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

An Approach to Dependable Embedded Software Development

State of The Art and Concept of Doctoral Thesis

Marek Paška

Technical Report No. DCSE/TR-2008-04
May, 2008

Distribution: public

Technical Report No. DCSE/TR-2008-04
May 2008

An Approach to Dependable Embedded Software Development

Marek Paška

Abstract

Developing a dependable software is a challenging problem. Dependability requirements are emphasized in the world of embedded systems that may be safety critical. This report summarizes state of the art of techniques suitable for dependable embedded software development: static type analysis, testing, simulation, formal methods. Software engineering techniques such as aspect oriented programming and generative programming are also discussed. Finally we propose a technique based on high level dynamic languages and code generation.

Contents

1 Introduction 2

2 Embedded Devices 2
- 2.1 Microprocessors in Embedded Devices . . . 2
- 2.2 Classification of Embedded Systems . . . 3
  - 2.2.1 Size-based Classification . . . 3
  - 2.2.2 Centralized versus Distributed . . . 4

3 Software in Embedded Devices 5
- 3.1 Reactive Systems . . . 5
- 3.2 Real-Time Systems . . . 6
- 3.3 Program Errors . . . 7
  - 3.3.1 Errors and Failures . . . 7
  - 3.3.2 Fail-Safe and Fail-Operational Systems . . . 8

4 Static Verification 8
- 4.1 Type Checking . . . 8
  - 4.1.1 Type Checking Approaches . . . 9
  - 4.1.2 Type Checking in Ada . . . 9
  - 4.1.3 Conclusion . . . 12
- 4.2 Formal Verification Theory . . . 12
  - 4.2.1 Model-Checking Problem . . . 13
  - 4.2.2 Kripke Structure . . . 13
  - 4.2.3 Linear Temporal Logic . . . 14
  - 4.2.4 Büchi Automaton . . . 16
  - 4.2.5 Model Checking Algorithm . . . 17
  - 4.2.6 State-space Explosion Problem . . . 17
- 4.3 Formal Verification in Practice . . . 18
  - 4.3.1 Manual Creation of a Formal Model . . . 18
  - 4.3.2 Formal Model Extraction . . . 19

4.3.3 Java Pathfinder 2 20
4.3.4 SPIN 21

5 Run-time Verification 21
5.1 Testing 21
5.2 Simulation 22
5.2.1 Simulation-based Testing 22
5.2.2 Simulation-based Checking 22
5.2.3 Conclusion 23
5.3 Design by Contract 23
5.3.1 Conclusion 24
5.4 LTL Run-time Verification 24
5.4.1 Java Logical Observer 25
5.4.2 Conclusion 25

6 Tools and Practices for Dependable Software 25
6.1 Generative Programming 25
6.1.1 Modeling Languages 26
6.1.2 Domain-specific Languages 26
6.2 Aspect Oriented Programming 26
6.2.1 Separation of Concerns 26
6.2.2 Applying Aspects 27
6.2.3 Implementation 28
6.2.4 Conclusion 29
6.3 High Level Dynamic Approach 29
6.3.1 High Level Language Definition 29
6.3.2 Flexibility of Dynamically Typed Languages 30
6.3.3 Compilation to Native Code 31
6.3.4 Conclusion 34

7 Conclusion 34
7.1 Traditional Approach 34

7.2 Model-driven Approach 34
7.3 Proposed Approach 35
7.4 Goals of the Ph.D. Thesis 36

1 Introduction

Embedded systems are a wide and increasing domain. As computer systems become cheaper and more reliable, they are utilized in wider areas of human activities. Developing software for such systems involves dealing with a number or specific constrains, mainly computing resource limitations (CPU and memory). The traditional programming languages used to develop such systems are C and assembly.

Embedded systems are often used in safety-critical applications, e.g., in aerospace field (fly-by-wire). So there is a very strong need for dependability of such systems, even if they are not safety critical. For example, software of a network printer should also ”never break”.

Formal methods can contribute substantially to the reliability of embedded systems. Unfortunately, the state of the practice is far behind the state of the art of formal methods [1]. This fact has several reasons: formal methods are considered hard, require special languages and tools, that the developers are usually not familiar with.

2 Embedded Devices

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 traditional computer design methodologies and tools for embedded applications [5]. The reliability requirements are imposed due to the fact, that embedded systems typically has to work without human intervention, in fact they are often designed to substitute supervision of a human operator.

2.1 Microprocessors in Embedded Devices

There are three main ways, how the CPU is designed to implement a system's desired functionality:

- General-purpose processor
- Application-specific processor

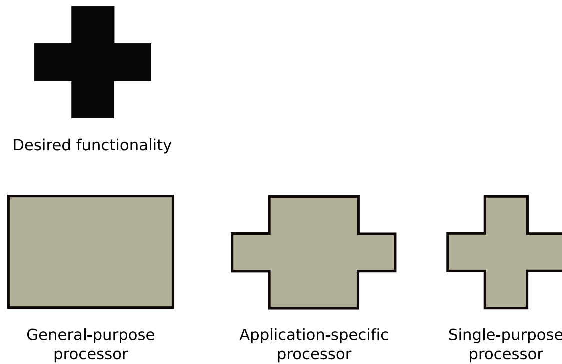

Figure 1: CPU types

- Single-purpose processor

The three types differ in the way the system can be programmed. Single-purpose processor does not have to be programmable. General-purpose and application-specific processors usually run a program. Single-purpose processor can offer desired functionality with the lowest cost and power requirements. The application-specific processor is a compromise, it has some optimizations for a certain application domain while staying customizable by a program [6]. See also fig. 1.

In this report, we deal with program verification and thus we focus on general-purpose processors.

General-Purpose Processor is designed for a variety of computation tasks. Although general-purpose processors are complicated and offer rich functionality, the unit cost is relatively low because manufacturer spreads NRE[1] over large numbers of units. Can yield good performance, size and power as they are carefully designed since higher NRE is acceptable. Usage of a general-purpose processor allows an embedded system to be quickly designed because it involves only software development, not a processor design.

2.2 Classification of Embedded Systems

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

2.2.1 Size-based Classification

Very small systems (i):

[1]Non-recurring engineering, i.e., once paid cost of development

- 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[2].
- 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 (ii) and (iii).

2.2.2 Centralized versus Distributed

An embedded computer system can be either centralized or distributed.

When a system is centralized, all the functionality is concentrated into one node. The main advantage of centralized approach is saving of hardware resources. Even low-cost CPUs are able to handle multiple tasks at the same time. The system is also simpler as no communication between nodes is needed.

When the complexity of an embedded system exceeds certain limits, the construction of dependable centralised system could be difficult and the system has to be

[2]Memory Management Unit

partitioned into several nodes. The main disadvantage of a distributed system is the need of a communication system that has to have the same properties as has the original system, e.g., dependability, real-time properties.

For the distributed system, it is important that the properties that have been established at the subsystem level are maintained during system integration. Such a property of the system is called *composability*.

The major advantage of distributed systems is an ability to construct *error-containment regions*, i.e., when an error within the system occurs it is isolated in one part of the system. It is difficult to implement error-containment regions in a centralized system, because many system resources are used by several tasks.

The recent trend in embedded system design is a distributed system on a single chip. That means, several components on a chip are loosely connected and communicate to each other via standard protocol, e.g., Ethernet [15]. The error-containment regions thus can be properly defined.

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

3.1 Reactive Systems

Embedded programs usually do not use 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 an 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 [9], 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 periodical change of internal clock. When a certain time interval elapses, the state of the environment is obtained and appropriate actions executed. Note that behavior of such systems is generally more predictable than in the case of event-triggered counterparts.

3.2 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 computation is delivered still depends, 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.

3.3 Program Errors

3.3.1 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 [16]:

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 [18]:

- *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 behaviour of a system for a significant portion of mission time. On the contrary, a temporary failure alters the behaviour 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 [14].

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 mus 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.

3.3.2 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 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.

Contrary, the example of 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.

4 Static Verification

4.1 Type Checking

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

The aim of the *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.

4.1.1 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 variable names, 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, the compiler ensures that operations only occur on operand types that are valid for the operation. This early error detection prevents programmer from reasonable class of errors. Many wide-spread languages employ static type checking: C, C++, Java, Ada.

In *dynamic type checking* the type information is associated with object *values* rather than variable names. As the variables change 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' (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 another 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.

4.1.2 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 [7].

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 in some cases even array indices. One of the design goals is memory safety, that means, 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

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

Figure 2: 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: Typecasting in Ada

by its compiler is compact and efficient. This makes Ada very suitable tool for embedded software development, even in challenges areas of avionic and space applications [8].

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 primitive 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 2 shows a definition of two new types: `Apples` and `Oranges`. These new types inherits semantics from 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 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.

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

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

function "*" (Left, Right : Square_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; --OK
    bad_area = width * height --yields a compilation error
end
```

Figure 4: Physical Computation

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

Figure 5: Subtype Example

In Ada, one can also define semantic for user defined types, in our example that means, when we multiply `Meters` with `Meters`, the result is of type `Square_Meters`. This is done by appropriate overload of multiplication operator for `Meters`, see figure 4.

Integrity Checks

Ada types also employs value constraints, i.e., type bears a range of values it can contain. This is often combined with Ada subtypes. 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 provided. See figure 5 for an example of the `Angle` subtype that can be always assigned to a variable of the `Float` type.

4.1.3 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 a program run-time behavior is correct as well. For instance when assigning value -1 to the type `Positive_Integer`, a run-time exception is raised and the correct exception handling cannot be examined at compile-time.

4.2 Formal Verification Theory

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

4.2.1 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 behaviours that traverse through them [17].

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 behaviour (properties) of the model. Linear temporal logic formulae are common.

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 when M does not satisfy ϕ, it provides counterexample, i.e., state sequence from the initial state of the examined system to the state that violates a demanded property.

4.2.2 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 more compact and abstract form. The common approach is to represent the system as the *Kripke structure*. 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 (see section 4.2.3). Formal definition follows:

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

- S a finite set of states,
- $R \subseteq S \times S$ a set of directed edges,
- $L : S \rightarrow 2^P$ 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 short: $s \models p_i$.

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 $s_0 \in S$.

4.2.3 Linear Temporal Logic

Linear temporal logic (LTL) is a modal logic with modalities referring to time [10]. 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)

$$\pi = \pi_0\pi_1\pi_2... \tag{1}$$

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):

$$\varphi := a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \boldsymbol{X}\varphi \mid \varphi\boldsymbol{U}\varphi \tag{2}$$

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,

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

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

$(\pi, j) \models \neg\varphi \iff (\pi, j) \not\models \varphi$

$(\pi, j) \models \varphi \wedge \psi \iff (\pi, j) \models \varphi$ and $(\pi, j) \models \psi$

$(\pi, j) \models \boldsymbol{X}\varphi \iff (\pi, j+1) \models \varphi$

$(\pi, j) \models \varphi\boldsymbol{U}\psi \iff$ 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 integer value to the 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 [12], 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, safety property should specify 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 [10] 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 [11]:

- *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 occurs 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, *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 procedure is exponential.

4.2.4 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: $(s_i, \pi_i, s_{i+1}) \in \Delta, \forall i \in \boldsymbol{N}$. 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 [13], an efficient algorithm for translation of LTL formulae to Büchi automata was published in [19] and [20].

4.2.5 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.

4.2.6 State-space Explosion Problem

The major drawback of model-checking is that it scales badly. When a model size grows linearly, the state space of the model tends to grow exponentially, the problem is referred 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. State model of thread interleaving is created by the Cartesian product of the state models of the individual threads.

Nevertheless 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 exact integer value, only property of the value is stored, e.g., *Negative, Zero, Positive*. This can be done when a certain specification does not depend on 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 statements in C, and if Π holds for a reduced version of P, it also hold for P.

4.3 Formal Verification in Practice

4.3.1 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 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 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 preserves all 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 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 binary Dijkstra semaphore is shown in figure 6. 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 a symbol `p` before it enters the critical section. When user process is leaving the critical section, symbol `v` is sent back to the `dijkstra`

```
#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 6: Semaphore in Promela

process. The `dijkstra` process controls the mutual exclusion by sending `p` symbol when semaphore is open and accepting `v` symbol when semaphore is closed (`open == 0`). Note that 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*.

4.3.2 Formal Model Extraction

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

Example of such a tool is Bandera [22]. 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 mus be reduced in order to make the verification feasible. Bandera provides several optimizations for state space reduction, mainly abstraction and slicing.

4.3.3 Java Pathfinder 2

Java Pathfinder 2 (JPF2) [24] is an explicit model checker for Java developed at NASA. Its predecessor Java Pathfinder 1 [23] attempted to translate Java source code to Promela language, though it is now retired.

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

Conventional JVM executes Java bytecode 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 firstly 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 bytecode instruction, to one line of Java code, or to a block of code.

JPF2 also supports non-determinism to be injected into 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.

4.3.4 SPIN

SPIN [2] is an explicit model checker developed in 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 in section 4.3.1. Specification of valid behaviour 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 pure formal verification. The Promela model along with LTL-based properties is translated to C code, i.e., special-purpose model checker written in C language is generated. The model-checking process itself is performed by a 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.

5 Run-time Verification

5.1 Testing

Software testing is a very general term for process of investigation quality of 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 all combinations of inputs and preconditions is feasible only for trivial programs. Properly designed test should, however, test as much as possible. 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 that testing achieves 100% of some form of 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 input the 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, which is extremely useful for fault-tolerant systems development (*fault injection*).

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 development.

5.2 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.

5.2.1 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 assumed 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.

5.2.2 Simulation-based Checking

Further step is to turn the investigated program into simulation process as well. Example of such an approach can be found in [26] where a Java concurrent program is checked on top of J-Sim [27]. 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 code

that can be handled by J-Sim. Special conversion tool called J-SourceMorph is provided for that task [28]. 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` or `notify` method) are replaced by J-Sim equivalents. This turns concurrent Java program into 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. Simulation can help to discover thread interaction errors, e.g., deadlocks, because the simulated thread scheduler is able to provide more random scheduling then the standard JVM scheduler. Also time-related properties can be more easily examined in the simulated program run.

5.2.3 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 simulation approach is able to deal with time-related properties of the tested software system.

5.3 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 [25]. Eiffel is statically typed object-oriented imperative language with built-in support for DbC. Another language worth mentioning for built-in DbC support is the D language [21]. The paradigm can be relatively easily used in many common languages.

The Eiffel DbC stands on four constructs:

- *Precondition:* An assertion that must hold, i.e., to be true, *before* a method is executed.
- *Postcondition:* An assertion that must hold *after* a method is finished.

```
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 7: Factorial with Contracts

- *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 7. 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.

5.3.1 Conclusion

Adding assertion to a program is a good programming practice. DbC is only more precise application of this practice. DbC is a general principle, though some languages provide built-in support for it.

5.4 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 the against

running program. The main difference between the 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. Reasoning about infinite execution paths only makes sense if one is able to detect cycles in the execution flow, which is usually not possible in today's tools. Note that Büchi automaton described in section 4.2.4 is constructed to recognize *infinite* traces. LTL run-time verification can employ *alternating finite automaton* [42] to cope with finite traces.

5.4.1 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 [41]. 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. 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 6.2) and uses AspectJ to inject a verification code into the code of the original program.

5.4.2 Conclusion

LTL runtime verification is a valuable technique, however it cannot provide proof of correctness as it investigates only actual execution path. The major advantage is that it verifies concrete implementation, because some information is available only at run-time.

6 Tools and Practices for Dependable Software

6.1 Generative Programming

Generative programming is a process of creating a program code that is done 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 native code. Without compilers and automated low-level code generation, the creation of large applications would be unfeasible.

6.1.1 Modeling Languages

Generative programming can be used to generate a program code from a model of the intended program. The model of the program is created 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 to 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 guarantee that the generated program code maintains the properties of the model.

6.1.2 Domain-specific Languages

Another possibility is to generate a program code from a *domain-specific language* (DSL). The main advantage of using domain-specific languages is that the DSL-based description contains all information needed to generate the program code. For instance [31] presents a language called *Action Language* for specification of behaviour 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 [32] and [33]. 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.

6.2 Aspect Oriented Programming

6.2.1 Separation of Concerns

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 break-

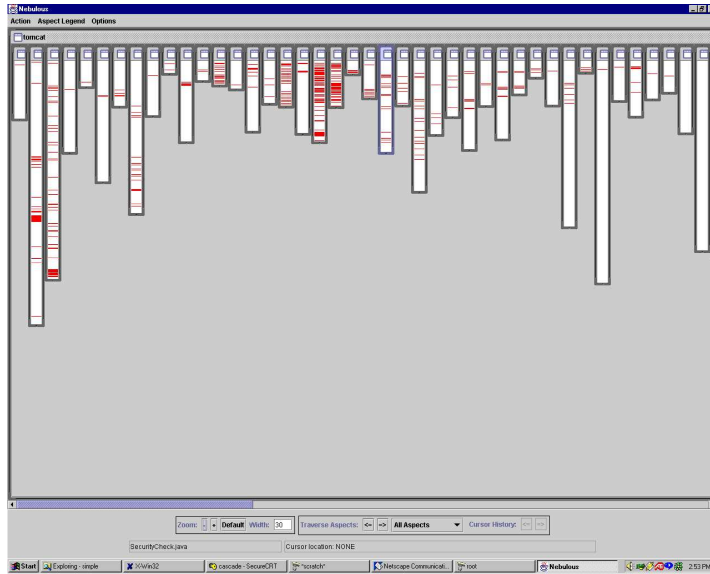

Figure 8: Logging Aspect in Apache Tomcat

ing 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 large portion of the code base. Good examples of such concerns are: logging, security policy, and transactional processing.

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 8 [36], you can see the logging aspect in Apache Tomcat: the white vertical bars represent individual packages, red colour within bars represents code for logging.

6.2.2 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* [29]. First of all, there are some terms that should be explained; the terms were established by first widely used AOP implementation for Java, AspectJ [34].

- *Advice* is a piece of code that implements an aspect. For the case of logging aspect, it should contain call of some logging routine.

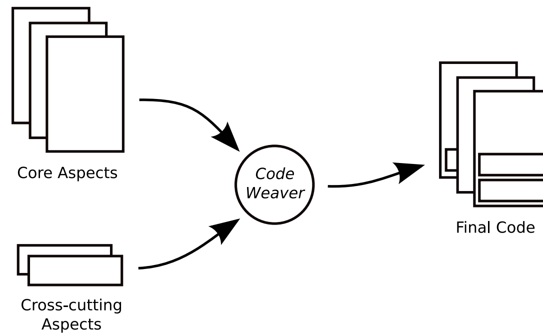

Figure 9: Aspect Weaving

- *Join-point* is a point in 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 transfers money form one account to another. Demanded cross-cut should be defined as follows: method's name contains "transfer" and the method has two parameters of the `Account` class.

6.2.3 Implementation

The AOP paradigm is not directly supported by any of mainstream programming languages. The code of aspect advices has to be injected into the core application code (the core aspect) by some kind of preprocessor. The injection process is referred as *aspect weaving*. The weaver takes core code written in particular programming language and aspects (that are usually written in a aspect-enhanced superset of the used programming language) and produces the final code in the original language. The process is depicted in figure 9.

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 [35].

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.

6.2.4 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. The code dependability is improved by an automatic aspect weaving that is error-prone when done manually.

6.3 High Level Dynamic Approach

6.3.1 High Level Language Definition

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. Firstly, assembly code (i.e., human readable notation of CPU instructions) is now rarely written by hand. Secondly, in the mainstream use there are languages with much higher level of abstraction than C.

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

- Memory management is automatic, some kind of *garbage collection* reclaims unreferenced objects. Explicit memory access via pointers, is forbidden.
- Powerful data types such as variable-size arrays, associative arrays (dictionaries) or sets are incorporated into the language. Language should have an ability to express literals of these data types, e.g., `[1,2,4]` is a list literal. If the powerful data types are not part of the language, they can be provided in a form of tightly integrated standard library.
- Support for object-oriented paradigm is desirable.
  - Classes are first-class objects, i.e., they can be manipulated as any other object instances.
  - Functions (or object methods) are first-class objects. They can be created at run-time, passed as an argument to another function, or returned as a result of a function call.

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.

6.3.2 Flexibility of Dynamically Typed Languages

In statically typed object-oriented languages such as C++ or Java, objects have well defined interface. Every method that is to be called must be known before the call is actually performed (at compilation time).

In a dynamically typed language, method calling can be more naturally viewed as *message passing*. That means, we can naturally "call" an arbitrary method, even a method that particular object does not provide. In this conception, method call is just a sent message and object has a mechanism how to handle completely unknown messages.

No Difference between Compile- and Run-time

Modern object oriented languages like Java or C# has an introspection ability, usually called *reflection*. By reflection, we can examine an unknown object at run-time, for instance, acquire list of methods it implements. It is also possible to call any of the methods from the acquired list. Note that reflection in fact bypasses type checking and allows some sort of "message passing", however, programs usually use reflection only for special purposes because it is not very convenient for a programmer; such a method call is also in order of magnitude slower than native call.

Dynamic languages take this approach further. An arbitrary object can be not only examined at run-time, but also altered at run-time, for instance it is possible to add/remove/rename methods. This usually depends whether a particular language is compiled o interpreted. For example in Java, a programmer writes a class definition (by implementing a set of methods) and compiles it. A set of methods, the objects (instances) of the class will ever provide, is determined when the class definition is compiled. On the other hand, in Python or Ruby, one can imagine definition of a new class as follows: when an interpreter reach a point in source code where a new class is defined, it creates an empty class (without implementation). Then, when it reaches a definition of a method that belongs to the class, it appends it to the class definition. The set of methods the class provides is constructed dynamically and class definition is never closed.

That means there is virtually no difference between "compile-time" (i.e., a time when a program code is "created") and run-time: functionality of a particular object is usually specified in the program source code, but it can be altered during program execution by the program itself.

Proxy Objects

In dynamically typed language, object semantic is determined by a set of messages that object can response to, rather than its inheritance from a particular class [37]. Keeping message passing conception in mind, it is extremely easy to construct an object that forwards every incoming message to another object—a *transparent proxy*. There is virtually no difference whether a client object communicates with the proxy object or with the original object.

The idea of proxy objects can be used for various purposes, for instance, a transparent proxy object can pass messages to another computer by network, to allow simple distributed computation. Proxy object can be used as an implementation of AOP, i.e., the proxy adds some functionality such as logging facility that the original object does not have.

Open for New Paradigms

In statically typed languages, there is more or less statically determined set of programming paradigms they support. Though, new paradigms can be used, it is usually inconvenient. For instance, it is possible to use object-oriented paradigm in pure C, but it is never so natural as in C++ or Objective C. Incorporation of new paradigms such as *aspect oriented programming* or *design by contract* usually relays on some kind of preprocessors.

In dynamically typed languages, on the other hand, the determination is not so strict. The ability to alter classes and functions as ordinary data objects opens the language for new paradigms. Moreover, the support for new paradigm can be simple but powerful, for instance, *design by contract* implementation for Python (written in Python itself) contains only dozens of lines of code [30].

6.3.3 Compilation to Native Code

Programs written in high level languages usually do not run directly in an operating system, i.e., instead of compiling it to native code, they prefer interpretation or some kind of virtual machine. In order to utilize program written in a high level language on an embedded device, translation to an efficient native code is desirable.

Compilation of dynamic languages is not straightforward. When a program is translated to native code, various features that depend on run-time modification of the program are not available. Many interpreted high level languages have, for instance, an ability to interpret a string containing a program source code as a piece of real program, the function is usually called `eval`. Such kind of

functionality cannot be provided in a compiled program because it depends on interpretation.

The limitations mentioned do not mean that a program suitable for compilation has to be static at all. For instance a subset of Python [40] that is suitable for compilation to the native code stays fairly dynamic.

Abstract Interpretation

A traditional compiler parses a program source code, creates an abstract syntax tree (AST), and generates output code directly from the tree. To compile dynamic programs, this pattern does not work, because the structure of the program is "created" during execution (interpretation).

However, dynamic program can be compiled by *abstract interpretation*. It can be viewed as a partial execution of a computer program which gains information about its semantics (e.g., control structure, flow of information) without performing all the calculations. The technique allows a program to behave dynamically (for some bounded time) and then to generate a fairly static code. The process presented in [38] works as follows:

1. Dynamic program source code is run in the interpreted manner. Classes and objects are constructed dynamically. All run-time alterations of the program structure are available, that allows transformations such as AOP; even `eval` construct is permitted. The result of this step is an initialized object system in the memory of the interpreter.

2. In the second step, abstract interpretation itself is performed. The abstract interpretation starts from a selected entry point, i.e., a procedure/function, and follows the control structure of the program reachable from the entry point. To make this process finite, all alterations of the program structure are disabled. Output code is emitted during the interpretation.

The translation based on the abstract interpretation is very flexible. It is possible to generate various final programs from one dynamic source code depending on dynamic transformations (e.g., AOP) performed and entry point selection.

For the purpose of dependable embedded software development, there is a promising idea of translating critical part of a dynamic program (e.g., thread synchronization aspect), to some verification language (e.g., Promela). Thread synchronization as a separate aspect is described in [34].

```
def triple(x):
    return x + x + x
```

Figure 10: Type-agnostic Python Code

Type Inference

The translation process also has to perform *type inference*. In dynamic languages, type checking is done at run-time and many pieces of code are type agnostic. For instance, Python function `triple` (see figure 10) works with any type that is suitable for addition (provides the "`+`" operator): an integer, a float or a string. Type inference determines for which types the function is actually used and what type-aware native code variants to be generated.

Low-level aspects

The high level code does not contain low-level aspects such as memory management or threading implementation, although low-level native code has to address these issues.

In high level languages, garbage collection (GC) is used to manage memory resources. The semantic of garbage collection is always the same: it reclaims objects that are no longer used by the program. However, the garbage collection process can be implemented in various ways. *Reference counting* and *mark-and-sweep* are two main GC approaches, each of them having pros and cons.

When high the level code is translated to the low-level static code, the memory management is just a parameter of the translation process. GC approach suitable for a particular application can be generated. Note that memory management is typical cross-cutting aspect, it is scattered across all the code base and without generative programming, it is hard to change it.

Threading is another low level aspect. Programs may use threads provided by an operating system or threads implemented in user-space. Threads can be either preemptive or cooperative. Generative programming allows various user-space level implementations to be easily provided, for example Python translation to native code described in [39] allows following features:

- Recursion that does not depend on system stack, i.e., is virtually infinite.
- State of a running thread can be saved to a persistent memory and restored later, when needed.
- Co-routines.

6.3.4 Conclusion

High level dynamically typed languages such as Python or Ruby are very useful for rapid application prototyping for their extreme flexibility. With recent methods like translation to static low-level native code based on abstract interpretation, it is possible to take advantage of high level dynamic languages also for embedded applications.

7 Conclusion

The way leading to the dependable embedded software is long and full of obstacles.

7.1 Traditional Approach

The traditional way of developing an embedded software relies mainly on statically typed languages like C, C++, Java, and Ada. These languages offer an error detection based on the compile-time type analysis. The type correctness of a program definitely does not mean that the program is correct at all. A development process can be improved by recent techniques like *aspect oriented programming* or *design by contract*. Verification relies mainly on testing.

Advanced testing methods can significantly increase the level of confidence that the program is correct, however the level of confidence is far behind formal methods.

The traditional approach has two main drawbacks:

- A program code written in relatively low level language (C is the best example) is polluted by implementation details (e.g., explicit memory management). Program code with lower level of abstraction is harder to test and verify, for instance, formal model extraction is very difficult.
- Static typing significantly reduces a flexibility of a program code. Code entities in dynamically typed languages are not so tightly coupled and test code can be thus injected more easily.

7.2 Model-driven Approach

The best assurance of the flawless software can be provided by formal methods. With this approach, the requirements of the intended applications are expressed

in a formalism. The formal description that is written in a special purpose language (e.g., Promela) can be directly verified by powerful techniques such as model-checking.

Apart of undisputed strengths of this approach, there exist also weaknesses:

- When a formal model is evolved to a final implementation code, there is no longer guaranteed that the original properties of the formal model are maintained in the final implementation code. The wide semantic gap between the abstract formal description and the low-level implementation language makes the translation difficult.
- Formal methods are considered hard because they require special tools and languages.

7.3 Proposed Approach

We propose a dependable embedded software to be developed in a general purpose language with high level of abstraction. High level languages such as Python or Ruby are generally accepted tools for rapid prototyping and recently gained a big attention on the field of web applications. In conjunction with generative programming, embedded systems can also benefit from the high level approach.

Major advantage of development in language with higher level of abstraction than, say, Java, is that the code is significantly shorter and contains less implementation details. Dynamic typing also makes the code more compact and much more flexible—paradigms such as *aspect oriented programming* or *design by contract* can be easily incorporated.

The key for utilization of high level languages is *generative programming*. A high level code is not suitable to run directly on an embedded device due to the performance reasons. However, more efficient native code can be generated from the high level description. The translation process described in [38] translates rich enough subset of Python to the native code or Java bytecode. The performance of the generated code is close to hand-written C code [43]. Notable property of the translation is that low-level aspects of the generated code can be changed from the high level perspective, for instance, memory management can be customized by selecting various garbage collection algorithms.

The high level description is also convenient for testing, program analysis, and verification:

- Dynamic typing offers easier testing as the test code can be injected more easily because source code entities are not so tightly coupled as in statically typed languages.

- Dynamically typed languages does not provide compile-time type checking, however the type correctness can be examined by type inference.
- Compact high-level representation is friendly for formal model extraction.
- Instead of translating the high-level code to the final efficient native code, special purpose verification code can be generated, e.g., Promela. Also special-purpose model checker can be generated.

We use the high level language as a modeling (specification) language and production code is generated from the model. However, the high level "specification" language is in fact a general purpose programming language—so it is possible to use variety of existing libraries and tools.

7.4 Goals of the Ph.D. Thesis

We propose a verified and efficient embedded system software to be written in a high level language, that allows smooth evolution from a prototype to a dependable and deployable product. The overall goal of the thesis is to prove whether the proposed approach described in previous paragraphs is viable. The result of the thesis should be a methodology for embedded software development.

The future work should focus mainly on the following subjects:

- Select a high level language and development tools suitable for translation to the code appropriate for embedded applications.
- Design a set of verification methods that take advantage of code generation and advanced methods such as model-checking. Explore how a formal model can be extracted from a high level language code.
- Design a set of error-detection techniques known from statically typed languages that can be done in dynamically typed language via techniques such as abstract interpretation and type inference.
- Show that the generated native code is suitable for running on an embedded device.

References

[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] G. J. Holzmann: "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, Vol. 23(5), May 1997: pp. 279-295.

[3] L. Aubry, D. Douard, A. Fayolle: "Case Study On Using PyPy For Embedded Devices", 2007, IST FP6-004779, 2007

[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] F. Vahid, T. Givargis: "Embedded System Design: A Unified Hardware/Software Introduction", John Wiley & Sons; ISBN: 0471386782. Copyright (c) 2002.

[7] J. Barnes: "Programming in Ada 95", Addison-Wesley Professional; 2 edition, ISBN 978-0201342932, 1998

[8] GNAT Pro High-Integrity Family, *http://www.adacore.com/home/gnatpro/safety-critical*

[9] H. Kopetz: "Real-Time Systems, Design Principles for Distributed Embedded Applications", Kluwer Academic Publishers, Netherlands, 1997.

[10] Z. Manna, A. Pnueli: "A hierarchy of temporal properties", In *Proc. ACM Symposium on Principles of Distributed Computing.* ACM Press, 1990

[11] R. Pelanek: "LTL Model Checking, Faculty of Informatics", Masaryk University, Brno, Master's Thesis, 2003

[12] L. Lamport: "Proving the correctness of multiprocess programs", *IEEE Trans. Software Engin.* 3, 1977

[13] M. Mukund: "Finite-state Automata on Infinite Inputs", Tutorial talk, *Sixth National Seminar on Theoretical Computer Science*, Banasthali Vidyapith, Banasthali, Rajasthan, August 1996.

[14] ARINC (1992). Software Considerations in Airbone Systems and Equipment Certification. Document RTCA/DO-178B. ARINC, Annapolis, Maryland

[15] R. Obermaisser, H. Kopetz, C. El Salloum, B. Huber: "Embedded System Design: Topics, Techniques and Trends", Springer Boston 2007, ISBN 978-0-387-72257-3, pages 339-352

[16] P. Herout: "A Proposal of Reliable Embeded Microcomputer", PhD Thesis, University of West Bohemia, Pilsen, 1999

[17] E. M. Clarke, O. Grumberg, D. A. Peled: "Model Checking", The MIT Press, Cambridge, Massachusetts, 1999.

[18] J.C. Geffroy, G. Motet: "Design of Dependable Computing Systems", Kluwer Academic Publishers, Netherlands 2002, ISBN 1-4020-0437-0

[19] M. Daniele, F. Giunchiglia, M. Y. Vardi: "Improved Automata Generation for Linear Temporal Logic", *Computer Aided Verification*, 1999

[20] 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

[21] D Programming Language, *http://www.digitalmars.com/d/*

[22] J. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, H. Zheng: "Bandera: Extracting Finite-state Models from Java Source Code", Department of Information and Computer Science, University of Hawaii, Department of Computing and Information Sciences, Kansas State University

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

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

[25] B. Meyer: "Object-oriented Software Construction", University Press, Cambridge, 1988

[26] J. Kačer: "Simulation-Based Checking of Java Concurrent Programs", University of West Bohemia, 2005

[27] 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

[28] J-SourceMorph, *http://www.j-sourcemorph.zcu.cz/*

[29] K. Czarnecki, U. Eisenecker: "Generative Programming– Methods, Tools, and Applications", Addison-Wesley,2000.

[30] PyDBC, *http://www.nongnu.org/pydbc/*

[31] 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

[32] M. Sveda, R. Vrba: "Executable Specifications for Distributed Embedded Systems", IEEE Computer, IEEE Computer Society Press, 2001

[33] M. Sveda: "Rapid Prototyping of Networked Embedded Systems", Proceedings of the IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, 2003

[34] R. Laddad: "AspectJ in Action", Manning Publications Co., 2003 ISBN 1-930110-93-6

[35] O. Rohlik, I. Birrer, P. Chevalley: "Adapting Control Software Systems Through Aspect-Oriented Programming", *Proceedings of the IFAC World Congress* 2005, Prague, Czech Republic

[36] M. Kersten, A. Colyer: "aspectj tools", *http://kerstens.org/mik/publications/aspectj-eclipse-oopsla2002.ppt*

[37] Duck Typing, *http://en.wikipedia.org/wiki/Duck_typing*

[38] 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

[39] C. F. Bolz, A. Rigo, "Support for Massive Parallelism and Publish about Optimisation results, Practical Usages and Approaches for Translation Aspects", IST FP6-004779, *http://codespeak.net/pypy/extradoc/eu-reportD07.1_Massive_Parallelism_and_Translation_Aspects-2007-02-28.pdf*, 2007

[40] RPython - Restricted Python, *http://codespeak.net/pypy/dist/pypy/doc/coding-guide.html#restricted-python*

[41] Eric Bodden: "J-LO A tool for runtime-checking temporal assertions", Master Thesis, RWTH Aachen University, 2005

[42] P. Gastin and D. Oddoux: "Fast LTL to Büchi Automata Translation", In CAV, 2001.

[43] M. Paška: "Testing of Embedded Systems Using Application Level Threads," *Počítačové architektury a diagnostika 2007 - sborník příspěvků* ISBN 978-80-7043-605-9, 2007