, 2010.

[43] A. Courbot, G. Grimaud, J. J. Vandewalle, D. Simplot-Ryl, *Application-Driven Customization of an Embedded Java Virtual Machine*, EUC Workshops 2005: 81-90, 2005.

[44] M. Weiss, F. Ferrière, B. Delsart, Ch. Fabre, F. Hirsch, E. A. Johnson, V. Joloboff, F. Roy, F. Siebert, X. Spengler, *TurboJ, a Java Bytecode-to-Native Compiler*, Lecture Notes in Computer Science, ISBN: 978-3-540-65075-1, Springer, 2004.

[45] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, Turnbull, *The Real-Time Specification for Java*, Addison-Wesley, 2000.

[46] David F. Bacon, Perry Cheng, V. T. Rajan, *A Real-time Garbage Collector with Low Overhead and Consistent Utilization*, LCTES ’03 Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems, ACM New York, NY, 2003.

[47] A. Corsaro, D. C. Schmidt, *The Design and Performance of the jRate Real-time Java Implementation*, In International Symposium on Distributed Objects and Applications (DOA), 2002.

[48] Paul Biggar, Edsko de Vries, David Gregg, *A practical solution for scripting language compilers*, SAC ’09: Proceedings of the 2009 ACM symposium on Applied Computing, ISBN 978-1-60558-166-8, Honolulu, Hawaii, 2009.

[49] 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.

[50] *Programming Language Popularity*, [Online], http://langpop.com/, March 2010.

[51] W. Chun, G. van Rossum, *Python@Google*, [Online], http://www.google.com/events/io/2011/sessions/python-google.html, Google I/O 2011, San Francisco, 2011.

[52] M. Chang, M. Bebenita, A. Yermolovich, A. Gal, M. Franz, *Efficient Just-In-Time Execution of Dynamically Typed Languages Via Code Specialization Using Precise Runtime Type Inference*, University of California, Irvine, http://www.ics.uci.edu/ franz/Site/pubs-pdf/ICS-TR-07-10.pdf, 2007.

[53] *Tiobe Index*, [Online], http://www.tiobe.com/index.php /content/paperinfo/tpci/index.html, March 2010.

[54] Masaharu Goto, *C++ Interpreter - CINT*, CQ publishing, ISBN4-789-3085-3 (Japanese), 1999.

[55] G. Le Lann, *An Analysis of the Ariane 5 Flight 501 Failure - A System Engineering Perspective*, 10th IEEE Intl. ECBS Conference. pp. 339–346, 1997.

[56] *Rake*, example script, [Online], http://en.wikipedia.org/wiki/Rake_(software), 2011.

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

[58] K. Beck, *Test-Driven Development: By Example*, Addison-Wesley Professional, ISBN 0321146530, 9780321146533, 2003.

[59] G. Rossum et al., *Python Documentation*, Release 2.4.4 [Online], http://docs.python.org/release/2.4.4/, 2006.

[60] K.-P. Yee, G. Rossum, *Iterators*, [Online], http://www.python.org/dev/peps/pep-0234/, 2001.

[61] N. Schemenauer, T. Peters, M. L. Hetland, *Simple Generators*, [Online], http://www.python.org/dev/peps/pep-0255/, 2001.

[62] K. D. Smith, J. J. Jewett, S. Montanaro, A. Baxter, *Decorators for Functions and Methods*, [Online], http://www.python.org/dev/peps/pep-0318/, 2004.

[63] H. J. Boehm, *Bounding Space Usage of Conservative Garbage Collectors*, 29th ACM Symposium on Principles of Programming Languages, Portlant, OR USA, 2002.

[64] C. A. R. Hoare, *Monitors: an operating system structuring concept*, Comm. A.C.M. 17(10), 549–57, 1974.

[65] 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.

[66] A. C. Nguyen, S. C. Khoo, *Towards Automation of LTL Verification for Java Pahtfinder*, NUROP Congress, Singapore, 2010.

[67] 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.

[68] *The Computer Language Benchmarks Game*, [Online], http://shootout.alioth.debian.org/, 2011.

[69] K. Anderson, D. Rettig, *Performing Lisp Analysis of the FANNKUCH Benchmark*, [Online], http://citeseerx.ist.psu.edu/viewdoc /summary?doi=10.1.1.35.5124, 1995.

[70] M. Paška, *Installing and Using PyPy Standalone Compiler with Parlib Framework*, Technical Report no. DCSE/TR-2012-09, https://www.kiv.zcu.cz/cz/vyzkum/publikace/technicke-zpravy/, University of West Bohemia, Pilsen, 2012.

Appendix A

Author's Publications and Lectures

A.1 Publications Related to the Doctoral Thesis

1. M. Paška, *Testing of Embedded Systems Using Application Level Threads*, Počítačové architektury a diagnostika 2007 - sborník příspěvků, ISBN 978-80-7043-605-9, Srní, Czech Republic, 2007.

2. M. Paška, *Generative Programming for Embedded Devices*, The 2nd Young Researchers Conference on Applied Sciences - Conference Proceedings Book, University of West Bohemia, Pilsen, 2008.

3. M. Paška, *An Approach to Dependable Embedded Software Development*, State of the Art and Future Research, Technical Report no. DCSE/TR-2008-04, University of West Bohemia, Pilsen, 2008.

4. 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.

5. M. Paška, S. Racek, *Generative Programming and Formal Verification: Case Study*, Proceedings of the Tenth International Converence on Informatics, INFORMATICS 2009, ISBN 978-80-8086-126-1, Herľany, Slovakia, November 2009.

6. M. Paška, *Utilization of Linear Temporal Logic for Generated C Program Code*, Aplimat 2011, Slovak University of Technology in Bratislava, Slovakia, ISBN 978-80-89313-51-8, 2011.

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.

8. R. Lipka, M. Paška, *Defect Discovery by Component Simulation versus Model Checking*, [Submited], 2012.

9. M. Paška, *Installing and Using PyPy Standalone Compiler with Parlib Framework*, Technical Report no. DCSE/TR-2012-09, University of West Bohemia, https://www.kiv.zcu.cz/cz/vyzkum/publikace/technicke-zpravy/, Pilsen, 2012.

A.2 Other Publications

1. M. Paška, *Program pro práci s markovskými modely*, Sborník príspevkov konference PAD 2006, Papradno, SK, ISBN 80-969202-2-7, 2006.

2. M Paška, *Lightweight Distributed Computing in Comparison With Java RMI*, The 1st Young Researchers Conference on Applied Sciences - Conference Proceedings Book, University of West Bohemia, Pilsen, ISBN 978-80-7043-574-8, 2007.

3. M. Paška, P. Dvořák, *Model for Aging Management of Large Instrumentation and Control Systems*, MOSIS 2007, Proceedings of the 41st Conference "Modelling and Simulation of Systems", Rožnov pod Radhoštěm, CZ, ISBN 978-80-86840-30-7, 2007.

4. M. Paška, P. Dvořák, S. Racek, E. Janeček, *Model Based Support for Life Cycle Management of I&C Systems*, Proceedings of EUROCON 2007, IEEE Region 8, Warsaw, PL, ISBN 1-4244-0813-X, IEEE Catalog Number 07EX1617C, 2007.

5. P. Dvořák, M. Paška, *Modely pro sledování spolehlivosti a životnosti komlexních řídicích systémů*, Technical Report no. DCSE/TR-2007-17, JE ČEZ a. s., technical report, University of West Bohemia, http://www.kiv.zcu.cz/cz/vyzkum/publikace/technicke-zpravy/2007, Pilsen, 2007.

A.3 Related Lectures Given

1. M. Paška, *Generative Programming for Embedded Devices*, DSS Working Group Seminar, University of West Bohemia, Pilsen, 24th November 2008.

2. M. Paška, *Generative Programming with Support for Formal Verification*, DSS Working Group Seminar, University of West Bohemia, Pilsen, 13th May 2009.

3. M. Paška, *Generovaný C kód s dokázanými vlastnostmi definovanými v lineární temporální logice*, DSS Working Group Seminar, University of West Bohemia, Pilsen, 22nd November 2010.

4. M. Paška, *An Approach to Embedded System Development Based on Dynamically-typed Language*, D3S Seminar – Department Meeting, Faculty of Mathematics and Physics, Charles University in Prague, Prague, 10th May 2011.

A.4 Teaching Activities

- 2006-2007 KIV/UPS—Fundamentals of Computer Networks
- 2007-2008 KIV/ZIT—Fundaments of Information Technologies