Persistent Contextual Values as Inter-process Layers

Markus Raab

Institute of Computer Languages
Vienna University of Technology, Austria
markus.raab@complang.tuwien.ac.at

Abstract

Mobile applications today often fail to be context aware when they also need to be customizable and efficient at run-time. Context-oriented programming allows programmers to develop applications that are more context aware. Its central construct, the so-called layer, however, is not customizable. We propose to use novel persistent contextual values for mobile development. Persistent contextual values automatically adapt their value to the context. Furthermore they provide access without overhead. Key-value configuration files contain the specification of contextual values and the persisted contextual values themselves. By modifying the configuration files, the contextual values can easily be customized for every context. From the specification, we generate code to simplify development. Our implementation, called Elektra, permits development in several languages including C++ and Java. In a benchmark we compare layer activations between threads and between applications. In a case study involving a web-server on a mobile embedded device the performance overhead is minimal, even with many context switches.

Categories and Subject Descriptors H.2.3 [*Languages*]: Persistent programming languages

Keywords configuration specification, benchmark

1. Introduction

Context-oriented programming (COP) aims at software modularization, with focus on considering contextual behavior [4]. It enables us to implement context-aware behavior separately. Central to COP are *layers*, i.e. the modules for such code. Active layers encode the current context of an application.

Contextual values (CVs) are variables whose values depend on context, i.e. active layers. CVs fit nicely into COP ideas because of their seamless integration with layers. Side-effects of CVs are limited to their respective context. Their main advantage is their simplicity because CVs "boil down to a trivial generalization of the idea of thread-local values" [16].

Currently many COP languages do not provide straightforward, context-aware activation of layers. Instead they mainly support code-snippets for activation [6], e.g.:

```
activate (Austria) if (gps_pos=="austria")
```

In this example, we activate the layer `Austria` according to the current GPS position. Developers can easily miss some context information at some places, considering that activations are spread across the whole source code.

Another problem COP languages currently face is that layer information cannot easily be shared between applications. Solutions with context-oriented middleware are often not light-weight and need special tools for introspection and debugging. We propose to persist layer information into configuration files. While it is well-known how to persist objects, we lack similar techniques for layers. Because of this issue, it is difficult to separate context sensors and applications. Often the same sensed context, however, is needed in several applications which leads to duplication of efforts.

In this paper we describe the novel idea to use persistent contextual values for layer activation. Our goal is to combine layer activations with CVs that are shared across applications. In our approach we will use CVs as parameters for `activate` and `with` constructs. COP languages commonly use these constructs for layer activation [6].

Our main contribution is a fully implemented framework fulfilling the above goals. It has already successfully been used in embedded systems. It does not make any assumptions of the application's architecture. Our implementation provides (1) generated CVs to be used like variables, and (2) initialization, persistency, and notification for them. It supports many languages, including C++ and Java.

Because CVs are context-aware they will automatically consider their context. By persisting CVs we synchronize layer activations between applications. Our technique even works across different programming languages.

To be better suitable for mobile development one additional goal is that accessing CVs should not add overhead. Instead it should deliver the same performance as reading na-

tive variables. Thus our approach relies on explicit activation and a cache for every CV [14].

We avoid the `if` in the `Austria` layer activation above. Instead we would activate the contextual value `position`:

```
activate (position)
```

This way, due to CV semantics, contexts of `position` are automatically considered. The contextual value would be persistent and shared between applications. When a context sensor has new information available, other applications would be notified via our framework.

For example, think about internationalized software. CVs help us to easily show correct translated messages. Currently, however, we have to activate the correct layers in every COP application individually. There was no easy way for one application to tell all other applications that the language has changed. We want to be able to activate layers with CVs:

```
1 void greet(Person & p, Language & lang) {
2   p.activate(lang)
3   cout << p.greeting << endl;
4 }
```

A code generator yields the classes `Person` and `Language`. They implement CVs semantics. In the Line 2 we activate the contextual value `lang`. A previously persisted value initializes CVs at application startup. This initialization provides sensible defaults changeable by user settings or context.

We answer the following research questions:

RQ1 : How can contextual values be used for layer activation? Which limitations does it have?

RQ2 : What are the costs of inter-process layer activation?

The questions are significant because it is common that we face a number of applications and programming languages in our system. Such a combination, however, currently is not well supported by COP. Our idea is not only feasible but practical as demonstrated by a prototype.

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

2. Background and Syntax

We implemented our approach as tool called Elektra. In this section we will explain the essence and syntax previous versions of Elektra. Earlier versions lacked possibilities to use CVs for activations and needed extra layer specifications.

CVs are very useful to interact with a program execution environment (PEE), i.e. configuration files and environment variables. CVs ensure that the context always is taken into account when accessing the PEE. PEE is an elegant way to persist CVs: we write key-value pairs into configuration files.

For PEE the performance focus lies on retrieving values. Modification of the PEE is usually only done manually by users when they change settings. Thus, when changing PEE, proper validation is more important than performance. In earlier work we demonstrated that PEE can be tightly integrated with CVs [10, 14]. We propose the adoption of PEE as CV storage and maintain the goal to have fast access when reading CVs.

In our approach, a small library abstracts from syntax and location of the configuration files. Elektra uses the PEE, such as configuration files, to initialize contextual values. Elektra supports over 190 configuration file formats, including INI, XML and JSON formats. In this paper we will use a simple key-value syntax to illustrate the content of the key-value database. To easily distinguish configuration and its specification in this paper, we use: (1) assignment with `=` for configurations containing the values of every CV, (2) keys written in `[]` and assignment with `:=` only for specifications:

```
1 path/key=value
2 [path/key]
3 property1:=propvalue1
4 property2:=propvalue2
```

In the first line of the example above we *configure* the option identified with the key `path/key`. As hierarchy separator we use `/`. The key `key` is below `path` and has the value `value`. In lines 3 and 4 we specify two properties for the same key `path/key`: they are called `property`N and have respective values `propvalue`N. The key, value and the properties are stored in the key-value database, for example:

```
1 /*/*/person/greeting=Hi!
2 /german/*/person/greeting=Guten Tag!
3 /german/austria/person/greeting=Servus!
```

In this example, the CV has 3 different possible values with `*` as wildcard expression. Furthermore, within the same configuration files, we can also *specify* contextual values. For example, we specify the CV `greeting`:

```
1 [/%language%/%country%/person/greeting]
2   type:=string
3   description:=hello in all languages
```

The respective key in Line 1, i.e. the full string within `[]`, is a contextual value's name. Lines 2-3 further specify the contextual value with `:=` assignment. Here we specify that the CV `greeting` has the type `string`. In Elektra a code-generator synthesizes context-aware classes using contextual value specifications. The tool generates the code for the underlying CV-classes `Person` and `Greeting`.

Often it is useful to give layers a name [6]. Our approach consistently gives every layer a name, written within `[]`. All generated classes are nested in one hierarchy with `/` as root.

A single CV has many values for each context. In the specification strings enclosed in `%`...`%`, e.g. `%language%`, are placeholders for layer values. For contextual interpretations we substitute the placeholders with values given by layers. Unique keys to lookup individual values are determined by

substituting all placeholders with values from the layers. We use these keys to lookup values in the configuration files.

When no layer was found, the * in the configuration file will match. The character means that the layer is inactive or empty. In the above example, if the only active layer is `language` with the value `german`, an instance of the class `Greeting` has the value `Guten Tag!`.

Developers directly utilize the contextual values in their own code. CVs are used in the same way as variables. As example we extend our previous C++ snippet:

```
1 void visit(Person & p) {
2   p.context().with<CountryAustriaLayer>()
3     .with<LanguageGermanLayer>()([&] {
4     cout << "visit " << ++p.visits
5         << " in "
6         << p.context()["country"]
7         << ": " << p.greeting << endl;
8   });
9 }
```

The `Person`-object `p` is a contextual value passed via reference parameter on Line 1. Dynamically scoped context is specified via the `with` construct in lines 2 and 3. `CountryAustria` and `LanguageGerman` are layers, but not persistent contextual values. Within the dynamic scope of the block after `with` statements, the content of CVs can differ.

For example, when we modify the nested contextual value `visits` in Line 4, the changes are only visible within the `with` block. In Line 8, the previous values are restored.

Our approach has introspection capabilities. We easily can inquiry layer information as done in Line 6. The introspection is useful for debugging and assertions [14].

The issue with the former approach is the implementation of the layers used in lines 2 and 3. The previous approach forced us to implement a layer for every contextual variation. Developers needed to manually implement features such as thread safety, contextual awareness and persistence for every layer. With many layers for highly-dynamic context-aware applications their implementation can be a burden. In this paper we describe how we can avoid this extra effort and exclusively use CVs specified in configuration files.

3. Inter-process Layers

We extend Elektra with the possibility to directly activate CVs. We consider a layer to be active, when a CV is non-empty. As side-effect, this idea enables inter-process activation of layers because of the persistency of CVs. To synchronize layers with CVs we lack two essential features:

- We need an intra-process notification which allows us to update the context of CVs when other CVs, representing layers, change.
- We need an inter-process notification to know when to reread configuration files.

3.1 Contextual Activation

One of the main benefits of activation of CVs is that we automatically get contextual activation. We do not have to worry for every activation if every dependent context is considered. For example, we have the following CVs:

```
1 [/location]
2   type:=Position
3   description:=GPS position
4 [/%location%/country]
5   type:=string
```

And we want to successively activate these CVs:

```
1 void greet (Person & p,Country & country,
2             Location & location) {
3   p.activate(country);
4   p.activate(location);
5   cout << p.greeting << endl;
6 }
```

If `location` and `country` were layers as described in background exactly these layers would be activated in the given order. This means that the activation in Line 3 will not consider the location activated in Line 4.

But because `location` and `country` are CVs, they take context into account and the activations influence each other. In this example after the activation of `location`, `country` will be updated according the new position. Line 4 will also update the `Country` layer according to the established context.

For example, if `location` is an empty string, no layer is activated. With symbolic links in the CVs specification [11] one can implement even more complex scenarios. In Line 5 the greeting will be according to local customs or some default if the CVs `country` and `location` are inactive.

The value of CVs usually is determined by context sensors that run as separate active processes. Their task is to track low-level sensor values such as GPS and pool them to high-level context as needed by other processes. The main advantage of this approach is the reuse of context information and the decoupling between processes.

3.2 Specification

For convenience we decided that by default the last part (separated with `/`) of the key is the layers' name. We already established that the keys are specified within `[]`. This convention gives most CVs an appropriate layer name:

```
1 [/%language%/country]
```

In above example, the CV-class `Country` will automatically have the layer name `country` when activated. In some situations the last part of the name is not the right choice. For example, we use a country code to determine a country:

```
1 [/%language%/country/code]
2   type:=string
3   layer/name:=country
```

Layer names, unlike CV-classes, do not provide a hierarchy. We likely do not want the layer to be named `code`, thus we rename it to `country` as specified by `layer/name`. Then activation of such a CV activates the layer `country`.

3.3 Intra-process Notification

In previous versions of Elektra it was assumed that every change of a CV is caused by the assignment to a CV. In this extension we avoid this assumption. We introduce a reload mechanism when the underlying persistent CVs change.

We implement such a mechanism by an intra-process notification. For this functionality we use the observer pattern. The CVs act as observers, the context is the concrete subject. For in-memory synchronization the method `sync` can be used:

```
1 void doSync(Country const& c) {
2   kdb.get(c.values());
3   c.context().sync();
4   // c and other CV are updated
5   // and different layers are active
6 }
```

In Line 2 we fetch all values for every context from the configuration files. In Line 3 we call the in-memory update of all CVs to reload their value. The `sync` invocation also makes sure that the correct layers are active.

The persistency layer has two further methods for synchronization in both directions: With `kdb.get` modified configuration files are parsed, with `kdb.set` changes are written to the configuration files. This way the application developer decides about the behavior in the case of conflicts. Because a three-way merge is the most-requested behavior, a convenience API exists for this case.

Because we want CVs to act as layers we cannot update the observers in an arbitrary order. Instead we need to consider the dependencies between CVs. CVs with placeholders depend on CVs that have the placeholders name as their layer name. For example, `Country` needs to be updated before `Location` because `Country` has `%location%` as dependency.

We start by updating CVs that do not have dependencies. Then we update CVs that are dependent on layers that were updated before. Elektra solves this ordering problem with a topological sort based on Kahn [5].

Because in our approach hooks can be registered to be executed on layer activation [10], users might need a specific order. Thus users can describe a specific order of activation. For example:

```
1 [/country]
2   layer/order:=0
3 [/language]
4   layer/order:=1
```

In the example, no dependency is given. Thus any order would be correctly topological sorted. But because of the users preference in `layer/order`, `country` is activated before `language`. This way the user can be sure that `country` hooks would be executed before `language` hooks. If the `layer/order` conflicts with topological sort, `layer/order` can only be fulfilled partly and a warning is emitted.

With layers and CVs as completely separate concepts cycles were not possible: CVs depended on layers, but not the other way round. An issue of our approach is that we introduce potentially cyclic dependencies. For example:

```
1 [/%country%/language]
2   type:=string
3 [/%language%/country]
4   type:=string
```

If we activate `country` we would need the value of the current `language` layer and vice-verse. Note that `layer/order` introduced before does not help with this issue. With specific values stored, every activation leads to toggling values:

```
1 /swiss/language=de
2 /luxembourg/language=fr
3 /fr/country=swiss
4 /de/country=luxembourg
```

Such cycles usually stem from design errors and are unwanted. In our approach we prohibit such cycles. We introduce a limitation that causes some previously valid specification files to be rejected. These cases can already be detected when parsing the specification. If the specification nevertheless is faulty, the user will receive a run-time exception.

3.4 Synchronization Points

Because CVs are frequently accessed, we want to avoid any overhead when reading the value of CVs. We achieve this behavior by requiring the developer to define synchronization points. At synchronization points new values are pushed to CVs using the observer pattern. Only during synchronization points performance overhead occurs. Otherwise reading CVs has the same overhead as accessing native variables [14]. Another advantage of explicit synchronization points is that the user has full control over costs occurring in the program.

Making synchronization points explicit might seem to be cumbersome to program. But in practice it is often obvious where synchronization should occur: when a user starts a new interaction. With a use-case-based software engineering approach one can systematically find all such places. Forgetting about a synchronization point will only affect the specific interaction. In mostly single-threaded applications, which do not sense context itself, it is even simpler: one only synchronizes the main-thread when the application is notified.

The synchronization points define when intra-process updates will take place. The context will push all changes to the respective CVs. For example:

```
1 void userInteraction(Accuracy const& a) {
2   a.context().sync(); // a might change
3   for (long i=0; i<a; ++i) {
4   /* a does not change here */ }
5 }
```

In the example above, we introduce a synchronization point in Line 2. All mentioned reloading features only happen during this invocation. The contextual value `a` is not modified by context changes or changes of persistent CVs at other places. This means, that the programmer can be sure that `a` is not changed during the loop starting on Line 3.

3.5 Assignment

Another property of our approach is that after an initial activation, every (de)activation can occur via changing the values of CVs:

```
1  void assignLanguage(Language & lang) {
2    lang.context().activate(lang);
3    lang = "";
4    // layer lang deactivated
5    lang = "spanish";
6    // layer switch to spanish
7    lang.context().deactivate(lang);
8    lang = "english";
9    // layer still deactivated
10 }
```

The precondition that layers are influenced via assignments is fulfilled in Line 2. In Line 3 we see an assignment to an empty string. Layers with an empty value influence CVs in the same way as deactivated layers. But only after explicit deactivation in Line 7, changes of the CV `lang` do not influence other CVs anymore.

Synchronization via `sync` activate or deactivate layers in the same way as the assignment does. One can think of `sync` as correctly ordered assignment of every CV.

3.6 Inter-process Notification

Because of diverse requirements we took care that Elektra follows very modular design principles [9, 12]. The inter-process notification requirements differ from system to system. We decided to implement inter-process notification in plugins.

Whenever a process modifies the underlying configuration files plugins take care of notification. It is trivial to include notification mechanism that already implement a message bus. In the plugin you only have to publish the message without any further concerns. In every interested process one has to implement a listener that fetches the updated configuration. After every thread has passed a synchronization point, the application is fully updated to the new configuration.

Implementing an inter-process notification without a message bus is more challenging. Nevertheless such an endeavor is useful for legacy applications. Notification via signals are very popular because they are part of C89 and POSIX.

The idea is that every process using Elektra registers its process identifications (PIDs) at startup. Whenever configuration files change, a plugin sends a signal to all registered PIDs. Within individual applications one has to install a signal handler. The signal handler is limited to atomic changes. Thus it can only flag that such an event occurred and other threads can pickup the changes later.

The update to current persistent CVs themselves is via `kdb.get` as shown before. On conflicts Elektra usually uses a three-way merge to not lose data on concurrent updates of other processes.

4. Evaluation

We benchmarked Elektra on a hp® EliteBook 8570w using the central processor unit (CPU) Intel® Core™ i7-3740QM with 4 cores @ 2.70GHz. The operating system was Debian GNU/Linux Jessy 8.4 amd64. We used the compiler gcc 4.9.2-10.

4.1 Microbenchmarks

We start with microbenchmarks that measure the cost of different synchronization methods. As we see hard coded in the code snippets below, we used 1000 iterations. We only show the mean value of the 11 measurements we did for every microbenchmark. We created four microbenchmarks for each line in Figure 1. We designed the microbenchmarks in a way that they are valuable for the decision when which activation strategy should be used. First we start to explain the individual microbenchmarks, then discuss the results. For all four microbenchmarks we will use the same setup:

```
1 Context c;
2 Timer t;
3 CV tcv;
```

The first test, called `benchmarkActivate`, measures activation with layers. In Line 2 we start the measurement and stop it at Line 10. In lines 5-7 the relevant action takes place. In Line 8 another contextual value gets accessed to avoid too aggressive compiler optimizations. Note that reading Elektra CVs is without any overhead compared to access of native variables, so the Line 8 does not influence the benchmark measurable:

```
1  void benchmarkActivate(CV & cv) {
2    t.start ();
3    for (long i = 0; i < 1000; ++i)
4    {
5      if (n>0) c.activate<Layer0>();
6      // ..
7      if (n>N) c.activate<LayerN>();
8      x ^= tcv + tcv;
9    }
10   t.stop ();
11 }
```

In the second microbenchmark (`benchmarkActivateCV`), we used the CV-activation feature introduced in this paper. In the relevant lines 5 to 7, we now have activation of 0 to N CVs. In this benchmark, activation of layers only happens implicitly. Note that again the numbers of activations increase with the number of CVs:

```
1  void benchmarkActivateCV(vector<CV>& cv){
2    t.start ();
3    for (long i = 0; i < 1000; ++i)
4    {
5      if (n>0) c.activate(cv[0]);
6      // ..
7      if (n>N) c.activate(cv[N]);
8      x ^= tcv + tcv;
9    }
10   t.stop ();
11 }
```

As third benchmark, we used the `sync` feature introduced in this paper. The synchronization in Line 6 will synchronize all N different CVs. In this benchmark we do not reload from persistent storage. Nevertheless, the implementation must recalculate every activation of every CV.

```
1  void benchmarkSync(vector<CV> & cv) {
2    // cv contains 0..n-1 CVs
3    t.start ();
4    for (long i = 0; i < 1000; ++i)
5    {
6      c.sync();
7      x ^= tcv + tcv;
8    }
9    t.stop ();
10 }
```

In the last microbenchmark, we fully reload configuration files from disk. The challenge of this benchmark is an optimization of Elektra. The operation to fetch data from persistent storage would not reload without changes. Thus we used for every iteration a different handle to fetch from persistent storage (lines 2 and 3). This ensures that the configuration file is actually reread:

```
1  void benchmarkReload(vector<CV> & cv) {
2    vector<kdb::KDB> kdb;
3    kdb.resize(1000);
4    t.start ();
5    for (long i = 0; i < 1000; ++i)
6    {
7      kdb[i].get(c.values(), "/test");
8      c.sync();
9      x ^= tcv + tcv;
10   }
11   t.stop ();
12 }
```

We see in Figure 1 that the overhead increases linearly the more CVs or layers are involved. More flexible activation scenarios are more expensive. Especially, reloading from configuration files introduces a relatively large, but constant, offset. Note that we used a text configuration file parser.

4.2 Case Study

In a case study we implemented a web server that outputs localized HTML pages. Remote users can connect to the

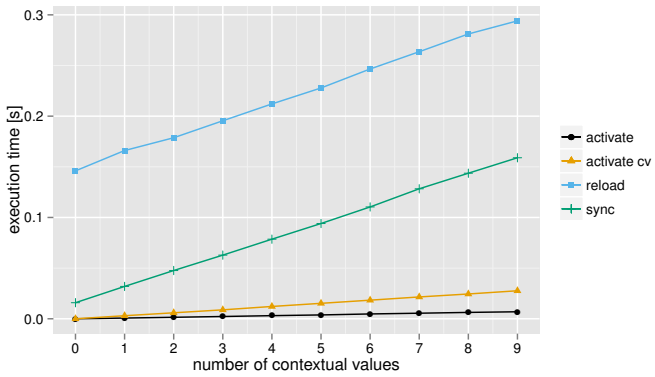

Figure 1. Comparison of 1000 iterations in four setups: by using reloading from persistent storage (reload), syncing all CVs in memory (sync), activation of CVs (activate CV), and only switching layers (activate). We vary the number of activations or CVs.

web server that is installed on an embedded device. For localization and session handling the web server heavily relies on CVs. Because of the features CVs provide, these parts were trivial to implement. Additionally the web server works together with context sensors that modify persistent CVs. For example, one sensor detected motion.

During implementation we found CVs very useful. The specification file was 87 lines and contained 17 CVs. From this specification file 3623 lines of code defining all CVs as classes in one large hierarchy were generated. The boilerplate code (lines 1 to 10) left to write is minimal:

```
1  #include <kdbenvironment.hpp>
2  #include <kdbparse.hpp>
3  Coordinator c;
4  using namespace kdb;
5  int main(int argc, char**argv) {
6    KeySet ks;
7    ThreadContext tc(c);
8    parseConfigfiles(ks);
9    parseCommandline(ks, argc, argv);
10   Environment env(ks, tc);
11   // the rest of the program. e.g.:
12   visit(env.person);
13 }
```

In Line 12 we see how we can access CVs in the hierarchy. We call the function we defined earlier in the paper.

For the benchmark we created one additional thread. In the thread the web server periodically syncs with persistent changes in the same way as `benchmarkReload` does. We found out that beginning with 2200 requests/seconds the reply/seconds did not increase significantly anymore. To measure if context changes influence the functionality of the web server we used httperf on the same machine `1`:

```
1 httperf --hog --timeout=1 --rate=2200
2 --num-conn=50000 --num-call=1 --server=l
```

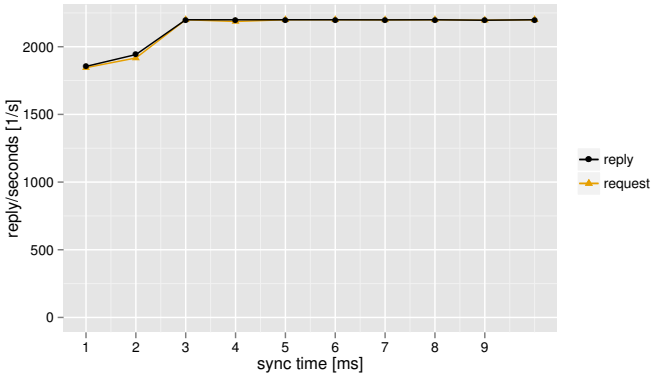

Figure 2. Request/reply rate of a web server. The sync time is a sleep interval in milliseconds to wait for the next synchronization with persistent CVs.

As you see in Figure 2 the sync rate barely has influence on the request and replies the web server can deliver. Only at sync rates below 3 ms there is an effect. A possible explanation is that other CPU cores can compensate easily as long as the required lock during `sync` does not dominate. Timeouts of requests were nearly always 0 except for sync rates of about 3 ms or when influencing the setup. We see that Elektra can be practically used for embedded applications and that the number of context changes only has minimal effect, even across applications.

5. Related Work

Löwis et al. [18] also implemented CVs. They call them context variables. Their layer activation (they call it binding of CVs) requires the programmer to explicitly declare layers. Thus their approach is very similar to previous versions of Elektra and would benefit from the approach described here.

Kamina et al. [6] proposed a generalized activation mechanism based on contexts and subscribers. Their implicit activation, however, has severe impact on the performance.

Very early work (1979) on contextual values is done by Asirelli et al. [1]. Their contextual values are depending on state. We could not find specific information, but their system may have achieved something similar to Elektra. Different from Elektra, they use context-value pairs which do not encode variability information in their keys. Montangero et al. [8] described garbage collection techniques for CVs.

Wang et al. [19] proposed a new metric useful for testing context-aware applications. Information present in the specification of Elektra helps to improve testing, too.

Elmongui et al. [3] described context-aware DBMS. Because Elektra focuses on persisting CVs the approaches seem to be complementary: Such query languages could be an extension for some use cases of Elektra.

We also proposed to move the context awareness to a key-value database [13]. Using interception techniques, unmodified applications were made more context aware.

Mens et al. [7] created a taxonomy of context-aware software variability approaches. They explicitly mention the execution environment to be an important source for context. Based on the taxonomy, Elektra has a closed form of variability, when only the values of CVs can change. Then changes of behavior are limited to the code already present in the program. Elektra has its focus on contextual (not context) features. Elektra uses an one-branch context tree: Without placeholders CVs are automatically non-context-aware configuration items. Additionally, Elektra supports programmer-declared dependencies.

Umuhoza et al. [17] compared different ways for code generations on mobile development. Their methods do not have specific support for context.

Williams et al. [20] and Biegel et al. [2] presented frameworks for development of context-aware applications. ContextPhone [15] is a further prototyping platform for context-aware mobile applications. Their approaches focus on developing context sensors, which complements our approach.

6. Conclusion

In this paper we presented the idea to use persistent contextual values (CVs) as information source whether layers are active. We discussed several benefits and limitations of the approach: (1) The activations themselves take context into account. (2) Persistent CVs can be used to synchronize layers across different applications in different programming languages.

The approach simplifies taking current context into account and sharing context with other applications. Furthermore it provides support for individual customization. End-users can add or redefine configuration in specific context.

In the benchmarks, we showed that activation of CVs is not much more expensive than original layer-activation. Synchronization of all CVs with configuration files, however, is more costly. Luckily, the overhead is only constant. In a real-world benchmark we showed that the sync rate barely has an influence in a web-server setup.

Our contributions are:

1. This combination of performance, context awareness and customization is unique to our approach.
2. Elektra enables programmers to use CVs with code generation in multi-threaded and multi-process applications. CVs can even be shared across applications.
3. Our implementation is free software and can be downloaded from `http://www.libelektra.org`. It supports mobile development in C++, Java and other languages.
4. In a case study we described our experience with mobile development, and analyzed the performance in microbenchmarks and with a web server.

Acknowledgments

Many thanks to Franz Puntigam and the anonymous reviewers for a detailed review of this paper. Additionally, many thanks to all persons contributing to Elektra.

References

[1] P. Asirelli, P. Degano, G. Levi, A. Martèlli, U. Montanari, G. Pacini, F. Sirovich, and F. Turini. A flexible environment for program development based on a symbolic interpreter. In *Proceedings of the 4th International Conference on Software Engineering*, ICSE '79, pages 251–263, Piscataway, NJ, USA, 1979. IEEE Press. URL `http://dl.acm.org/citation.cfm?id=800091.802946`.

[2] G. Biegel and V. Cahill. A framework for developing mobile, context-aware applications. In *Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on*, pages 361–365, March 2004. doi: 10.1109/PERCOM.2004.1276875.

[3] H. G. Elmongui, W. G. Aref, and M. F. Mokbel. Chameleon: Context-awareness inside DBMSs. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1335–1338, March 2009. doi: 10.1109/ICDE.2009.234.

[4] H. Jong-yi, S. Eui-ho, and K. Sung-Jin. Context-aware systems: A literature review and classification. *Expert Systems with Applications*, 36(4):8509 – 8522, 2009. ISSN 0957-4174. URL `http://dx.doi.org/10.1016/j.eswa.2008.10.071`.

[5] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, Nov. 1962. ISSN 0001-0782. doi: 10.1145/368996.369025. URL `http://dx.doi.org/10.1145/368996.369025`.

[6] T. Kamina, T. Aotani, and H. Masuhara. Generalized layer activation mechanism through contexts and subscribers. In *Proceedings of the 14th International Conference on Modularity*, MODULARITY 2015, pages 14–28, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3249-1. doi: 10.1145/2724525.2724570. URL `http://dx.doi.org/10.1145/2724525.2724570`.

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

[8] C. Montangero, G. Pacini, and F. Turini. MAGMA-Lisp: A "machine language" for artificial intelligence. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'75, pages 556–561, San Francisco, CA, USA, 1975. Morgan Kaufmann Publishers Inc. URL `http://dl.acm.org/citation.cfm?id=1624626.1624713`.

[9] M. Raab. A modular approach to configuration storage. *Master's thesis, Vienna University of Technology*, 2010.

[10] M. Raab. 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, April 2015. doi: 10.1109/ISORCW.2015.52.

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

[12] M. Raab. 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, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4033-5. doi: 10.1145/2892664.2892691. URL `http://dx.doi.org/10.1145/2892664.2892691`.

[13] M. Raab. Unanticipated context awareness for software configuration access using the getenv API. In *Computer and Information Science*, pages 41–57. Springer International Publishing, Cham, 2016. ISBN 978-3-319-40171-3. doi: 10.1007/978-3-319-40171-3_4. URL `http://dx.doi.org/10.1007/978-3-319-40171-3_4`.

[14] M. Raab and F. Puntigam. Program execution environments as contextual values. In *Proceedings of 6th International Workshop on Context-Oriented Programming*, pages 8:1–8:6, NY, USA, 2014. ACM. ISBN 978-1-4503-2861-6. URL `http://dx.doi.org/10.1145/2637066.2637074`.

[15] M. Raento, A. Oulasvirta, R. Petit, and H. Toivonen. ContextPhone: a prototyping platform for context-aware mobile applications. *IEEE Pervasive Computing*, 4(2):51–59, Jan 2005. ISSN 1536-1268. doi: 10.1109/MPRV.2005.29.

[16] E. Tanter. Contextual values. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 3:1–3:10, NY, USA, 2008. ACM. ISBN 978-1-60558-270-2. doi: 10.1145/1408681.1408684. URL `http://dx.doi.org/10.1145/1408681.1408684`.

[17] E. Umuhoza, H. Ed-douibi, M. Brambilla, J. Cabot, and A. Bongio. Automatic code generation for cross-platform, multi-device mobile apps: Some reflections from an industrial experience. In *Proceedings of the 3rd International Workshop on Mobile Development Lifecycle*, MobileDeLi 2015, pages 37–44, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3906-3. doi: 10.1145/2846661.2846666. URL `http://dx.doi.org/10.1145/2846661.2846666`.

[18] M. von Löwis, M. Denker, and O. Nierstrasz. Context-oriented programming: Beyond layers. In *Proceedings of the 2007 International Conference on Dynamic Languages*, ICDL '07, pages 143–156, NY, USA, 2007. ACM. ISBN 978-1-60558-084-5. URL `http://dx.doi.org/10.1145/1352678.1352688`.

[19] H. Wang and W. K. Chan. Weaving context sensitivity into test suite construction. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pages 610–614, Nov 2009. doi: 10.1109/ASE.2009.79.

[20] E. Williams and J. Gray. Contextion: A framework for developing context-aware mobile applications. In *Proceedings of the 2nd International Workshop on Mobile Development Lifecycle*, MobileDeLi '14, pages 27–31, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2190-7. doi: 10.1145/2688412.2688416. URL `http://dx.doi.org/10.1145/2688412.2688416`.