Introducing Context Awareness in Unmodified, Context-unaware Software

Markus Raab, Gergö Barany

Introducing Context Awareness in Unmodified, Context-unaware Software

Markus Raab[1] and Gergö Barany[2*]
[1]*Institute of Computer Languages, Vienna University of Technology, Vienna, Austria*
[2]*Inria Paris, France*
markus.raab@complang.tuwien.ac.at, gergo.barany@inria.fr

Keywords: Context-aware Software Engineering, Configuration Specification

Abstract: Software tends to be highly configurable, but most applications are hardly context aware. For example, a web browser provides many settings to configure printers and proxies, but nevertheless it is unable to dynamically adapt to a new workplace. In this paper we aim to empirically demonstrate that by dynamic and automatic reconfiguration of unmodified software we can systematically introduce context awareness. In 16 real-world applications comprising 50 million lines of code we empirically investigate which of the 2,683 run-time configuration accesses (1) already take context into account, or (2) can be manipulated at run-time to do so. The results show that context awareness can be exploited far beyond the developers' initial intentions. Our tool Elektra dynamically intercepts the run-time configuration accesses and replaces them with a context aware implementation. Users only need to specify contexts and add context sensors to make use of this potential.

1 Introduction

Context—information about the environment in which software executes—strongly influences the behavior we expect from software, and most software is subject to context. As our running example, we describe a web browser with its local network settings as context: In different networks, web browsers may require different proxy settings for Internet access. The default printer might also have to be changed to a physically co-located one.

If software (more) readily adapts its behavior automatically to its current context, we call it (more) *context aware* (Alegre et al., 2016). Context awareness fundamentally increases user experience (Dey and Abowd, 2000). For example, if a web browser considers its network context, users will be able to display a web page regardless of which proxy is required by the network.

Context-oriented software engineering (COSE) puts context awareness in its focus. Previous COSE approaches required developers to consider every context already at design time (Kamina et al., 2014). Thus they were not applicable for already existing large software projects and in particular for legacy software.

To improve on these issues, we propose to move COSE to deployment time. This way we delay decisions about supported contexts. Three classes of stakeholders participate: (1) the developers, who still focus on configurability without the need to explicitly implement context awareness; (2) the administrators, who enable context-awareness in applications with our novel COSE process during deployment; and (3) the end users, who enjoy more context-aware applications.

1.1 Research Questions

We claim that it is possible and practical to take unmodified software, and by run-time reconfiguration, improve their context awareness. Our goal is to exploit already existing *run-time configuration accesses* (RCAs) in free and open source software (FLOSS).

To validate our claim, we answer 3 research questions:

First we need to show that enough RCAs are present in FLOSS to support run-time reconfiguration. To confirm that, in Section 4 we analyze the source code in a sample of 16 popular and large-scale FLOSS applications and count RCAs to answer **RQ 1:** How often are RCAs used in FLOSS?

To find out whether RCAs occur sufficiently frequently during run time, a dynamic analysis is needed.

*This work was performed while the author was at CEA LIST Software Reliability Laboratory, France, and supported by the French National Research Agency (ANR), project AnaStaSec, ANR-14-CE28-0014.

In Section 5 we evaluate case studies with the same 16 applications and answer **RQ 2:** How many RCAs can be made context aware?

In Section 6 we investigate if our proposed solution is efficient enough by answering **RQ 3:** What is the overhead of context-aware RCAs?

1.2 Contributions

This paper is (to the best of our knowledge) the first endeavor to empirically investigate context awareness in large-scale FLOSS applications:

- We collected profound evidence that RCAs in FLOSS applications can be used to improve context awareness.
- In case studies with 16 real-world applications we found out that COSE improves unmodified FLOSS applications.
- No previous evaluation of context-aware applications was conducted using such large, complex, and popular applications.

Contribution 1: In a source-code analysis of 16 FLOSS applications we observe that a particular kind of RCAs, namely invocation of the `getenv` function, is used pervasively (2,683 call sites). (Section 4)
Contribution 2: We confirm that RCAs are used ubiquitously also at run time. We systematically investigated which RCAs can be used to improve context awareness in all 16 applications. We improved context awareness in nearly every studied application and found promising candidates in the others. For example, from 316 candidates in browsers, in total we found 40 RCAs that certainly enable context awareness. In some cases applications were made completely aware of individual contexts. Furthermore, we could integrate all 1,957 configurations settings of Firefox, which provided seamless adaption to workplaces. We never needed to modify the source code. (Section 5)
Contribution 3: By evaluating performance characteristics of browsers in realistic proxy transitions, we found that in minimalist applications there is significant overhead. For feature-rich applications such as Firefox, however, the overhead of our tool is below 1 %. (Section 6)

2 Preliminaries & Motivation

An important aspect of software configuration is to specify *run-time configuration accesses* (RCAs). RCAs are the places within code that define different behavior based on configuration. Usually RCAs are calls to configuration *application programming interfaces* (APIs) such as `getenv` but can also be direct accesses to data structures.

The `getenv` function is a low-level configuration RCA. It accesses the *environment variables* of the current process, which is set by the caller (e. g., the user's shell) when a program is started. After the start of the process, its environment variables can no longer be changed externally, only from within the process using the `setenv` function. We chose it for most of our investigations because it is widely standardized and available in many programming languages.
For example, in web browsers we find code such as:

```
getenv ("http_proxy");
```

In this example, the return value of `getenv` can contain an outdated proxy after network changes because environment variables are not updated for processes.

Context-oriented programming (COP) allows developers to naturally separate multi-dimensional concerns (Dey and Abowd, 2000; Salvaneschi et al., 2012; Schippers et al., 2010). COP is one way to specify programs that adapt their behavior to the context.

Layers are the foundation of COP (Appeltauer et al., 2009; Costanza et al., 2006; von Löwis et al., 2007; Wasty et al., 2010). Every layer constitutes one dimension of context that cuts across the software system. The (de)activation of layers occurs during program execution. All active layers together define the current context of the program.

Contextual values originate from Lisp systems (Asirelli et al., 1979). Tanter revived contextual values as a lightweight subset of COP. They "boil down to a trivial generalization of the idea of thread-local values" (Tanter, 2008) and can be described as variables whose values depend on the current context. Contextual values naturally work along with the concept of layers.

In the present paper we will interpret access to configuration settings as contextual values. Every RCA, such as a `getenv` invocation, will be considered as reading a contextual value.

Context awareness is a property of a program and defines the degree of context taken into account. For every possible context a combination of layers considers necessary adaptations. Organizing such dynamic behavioral changes requires careful engineering (Salvaneschi et al., 2012). Alternatively, contextual values are by design always context aware (Raab, 2016c).

Context-oriented software engineering (COSE) provides a "methodology that guides us to a specification of context-dependent requirements" and a systematic mapping from context-dependent use cases to layers (Kamina et al., 2014).

Context sensors (Dey and Abowd, 2000; Baldauf et al., 2007) are hardware and software with the main purpose of activating layers according to context changes. We will use them as separate processes that wait for context change events (Raab, 2016c).

Yin et al. (Yin et al., 2011) found out that in "a large portion (46.3 % to 61.9 %) of the parameter misconfigurations" the context was not considered. Our goal is to explicitly specify the contextual values that make the execution of the applications more context-aware. This specification leads to a better understanding of the context in teams maintaining the software. We are positive that the specification helps reduce external misconfiguration errors as a side effect.

For newly written context-aware software, current COP and COSE approaches would be a viable choice (Jong-yi et al., 2009). For large FLOSS projects, however, rewriting the whole source code is not feasible.

3 Elektra

Elektra is a library developed by one of the authors, which implements uniform, consistent and context-aware configuration access. In the present paper we describe an approach to use Elektra as a tool to integrate unmodified applications. The approach is to apply Elektra in COSE processes at deployment.

Elektra (Raab, 2016a) works as follows: At application start, Elektra initializes itself by parsing configuration files. The configuration files contain both the specifications and configurations for contextual values. Elektra supports over 190 configuration file formats including the widely-used INI, XML and JSON formats. The support for these formats enables Elektra to directly manipulate configuration of applications.

In its essence, Elektra provides a key-value database with unique keys and a specification for every configuration setting.

3.1 Interception

To work with unmodified applications, Elektra intercepts important library calls, including the following:

- Each `getenv` invocation to provide context-aware access for environment variables.
- Each `open` invocation to return configuration files with configuration settings respecting the current context.

Interceptions of library calls are platform-dependent but are available for every major OS, e. g., `LD_PRELOAD` and `/etc/ld.so.preload` for Linux. Instead of requiring developers to implement new behavior for context adoption, we rely on already existing behavioral adoptions that are guarded by RCAs.

3.2 Context Specification

Elektra itself is configured via a configuration specification language. In this paper we will use a simple key-value syntax to illustrate the specification of the key-value database and its contents. For example, let us specify the contextual value `getenv/http_proxy`:

```
[getenv/http_proxy]
context=http_proxy/%interface%/%network%
```

The key within `[]` represents a unique identifier to a configuration setting. Entries in the database are organized hierarchically with `/` as the level separator. The `getenv`-interceptor reads its configuration from keys starting with `getenv/`. We configure it to handle `getenv` invocations with the parameter `"http_proxy"`.

In the example above, the only property for this key, i. e., `context`, specifies that the value to be returned from such invocations should be context aware. Using the `%...%` syntax we specify placeholders to be substituted by the values of layers. The `getenv` invocation returns the proxy configured for the currently active *`interface`* and *`network`* layers. This functionality allows us to modify configuration settings passed to applications: An application that requests a configuration setting from the environment transparently receives a setting from our key-value database instead. By honoring context in the lookup we introduce context awareness in the client software.

The context-aware lookup makes sure that the returned value recursively respects context specifications. With changing context, i. e., different values in layers, the same requested key has different values. We express the possible values using straight-forward pattern matching. E. g., the placeholder `*` will match any layer that was not matched specifically by name:

```
http_proxy/wlan/home= proxy.example.org
http_proxy/eth/work = proxy.example.com
http_proxy/*/* = default.example.com
```

Personalization is an important aspect of context-aware systems (Alegre et al., 2016). In Elektra we personalize applications by changing such configuration values for every individual context.

3.3 Context Changes and Sensors

When the context changes, this information must be communicated to applications. For the present work,

we use external *context sensors*: small programs running in separate processes that monitor the context of interest. When the sensor detects a change, it updates the corresponding layer's value in the key-value database. Future requests for contextual values via Elektra (through intercepted `getenv` or `open` invocations) will use these updated layer values in their lookups. Having context sensors running in their own process separates concerns between the application and the code detecting context changes.

In the running example, users switch networks by changing their location or by connecting a network cable. A sensor detects this change and updates Elektra's database accordingly. We used hooks in network interfaces to implement this use case. Assume that the ***interface*** changes to `eth` and the ***network*** to `work`. Then the next `getenv("http_proxy")` invocation will return `proxy.example.com`. This is an increase in context awareness: Normally `getenv` is *not* context-aware because the program's environment is initialized at startup and cannot be modified externally, only by the program itself using `setenv`. Thus the standard `getenv` function always returns the same value for a given argument. In contrast, Elektra's modified `getenv` returns different values if the underlying context changes.

Elektra is not limited to pulling configuration settings while RCAs are executed. Instead Elektra can push information to applications by notifying them to reload their configuration, e. g., via signals for daemons or socket communication for Firefox.

4 RQ1: Use of `getenv`

In this section we collect empirical evidence of `getenv` invocations in the source code of applications.

4.1 Methodology

We count the total lines of code and occurrences of `getenv` in selected applications. Obvious wrapper functions (e. g., `LYGetEnv` in Lynx) are treated identically to `getenv` itself.

To improve external validity we carefully sampled 16 applications. We started by including large applications that have a thriving community. In addition we took care to have a broad range of diverse applications. We searched for further popular applications to reduce the familiarity heuristic. If Internet pages repeatedly mentioned some applications, we considered them for inclusion. Finally, we set a focus on browsers to have a better picture for a specific domain.

The evaluation of the paper consistently uses the same applications with the same versions. We analyzed applications in the version as included in Debian Jessie 8 amd64 available at `snapshot.debian.net`. Considering these factors we compiled the following list of 16 applications and versions:

application	version	**application**	version
0ad	0.0.17	Gimp	2.8.14
Akonadi	1.13.0	Inkscape	0.48.5
Chromium	45.0.2454	Ipe	7.1.4
Curl	7.38.0	Libreoffice	4.3.3
Eclipse	3.8.1	Lynx	2.8.9dev1
Evolution	3.12.9	Man	2.7.0.2
Firefox	38.3.0esr	Smplayer	14.9.0 ds0
Gcc	4.9.2	Wget	1.16

We used Cloc 1.60 to determine the code size of the applications in the versions as listed above. We used `grep -rio` to find all textual `getenv` occurrences. Finally we manually looked at every `getenv` occurrence to check whether it is an invocation or something else like text in a comment.

4.2 Results

In the following table below the column *1k lines of code* shows the code size of the applications, expressed as multiples of 1,000 lines of code. For the column *counted getenv* we manually counted `getenv` invocations.

application	**1k lines of code**	**counted getenv**	**lines per getenv**
0ad	474	55	8,617
Akonadi	37	13	2,863
Chromium	18,032	770	23,418
Curl	249	53	4,705
Eclipse	3,312	40	82,793
Evolution	673	23	29,252
Gcc	6,851	377	18,172
Firefox	12,395	788	15,730
Gimp	902	56	16,102
Inkscape	480	19	25,255
Ipe	116	21	5,529
Libreoffice	5,482	284	19,304
Lynx	192	89	2,157
Man	142	62	2,293
Smplayer	76	1	76,170
Wget	143	32	4,456
Total	49,556	2,683	18,470
Median	477	54	

The applications we analyzed have 2,683 `getenv` invocations in 50 million lines of codes. We excluded textual occurrences in wrappers, comments, ChangeLogs or similar.

Finding 1: We demonstrate that `getenv` is used pervasively by finding 2,683 invocations. This is one `getenv` occurrence in 18,470 lines of code.

5 RQ2: Run-Time Behavior

In this section we validate the applicability of our approach at run-time. Run-time analysis considers `getenv` invocations by all participating libraries, complementing our source-code analysis. We will investigate how often changed return values of `getenv` invocations actually modify the application's behavior to improve context awareness.

This study is a partial replication of our previous study (Raab, 2016d). Unlike the previous study, we compare the context awareness of every single parameter. Furthermore, we added more applications and introduce `open` interception. For brevity, however, we only report about selected and representative cases.

5.1 Methodology

We applied our approach for all 16 applications. As described in Section 3, we use the library preload mechanism to use Elektra's implementations of `getenv` and `open` instead of the ones in the standard library.

First we started the 16 applications and clicked through the user interface. While doing so, we logged every `getenv` invocation and its parameters. To check if the `getenv` invocation is used as if standard `getenv` were context aware, we modified the return values of `getenv` while the application was running. Then we repeated the user-interaction to see if the `getenv` invocation influences the behavior.

5.2 Results

application	getenv all	all uniq	later uniq	later config	context aware
Chromium	2,723	1,056	73	≥ 24	≥ 1
Curl	87	14	9	6	6
Firefox	8,185	273	210	118	≥ 15
Lynx	1,428	45	23	19	16
Wget	13	7	1	1	1

The table above shows the number of `getenv` invocations on a freshly installed Debian system. The number of invocations varies widely from system to system depending on configuration and installed software. E. g., on other machines, we observed up to four times more unique `getenv` invocations during run-time for Firefox. Sometimes we found settings that are likely to be context aware (indicated by $\geq$) but lacked the resources to investigate them in detail.

In the column *getenv all* we see how many times the browsers called `getenv` in total. The next column shows the number of `getenv` invocations with unique parameters. The column *later uniq* only considers `getenv` invocations with unique parameters and only after startup. The next column are candidates for context awareness: they are additionally related to configuration. The last column shows which of the candidates actually successfully influenced behavior at run time without reloading.

In the analysis we found many `getenv` invocations after startup. Loops implementing user interactions often repeatedly call functions that redo the same `getenv` invocations.

5.3 Case Study: Firefox

In a case study we conducted the complete COSE process. We selected Firefox and specified `http_proxy` and `PRINTER_LIST` as configuration options of interest as shown in Section 3. We implemented the layer-changes with one-line hooks in the `/etc/NetworkManager` scripts.

Then we needed to specify printers/proxy for every context line-by-line. Within a day, Firefox fully-automatically selected nearby printers and proxies immediately on network changes (available printers are even modified while the printer dialog is open).

Elektra also allows us to modify options in configuration files. We needed 9 hours to configure Firefox to enable rereading its configuration files. In 2 more hours we implemented an Elektra plugin for Firefox's configuration files. With `open` interception we have a context-aware mechanism for all of 1,957 configuration options available in Firefox's configuration files (Jin et al., 2014).

Finding 2: In each of the 16 application user interactions caused `getenv` invocations, often useful to make features flawlessly context aware.

We successfully used Elektra in a real-world case study with Firefox. To enable our implementation of retrofitting context-awareness for flexible workplaces, only three actions were required: (1) specify contextual values, (2) create context mapping for every workplace, and (3) add context sensors to switch layers.

Implication: Our tool can be practically applied in real-world case studies with small effort.

6 RQ3: Performance Evaluation

To evaluate the performance we profiled different browsers during a proxy transition on a hp® EliteBook 8570w. In the experiment we opened a web page, then changed the context, and finally opened a different

web page. For the proxy transition Elektra performs the following steps for every process: First the process needs to parse the context specification. Then the actual `getenv` is replaced with our context-aware implementation. On layer transitions, the configuration file needs to be reread.

We measured the number of executed CPU instructions with Valgrind's tool Callgrind. We report inclusive costs, i. e., the cost of the `getenv` invocation including every callee. Thus Valgrind simulates a CPU, the results are deterministic.

Results: In the first benchmark we will use the Lynx browser. It is written in a lean way and has negligible startup times. Such an efficient implementation allows more precise exploration of the impact the context switches have.

First we started Lynx without Elektra and visited two links. Valgrind counted 92,888,073 instructions (median of three invocations). Then when we activated Elektra and changed the proxy before visiting the second link, we counted 114,049,336 instructions, which are about 18.5 % more instructions. We used the context specification with the two layers ***network*** and ***interface***.

Without context-aware `getenv`, the `getenv` invocations needed 0.33 % of all instructions. If we modify `getenv/http_proxy` directly (without using layers) `getenv` needs 24.51 %. If using the setup with the two layers, `getenv` invocations needed 25.27 %.

With Firefox the comparison was more difficult because it consumes resources even without user interaction. The startup times with Valgrind are nearly two minutes. We estimated the overhead by looking at the profile data, similar to the `getenv` overhead in Lynx. Overall 20,362,848,539 instructions were needed to display two web pages. Internal Elektra overhead, by summing up all costs from the Elektra library, are 68,750,481, i. e., 0.39 %. The `g_getenv` function (a wrapper for `getenv` used within Firefox) needs 16,614,089 instructions (i. e., 0.08 %) instead of 22,703 instructions (i. e., 0.00 %) without Elektra.

> **Finding 3:** In minimalist applications such as Lynx our approach can cause some overhead. The number of participating layers caused only minimal differences. For feature-rich applications the overhead is below 1 %.

7 Related Work

Xu et al. investigated which configuration settings are actually used in practice (Xu et al., 2015). They argue that users get confused by too many settings. We fully agree and think that our approach helps here by automatically deducing most settings from context. Then developers can remove these settings from user guides. Advanced users, however, still can override context-aware configuration settings.

Jin et al. describe different challenges in configuring real-world systems (Jin et al., 2014). Our approach addresses them by (1) working across language barriers, (2) having a holistic integration of different RCAs, and (3) making sure that RCAs do not return outdated values. The authors uncovered 1,957 settings for Firefox and assumed that only a "small part" of settings is missing. But our study shows, that even `getenv` alone adds a large amount of otherwise unconsidered configuration settings.

Most other approaches require modifications in the source code. Tanter et al. (Tanter et al., 2006) propose to make aspects context aware. COP (Salvaneschi et al., 2012; Appeltauer et al., 2009; von Löwis et al., 2007; Baldauf et al., 2007; Jong-yi et al., 2009; Raab, 2015a) improves the modularity in programs. In earlier work we investigated how contextual values are synthesized with code generation (Raab, 2015a; Raab and Puntigam, 2014; Raab, 2015b). A survey discusses many different approaches how to implement context-aware applications (Alegre et al., 2016). Mens et al. created a taxonomy for context-aware variability approaches (Mens et al., 2016). All these approaches require at least some context-specific design upfront the implementation. The specification language Elektra does not only improve context awareness, but fosters system integration (Raab, 2016b). Some approaches focus on deployment, but require decisions at design time (Lee et al., 2014). Alexandrov et al. facilitates intercepting of library calls to improve user experience, but with a different goal (Alexandrov et al., 1998).

The survey of Xu and Zhou gives an overview of the different approaches for improving on configuration problems (Xu and Zhou, 2015). In contrast, our basic idea is to automatically derive correct configuration settings from context.

8 Threats to Validity

Internal: Both the code and run-time analysis have the danger of subjective classification and oversight. To minimize such errors we included second opinions and only report large differences. Additionally the combination of code and run-time analyses yields a more complete picture as proposed for mixed methods (Ihantola and Kihn, 2011).

External: An important concern is whether the evaluated applications and their developers are representative. We address it by studying a high number of diverse applications. We included both small and large applications. We took care that different domains, development teams and programming languages are represented. In particular the browsers are used heavily in mobile contexts.

We have to acknowledge that most software we evaluated is written in C/C++. Nevertheless, Java, JavaScript and Python were well represented with 4.3, 3.3, and 1.1 million lines of code, respectively. Furthermore, we added Eclipse to also have a large project mainly implemented in Java. Since we found no context-aware application of reasonable size with an active community, we could not include such applications into the analysis. Hence our claims exclude applications developed with context-awareness as goal.

An equally important concern is whether `getenv`, our main subject of study, represents every form of RCA. Based on many previous studies (Jin et al., 2014; Rabkin and Katz, 2011; Xu et al., 2013), run-time RCAs are in their essence simple key-value accesses. Higher-level RCAs, e. g. with type-safety, would only complicate the implementation.

Because we did an in-depth source code analysis, we could not pick closed-source applications. A significant portion of the evaluated software, however, has at least roots as closed-source applications. Also based on experience within companies, we are positive that our conclusions hold for closed-source applications.

It is well known that in experimental analysis high standards are required (Johnson, 2002); e. g., to mitigate measurement issues we always used two different profiling tools.

Overall we cannot draw any general conclusions applicable to every form of configuration. In particular we focus on observations how RCAs are used in FLOSS applications. Thus results should be interpreted and generalized with awareness of our focus. Nevertheless our study provides profound insights about connections between configuration and context awareness.

9 Conclusion and Future Work

In this paper we claimed that unmodified applications can become more context aware. We demonstrated that such an approach exists and is practical. We evaluated the approach on 16 large, real-world FLOSS applications. By configuring a simple tool, context awareness was improved in case studies, often even flawlessly. We applied a straightforward context-oriented software engineering process which enables systematic applicability during deployment.

Our work shows that it is realistic to deduce configuration settings from context. We are positive that doing so contributes to reduce one of the major source of configuration errors, i. e., forgetting about context in configuration.

We propose that environment variables should be specified and documented like other configuration settings. Our approach shows that it is not necessary that developers foresee every possible context. Instead layers and configuration settings per context are introduced during deployment. Our approach is modular because context sensors are implemented separately from applications.

Elektra is available as free software from

`http://www.libelektra.org`

and is more general than described in this paper: For example, it can be used for newly developed context-aware software by generating contextual values. This paper focuses on its use for unmodified software.

The source code analysis suggest that dependency injection ('hijacking' existing `getenv`/`open` invocations or other APIs) makes it easy to introduce context awareness. Elektra is not limited to intercepting `getenv` and `open`. For example, we implemented the `gsettings` API which has the potential to make GNOME settings context-aware. As future work Elektra can be extended to make even more forms of configuration context-aware (configuration for modules, plugins, dependency injections, rich mobile APIs etc.).

Although we did not find a single occurrence where existing context awareness conflicted with our approach, combining Elektra with already context-aware software is future work. Another research direction is to investigate how many applications allow reloading of configuration settings. Enabling Elektra to push configuration settings to more applications improves the user experience again.

REFERENCES

Alegre, U., Augusto, J. C., and Clark, T. (2016). Engineering context-aware systems and applications: A survey. *Journal of Systems and Software*, 117:55 – 83.

Alexandrov, A. D., Ibel, M., Schauser, K. E., and Scheiman, C. J. (1998). UFO: A personal global file system based on user-level extensions to the operating system. *ACM Trans. Comput. Syst.*, 16(3):207–233.

Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., and Perscheid, M. (2009). A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, COP '09, New York, NY, USA. ACM.

Asirelli, P., Degano, P., Levi, G., Martèlli, A., Montanari, U., Pacini, G., Sirovich, F., and Turini, F. (1979). A flexible environment for program development based on a symbolic interpreter. ICSE '79, pages 251–263, Piscataway, NJ, USA. IEEE Press.

Baldauf, M., Dustdar, S., and Rosenberg, F. (2007). A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277.

Costanza, P., Hirschfeld, R., and De Meuter, W. (2006). Efficient layer activation for switching context-dependent behavior. In Lightfoot, D. and Szyperski, C., editors, *Modular Programming Languages*, volume 4228 of *LNCS*, pages 84–103. Springer.

Dey, A. K. and Abowd, G. D. (2000). 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, NY. ACM.

Ihantola, E. and Kihn, L. (2011). Threats to validity and reliability in mixed methods accounting research. *Qualitative Research in Accounting & Management*, 8(1):39–58.

Jin, D., Qu, X., Cohen, M. B., and Robinson, B. (2014). Configurations everywhere: Implications for testing and debugging in practice. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 215–224.

Johnson, D. S. (2002). A theoretician's guide to the experimental analysis of algorithms. In *Proceedings of the 5th and 6th DIMACS Implementation Challenges*.

Jong-yi, H., Eui-ho, S., and Sung-Jin, K. (2009). Context-aware systems: A literature review and classification. *Expert Systems with Applications*, 36(4):8509 – 8522.

Kamina, T., Aotani, T., Masuhara, H., and Tamai, T. (2014). Context-oriented software engineering: A modularity vision. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 85–98, New York, NY, USA. ACM.

Lee, K. C. A., Segarra, M.-T., and Guelec, S. (2014). A deployment-oriented development process based on context variability modeling. In *Model-Driven Engineering and Software Development (MODELSWARD), 2nd International Conference*, pages 454–459. IEEE.

Mens, K., Capilla, R., Cardozo, N., Dumas, B., et al. (2016). A taxonomy of context-aware software variability approaches. In *Workshop on Live Adaptation of Software Systems, collocated with Modularity 2016 conference*.

Raab, M. (2015a). 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)*, pages 34–41.

Raab, M. (2015b). Sharing software configuration via specified links and transformation rules. In *Technical Report from KPS 2015*, volume 18. Vienna University of Technology, Complang Group.

Raab, M. (2016a). Elektra: universal framework to access configuration parameters. *The Journal of Open Source Software*, 1(8).

Raab, M. (2016b). Improving system integration using a modular configuration specification language. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 152–157. ACM.

Raab, M. (2016c). Persistent contextual values as interprocess layers. In *Proceedings of the 1st International Workshop on Mobile Development*, Mobile! 2016, pages 9–16. ACM.

Raab, M. (2016d). Unanticipated context awareness for software configuration access using the getenv API. In *Computer and Information Science*, pages 41–57. Springer International Publishing, Cham.

Raab, M. and Puntigam, F. (2014). Program execution environments as contextual values. In *Proceedings of 6th International Workshop on Context-Oriented Programming*, pages 8:1–8:6, New York, NY, USA. ACM.

Rabkin, A. and Katz, R. (2011). Static extraction of program configuration options. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 131–140. IEEE.

Salvaneschi, G., Ghezzi, C., and Pradella, M. (2012). Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801 – 1817.

Schippers, H., Molderez, T., and Janssens, D. (2010). A graph-based operational semantics for context-oriented programming. In *Proceedings of the 2nd International Workshop on Context-Oriented Programming*, COP '10, New York, NY, USA. ACM.

Tanter, E. (2008). Contextual values. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 3:1–3:10, New York, NY, USA. ACM.

Tanter, É., Gybels, K., Denker, M., and Bergel, A. (2006). *Context-Aware Aspects*, pages 227–242. Springer.

von Löwis, M., Denker, M., and Nierstrasz, O. (2007). Context-oriented programming: Beyond layers. In *Proceedings of the 2007 International Conference on Dynamic Languages*, ICDL '07, pages 143–156. ACM.

Wasty, B. H., Semmo, A., Appeltauer, M., Steinert, B., and Hirschfeld, R. (2010). ContextLua: Dynamic behavioral variations in computer games. In *Proceedings of the 2nd International Workshop on Context-Oriented Programming*, COP '10, pages 5:1–5:6. ACM.

Xu, T., Jin, L., Fan, X., Zhou, Y., Pasupathy, S., and Talwadker, R. (2015). Hey, you have given me too many knobs! Understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 307–319. ACM.

Xu, T., Zhang, J., Huang, P., Zheng, J., Sheng, T., Yuan, D., Zhou, Y., and Pasupathy, S. (2013). Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 244–259. ACM.

Xu, T. and Zhou, Y. (2015). Systems approaches to tackling configuration errors: A survey. *ACM Comput. Surv.*, 47(4):70:1–70:41.

Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L. N., and Pasupathy, S. (2011). An empirical study on configuration errors in commercial and open source systems. SOSP '11, pages 159–172.