Context-Aware Software Variability through Adaptable Interpreters

Walter Cazzola and Albert Shaqiri, Università degli Studi di Milano

// A proposed approach moves variability support from the programming language to the language implementation level. This enables contextual variability in any application independently of whether the underlying language supports context-oriented programming. A Neverlang-based prototype implementation illustrates this approach. //

SUPPORT FOR CONTEXTUAL variability is increasingly in demand for software development tools, especially for ubiquitous and mobile computing. The poor or nonexistent behavioral-variability support provided by traditional software development tools gave rise to approaches that support contextual variability.

In particular, *context-oriented programming* (COP) addresses contextual variability from a programming-language perspective. Typically, COP languages provide abstractions to implement behavioral variations and the variation activation logic. Behavioral variations are modularization units, typically defined as partial-method definitions,[1] that implement the desired context-dependent behavior. Variations are then grouped into layers that the developer explicitly triggers through a layer activation or deactivation mechanism. (Throughout this article, we refer to this widespread model of grouping and activating or deactivating behavior. However, alternatives to layers exist—for example, context objects[2] and predicates.[3] Also, layer context management can be event-driven.[4])

Traditional COP languages are suitable when behavioral variability is designed a priori and implemented from scratch. On the other hand, if behavioral variability is introduced a posteriori or is implicit in a language construct, the application code will require extensive reorganization.

To help programmers avoid such labor, we've developed an approach that moves variability support from the programming language to the language implementation level. This enables contextual variability in any application independently of whether the underlying language supports COP.

Our prototype implementation is based on the Neverlang language development framework,[5] whose support for modularity greatly eases the development of context-aware programming-language interpreters. Neverlang has a multiple-semantic-action dispatcher[6] and supports dynamic adaptation of interpreters.[7] On top of these features, we developed support for interpreter-level layers.

```
 1 var MAX_ITER=50, ZOOM=450, HEIGHT=300, WIDTH=300;
 2 var I = create2DArray(WIDTH,HEIGHT);
 3 var zx=0, zy=0, cX=0, cY=0, tmp=0, iter=0;
 4 for(var y=0; y<HEIGHT; ++y) {
 5     for(var x=0; x<WIDTH; ++x) {
 6         zx=0; zy=0; cX = (x - 400) / ZOOM;
 7         cY=(y-300)/ZOOM; iter=MAX_ITER;
 8         while ((((zx*zx)+(zy*zy))<4) && (iter>0)) {
 9             var tmp=(zx*zx)-(zy*zy)+cX;
10             zy = (2.0*(zx*zy))+cY; zx=tmp; iter=iter-1;
11         };
12         I[x][y]=iter | (iter << 8);
13     };
14 };
```

FIGURE 1. A JavaScript implementation of the escape-time algorithm for calculating a Mandelbrot set. The two loops in lines 4 and 5 are responsible for the whole set calculation. They account for most of the execution time, which heavily increases with the calculated set's size.

Moving to the Language Implementation Level

Consider the escape-time algorithm for calculating a Mandelbrot set; Figure 1 shows a possible JavaScript implementation. The details don't matter; we focus only on the two loops in lines 4 and 5, which are responsible for the whole set calculation. They account for most of the execution time, which heavily increases with the calculated set's size.

It's well known that you can speed up this algorithm by unrolling these loops and executing each of their stages in parallel (for example, on different cores). Parallel execution is faster than sequential execution but consumes more energy and could more quickly drain the battery of a laptop executing the code. Owing to this tradeoff, we'd like to switch between sequential and parallel execution, depending on whether the laptop is plugged into the electrical grid or running on a battery.

This example is deliberately simple to avoid obscuring basic ideas with unnecessary algorithmic details. At the same time, it's representative of a range of context-aware applications whose execution depends on the resource-saving context.

With a traditional COP approach, we'd have to introduce several modifications to our code. For example, if the original code is in a language with no COP extension, we'd have to adopt a new COP language and largely rewrite the original code. We'd have to reorganize the code to explicitly modularize behavioral variations and add the activation logic to switch between sequential and parallel mode according to the context information.

Although in this specific case, this process might not present a serious problem, it's generally invasive and potentially error-prone, and we'd prefer to avoid it. Indeed, we might achieve the desired behavior without changing a single line of code if we take into account that the sequential and parallel behavioral variations are aligned closely with the `for` constructs in Figure 1. In other words, the application's behavior regarding the desired goal depends on whether the `for` constructs run sequentially or in parallel. So, if we could change in the underlying interpreter how these loops behave, we'd affect the application's behavior without modifying its code.

This problem emerges in applications in which behavioral variations aren't (or can't be) explicitly modularized but are implicit in a language feature. (A language feature is an abstract concept or construct belonging to a programming language[5]—for example, class definition, inheritance, or loop.) The problem also occurs when applications originally weren't designed to be context-dependent or when no COP extension exists for the language. Introducing variability to traditional approaches in such situations can be invasive and undesirable.

To overcome these drawbacks, we move variability support to the programming-language interpreter. Instead of providing explicit language abstractions for handling the context, we aim to

- provide interpreter-level semantic variations of standard, non-COP language features in terms of the language in which the running application is written and
- activate or deactivate them when the context changes.

This approach completely separates behavioral variations and their triggering logic from the application logic, with clear benefits for future maintenance and evolution.

Clearly, our approach differs substantially from traditional COP and isn't meant to replace well-established approaches. Instead, it's useful when adopting other approaches is impossible or would lead to onerous rewriting of the original code.

Contextual Variability through Adaptable Interpreters

A programming language is basically a set of language features developers use to implement applications. To implement an interpreter for such a programming language, language developers must specify how the language features behave. Before interpretation, the application source code is typically transformed into a tree representation in which each node corresponds to a language feature used in the source code. For example, the expression `1 + 2` could be represented by a subtree whose root node represents the addition operation and whose child nodes represent the operands (see Figure 2a).

A language feature is associated with a semantic action that determines how the feature behaves. For example, the addition operation, represented by the + node in Figure 2a, could be associated with a semantic action that adds the operands whose values are stored in the node's children. The rectangle in Figure 2a shows the pseudocode describing the addition construct's behavior.

So, interpreting a program is as simple as traversing the tree and executing the defined semantic actions when nodes are visited. Hence, an application's behavior strongly depends on the behavior of the underlying language features. We can thus affect how the application behaves by changing the interpreter-level semantic actions associated with the desired language features.

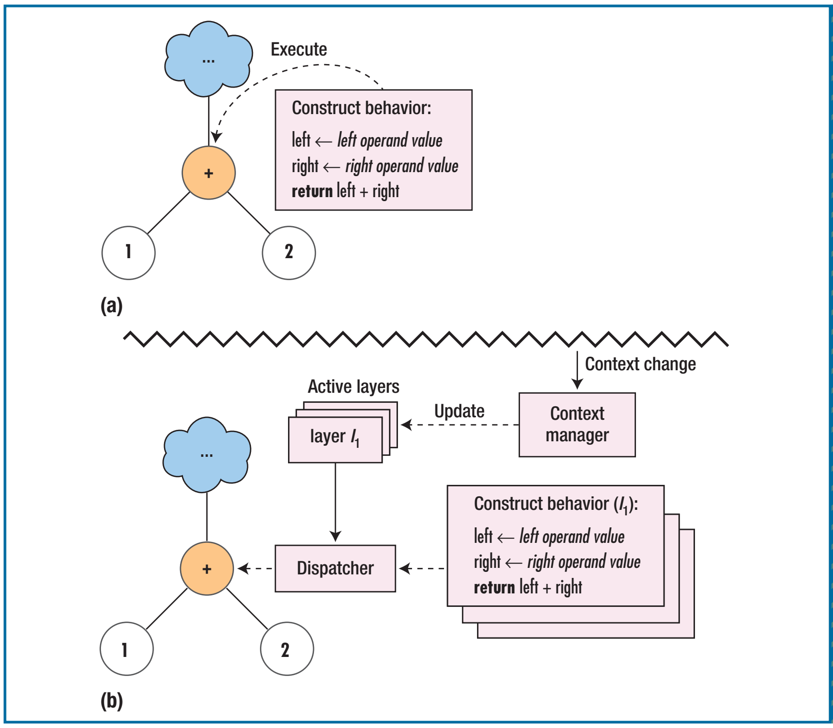

FIGURE 2. An interpretation model. (a) Application interpretation: the association between nodes and semantic actions. (b) Multiple behavioral variations dispatched according to the current context.

Behavioral Variations

As we mentioned before, traditional COP provides behavioral variations as partial methods grouped into layers. When no layer is active, a call to method `m` triggers the standard behavior variation defined in the body of `m`. If a particular layer is active, the variation of `m` defined in that layer executes. In a sense, `m` is associated with many behaviors defined in different method bodies, and the active layers (the context) determine the right variation.

We associate interpreter-level behavioral variations with a language feature and dispatch the related semantic actions according to some context information. In this view, a language feature and its associated interpreter-level semantic action correspond respectively to a partial method's name and body in COP. Indeed, when the code uses a language feature, our approach uses that feature (the "partial method's name") to identify and execute the associated semantic action (the "partial method's body").

In this approach, variations are implicit in a language feature (because they're implemented in the underlying interpreter) and needn't be explicitly implemented in the application. In Figure 2b, the addition operation is associated with many actions, each of which has a label that determines to which layer that action belongs. (For example, in Figure 2b, the visible action belongs to layer l_1.) When a node is visited, the dispatcher executes the associated actions belonging to the active layers.

It's a big challenge to identify language features whose behavior must be changed to achieve the desired

```
module nlg.JS.ForLoop {
  reference syntax {
    For: Stm←... /* syntax definition */;
  }

  role (evaluation) {
    For: .{
      ...
      int start=(int)$For[2].Value;
      int end=(int)$For[3].Value;
      int step=(int)$For[4].Value;

      // execute the loop body sequentially
      for(int i=start;i<end;i=i+step) {
        ...
        // execute the loop body
        eval $For[4];
      }
    }.
  }
}
```

(a)

```
module nlg.JS.ParForLoop {
  imports { java.util.concurrent.*; }

  reference syntax from nlg.JS.ForLoop;

  role (evaluation) {
    For (performance): .{
      ...
      ExecutorService exec=
        Executors.newFixedThreadPool(4);
      // execute the loop body in parallel
      for (int i=start;i<=end;i=i+step) {
        ...
        // ThreadVisit implements Runnable
        ThreadVisit t=new ThreadVisit(...);
        exec.submit(t);
      }
      ...
    }.
  }
}
```

(b)

FIGURE 3. Behavioral variations of the **for** construct implemented in Neverlang. (a) A sequential variation. (b) A parallel variation. Ellipses replace discussion-irrelevant code.

application-level effects. To address this problem, Ruzanna Chitchyan and her colleagues introduced the concept of *micro-languages*.[8,9] A micro-language associates an application feature with the language features used to implement it. That is, it identifies the language features responsible for the portion of the application you want to change. For instance, in the Mandelbrot example, we'd like to change how the "set calculation" application feature (lines 4 to 14 in Figure 1) behaves. A micro-language associated with this application feature would contain, among other things, the **for** feature that we need to change.

Layer Activation

In our approach, layers aren't explicitly activated in the application code. Instead, an external context manager notifies the interpreter about the context change by activating or deactivating layers (see Figure 2b). A context manager is a custom application that handles the context and interacts with the interpreter. In our implementation, the interaction is based on a command-line tool we describe later.

The interpreter handles a global list of active layers whose order determines the order in which semantic actions execute. Unless specified otherwise, the order of layers is determined by the order in which the context manager activates or deactivates them. Layer activation or deactivation can occur at any moment of the application interpretation.

Using the Approach

To illustrate how a practitioner would use our approach, we refer to the Mandelbrot example in Figure 1. In general, there are two steps:

1. Develop or reuse behavioral variations.
2. Define the activation logic.

In this example, the default interpreter implementation already provides the sequential behavioral variation. Figure 3a shows this variation's pared-to-the-bone Neverlang implementation. If the application developer is also a language engineer, he or she could develop the parallel variation in Figure 3b; otherwise, he or she could reuse one that other developers have already implemented. The main difference between the two variations is that the first sequentially executes a loop body, whereas the second spawns a thread for each loop run. The highlighted code shows that the parallel implementation is grouped into the **performance** layer, whereas the sequential variation is unlabeled, meaning that it pertains to the **standard** layer.

The context manager in Figure 4 is a simple Bash script that periodically checks whether the laptop executing the code is running on the electrical grid or a battery. When a change is detected, the script activates or deactivates the appropriate layers. **mda** is a command-line tool that lets users dynamically modify a running Neverlang-based interpreter that's identified by a name binding—**Mandelbrot** in our example. During the interpretation, the node instances of the **for** loops will be visited several times. If the **performance** layer is active, the execution will be in parallel. This is possible because the loop stages are independent.

Applicability

Our approach is best for, but isn't limited to, situations with these characteristics:

- The application-level language doesn't support COP.
- Behavior activation is asynchronous with the application control flow and can't be efficiently coded with standard COP abstractions.
- Behavioral variations can't be efficiently modularized in the application-level language because they're implicit in a language feature.

Interpreter-level COP provides context-handling features to non-COP languages. Also, behavior activation is asynchronous with the application control flow. Although explicit layer activation is suitable in many scenarios, an implicit-activation mechanism is mandatory when the context might change anytime during program execution.[4,10] The third characteristic is particularly prominent in domain-specific languages in which context and language constructs are aligned with domain concepts. The more they're aligned, the easier defining behavioral variations becomes. In the best-case scenario, a construct will correspond perfectly to a domain concept whose behavior must vary in response to domain context changes.

This approach has three main possible applications. First, *sandboxing* disables or modifies a specific language construct's behavior in response to a context change. Second, *input sanitization* employs input constructs to additionally elaborate an input if its origins are suspicious. Finally, *approximate programming* adapts constructs to consume less energy or time at the expense of precision.[11]

Our approach offers some advantages over existing context-oriented approaches, especially when behavioral variations are aligned with language features. The application code remains untouched, which contributes to clearer understanding and maintenance. You don't have to adopt a specific COP language, nor do you have to rewrite your software.

Ensuring behavioral consistency, a problem somewhat present in COP research,[12] becomes difficult as the software grows. Constraint enforcement,[13] context dependencies,[2] and other context-switching strategies[14] somewhat alleviate this problem. Neverlang has a type system that prevents interpreters from being broken, although it can't guarantee that the application will behave as expected.

Although separating the adaptation and the application logic is beneficial, it limits control of adaptation. Furthermore, the global state of active layers doesn't provide means for fine-grained adaptation. So, the need exists for further investigation to provide more control at the interpreter level.

Also, it's unreasonable to expect application developers to be language engineers. To bridge this gap, we're doing three things. First, we're identifying and developing the most common behavioral variations for a set of language features that will be provided as black boxes to plug into interpreters. Second, we're developing a micro-language-based support framework that should help developers identify language features to adapt to achieve the desired behavior. Finally, we plan to further elaborate our interactive configuration tool for language product lines[15,16] to help users choose from the available behaviors.

```
[[$(acpi -a)==*"off"*]]; plugged=$?

while true; do
  [[$(acpi -a)==*"off"*]]; now=$?
    if ["$plugged"!="$now"]; then
      if ["$now"==true]; then
        # plugged->battery
        mda Mandelbrot deactivate performance
        mda Mandelbrot activate standard
      else
        # battery->plugged
        mda Mandelbrot deactivate standard
        mda Mandelbrot activate performance
      fi
      plugged="$now"
    fi
  sleep 1
done
```

FIGURE 4. The context manager script, written in Bash. The script periodically checks whether the laptop executing the code is running on the electrical grid or a battery. The code will execute sequentially in the former case and in parallel in the latter case.

References

1. R. Hirschfeld, P. Costanza, and O. Nierstrasz, "Context-Oriented Programming," *J. Object Technology*, vol. 7, no. 3, 2008, pp. 125–151.
2. S. González et al., "Subjective-C: Bringing Context to Mobile Platform Programming," *Software Language Engineering*, LNCS 6563, Springer, 2010, pp. 246–265.
3. J. Vallejos et al., "Predicated Generic Functions: Enabling Context-Dependent Method Dispatch," *Software Composition*, LNCS 6144, Springer, 2010, pp. 66–81.
4. M. Appeltauer et al., "Event-Specific Software Composition in

ABOUT THE AUTHORS

WALTER CAZZOLA is an associate professor in the Department of Computer Science at Università degli Studi di Milano and the chair of the university's ADAPT Lab. His research interests include reflection, aspect- and context-oriented programming, software evolution, and programming languages and their implementation. Cazzola received a PhD in computer science from Università degli Studi di Milano. Contact him at cazzola@di.unimi.it.

ALBERT SHAQIRI is a PhD candidate in computer science at Università degli Studi di Milano and a member of the university's ADAPT Lab. His research interests include modular development of programming languages, dynamic interpreter optimization, and dynamic software updating. Shaqiri received a master's in computer science from Università del Piemonte Orientale. Contact him at shaqiri@di.unimi.it.

Context-Oriented Programming," *Software Composition*, LNCS 6144, Springer, 2010, pp. 50–65.

5. E. Vacchi and W. Cazzola, "Neverlang: A Framework for Feature-Oriented Language Development," *Computer Languages, Systems and Structures*, vol. 43, no. 3, 2015, pp. 1–40.
6. W. Cazzola and A. Shaqiri, "Modularity and Optimization in Synergy," *Proc. 15th Int'l Conf. Modularity* (Modularity 16), 2016, pp. 70–81.
7. W. Cazzola and A. Shaqiri, "Open Programming Language Interpreters," *The Art, Science, and Engineering of Programming*, vol. 1, no. 2, 2017, article 5.
8. R. Chitchyan, W. Cazzola, and A. Rashid, "Engineering Sustainability through Language," *Proc. 37th Int'l Conf. Software Eng.* (ICSE 15), 2015, pp. 501–504.
9. W. Cazzola et al., "μ-DSU: A Micro-language Based Approach to Dynamic Software Updating," to be published in *Computer Languages, Systems and Structures*.
10. M. von Löwis, M. Denker, and O. Nierstrasz, "Context-Oriented Programming: Beyond Layers," *Proc. 2007 Int'l Conf. Dynamic Languages* (ICDL 07), 2007, pp. 143–156.
11. J. Park et al., "FlexJava: Language Support for Safe and Modular Approximate Programming," *Proc. 10th Joint Meeting Foundations of Software Eng.* (ESEC/FSE 15), 2015, pp. 745–757.
12. G. Salvaneschi, C. Ghezzi, and M. Pradella, "Context-Oriented Programming: A Software Engineering Perspective," *J. Systems and Software*, vol. 85, no. 8, 2012, pp. 1801–1817.
13. P. Costanza and R. Hirschefeld, "Reflective Layer Activation in ContextL," *Proc. 2007 ACM Symp. Applied Computing* (SAC 07), 2007, pp. 1280–1285.
14. N. Cardozo et al., "Safer Context (De)Activation: Through the Prompt-Loyal Strategy," *Proc. 3rd Int'l Workshop Context-Oriented Programming* (COP 11), 2011, article 2.
15. T. Kühn, W. Cazzola, and D.M. Olivares, "Choosy and Picky: Configuration of Language Product Lines," *Proc. 19th Int'l Software Product Line Conf.* (SPLC 15), 2015, pp. 71–80.
16. E. Vacchi et al., "Variability Support in Domain-Specific Language Development," *Software Language Engineering*, LNCS 8225, Springer, 2013, pp. 76–95.