Unanticipated Context Awareness for Software Configuration Access using the getenv API

Markus Raab

Abstract Configuration files, command-line arguments and environment variables are the dominant tools for local configuration management today. When accessing such program execution environments, however, most applications do not take context, e.g. the system they run on, into account. The aim of this paper is to integrate unmodified applications into a coherent and context-aware system by instrumenting the getenv API. We propose a global database stored in configuration files that includes specifications for contextual interpretations and a novel matching algorithm. In a case study we analyze a complete Debian operating system where every getenv API call is intercepted. We evaluate usage patterns of 16 real-world applications and systems and report on limitations of unforeseen context changes. The results show that getenv is used extensively for variability. The tool has acceptable overhead and improves context-awareness of many applications.

1 Introduction

The goal of context-oriented programming (COP) is to avoid the tedious, time-consuming and error-prone task of implementing context awareness manually, and instead adapt the application's behavior using the concept of layers [1, 12]. Each layer represents one dimension of the context relevant to the application. *Contextual values* [27] act as variables whose values depend on layers. A *program execution environment* consists of the environment variables and key/value pairs retrieved from configuration files. A program execution environment can be tightly integrated with contextual values [21]. *Context awareness* [5] is a property of software and refers to its ability to correctly adapt to the current context. Our aim is to make applications context-aware that previously were not.

For example, an important context for a browser is the network it uses. In a different network, different proxy settings are required to successfully retrieve a web page. We want the browser to automatically adapt itself to the network actually present, i.e., make it context-aware in respect to the network.

Markus Raab
Institute of Computer Languages, Vienna University of Technology
e-mail: markus.raab@complang.tuwien.ac.at

Although COP eases the writing of new software, there remains a huge corpus of legacy software that cannot profit from context awareness. Our paper aims at intercepting the standard API `getenv` in a way that COP-techniques are applied to unmodified applications. We focus on `getenv` because we found that it is used extensively. Our interception technique, however, does not make any assumption on the API. We recommend to specify the values and the context of the program execution environments separately. This configuration specification contains place-holders, each representing a dimension of the context:

```
[/phone/call/vibration]
  type=boolean
  context=/phone/call/%inpocket%/vibration
```

In this example, `vibration` is a contextual value of type `boolean` and `%inpocket%` a placeholder to be substituted in contextual interpretations. Thus, the value of `vibration` changes whenever `inpocket` changes. E.g., when a context sensor measures body temperature only on one side of the gadget, it will change the value of `%inpocket%`. Thus, when the mobile phone is in the pocket, it will turn on vibration. When the mobile phone is lying on a table, it will turn off vibration to prevent falling down when someone calls. If needed, users can even specify further context. For example, some users dislike the context-dependent feature as described. Our approach inherently allows users to reconfigure every parameter in every context. To turn on vibration if the phone is *not* in the pocket, we configure our device differently:

```
/phone/call/inpocket/vibration = off
/phone/call/notinpocket/vibration = on
```

In this paper we analyze the popular `getenv()` API. The function `getenv()` is standardized by SVr4, POSIX.1-2001, 4.3BSD, C89, and C99. Because of this standardization and ease of use it is adopted virtually everywhere, even in core libraries such as `libc`. It allows developers to query the environment. Using standard `getenv` implementations developers have to act carefully: settings valid in the current context can differ from those received through `getenv`. To reduce the danger of assuming wrong context information we propose to use a context-aware implementation. We implement it in the whole system by intercepting every `getenv` API call. Our contributions are:

- We allow unmodified applications to use contextual values. In these standard applications the developers did not initially think of context awareness.
- We conduct an extensive case study and analyze 16 applications and systems.

These contributions are of practical relevance. While other approaches require code rewriting [3, 4], our approach is suitable for legacy applications, flexible and open for extensions. We tackle the research question: "How can we integrate unmodified applications into a coherent, context-aware system?"

The paper is structured as follows: In Section 2 we elaborate on the background. In Section 3 we explain our approach and in Section 4 we evaluate it. The validity of the evaluation is discussed in Section 5. After considering related work in Section 6 we conclude the paper in Section 7.

2 Background

Context-oriented programming (COP) enables us to naturally separate multi-dimensional concerns [5, 24, 26]. In some sense it extends object-oriented programming. Activation and deactivation of *layers* belong to its main concepts. Every layer represents a dimension of context that cuts across the system. All active layers together form the context the program currently is in.

The (de)activation of layers occur at any time during program execution. A currently active stack of layers determines the context the program or thread is in. COP allows us to specify programs with adaptable, dynamic behavior. Later approaches [14] go beyond object-oriented programming: they support program construction with layers only. Furthermore, later work considers software engineering perspectives [24] and modularity visions [13].

Tanter suggested a lightweight subset of COP: *Contextual values*. They are easier to understand because they "boil down to a trivial generalization of the idea of thread-local values" [27]. They are variables whose values depend on the current context. Contextual values originate from COP and naturally work along with the concepts of dynamic scoping and layers.

For newly written context-aware software, COP is a viable choice. For legacy software, however, rewriting seems unrealistic. So in this paper we introduce a new approach that does not require modifications of the application.

3 EnvElektra

In our approach, we want to intercept every call to the `getenv` API. Whenever an application calls the API, we want to invoke a context-aware implementation instead. EnvElektra, which is our research tool, contains such a `getenv()` implementation. The implementation contains a novel matching algorithm for context awareness. When EnvElektra is installed and activated on a system, the matching algorithm will be used for every call of `getenv()` done by any application.

The basic idea of EnvElektra's `getenv()` implementation is as follows: First, it ensures that the data structure is up-to-date. Second, the matching algorithm calculates a new key for the parameter of `getenv()` using the context specification. Third, this key is searched in the data structure. With the found key, we recursively descend until every relevant context is considered.

The library *LibElektra* [21] (shown in Fig. 1) maps the *program execution environments* (e.g., command-line arguments and configuration files) to the in-memory key/value pairs. LibElektra includes start-up code that initializes all key/value pairs from a key database. The key database is modular via plugins [18]. The plugins allow us to use different syntax for configuration files.

Fig. 1 also depicts the EnvElektra architecture. The system with EnvElektra has to provide three artifacts (bold, blue boxes): (1) unmodified applications that become context-aware, (2) context specifications, and (3) context sensors for out-of-

Fig. 1 Architecture of EnvElektra. The common data structure is a set of key/value pairs (middle). Bold, blue boxes need to be provided by users of EnvElektra.
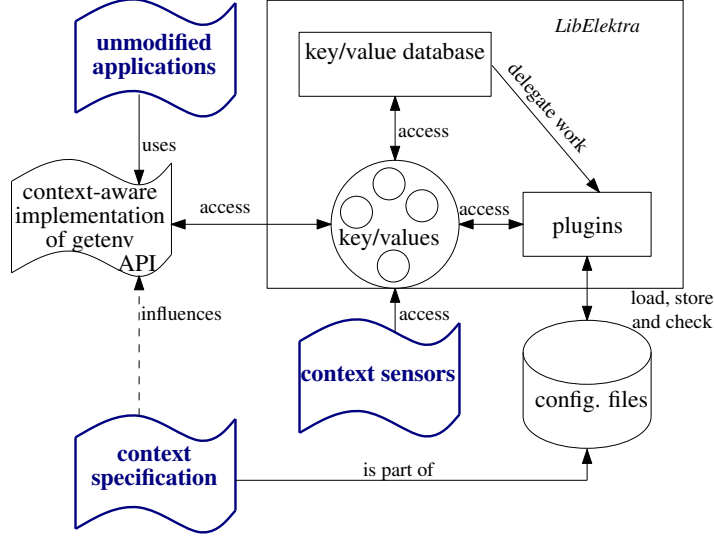
process layer (de)activation. In the remainder of this chapter we will explain the user-provided artifacts and the matching algorithm. Finally, we will give a full example demonstrating how the system works interconnected.

3.1 Context Sensors

An essential issue to enjoy global, context-aware configuration access without modifying the application is an out-of-process layer (de)activation. We will show why such context sensors require us to use a database.

The original function `getenv()` retrieves values from the environment. Internally, it uses the data structure `char** environ`. By design, `environ` is copied into every process and will not receive any external changes afterwards. Thus, `environ` cannot consider out-of-process changes and cannot be used in EnvElektra.

We prefer to use configuration files that are read by the application itself. Then security is correctly handled by the operating system. In EnvElektra the administrator decides which configuration files are used, possibly with different syntax for each file [20]. EnvElektra makes sure that all applications have the same global view of the system's configuration files leading to a consistently configured system. This way values returned by `getenv()` will not be different from values retrieved from configuration files. The configuration files are viewed as a key/value database suitable for `getenv` lookups.

Context sensors observe the system and change the database when they detect context changes. They are responsible to modify the layers accordingly. Context sensors write their layer information into `/env/layer`. The key `/env/layer` is part of the database and resides within one of the configuration files. The use of files enables out-of-process communication between context sensor and applications. Thus, context changes can have an immediate effect on applications.
We identified two different kinds of context sensors to be used with our approach:

Information within the Database: Quite often, the necessary value is already present in the database. For example, in Linux many syscalls and the `/sys`-file system already provide much information. Using plugins, these sources are easily embedded within the database. Then we only need a symbolic link from `/env/layer` to the correct key. For example, if `/env/layer/nodename` points to `/syscall/uname/nodename`, then `%nodename%` will resolve to the nodename as returned by the `uname` system call. In EnvElektra we mount plugins into any part of the hierarchy [18].

Context Sensor Daemons: In other cases, we implement a daemon, i.e. an active process, that updates `/env/layer`. Doing so, we can implement hysteresis, value transformations, and even complex feedback control systems. For example, to update `%inpocket%` a daemon measures the temperatures and modifies `/env/layer/inpocket` whenever we cross a threshold value. Changes in the database influence all processes across the whole system.

3.2 Context Specification

Up to now, we have established a database that contains key/value pairs to be used in a `getenv()` implementation. We have to make the database context-aware with the layer-information present in `/env/layer`, e.g.:

```
/env/layer/inpocket = notinpocket
```

Furthermore, we specify which key is used in which contextual interpretation:

```
[/phone/call/vibration]
  type=boolean
  context=/phone/call/%inpocket%/vibration
```

Now, when an API accesses `/phone/call/vibration`, the lookup layer will search for `/phone/call/%inpocket%/vibration`. Layer interpretations are stored in the database below the key `/env/layer`. In this case the correct contextual interpretation of `%inpocket%` is `notinpocket`. Using more than one placeholder creates several dimensions of variability. Late-binding is necessary so that unmodified software benefit from contextual features. EnvElektra needs to resolve its context awareness as late as possible, i.e., on `getenv()` calls.

For example, if a phone-call application executes `getenv("vibration")` it will look up `/phone/call/vibration`. Because of the **context** specification, we know we want the key `/phone/call/%inpocket%/vibration` instead. For the correct

interpretation of `%inpocket%` we will lookup `/env/layer/inpocket` first. We get the value `notinpocket` for the layer `%inpocket%`. Thus, `getenv("vibration")` will return the value of `/phone/call/notinpocket/vibration`.

3.3 Matching algorithm

The core of our approach is the contextual lookup within our alternative implementation of the `getenv` API. In EnvElektra `getenv()` provides the context-aware variability. The essence of EnvElektra's `getenv()` implementation is:

```c
char* getenv(char* key) {
    if(needsReload(conf)) {
        reloadConfiguration(conf);
        reloadLayers(conf);
    }
    return contextLookup(conf, key);
}
```

Context is not static but dynamically changes over time. Our approach supports dynamic changes of context using `reloadLayers()` even though the original `getenv` implementation did not. The interception approach limits us to context-changes within `getenv()`: We cannot (de)activate layers at other places. Instead, we make sure that for every `contextLookup()` the correct context is used. The matching algorithm `contextLookup()` is recursively defined:

```c
char* contextLookup (KeySet* cfg, char* key) {
    m = lookupBySpecification (cfg, key, "context");
    if (m) return contextLookup (cfg, fix(m));
    else return lookup (cfg, key);
}
```

The idea of the algorithm is: First, we look whether a context is specified for the key. If it is, `contextLookup` descends recursively after replacing all placeholders in the key. If it is not, a ordinary lookup will be used. The full implementation features namespaces, symbolic links and defaults [20].

3.4 Example

We present a full example that demonstrates recursion with several layers. Suppose a mobile phone is lying on the table in a building during a meeting. To simplify the example, we assign constant values to the layers:

```
/env/layer/inpocket = notinpocket
/env/layer/inbuilding = inbuilding
/env/layer/inmeeting = inmeeting
```

In a real system, a sensor will continuously update the values. So far, we already discussed the layer `inpocket`. The layer `inbuilding` represents a value from a location context. Layers such as `inmeeting` are called *virtual sensors* [1]. In this case the value of the layer is calculated by a sensor querying the person's schedule. The application running on the phone uses the following non-context-aware code:

```c
char* use_vibration = getenv("vibration");
if (!strcmp(use_vibration, "on")) {/* activate vibration */}
```

We add context awareness with the following specification:

```
[/phone/call/vibration]
  type=boolean
  context=/phone/call/%inbuilding%/vibration
[/phone/call/inbuilding/vibration]
  type=boolean
  context=/phone/call/%inpocket%/%inmeeting%/vibration
[/phone/call/notinbuilding/vibration]
  type=boolean
  context=/phone/call/%handsfree%/vibration
```

Due to lack of space, we here specify only two of the six possible configurations:

```
/phone/call/inpocket/inmeeting/vibration = on
/phone/call/notinpocket/inmeeting/vibration = off
```

Suppose the mobile phone gets a call. By above `getenv` we request to lookup `/phone/call/vibration` to know whether vibration is turned on. In the first step, it will find the **context** and resolve `inbuilding`. In the next step, it will recursively search in the specification again, and find another **context** with `/phone/call/%inpocket%/%inmeeting%/vibration`. Then the placeholders are again replaced with the respective values. Resolving this key, the algorithm will not find another matching specification. Thus, it returns the configuration value of not in pocket and in meeting, i.e., `/phone/call/notinpocket/inmeeting/vibration`. Because this configuration value is **off**, the phone will not vibrate.

4 Evaluation

Our methodological foundation is built on "theory of cases" [6, 7]. Other research should supplement our work with further case and user studies.

We chose 16 popular systems for evaluation (as discussed in threats to validity in Section 5). We will solely focus on existing applications and their integration into a coherent system.

The evaluation was conducted on different machines using Debian GNU/Linux Jessie 8.1 amd64. For the evaluation we globally intercept `getenv()` using `/etc/ld.so.preload`. By listing EnvElektra in `/etc/ld.so.preload` it will be loaded before any other library. Thus its symbols will be preferred. Because of this preference EnvElektra will be used for every `getenv()`-call.

In each of the following subsections, we will answer one of the questions:

RQ1:   What are the usage patterns of `getenv()` in popular applications?
RQ2:   For which applications can we actually exploit `getenv()` to be used for unanticipated context awareness? What are the fundamental limitations?
RQ3:   What is the overhead that occurs in a system using EnvElektra?

4.1 RQ1: Usage Patterns

Only APIs that are actually called during runtime can be exploited for context awareness. To learn more about usage patterns, we count how often `getenv(key)` is executed.

application	lines of code	getenv all	getenv init	all unique	later unique	same
akonadi	37,214	10,357	8655	110	12	5126
chromium	18,032,183	6006	1803	1118	192	165
curl	249,380	19	8	12	8	4
eclipse	3,311,712	2790	2696	389	42	1495
evolution	672,789	4407	1488	1060	24	163
firefox	12,394,938	3371	2049	276	70	895
gimp	901,703	2551	1115	217	137	364
inkscape	479,849	722	457	160	51	166
libreoffice	5,482,215	3354	2891	258	59	1493
lynx	192,012	1931	961	27	27	923
man	142,183	2862	13	86	76	2
smplayer	76,170	212	164	71	8	53
wget	142,603	11	10	8	1	3
Mean	3,217,074	2969	1716	292	54	835
Median	479,849	2790	1115	160	42	166
Total	41,821,956	38,593	22,310	3792	707	10,852
KDE	*	*	9606	265	*	2634
GNOME	*	*	144	47	*	4
Debian	*	*	5317	430	*	286

* Any of the above applications can be started within the same session.

lines of code: Count lines of code with the tool `cloc`.

getenv all: Count all calls to getenv while using the application.

getenv init: Count all calls to getenv while starting the application.

all unique: From all getenv calls, how many different `keys` were used?

later unique: From getenv calls after initialization, how many different `keys` were used? For wget and curl the first download counts as initialization.

same: From the getenv calls during startup (during runtime an arbitrary high number could be acquired), what is the maximum number of queries with the same value for the parameter `key`?

To interpret the numbers correctly we have to know that the usage patterns vary widely even for the same application. For example, firefox started within GNOME requests 11 GNOME specific and 8 GTK specific environment variables (like `G_DEBUG`). If executed on a system with OpenGL enabled, 43 additional environment variables (like `__GL_EVENT_LOGLEVEL`) are used to determine OpenGL configurations. Additionally, the tested system requested three vendor (`NV`) specific variables. For KDE, `KDE_FULL_SESSION` was used as detection. Then 8 more KDE-specific and 15 more QT-specific environment variables were requested if started within KDE. Thus, the numbers depend on the desktop environment and hardware.

For better reproducibility, we freshly installed Debian Jessie KDE and GNOME variants, respectively. The only modification was the installation of EnvElektra. For example, on a daily used KDE with many installed applications, we measured 210.276 `getenv()` during startup, which is 21 more than with a freshly installed KDE. We see that the numbers also depend on the installed software.

The above 13 applications request an average of 2969 environment values (2790 median). Akonadi, configured to use IMAP, had the highest number of calls to `getenv`. The reason seems to be a potential misuse of a libc function which requested `LANGUAGE` 5126 times. During the KDE startup 27% of all `getenv` calls were `LANGUAGE`. We conclude that excessive use can be unintentional.

From the numbers in the table we conclude that `getenv()` is used extensively in all examined applications. Applications often reread environment parameters during user interactions. This statement is true for both large applications and small helper tools. As expected, large feature-rich applications request much more environment variables. The ratios of requested and unique environment variables varies greatly: it is 14% median, and in akonadi it is $\sim 1\%$. We see that applications tend to request the same variables often.

Our findings regarding **RQ1** are:

(1) We quantitatively show that `getenv()` is pervasive. We think that the usage patterns stem from a rather random use of `getenv()`: variability seems to be added ad-hoc whenever single developers needed it. Because `getenv()` has no noticeable performance implication and typically is not unit-tested, it is likely that quality assurance will not find unnecessary occurrences.

(2) Based on our observation, `getenv()` is used frequently after startup.

Implications: Developers seem to not optimize calls to `getenv()`. The resulting high number of `getenv()`-calls open up possibilities to influence the behavior of applications on context changes.

RQ2: Unanticipated Context Awareness

We already showed that the use of `getenv()` is pervasive, even after startup. Now, we want to find out whether changes in the context — and thus in the variables returned by `getenv()` — actually have an influence on the behavior.

We found that in help-, save- and open-dialogs different values returned by `getenv()` often influence the behavior of the application in a way easily visible to the user. These environment variables often have immediate and visible impact when changed dynamically. For example, gimp uses for every open dialog `G_FILENAME_ENCODING` and for every help dialog `GIMP2_HELP_URI`. On context changes, e.g. when we enter another network or mount a new file system, the software can automatically be adapted with EnvElektra.

Now, we investigate context awareness of proxy settings. A user changing the network with a different proxy should be able to continue browsing. `lynx` requests and correctly uses `http_proxy` for every single page. `curl` has the same behavior and reloads 7 additional environment variables every time. `wget` gives less control per download but still requests `http_proxy` for every page in recursive downloading mode. Firefox uses the proxy for most pages but pages in cache are displayed even when the proxy is unreachable. Chromium is the only browser not rereading `http_proxy`. Instead, it requests many internals such as `GOOGLE_API_KEY` during run-time. EnvElektra supports `http_proxy` well.

Our approach is very successful whenever an application executes other programs because during the startup of the programs the whole environment is always requested and used. Many programs use a pager or editor as external program. For example, `man` executes a pager for every displayed manpage.

For some applications it is possible to specify a configuration file using an environment variable. In EnvElektra configuration files can be mounted. Then they are a part of the database, which permits full configurability. For example, `less` executed within `man` uses the environment variable `LESSKEY`. In such cases our approach provides seamless context-aware configuration.

Some `getenv()` calls, however, do not have any user-visible impact. Instead, they seem to be left-overs. In LibreOffice, `WorkDirMustContainRemovableMedia` is obviously a workaround for a very specific problem. It is not documented and searching the web for it only reveals the use in the source code. Instead, `OOO_ENABLE_LOCALE_DATA_CHECKS` is an announced workaround. In GTK `GTK_TEST_TOUCHSCREEN` is requested extensively. According to the commit log it was explicitly introduced as a test feature.

Sometimes recurring `getenv` cannot be exploited to improve context awareness. For example, `LANGUAGE` is requested very often but does not influence the user-interface after startup. Here changes at runtime seem to have no impact. Such environment variables will only be context-aware during the start of an application.

A limitation of our approach is the impossibility to detect unwanted changes of environment variables. For example, the environment variable `CC` can change during compilation. Obviously, this easily leads to inconsistent compilation and linking.

In EnvElektra the runtime-context-change feature can easily be (de)activated for process hierarchies, though.

Not a single crash occurred in our experiments regardless of which values we modified. This behavior is not entirely surprising: First, software should validate values returned from `getenv()`. Thus, wrong values from `getenv()` are rejected. Second, we did no systematic stress testing but only searched for useful changes.

Our findings regarding **RQ2** are:

(1) We show that many practical use cases exist where context changes are applied successfully at runtime.

(2) Limitations include that some `getenv()` calls do not have visible impact and that context switches in rare cases lead to incorrect behavior.

Implications: EnvElektra increases the context awareness for the evaluated applications. Specific functionality is even flawlessly context-aware.

RQ3: Overhead

Finally, we want to evaluate whether the overhead of EnvElektra is acceptable. The benchmarks were conducted on a hp$^\circledR$ EliteBook 8570w using the central processor unit Intel$^\circledR$ Core$^{\text{TM}}$ i7-3740QM @ 2.70GHz. Overhead is measured with valgrind by running the executable without and with EnvElektra.

The glibc `getenv()` implementation linearly searches through the whole environment. On the one hand, our implementation does not have this constraint. Its complexity is $O(\log(n))$ compared to $O(n)$ for `environ` iteration. We do not use unordered hash maps because we need lexically ordered iteration, e.g. to iterate over all layers and during `reloadConfiguration()`. On the other hand, the contextual lookup involves recursion. Depending on the specification EnvElektra needs additional nested lookups.

In a benchmark we compared 1,000,000 `getenv()` calls with the same number of EnvElektra's lookups. We did 11 measurements and report the median value. For a small number (30) of environment variables, standard `getenv()` implementations (0.03 sec) clearly outperform EnvElektra's lookup (0.06 sec). For 100 environment variables (which is a typical value) they perform equally well: 0.076 sec for standard `getenv()` and 0.073 sec for EnvElektra's lookup. For more than 100 environment variables, EnvElektra's lookup outperforms `getenv()`.

Regarding the overall overhead, we first report about the diversity of the applications. For the startup of gimp the overhead of 2.6% is negligible. For the startup of firefox, however, the overhead is 6.5%. The reason is that Firefox performs `exec()` 5 times during startup. Then EnvElektra needs to be initialized and needs to parse its configuration files again. For very small applications, e.g. `curl` and `wget`, the parsing strongly affects the runtime overhead. If they download empty files, the overhead even dominates. The overhead between different applications varies greatly.

Next, we were interested in the impact on a system which executes many processes each with trivial tasks. An extreme example happens to be the compilation of C software projects with gcc. Because gcc spawns 5 subprocesses for the compilation of every `.c` file, the overhead seems to get immense. Actually, the overhead of a trivial program's compilation, only containing `int main(){}`, is 90%. The parsing of configuration files gets dominant. It is astonishing that the overhead of a compilation for a full project is only 14%. For this benchmark we compiled EnvElektra from scratch. The absolute times are 2:23 min total when compiling with EnvElektra and 2:05 min total without EnvElektra as measured with the `time` utility. The compilation executed 6847 processes, did 30862 `getenv` calls, 6199 of which contained `CC`. Even though trivial process executions have large overhead, the overall performance only suffers little, even in extreme cases.

We further were very interested in any other occurrence with a similar number of many process executions. The booting of Debian executes 732 processes. The most often requested environment variable was `SANE_DEBUG_SANEI_SCSI` with 286 occurrences. In the script `startkde`, 227 binaries are executed. The executed number of processes in the case of compilation actually seems to represent an exception. We conclude that occurrences where processes are spawned excessively are rare.

Finally, we want to discuss the overhead of the reload feature. We chose the following setup: We installed the webserver lighttpd locally. EnvElektra was active throughout the whole experiment. To download 10 files with 1MB to 10MB size each we executed `curl -o "#1 http://localhost/test/[1-10]"`. Without reloading this execution resulted in 83,786,947 instructions. With reloading EnvElektra every millisecond, valgrind counted 91,569,790 instructions. The reloading caused the configuration to be fetched 91 times instead of 4 times. Because of an optimization within EnvElektra only `stat` is used on the configuration files without parsing them again. Thus, the overhead is only 9.3%.

Different to the benchmark setup above we will now change the database once during program execution. Then EnvElektra will reread the respective configuration file. We have to take care that the changed value does not influence the control flow. For example, if we add the `no_proxy` variable, proxy setup is skipped and the performance even increases. Thus, we changed `COLUMNS`, which is requested for every download but does not influence the overhead more than unrelated parameters. When changing it during one of the ten requests the execution needed 95,248,722 instructions. We see that actual context changes have acceptable overhead of $\sim 4\%$.

Our findings regarding **RQ3** are:

(1) In applications that terminate very soon, e.g. only showing help text, the run-time overhead dominates. In practical use, however, EnvElektra only adds run-overhead from 2.6% to 14% (in extreme but realistic cases).

(2) Dynamic reload has about 10% overhead. On context changes the overhead increases again by about 4% in a realistic http-proxy-transition.

Implications: EnvElektra's run-time overhead typically is low and thus acceptable. For frequent context changes, optimizations would be preferable.

5 Threats to Validity

As in all quantitative studies our concern is if the evaluated software is representative. In **RQ1** we address it by using a significant number of diverse open-source software in terms of functionality, development teams and programming languages. We did not consider context awareness already present in applications. Although interception also works for closed-source software, we did not study it because of the impossibility to cross-check with source code. Anyhow some of the software, including libreoffice, chromium and eclipse, has at least origins in closed-source development. Thus, the results can be valid for closed-source software, too. While we think that the software we inspected represents some characteristics of variability APIs, more general conclusions need further work.

In the methodology of **RQ2**, we need to interpret whether contextual awareness can be exploited. We avoid subjective judgements about context awareness during program start. One could also modify the environment with a wrapper script to achieve similar results. We prefer to examine dynamic context changes which are impossible with former approaches. To improve reproducibility and objectivity we only consider visible changes in the user interface.

We exclusively measure calls of `getenv` but do not consider the use of the `environ` pointer, the third parameter of `main`, and `/proc`. We cannot guarantee full coverage. Therefore our evaluation actually underestimates the full potential.

We added optional logging to count the number of `getenv`. Logging, however, influences a system deeply. On one system two start-processes failed when logging was activated. We did not find other occurrences that caused differences in behavior. Thus, we always rerun our tests without logging.

The benchmarks are conducted comparatively and consider only a single implementation of `getenv`. Therefore run-time measurements may not apply for other versions or OSs. Additionally, the benchmarks yield very different results depending on the size of the used configuration files and the respective parser. To level out this problem, we took care that our setup is realistic. We used 8 different configuration files and especially chose parsers which are known to be slow. We think that it is straight-forward to reproduce our benchmarks in a way that they perform even better than the numbers we reported.

Overall, while we cannot draw general conclusions for context-aware configuration access in the `getenv` API, we think that our study unveils some important insights, particularly for open source software.

6 Related Work

Riva et al. [23] acquired software-engineering-related knowledge from studying context-aware software. Different from our approach, they reverse-architected existing context-aware support systems. We preferred to study the behaviour of well-known software when introducing context awareness.

Context-aware middleware [8, 9] is a well-established research direction. Env-Elektra could be seen as local context-aware middleware for configuration. Env-Elektra scores in situations where legacy software needs to be deployed.

Using the correct context is a subtopic of avoiding configuration errors. Yin et al. [28] researched different types of configuration-parameter-related mistakes. They investigated value-environment mistakes which can be caused by wrong contextual interpretation. Which errors actually are induced by incorrect contextual interpretation, however, is still an open question.

A lot of work exists about how to extract program configuration constraints from source code [16, 22]. The authors argue that even though many constraints are extracted, sometimes additional external knowledge is needed. We think that context awareness is such a constraint.

Context-oriented programming (COP) already has an important role within software-engineering [1, 12, 24]. COP mainly aims at more comprehensible programs expressing more context awareness. Our approach tackles the problem in a different direction: We add context awareness without changing the program.

Previous work [19] describes context-awareness by using explicit layer activations. Other than our approach, these methods cannot be used for already existing applications.

Niu et al. [17] report on a web-based framework which uses indoor location, which is an important context sensor. Software product line engineering [2, 25] deals with the question how to construct products by combining features. Configuration specification languages [10, 11] rarely have support for context. An exception is the context oriented component model PCOM [15]. Unlike our approach, these approaches cannot be used for already existing applications.

Yuan et al. [29] provided a quantitative characteristic study for software logging. Similar to our study they revealed that their object of study is used in four large open-source applications pervasively. Different to our approach, they researched how logging statements were introduced and changed, while we show how APIs for variability are intercepted for more context awareness.

7 Conclusion

In this paper, we described a context-aware database using configuration files. A `getenv` implementation uses it for context-aware configuration access. Applications facilitating this API profit from context awareness. Our approach is unique because it allows applications to be context-aware without any modifications.

We saw that `getenv()` in most software provides excessive variability which is currently underutilized. This variability benefits from context awareness. The paper gives ideas for programmers how `getenv()` can be used with more efficacy. Sometimes software is even capable to dynamically adapt to context changes even though the authors did not anticipate this use. In a benchmark we found out that while in

small synthetic benchmarks the overhead might be devastating, in practice it stays well with reasonable bounds.

Our results are:

- Presentation of an approach in which applications are more aware of their context
- A novel context-aware `getenv()` implementation downloadable from `http://www.libelektra.org`.
- Providing experimental validation by a case study of significant complexity.

References

1. Baldauf, M., Dustdar, S., Rosenberg, F.: A survey on context-aware systems. International Journal of Ad Hoc and Ubiquitous Computing **2**(4), 263–277 (2007)
2. Berger, T., Lettner, D., Rubin, J., Grünbacher, P., Silva, A., Becker, M., Chechik, M., Czarnecki, K.: What is a feature?: a qualitative study of features in industrial software product lines. In: Proceedings of the 19th International Conference on Software Product Line, pp. 16–25. ACM (2015)
3. Bockisch, C., Kanthak, S., Haupt, M., Arnold, M., Mezini, M.: Efficient control flow quantification. In: ACM SIGPLAN Notices, vol. 41, pp. 125–138. ACM (2006)
4. Costanza, P., Hirschfeld, R., De Meuter, W.: Efficient layer activation for switching context-dependent behavior. In: D. Lightfoot, C. Szyperski (eds.) Modular Programming Languages, *Lecture Notes in Computer Science*, vol. 4228, pp. 84–103. Springer (2006). URL `http://dx.doi.org/10.1007/11860990_7`
5. Dey, A.K., Abowd, G.D.: The what, who, where, when, why and how of context-awareness. In: CHI '00 Extended Abstracts on Human Factors in Computing Systems, CHI EA '00. ACM, NY (2000). URL `ftp://ftp.cc.gatech.edu/pub/gvu/tr/1999/99-22.pdf`
6. Easterbrook, S., Singer, J., Storey, M.A., Damian, D.: Selecting empirical methods for software engineering research. In: F. Shull, J. Singer, D. Sjøberg (eds.) Guide to Advanced Empirical Software Engineering, pp. 285–311. Springer (2008). URL `http://dx.doi.org/10.1007/978-1-84800-044-5_11`
7. Eisenhardt, K.M., Graebner, M.E.: Theory building from cases: opportunities and challenges. Academy of management journal **50**(1), 25–32 (2007)
8. Geihs, K., Barone, P., Eliassen, F., Floch, J., Fricke, R., Gjorven, E., Hallsteinsen, S., Horn, G., Khan, M.U., Mamelli, A., Papadopoulos, G.A., Paspallis, N., Reichle, R., Stav, E.: A comprehensive solution for application-level adaptation. Software: Practice and Experience **39**(4), 385–422 (2009). URL `http://dx.doi.org/10.1002/spe.900`
9. Gu, T., Pung, H.K., Zhang, D.Q.: A middleware for building context-aware mobile services. In: Vehicular Technology Conference, 2004. VTC 2004-Spring. 2004 IEEE 59th, vol. 5, pp. 2656–2660. IEEE (2004)
10. Günther, S., Cleenewerck, T., Jonckers, V.: Software variability: the design space of configuration languages. In: Proceedings of the 6th Workshop on Variability Modeling of Software-Intensive Systems, pp. 157–164. ACM (2012)
11. Hewson, J.A., Anderson, P., Gordon, A.D.: A declarative approach to automated configuration. In: LISA, vol. 12, pp. 51–66 (2012)

12. Jong-yi, H., Eui-ho, S., Sung-Jin, K.: Context-aware systems: A literature review and classification. Expert Systems with Applications **36**(4), 8509 – 8522 (2009). URL `http://dx.doi.org/10.1016/j.eswa.2008.10.071`
13. Kamina, T., Aotani, T., Masuhara, H., Tamai, T.: Context-oriented software engineering: A modularity vision. In: Proceedings of the 13th International Conference on Modularity, MODULARITY '14, pp. 85–98. ACM, New York, NY, USA (2014)
14. von Löwis, M., Denker, M., Nierstrasz, O.: Context-oriented programming: Beyond layers. In: Proceedings of the 2007 International Conference on Dynamic Languages, ICDL '07, pp. 143–156. ACM, NY, USA (2007). URL `http://dx.doi.org/10.1145/1352678.1352688`
15. Magableh, B., Barrett, S.: Primitive component architecture description language. In: Informatics and Systems (INFOS), 2010 The 7th International Conference on, pp. 1–7 (2010)
16. Nadi, S., Berger, T., Kästner, C., Czarnecki, K.: Mining configuration constraints: Static analyses and empirical results. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp. 140–151. ACM, New York, NY, USA (2014). DOI 10.1145/2568225.2568283. URL `http://dx.doi.org/10.1145/2568225.2568283`
17. Niu, L., Saiki, S., Matsumoto, S., Nakamura, M.: Wif4inl: Web-based integration framework for indoor location. International Journal of Pervasive Computing and Communications (2016)
18. Raab, M.: A modular approach to configuration storage. Master's thesis, Vienna University of Technology (2010)
19. Raab, M.: Global and thread-local activation of contextual program execution environments. In: Proceedings of the IEEE 18th International Symposium on Real-Time Distributed Computing Workshops (ISORCW/SEUS), pp. 34–41 (2015). DOI 10.1109/ISORCW.2015.52
20. Raab, M.: Sharing software configuration via specified links and transformation rules. In: Technical Report from KPS 2015, vol. 18. Vienna University of Technology, Complang Group (2015)
21. Raab, M., Puntigam, F.: Program execution environments as contextual values. In: Proceedings of 6th International Workshop on Context-Oriented Programming, pp. 8:1–8:6. ACM, NY, USA (2014). URL `http://dx.doi.org/10.1145/2637066.2637074`
22. Rabkin, A., Katz, R.: Static extraction of program configuration options. In: Software Engineering (ICSE), 2011 33rd International Conference on, pp. 131–140. IEEE (2011)
23. Riva, O., di Flora, C., Russo, S., Raatikainen, K.: Unearthing design patterns to support context-awareness. In: Pervasive Computing and Communications Workshops, 2006. PerCom Workshops 2006. Fourth Annual IEEE International Conference on, pp. 5 pp.–387 (2006). URL `http://dx.doi.org/10.1109/PERCOMW.2006.138`
24. Salvaneschi, G., Ghezzi, C., Pradella, M.: Context-oriented programming: A software engineering perspective. Journal of Systems and Software **85**(8), 1801 – 1817 (2012). URL `http://dx.doi.org/10.1016/j.jss.2012.03.024`
25. Schaefer, I., Hähnle, R.: Formal methods in software product line engineering. IEEE Computer **44**(2), 82–85 (2011)
26. Schippers, H., Molderez, T., Janssens, D.: A graph-based operational semantics for context-oriented programming. In: Proceedings of the 2Nd International Workshop on Context-Oriented Programming, COP '10. ACM, NY, USA (2010). DOI 10.1145/1930021.1930027. URL `http://dx.doi.org/10.1145/1930021.1930027`
27. Tanter, E.: Contextual values. In: Proceedings of the 2008 Symposium on Dynamic Languages, DLS '08, pp. 3:1–3:10. ACM, NY, USA (2008). DOI 10.1145/1408681.1408684. URL `http://dx.doi.org/10.1145/1408681.1408684`
28. Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L.N., Pasupathy, S.: An empirical study on configuration errors in commercial and open source systems. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, pp. 159–172. ACM, New York, NY, USA (2011). DOI 10.1145/2043556.2043572
29. Yuan, D., Park, S., Zhou, Y.: Characterizing logging practices in open-source software. In: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pp. 102–112. IEEE Press, Piscataway, NJ, USA (2012). URL `http://dl.acm.org/citation.cfm?id=2337223.2337236`