Run-time Validation of Behavioral Adaptations

Nicolás Cardozo[1], Laurent Christophe[2], Coen De Roover[3,2], Wolfgang De Meuter[2]

[1]Future Cities, DSG, Trinity College Dublin - College Green 2, Dublin 2, Ireland
[2]Software Languages lab, Vrije Universiteit Brussel - Pleinlaan 2, 1050 Brussels, Belgium
[3]Osaka University - 1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
cardozon@scss.tcd.ie, lachrist@ulb.ac.be, coen@ist.osaka-u.ac.jp, wdmeuter@vub.ac.be

ABSTRACT

Context-oriented programming enables the composition of behavioral adaptations into a running software system. Behavioral adaptations provide the most appropriate behavior of a system when their contexts are activated or deactivated, according to the situations at hand in the system's execution environment. Behavioral adaptations can be defined by third-party vendors or even be acquired at run time. As the systems grows in complexity it becomes increasingly difficult to reason about every possible runtime adaptation. Therefore, behavioral adaptations that lead to erroneous states or compromise the system's security might be difficult to detect statically. To prevent such undesired behavioral adaptations from happening, we introduce a run-time correctness verification approach. Our approach uses a symbolic execution engine to reason about the reachable states of the system, whenever contexts are activated or deactivated. Context activation and deactivation requests are allowed depending on the presence of erroneous states within reachable states. Our approach is a step forward to ensure the run-time correctness of software systems that adapt their behavior dynamically.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.4 [**Programming Languages**]: Software/Program Verification

General Terms

Languages, Verification

Keywords

Context-oriented programming, Correctness, Symbolic execution, Validation

1. INTRODUCTION

Current day software systems are able to adapt their behavior at run time according to the situations in their surrounding execution environment. Adaptive software systems can compose/withdraw application behavior based on information gathered from a sensor network or previous uses of the system. Context-Oriented Programming (COP) [4] is a programming paradigm that enables run-time adaptations of a software system. COP provides language level abstractions to define and compose fine-grained adaptations in a running system, respectively, as contexts and their associated behavioral adaptations.

Run-time adaptations, as proposed in COP, are modular and localized, enabling the extension and modification of existing third-party programs and frameworks [2]. As a consequence of the offered flexibility in modifying the behavior, multiple vendors can propose variations to a given service by adapting its basic behavior. Note that if composition and withdrawal of run-time adaptations is not dealt with carefully, it may lead to incorrect states of the system or even security breaches. On the one hand, adaptations of a service may not respect the API offered by the service, or depend on other transient services (*i.e.,* provided by other adaptations). These lead to incorrect states of the system either exhibiting incorrect computations or causing the application to be interrupted abruptly. On the other hand, security of the system may be compromised with the introduction of adaptations that, for example, bypass security checks of the system (*e.g.,* login–password verification).

Before an adaption is composed with or withdrawn from the run-time system, it should be explored to validate it does not yield run-time errors. A promising candidate for validating the correctness of adaptations is the use of symbolic execution. Symbolic execution [7] is a program analysis that traverses the source code of a program to generate constraints that should be satisfied in order for a concrete run to end up at given points in the code. In our approach, we invoke a new symbolic execution engine, Kusasa, at runtime to investigate whether it is safe to active or deactivate a context at a particular point in time. Before each behavioral adaptation, Kusasa symbolically explores the future execution steps. Depending on whether erroneous states are discovered in this exploration, the adaptation is performed. To give the programmer a more fine-grained control over our approach, we extend the semantics of context activation and deactivation. We introduce alternative contexts for Kusasa to try out in case the initial context (de)activation cannot take place. Additional parameters for precisely defining the safety policy of the symbolic exploration are also provided.

This paper represents the first use of a run-time symbolic execution engine for verifying the correctness of behavioral adaptations.

2. MOTIVATION: MULTI-VENDOR POSITIONING SERVICE

This section motivates the need for validating behavioral adaptations at run-time. To this end, we present a proof-of-concept application for mobile devices. The application consists of a geo-positioning service that can be extended or modified by different vendors. We develop the application using the Ambience [5] COP language.

The multi-vendor positioning service is an application that uses the GPS coordinate system for devices that can receive Global Navigation Satellite (GNS) signals. The service is provided through a simple API that consists of a function for displaying and a function for acquiring the current position. Both interacting functions are depicted in Snippet 1. The positioning service should only be available if there is an enabled GNS receiver, thus the service is provided as an adaptation rather than as a standalone application. An adaptation is defined by means of the `(defcontext ctx)` construct, which defines a context object, and the `(with−context (ctx )...)` construct, which defines the behavioral adaptations associated with that context object. The definition of the GPS adaptation including its context object and associated behavioral adaptations is shown in Snippet 2.

```
(defmethod display-position (pos)
 (format t "Current position is (~,3f, ~,3f)~%"
       (car res) (cdr res)))

(defmethod get-position ((user @person))
  (throw 'no-pos "No antenna"))
```

Snippet 1: Basic system behavior with no GNS antenna enabled.

```
(defcontext @gps-antenna)

(with−context (@gps-antenna)
     ;;gps object definition
  (defproto @gps-device (clone @object))
  (defmethod longitude ((gps @gps-device)) ...)
  (defmethod latitude ((gps @gps-device)) ...)
  (defparameter *gps* (clone @gps-device))

  (defmethod get-position ((user @person))
   (let (posx (x user))
      (posy (y user))
   (log-message :info "User ~a position (~,3f,
       ~,3f)~%" (name user) posx posy) '
   (cons posx posy)))
```

Snippet 2: GPS positioning adaptation.

Three different vendors decide to extend the existing service with their own specialized functionality. *Vendor1* offers a service, similar to the GPS adaptation, optimized for the European market by taking advantage of the Galileo positioning system. Whenever users are found to be in a European country, they will query this specialized service instead. Besides the way of acquired positions from the GNS satellite, *Vendor1* uses its own format for the retrieval of the position (*i.e.,* using a `@location` object), but reuses the display functionality offered by the original service. This service, as shown in Snippet 3, is offered in form of the adaptation characterized by the `@galileo` context object.

```
(with−context (@galileo)
 (defmethod get-position ((user @person))
  (let (loc (clone @location)))
  (setf (x-pos loc) (latitude *galileo* user))
  (setf (y-pos loc) (longitude *galileo* user))
  loc))
```

Snippet 3: Behavioral adaptation of *Vendor1*.

Note that the `get-position` behavioral adaptation associated with the `@galileo` context does not respect the interaction with the original `display-position` behavior, defined in Snippet 1. The interaction between these two methods yields an error when trying to use the `car` and `cdr` primitives on the object returned by `get-position`.

Vendor2 enhances the original service by providing an approximation to the positioning service whenever the GPS antenna is not available (*e.g.,* inside a building). This functionality is enabled in the `@compass` context, whenever a digital compass is available in the device. The introduced behavioral adaptation reuses previous calculations of the position (using the GPS adaptation) to give an approximated position of the user based on its direction and speed. Snippet 4 shows the definition of the `get-position` behavioral adaptation associated with the `@compass` context.

```
(with−context (@compass)
 (defproto @compass (clone @object))
 (defmethod direction ((comp @compass)) ...)
 (defparameter *compass* (clone @compass))

 (defmethod get-position ((user @person))
  (if (or (equal (xcoord user) nil) (equal (
      ycoord user) nil))
   (throw 'no-initial-pos "No initial position"
       )
   (cons (latitude (direction *compass*) user)
       (longitude (direction *compass*) user)))
       ))
```

Snippet 4: Behavioral adaptation for *Vendor2*.

Unlike for Vendor1, this behavioral adaptation respects the interaction between the `display-position` and `get-position` methods. However, for the successful execution of the introduced behavioral adaptation, the `@gps-antenna` context behavioral adaptations are required to have been used previously. This creates a fragile dependency on the `@gps-antenna` context. Execution of the system when the `@compass` context is active may or may not lead to errors.

Finally, *Vendor3* has some high-profile clients, whose position should never be disclosed. With this in mind, *Vendor3* is required to offer a privacy mode adaptation to a subset of its users. In this mode, characterized by the `@privacy` context, information must not be leaked out of the system. The behavioral adaptations associated with the `@privacy` context are defined in Snippet 5. This last adaptation redefines the `log-message` method by throwing an error instead of effectively logging potentially sensitive information. As the `@gps-antenna` context logs the position, an error will be thrown if it is active at the same time as the `@private` adaptation. The `@galileo` context is safe to use in combination

with the `@private` context.

```
(with-context (@privacy)
 (defmethod display-position (pos)
  (resend)
  (throw 'no-pos "Position is not available"))

 (defmethod log-message (args)
  (throw 'private-logging "Private information
      leaking")))
```

Snippet 5: Behavioral adaptation for *Vendor3*.

The above examples demonstrate that an adaptation's assumptions about available services and how they interact may not always be satisfied at run-time (*e.g.,* interaction between the `@compass` and `@gps-antenna` contexts). Therefore, behavioral adaptations must be validated at run time before they are composed with the system; this with the objective of avoiding errors.

3. CORRECTNESS OF BEHAVIORAL ADAPTATIONS

This section presents our approach for the validation of correctness of fine-grained behavioral adaptations as proposed by the COP paradigm. We define a behavioral adaptation $\mathcal{A}$ to be *valid* if and only if, for system $\mathcal{S}$ at state e, after $\mathcal{A}$ is composed with the system, then the composed adapted system $\mathcal{S} \circ \mathcal{A}$ does not reach a state e' where e' is an **error**. Our approach is based on Kusasa, a run-time symbolic execution engine, which over-approximates reachable states at context activation time for ensuring correctness.

3.1 Symbolic Execution

Before diving into our proposed approach, we introduce the basic notions of symbolic execution. Symbolic execution is a program analysis technique that traverses the source code of a program to generate constraints that should be satisfied, in order to reach specific points in the program during its concrete execution. Using constraint solvers such as Z3[1] or CVC4,[2] one can generate the inputs that would lead the execution to the targeted point. Symbolic execution can be used for many purposes, such as bug detection, program verification, debugging, maintenance, and fault localization [3].

Static symbolic execution involves executing the program with some values as symbols that range over a set of concrete values. The objective of static symbolic execution is to explore the tree of all possible computation branches a program can have [7]. This exploration approach has as drawback that it does not scale to large systems since the number of feasible paths is subject to combinatorial explosion [1].

Dynamic symbolic execution involves instrumenting a program to record symbolic constraints while it is concretely executed. A prominent application of dynamic symbolic execution is automatic test generation where test inputs are generated to maximize coverage. In automatic test generation, symbolic constraints are used to generate new inputs that would lead the execution to unexplored areas.

[1] `http://z3.codeplex.com/`
[2] `http://cvc4.cs.nyu.edu/web/`

Kusasa is a novel interpreter that executes a program by interleaving concrete execution with static symbolic exploration. During the static symbolic exploration phases of the system, Kusasa generates a tree of future computations starting from the current concrete execution point. When concrete execution is resumed, Kusasa moves from the root of the computation tree to one of its leaf nodes by resolving symbolic constraints and performing buffered input/output actions.

Unlike traditional static symbolic execution, Kusasa does not attempt to explore all possible computation paths in their entirety. In fact the symbolic exploration can be parameterized to restrict the exploration space.

3.2 Run-time Validation of Behavioral Adaptations

We now present our approach for validating behavioral adaptations at run-time. We consider validation of an active process of the run-time system to be performed whenever an adaptation is requested —rather than after adaptations have been composed.

We integrate Kusasa as the execution engine of the Ambience COP language. Specifically, to verify the correctness of adaptations we choose to trigger the Kusasa symbolic exploration phase at context activation time —that is, every time a context is requested for activation or deactivation. We propose the following extension of the constructs for context activation and deactivation to interact with Kusasa:

```
(activate   ctx ctx-list safety depth span)
(deactivate ctx ctx-list safety depth span)
```

The purpose of this extension is to couple validation through symbolic execution with the run-time activation of contexts, and hence, the dynamic composition of their associated behavioral adaptations. Table 1 details the different parameters to the (de)activation constructs.

Parameter	Default	Description
`ctx`		Context to be (de)activated.
`ctx-list`	nil	List of alternative contexts providing similar behavioral adaptations, in case (de)activating `ctx` fails.
`safety`	`'none`	A symbol specifying the safety policy.
`depth`	`0`	An integer indicating the targeted depth of the computation tree. `0` means the exploration stop only at final states.
`span`	`0`	A time span (in milliseconds) specifying the lease Kusasa has for the symbolic exploration. `0` means there is no time threshold.

Table 1: Activation/deactivation parameters API.

Every time a (de)activation is requested, the symbolic exploration phase of Kusasa is triggered to explore the future of the execution from the current execution state. First, Kusasa symbolically explores the execution paths corresponding to the (de)activation of `ctx`. Then it explores the (de)activation of each context in `ctx-list` in their given order. The exploration stops whenever the safety policy (`safety`) is satisfied; if no context were deemed safe, the adaptation does not occur. Such exploration process generates a computation tree whose root is located at the current execution point for each attempted adaptation.

The `depth` parameter represents the targeted depth of each computation tree —that is, how many steps into the future should be explored. The `span` parameter defines the maximum time span Kusasa should take for exploring the generated paths. The `span` parameter is important since the size of computation trees may grow exponentially with their depth. Note that Kusasa equitably distributes the time to investigate each context (de)activation; taking span/(|ctx-list| + 1) for expanding each computation tree.

We define three main policies to assess when the adaptation of the system leads to a correct state: (1) `none`: always allow the context switching, this is introduced to support the original (de)activate semantics. (2) `strict`: all computation paths must be safe —that is, the computation tree must be error-state free. (3) `lenient`: at least one computation path must be safe —that is, at least one branch of the computation tree must be error-state free. Independently from these policies, we also allow the programmer to specify that the targeted depth must be reached (*i.e.,* the time span is not used). Table 2 depicts the available safety policies.

For each safety policy, we interpret the response of Kusasa. For instance, a positive answer to a context activation using the `strict-complete` policy means that the computation tree generated by Kusasa is error free. Additionally, we know that each branch of the generated exploration tree either contains `depth` steps, or leads to a final success state. This means the next `depth` computation steps are safe.

There are three possible outcomes at the end of the Kusasa's symbolic exploration:

- The context `ctx` requested for (de)activation respects the specified safety policies and it is effectively (de)activated.

- The context `ctx` requested for (de)activation breaks the safety policy but one of the alternative contexts passes it. In case of activation, the best fallback adaptation that is safe will be activated. In the case of deactivation, the context is deactivated, and the best fallback adaptation that is safe is activated in its place.

- None of the given contexts satisfies the safety policy, thus no context is effectively (de)activated.

Note that the decision in the latter two cases to whether an alternative context is (de)activated still complies with the intention of adaptive systems. Since the desired adaptations (and its possible alternatives) are unsafe, the most appropriate behavior of the system would be not to compose unsafe adaptations. The role of the safety policies in the process is to dictate how conservative is the approach. For example, the use of the `strict` policy requires all computation path of the context to be (de)activated to be safe. If the condition is not satisfied, the execution of the system proceeds as if no context (de)activation was requested. In the following section we demonstrate how this process is used to validate different context activation situations in the multi-vendor positioning service.

4. EVALUATION

We evaluate the usefulness of our approach by demonstrating that the verification of behavioral adaptations at context activation time can help to prevent run-time errors of the system. To this end, we focus on three situations of the multi-vendor positioning service example and observe the behavior of our analysis for each of them. (1) The activation of the `@galileo` context using the `'strict` policy. r (2) The activation of the `@compass` context using the `'lenient` policy. (3) The activation of the `@private` context using a time span with the `'strict` policy.

(1) Galileo context.

In the first situation, described in Snippet 6, the user is asked whether he wants the position to be displayed. If the response is affirmative, the position is retrieved and displayed. Given that the user's profile signals the position to be in Europe, *Vendor1*'s service should be available to use the behavioral adaptations associated with context `@galileo`. Unfortunately, since the behavioral adaptation is inconsistent with the system's API, activating `@galileo` will lead to an error and it should be denied. Nonetheless, the context activation defines a fallback mechanism (Line 3 in Snippet 6). In case `@galileo` cannot be activated, then the activation of `@gps-antenna` is evaluated.

```
1 (defmethod main ()
2   ...  ;; get user information
3   (activate @galileo (list @gps-antenna) '
        strict)
4   (format "Display position? [y/n]")
5   (when (equal (read) "yes")
6     (defvar pos (get-position *user*))
7     (display-position pos)))
```

Snippet 6: Situation to active the `@galileo` context.

Upon activation request (Line 3), the execution of the system is paused, triggering the symbolic exploration phase. Kusasa first investigates the activation of the `@galileo` context, generating the left computation tree of Figure 1. This tree leads to an error state from calling the `car` primitive on a location object returned by the `get-position` behavioral adaptation. Since the activation of the context was requested using the `'strict` safety policy, this context activation is rejected. Subsequently, Kusasa investigates the activation of `@gps-antenna`. The generated computation tree (right tree of Figure 1), being free of error states, results in the activation of the `@gps-antenna` context.

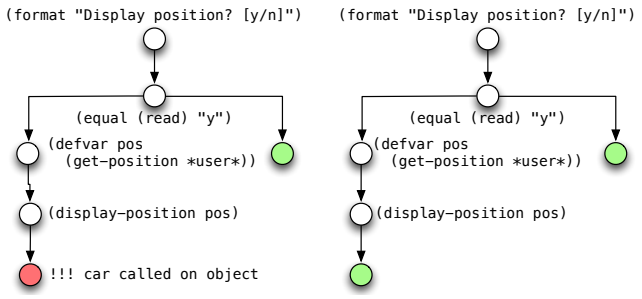

Figure 1: Kusasa generated computations trees: left(right) tree generated from the activation of `@galileo`(`@gps-antenna`).

(2) Compass context.

The second situation, described in Snippet 7, shows the use of the `@compass` context once a user location has been

Safety policy	Definition	Result	Guarantee (`depth` := n)
`none`	Always satisfied	Yes	None
		No	Not applicable
`strict`	*All* computation paths are safe	Yes	None
		No	An error *might* happen in maximum n steps
`lenient`	*One* computation path is safe	Yes	None
		No	An error *will* happen in maximum n steps
`strict-complete`	*All* computation paths are safe and the targeted depth is reached	Yes	The next n steps *will* be safe
		No	An error *might* happen in maximum n steps
`lenient-complete`	*One* computation path is safe and the targeted depth is reached	Yes	The next n steps *might* be safe
		No	An error *will* happen in maximum n steps

Table 2: Available safety policies and interpretation of their results.

calculated using the `@gps-antenna` context. Note that to display the position some arithmetic manipulation is done according to the sensor input and the device's screen (Line 9 in Snippet 7). *Vendor2* notices that the `@compass` behavioral adaptation fails from time to time, but the reason for this failure is unknown. As a consequence, activation of the `@compass` context should only happen when it is safe. In this situation, using the `'strict` safety policy is too restrictive because the symbolic exploration signals the division at Line 9 as a failure if the **read** method returns zero. Instead using the `'lenient` safety policy for activation would result in the correct activation of the context (Line 7 in Snippet 7) since it is only needed for one of the generated paths to reach a correct state.

```
1 (defmethod main ()
2   ... ;; get user and antenna information
3   (when *antenna*
4     (activate @gps-antenna)
5     (get-position *user*)
6     (deactivate @gps-antenna))
7   (activate @compas 'lenient)
8   (defvar pos (get-position))
9   (setvar pos (/ pos (read)))
10  (display-position pos))
```

Snippet 7: Situation to activate the `@compass` context.

As in the previous case, execution is paused upon context activation. In this situation Kusasa generates one of the computation trees depicted in Figure 3. If there is a GPS antenna nearby, Kusasa produces the left tree and the `@compass` context is activated since the program *might* succeed. If there is no GPS antenna, Kusasa produces the right tree and the `@compass` context is not activated.

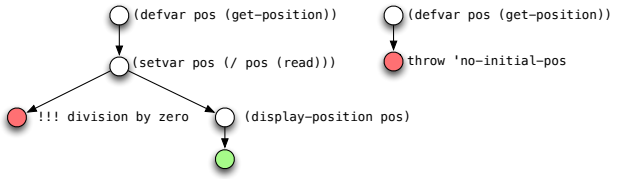

Figure 2: Kusasa generated computations trees: the left tree is generated when a GPS antenna is available, the right tree is generated otherwise.

(3) Privacy context.

Our final situation concerns the behavioral adaptations associated with the `@privacy` context. At first, the `@privacy` context is activated without a safety property. Kusasa then investigates which of the context (`@gps-antenna` or `@galileo`) is safe to activate. However, many statements may remain to be processed in the program. Hence, the symbolic exploration is explicitly restricted to last 1 second.

```
1 (defmethod main ()
2   ...  ;; get user information
3   (activate @privacy)
4   (activate @gps-antenna @galileo 'strict 0
        1000)
5   (defvar pos (get-position *user*))
6   ...) ;; many safe statements
```

Snippet 8: Situation to activate the `@privacy` context.

The first explored path corresponds to the `@gps-antenna` context. This exploration quickly finds an error state due to the call to `log-message` (Snippet 2), which is prohibited in the `@private` context. The second exploration, for the `@compass` context, satisfies the `'strict` policy although Kusasa might not have had time to complete the exploration of all paths.

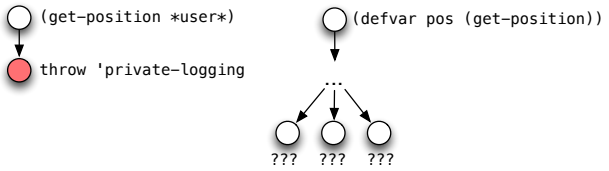

Figure 3: Kusasa generated computations tree: left(right) tree generated from the activation of `@gps-antenna`(`@galileo`) activation.

5. RELATED WORK

This paper presents a first approach for the run-time validation of fine-grained adaptations at the programming language level. Here we discuss other approaches that have been applied for the correctness verification of adaptations at coarser granularity levels, or other points in time during the adaptation process. We compare such approaches with our own, and discuss how they could compliment each other.

Kulkarni et al. [8] propose the correctness verification of adaptations in component-based systems. Their approach is

based on the notion of a transitional-invariant lattice. The approach consist in providing a specification of the system in terms of a lattice (with states as nodes and computations as edges) and invariant state predicates. Invariants can define states of the system during and after adaptation, so it is possible to verify the correctness of partial adaptation states; permitting identification of errors early in the adaptation process. At run time, components of the system are modified, if a transitional-invariant lattice is found at the end of the adaptation process, then the adaptation is said to be correct. Similar to our approach, Kulkarni et al. [8] use an abstract representation of the system (*i.e.,* a lattice) for the verification of correctness, and in their case this is verified against system invariants. Our use of a symbolic execution engine differs in two ways. On the one hand, our approach does not require a correctness specification for the system. We are, however, limited to detecting run-time errors ahead of time. On the other hand, using Kusasa, it is possible to evaluate different paths to achieve adaptation, which is not yet possible using transitional-invariant lattices.

Hayden et al. [6] introduce the concept of Client-Oriented specification (CO-spec) for the correctness verification of dynamic software upgrades. CO-specs are used to express interaction between the program and its updates (*e.g.,* backward-compatibility, post-updates, and interface changes). In order to verify CO-specs the program version is merged with its update via defined program transformations. Once the program has been merged off-the-shelf verification tools can be used to check its correctness. Similar to our proposal, Hayden et al. [6] use a symbolic engine (Otter) to simulate the update of the system. The system differs from our approach in two ways. First, to account for incompleteness of symbolic execution, the verification is complemented by a verification tool (Thor). A similar approach could be integrated in our proposal in the future. Second, unlike our approach, the verification of the system takes place once the program has been merged with its update. In that sense the verification using Otter can only signal that there are errors in the merged program. With Kusasa verification takes place before the programs are merged, so that, in case of errors the merging does not take place. Additionally, Kusasa offers the possibility of recovering from errors by using alternative adaptations.

6. CONCLUSION

Run-time adaptation of a system's behavior can lead to erroneous execution states because behavioral adaptations introduced into or removed from the system may violate the contract of its API, or overlook possible interactions with other active adaptations. This paper proposes to verify correctness of behavioral adaptations as their associated contexts are (de)activated. The cornerstone of our approach is the Kusasa symbolic execution engine. Kusasa is triggered upon context activation, generating an execution tree consisting of different branches for each context to be activated. If there are errors identified during the symbolic exploration of the tree, then the context (de)activation is not performed.

The use of Kusasa as a means to verify the correctness of behavioral adaptations is a first approach that tackles fine-grained adaptations and validates for correctness before the adaptation is actually composed with the system.

Nonetheless, we have identified various avenues for future work. First, the technique should be assessed using more comprehensive case studies to validate its real benefit and to measure its performance. Second, more refined means to bound the symbolic execution phase than pre-determined time and depth limits could be explored. Candidates include advanced leasing mechanisms and context-aware bounds. Finally, we could explore the use of assertions to enhance the correctness verification with respect to the behavior introduced/removed by adaptations.

Acknowledgements

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Center (`www.lero.ie`). We thank the anonymous reviewers for their comments on earlier versions of this paper.

References

[1] Anand, S., Godefroid, P., and Tillmann, N. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 367–381.

[2] Appeltauer, M., and Hirschfeld, R. Declarative layer composition in framework-based environments. In *Proceedings of the International Workshop on Context-Oriented Programming* (New York, NY, USA, 2012), COP '12, ACM, pp. 1:1–1:6.

[3] Clarke, L. A., and Richardson, D. J. Applications of symbolic evaluation. *Journal of Systems and Software 5*, 1 (1985), 15–35.

[4] Costanza, P., and Hirschfeld, R. Language constructs for context-oriented programming: an overview of ContextL. In *Proceedings of the Dynamic Languages Symposium* (Oct. 2005), ACM Press, pp. 1–10. Co-located with OOPSLA'05.

[5] González, S., Mens, K., and Cádiz, A. Context-Oriented Programming with the Ambient Object System. *Journal of Universal Computer Science 14*, 20 (2008), 3307–3332.

[6] Hayden, C., Magill, S., Hicks, M., Foster, N., and Foster, J. Specifying and verifying the correctness of dynamic software updates. In *Verified Software: Theories, Tools, Experiments*, R. Joshi, P. Müller, and A. Podelski, Eds., vol. 7152 of *LNCS*. Springer Berlin Heidelberg, 2012, pp. 278–293.

[7] King, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (July 1976), 385–394.

[8] Kulkarni, S. S., and Biyani, K. N. Correctness of component-based adaptation. In *Component-Based Software Engineering*, I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. Wallnau, Eds., vol. 3054 of *LNCS*. Springer Berlin Heidelberg, 2004, pp. 48–58.