The Journal of Systems & Software 198 (2023) 111614

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A framework for analyzing context-oriented programming languages[☆]

Achiya Elyasaf[a], Nicolás Cardozo[b], Arnon Sturm[a,*]

[a] Software and Information Systems Engineering Department, Ben-Gurion University of the Negev, Beer-Sheva, Israel
[b] Systems and Computing Engineering Department, Universidad de los Andes, Colombia

ARTICLE INFO

Article history:
Received 6 July 2022
Received in revised form 14 November 2022
Accepted 7 January 2023
Available online 18 January 2023

Keywords:
Context-oriented programming
Evaluation
Comparison

ABSTRACT

Context-aware systems keep on emerging as an intrinsic part of everyday activities. To cope with such situations, programming languages are extended to support the notion of context. Although context-oriented programming languages exist for over 15 years, they were tested for their suitability for developing context-aware systems only to a limited extent. In this paper, we propose a framework for analyzing the suitability of context-oriented languages from a wider viewpoint. Using this framework, we are able to examine context definition and activation, reasoning capabilities, process aspects of how to work with the language, and other pragmatic considerations. To demonstrate the use of the framework, we apply it to analyze three context-oriented programming languages: ServalCJ, Subjective-C, and COBPjs which represent the major Context-Oriented Programming themes. We evaluate the capabilities of each language using the purposed framework. Developers of context-oriented programming languages can use the framework to improve their languages and the associated development and supporting tools. Furthermore, such analysis can support users of context-oriented programming languages in deciding the language that best suits their needs.

1. Introduction

Context-aware systems gain increased attention and are embedded in every domain of our daily socio-technological life, for example, in health care (Bricon-Souf and Newman, 2007), education (Ireri et al., 2018), and ambient intelligent systems (Ramos et al., 2011). The paradigm of Context-Oriented Programming (COP) emerged to ease the implementation of such systems (Hirschfeld et al., 2008). Following this paradigm, various languages were extended to support the notion of context. These languages include JCop for Java (Inoue et al., 2014), ContextPy for Python (Pape et al., 2016), Subjective-C for Objective-C (González et al., 2011), or context-oriented behavioral programming (COBPjs) for behavioral programming (BPjs) (Elyasaf, 2021; Elyasaf et al., 2019a), to mention a few examples. These languages share the overall objective of dynamic adaptation, but present differences in their realization of such adaptations, the terminology used, and the programming interface offered to users. Moreover, the different concepts used in each language, and the lack of a standard evaluation, make the comparison between languages and their features challenging. The goal of this work is to facilitate the evaluation of Context-oriented Programming Languages (COPLs). The idea behind such a goal is motivated by the following considerations:

- The extent to which COPLs are analyzed is limited, as analysis attempts are dated from over a decade ago (Appeltauer et al., 2009; Salvaneschi et al., 2012a), and only focus on technical issues (Cardozo and Mens, 2022). Thus, a rigorous analysis of COP languages is required from various viewpoints, in particular examining the aspects that refer to their fitness for developing context-aware systems, from the point of view of system developers. Extending the analysis of COPLs from an exclusive implementation point of view.
- The analysis of COPLs focuses on a certain view. For example, Salvaneschi et al. (2012a) mainly focus on layered-based approaches, which indeed are dominating, although other approaches to COP exist, and are left out.

To address these considerations, in this work we propose a framework for analyzing context-oriented programming languages. The framework is composed of a set of criteria to be evaluated, together with a set of guidelines on how to evaluate these. The framework examines the concepts supported by the language, the development process support, and development pragmatics. From the viewpoint of COP developers, such a framework can serve as a standard evaluation for COP languages and to further analyze the existing gaps in the paradigm, and between languages. The results of such analysis can serve users in deciding upon the language (with its supporting tools) that best fits their

☆ Editor: Earl Barr.
* Corresponding author.
E-mail addresses: achiya@bgu.ac.il (A. Elyasaf), n.cardozo@uniandes.edu.co (N. Cardozo), sturm@bgu.ac.il (A. Sturm).

https://doi.org/10.1016/j.jss.2023.111614
0164-1212/© 2023 Elsevier Inc. All rights reserved.

needs. We validate the usefulness of the proposed framework for analyzing COPLs that originated from different paradigms, using three representatives of COPLs. This evaluation confirms that the framework is not biased toward one particular language or implementation technique. In particular, we applied the framework to (1) ServalCJ (Kamina et al., 2013b, 2014) — which follows the layer-based paradigm; (2) Subjective-C (González et al., 2011) which follows the object-oriented programming paradigm; and (3) COBPjs — which follows the Context-Oriented Behavioral Programming (COBP) paradigm (Elyasaf, 2021), originating from the domain of scenario-based programming. The evaluation was done separately by the first two authors, who are experts in the selected languages. The paper extends our previous work (Elyasaf and Sturm, 2021) by means of further refining the framework, extending the evaluation to other COPLs, and generalizing the findings, pointing to possible improvements.

The paper is organized as follows. In Section 2, we describe a smart home example we use throughout the paper for demonstration purposes. In Section 3, we introduce the framework. Sections 4–6 apply the framework to the three COPL exemplars. In Section 7, we discuss the analysis of the framework and the three COPLs. We conclude in Section 8, referring to the usefulness of the framework, and present perspectives for future work.

2. The smart home

We use a smart home system as an example to clarify different aspects within the framework. We begin with defining the basic requirements of this system and then evolve the requirements to allow the evaluation of context-oriented programming languages' maintainability.

2.1. Basic requirements

The basic requirements defined for the smart home consist of the behavior users can observe in some of the rooms. We posit a small comprehensive set of requirements to demonstrate the different features of COP.

1. A building has several rooms of different types, such as kitchen, bathroom, bedroom, living room, or hall.
2. Rooms have movement sensors.
3. Kitchens and bathrooms have two taps: one for hot water and the other for cold water. In addition, they have a button that controls the taps.
4. At night (9pm–8am), vacuuming is prohibited.
5. The effect of pressing the button in a room with taps is:
    (a) pour a small amount of *cold* water three times.
    (b) pour a small amount of *hot* water three times.

2.2. Evolved requirements

The original requirements do not pose restrictions in the order in which actions execute (*e.g.,* for the pouring actions of taps). Consider, for example, that clients now request an additional requirement for the smart home:

6. For kitchen rooms, two pouring actions of the same type (*i.e.,* cold or hot) cannot be executed consecutively.

3. The framework

In the following, we present a framework to analyze COPLs. The framework emerges from COPLs and context-aware systems studies (Appeltauer et al., 2009; Salvaneschi et al., 2012a; Cardozo and Mens, 2022), general programming languages and software engineering evaluation frameworks (Sturm and Shehory, 2003), and from our own experience in this domain. Different from the mainstream COPL research, in this framework, we aim at abstracting the main concepts and concerns of the paradigm, rather than focusing on implementation details. The framework's abstractions can cope with different implementation approaches. Note that while in the framework we emphasize criteria that are of high relevance to COPLs, it could be extended with general software engineering criteria (ISO/IEC 25010, 2011). We divide the framework into four major parts, in light of the possible usages of COPLs: (1) The goal of the language (why and when to use the language), (2) the concepts that should be considered by the language (what are the existing building blocks), (3) the process support when using the language (how to use the language), and (4) the language pragmatics (what is the available support when using the language).

Terminology

COP posits two abstractions to realize dynamic adaptations — *contexts* and *behavior variations*. Contexts refer to meaningful situations from the system's surrounding execution environment (sensed from the environment or internal monitored variables) (Dey, 2001). Behavior variations refer to the specific behavior associated with contexts (Mens et al., 2017), usually implemented as partial methods or mixins.

These two abstractions are described by all COP papers, though there are small differences in the definitions between the different approaches. To unify all COPLs under the same terminology, we posit a terminology that refines some of the COP terminology (summarized in Table 1).

(System) context. The context, or more generally, the system context, represents the internal and environmental system state. For example, the system context in the smart home example includes the states of all system objects, the current time, and the states of the executing behavior. Note that we do not refer here to implementation decisions and that the system context can be either shared by the entire application or not (as in most COPLs (Appeltauer et al., 2009)).

Behavior Context (BContext). A behavior context is a projection of the system context that is relevant for one or more system behavior. For example, the BContext of Requirement 4 is `Night` and therBContext of requirements 5a and 5b is `RoomWithTaps`. Each BContext may have several instances; for example, there may be several rooms that satisfy the condition "room with taps".

Behavioral variation. A behavioral variation (Salvaneschi et al., 2012a) is a context-dependent behavior, *i.e.,* a behavior that is used only under a certain BContext (*e.g.,* requirements 4 and 5) or multiple BContexts (*e.g.,* a behavior in a room during an emergency). Many COPLs use partial methods to represent the context-dependent behavior of a system or part of it. In the framework we are not concerned with the specific implementation details of behavioral variations, but rather are concerned with their association/binding to contexts.

Some COPLs assume a default main behavior and use behavioral variations to refine this behavior. Nevertheless, context-dependent requirements are not necessarily defined this way, and some COPLs do not follow this assumption. Thus, we refer to behavioral variations as context-dependent behavior, to allow types of variations.

BContexts and behavioral variations have attributes, such as *scoping*, *activation*, and are reified using specific implementation techniques, such as *layers*. Since these are attributes and/or implementation decisions for each COPL, we define them under the *Concepts* criteria (Criterion 2).

Table 1
Mapping the framework's terms to COP terms.

Framework term	COP term	Description
System context BContext	Context	The state of the entire system A projection of the system context
Behavioral variation	Behavioral variation	Context-dependent behavior/behavior that is used only under a certain BContext
–	Activation Scope	These are considered as attributes of behavioral variations and are thus not defined as separate terms
–	Layer	An implementation decision (rather than a term) that is relevant only for some COPLs

Criterion 1. The goal of the language

Following the exploration of Context-Oriented Programming Languages, we noticed that these emerged due to various reasons that affect the design of the language. The main reason for the emergence of COPLs is to enable the modification of the behavior of existing, non-COP-based systems. This means that the behavior of a system can be changed due to external factors to that behavior. Additionally, COPLs strive for a clear separation of concerns to support modularity and manage systems' complexity further. In addition to the modularity support provided by programming languages, COPLs aim at adding another level of modularity through the use of context. In that sense, modularity refers to the behavioral view. COPLs, and in particular contexts, can also serve as a bridge for requirements classification/gathering. This means that even before implementation, context can help in gathering related requirements. For example, in the smart home system, the night context can be used to "gather" all behavior requirements occurring during the night. Furthermore, COPLs may aim at shifting the programming paradigm and placing contexts as first-class citizens within the programming language. Such a shift may place contexts at the center of the entire development cycle, giving it a status similar to an object's status in OOP. Note that this goal refers to a change in the development process rather than a technical change in the implementation of a COPL. Since a COPL may address several goals, it is important to consider them when analyzing such a language.

Criterion 2. Concepts

We now present the main concepts in the framework. These concepts abstract the details and requirements of designing COPLs.

Criterion 2.1. Behavior Context (BContext). BContexts are projections of the system context that are relevant for one or more behavioral variations. A BContext has *preconditions* on the system context that define when it is active. In the smart home example, the precondition of the `RoomWithTaps` BContext may be defined as a room with a tap. The preconditions may also refer to behavioral aspects, for example: "the button was pressed three times". A BContext may also encapsulate data to be used by the relevant behavior. For example, an instance of the `RoomWithTaps` BContext may encapsulate the IP addresses of the button and the tap.

A BContext can be specified by two types of data. It can be static, *i.e.,* predefined and constant (for example, kitchen), and it can be dynamic; that is, its structure is defined, yet it is populated at run time. For example, the period of the day (*e.g.,* night), or a specific sequence of open/close operations of the taps.

In the framework, we are interested in examining whether a COPL explicitly refers to BContexts. When analyzing the notion of BContext, we should consider the following:

1. Does it encapsulate data to be utilized by the system behavior?
2. Does it refer to data as preconditions?
3. Does it refer to behavior as preconditions?
4. Can it be used by more than one behavioral variation?
5. Does it support both static and dynamic BContexts?

Criterion 2.2. Inter BContext relationship. Different BContexts may have various types of relationships that can be defined either explicitly or implicitly. In the following, we elaborate on common relationships we identified:

- **Structural** (*Association* and *Generalization*). Association refers to cases in which a BContext is associated with another. For example, a room might be associated with various sensors. Generalization refers to cases in which a BContext is refined. In the smart home system, for example, a `kitchen` is a refinement of a `room`. In such a case, it means that all properties, constraints, dependencies, and relationships that refer to the general context are applied to the specific context unless specified otherwise.
- **Dependency**. This refers to BContexts that rely on others. For example, the context `LockedRoom` depends on the existence of a `Room` context.
- **Temporal**. This refers to temporal relationships between BContexts. For example, the order according to which the `Day` and `Night` BContexts are activated.
- **Priority**. In some cases, several BContexts are simultaneously active, which may result in several active context-dependent behaviors. This may be problematic if two contradicting pieces of behavior are active at the same time. For example, one behavior states that rooms should be locked at night and another one states that rooms should be unlocked in case of emergency. Thus, there is a contradiction if there is an emergency at night. There are many ways to overcome such contradictions, for example, by defining the priority of each BContext. In our case, the `Emergency` BContext can be set with a higher priority than the `Night` BContext. While priorities may be dangerous as they require a cross-aspect perspective on the system, it is still a popular solution (as in rule-based systems). Yet, other solutions may be more robust to continuous changes, such as explicitly defining the behavior when two BContexts are simultaneously active.
- **Constraints**. All the previous relationship types, except for structural, imply constraints on the occurrences of BContexts. The constraints relationship is a generalization of these previous relationships and includes other constraints of BContexts; for example, there can be only one occurrence of the `Night` context.

Criterion 2.3. The BContext life cycle. An important observation is that BContexts have both specification and occurrences (*i.e.,* instances) determined by their life-cycle. For example, the `Room` BContext may have several occurrences, one for each room in the house.

- *BContext Specification* refers to its definition. This activity is usually done before the system runs, though it can also be done at run time, as in self-adaptive systems. The specification includes the data related to the BContext and its associated preconditions. In the smart-home example, we can specify the following BContexts (not necessarily for the current requirements): `Night`, which is determined by a precondition on time; `LockedRoom`, where the precondition can be specified using a boolean attribute of a Room; or `PersonInTheRoom`, which is activated upon the occurrence of three sensor events of movement detection (to avoid false detection).
- *BContext Activation* refers to the instantiation of a BContext, based on its preconditions, as determined by its specification. This means that the system tracks the state of the system context and the set of preconditions. Once the state of a BContext is sensed to satisfy its preconditions, a new occurrence of that BContext is activated. Note that in reaction to the BContext activation, the behavior bound to this BContext should be considered. For example, when a new occurrence of the `night` BContext is activated, the lock behavior should be executed (unless there are conflicts).
- *BContext Deactivation* refers to the point at which a BContext instance is no longer relevant. This deactivation may: (1) have no effect on the current behavior; (2) trigger a new behavior; (3) deactivate behavioral variations that are bound to this instance (see Criterion 2.5); or (4) change the behavior of behavioral variations that are bound to this instance.

Criterion 2.4. Behavioral variation. This corresponds to a context-dependent behavior that is used only under a certain BContext. The behavioral variation may also be used under a set of BContexts, for example, the behavior in a room during an emergency. This can be implemented as a BContext with multiple preconditions, however, some COPLs may allow binding behavioral variations with multiple BContexts.

The following list summarizes the COPL aspects that need to be considered:

1. Does it require a default behavior?
2. Can it be bound to multiple BContexts?
3. Is it executed for each BContext instance?
4. Is it scoped to specific BContext instances? If not, can it be scoped to other parts of the system?
5. Can it access the data encapsulated in the BContext instance(s)?
6. How is the order of execution decided when multiple behavioral variations are simultaneously active? Is it random or does it have to be explicitly specified? Is it decided prior to the execution or dynamically?
7. Is it specified as a single partial method/mixin or multiple?

Criterion 2.5. The behavioral-variation life cycle. Behavioral variations have the same life cycle as BContexts.

- *Behavioral-variation specification*. The specification of behavioral variations activity is usually done before the system runs, though it can also be done at run time (*e.g.,* using dependency injection or emergence (Cardozo, 2016; Cardozo and Dusparic, 2021)). The specification includes the behavior itself (*e.g.,* partial methods) and the binding to a specific BContext. In particular, it is important to understand if all behavioral variations that are bound to the same BContext are grouped together (*e.g.,* under the same layer) or are defined using other idioms (*e.g.,* inside classes).
- *Behavioral-variation Activation*. In COPLs, BContexts and their behavior are closely related. Thus, there is a need to discuss the relationship between BContext activation and behavior activation. When analyzing a COPL, it is beneficial to explicate the differences and relationships among these activations. In particular, the following should be discussed: (1) Does the activation of a BContext automatically activate the behavior? (2) Should behavior be explicitly activated?
- *Behavioral-variation Deactivation*. Similar to behavior activation, behavior deactivation can also present differences between COPLs. As a behavioral variation becomes inactive, three situations can occur (Cardozo et al., 2011a): (1) The execution can be interrupted to promptly effectuate the context change and apply the remaining active BContexts; (2) In addition to the prompt execution, clean-up tasks can be executed to avoid reaching inconsistent or erroneous systems states due to the interruption; (3) The execution can be loyal, and wait until the specialized behavior finishes its execution, and then effectuate the context change.

Criterion 3. Process support

Process support is extremely relevant when evaluating COPLs as it determines how to work with the language in the most proper (abstract) way, instead of focusing on the technicalities. When referring to the process of using a specific COPL, we find it important to examine the following points:

Criterion 3.1. Method and guidelines. Are there guidelines or a predefined methodology to guide programmers on the development with the COPL?

Criterion 3.2. Reasoning. The interactions between the system behavior and the system context may produce unexpected behavior that is hard to foresee. This may result in inconsistencies in the application logic, such as the inconsistency between the `Night` and `Emergency` BContexts described before. This problem is also known as *behavioral consistency* (Cardozo et al., 2012b; Salvaneschi et al., 2012a; Cardozo et al., 2015). As previously explained, one way to address such a contradiction is by specifying context constraints, either explicitly or implicitly. Yet, there are two sub-problems with this solution: First, we need a way to identify such inconsistencies and pinpoint their root cause. Second, the constraints may result in another undesired behavior. Thus, it is essential to be able to either verify the correctness of explicitly defined constraints or generate them automatically.

Reasoning techniques can effectively address these two sub-problems. Therefore, we are interested in examining whether there are development-time and run-time facilities for checking the correctness of (1) the basic application logic; and (2) the activation mechanisms.

Criterion 3.3. Testing. The COP paradigm introduces challenges in testing such programs. This is due to multiple BContexts and behavioral variations that may affect the system's behavior. Thus, it is important to understand whether a COPL: (1) has a testing framework associated with it, and (2) proposes guidelines on how to test the programs.

Criterion 4. Pragmatics

Pragmatics refers to the practical aspects of deploying and using a COPL. We believe that if a language's supporting tools and capabilities are adequate, it may well serve its goals and further indicate its maturity. We suggest examining the following aspects:

Criterion 4.1. Usage. In what context is the language being used? Academic? Industrial? Only by the authors? Do additional users actually use the language?

Criterion 4.2. Resources. What resources are available for using the language? (1) Are there any specific editors to support the language? (2) Is there a forum that supports the language? (3) Are there tutorials, videos, documentation, and APIs to support the language usage? (4) How does one debug a context-oriented program? (5) Are there other supporting tools? For example, are there development and run time IDE extensions to allocate BContexts and their bound behaviors? (6) Are there any libraries supporting the development of context-oriented applications?

Criterion 4.3. Required expertise. What is the required background for learning the language? Usually, COPLs are based on existing general-purpose languages. Yet, there might be languages that use special idioms to implement the sought features. Relying on existing language may require further learning, yet users may use libraries available in the host language ecosystem.

Criterion 4.4. Domain applicability. Is the COPL suitable for a particular application domain (*e.g.,* real-time or web applications)? Are there any application domains that are not suitable?

Criterion 4.5. Scalability. Can the language and its execution environment cope with large-scale applications?

Criterion 4.6. Maintainability. It may be difficult to organize a COPL codebase without compromising the maintainability and separation of concerns. This is because the main modularization direction of the underlying programming paradigm (*e.g.,* classes in OOP) is often orthogonal to the context-depended behavior, as it may be an aspect that crosscuts the application logic (Salvaneschi et al., 2012a). In light of this problem, we are interested in understanding how the language copes with issues related to code maintainability. These include: (1) How does the introduction of new requirements affect the code? what are the types of required changes? (2) How do programmers identify the binding between behavior and their contexts? (3) What are the consequences of BContext refactoring? and (4) How do programmers identify places to introduce changes (which behavior and associated context should be changed)? What are the effects of these changes?

Criterion 4.7. Language dependability. When adopting a COPL, developers need to understand whether it changes essential elements of the base language that might be used by external frameworks and libraries.

Criterion 4.8. Evaluation. When selecting a language to work with, it is always beneficial to look for evidence of its advantages. Thus, it is important to understand what kind of feedback, lessons learned, and understandings exist. For example, is there any evaluation of the COPL? User study? Performance benchmark? Controlled experiments?

4. Applying the framework to ServalCJ

To start the analysis of COPLs using our framework, we apply it to the ServalCJ (Kamina et al., 2013b, 2014) language, formerly known as EventCJ (Kamina et al., 2011). ServalCJ is selected as a representative of layer-based COPLs, one of the main implementation techniques followed by COPLs. The reason for choosing this language for our analysis is two-fold. On the one hand, the inception of COP languages arose with a proposal based on layer-based context orientation (Costanza and Hirschfeld, 2005). On the other hand, ServalCJ is the most robust and active language in this family of languages.

4.1. ServalCJ in a nutshell

ServalCJ (Kamina et al., 2014) is a Java-like language that supports COP, implemented using an extension of the AspectBench compiler (Allan et al., 2005) to enable dynamic adaptations through layer activation. The language supports an implicit activation mechanism enacting layers in response to events or conditional predicates. In particular, ServalCJ offers extended support for multiple layer activation mechanisms (*i.e.,* implicit, explicit, event-based) using a reactive programming style implementation and evaluation based on reactive values (Kamina et al., 2017). This variant of COP allows a layer to be (de)activated immediately when its associated activation conditions change. ServalCJ supports two scope strategies: per-object and global layer activations.

In ServalCJ, BContexts are named 'contexts' defined as standalone predicates specifying the conditions in which the context is active. For example, in the smart home scenario, we can define a named context `Kitchen` to be applicable whenever a user's location is detected to be in that room, as shown in Lines 3–6 of Snippet 1. Additionally, BContexts can be defined as logic predicates specifying the combination of other BContexts. For example, we define a named context `Hot` taking place whenever the `RoomWithTaps` context is active and the `Cold` context is not (Lines 8–11).

```
 1 // Context definition (BContext) and conditions
 2 // to observe the different contexts
 3 Kitchen(Location loc, Geocoder gcd) :: gcd
 4   .getFromLocation(loc.getLatitude(),
 5            loc.getLongitude())
 6   .get(o).getLocality().equals("kitchen")

 8 RoomWithTaps :: Kitchen || Bathroom

10 Hot :: RoomWithTaps && !Cold
11 Cold :: RoomWithTaps && !Hot
```

Snippet 1: System and BContext definition example for the smart home example in ServalCJ.

ServalCJ follows the layer-in-class model for the definition of layers and behavioral variations, in which the main modularization abstraction of the language are classes and layers. As shown in Snippet 2, the behavioral variations associated with a class are defined as layered methods inside said class. Layers are associated with BContexts through their name. In Criterion 2.5, we present a concrete example for the definition of behavioral variations (Snippet 4) for our smart home example.

```
class AdaptableObject {
  layer ContextName {
    //new behavioral variations
  }
}
```

Snippet 2: Definition of class modules with their layers inside.

Behavioral variations defined in layers become active at run time as the contexts themselves are (de)activated. ServalCJ offers different mechanisms to delimit (*i.e.,* scope) the effect of active contexts. ServalCJ activates contexts in a declarative fashion from their specification. These declarations cluster together all contexts relevant to specific environment variables in a **contextgroup**. Context groups can define the scope of availability if the activations are specified within. In Snippet 3, the context group is defined as global, which leads to its contexts impacting the whole system whenever their activation conditions are satisfied.

```
global contextgroup SmartHome(Location loc,
  Geocode) perthis(this(Location,
                  Geocode)) {
 activate Kitchen if(
  gcd.getFromLocation(loc.getLatitude(),
              loc.getLongitude())
  .get(o).getLocality().equals("kitchen")
 );
 ...
 activate Cold when RoomWithTaps && !Hot;
}
```

Snippet 3: System and BContext definition example for the smart home example in ServalCJ.

4.2. Analysis of ServalCJ

Criterion 1. The goal of ServalCJ

ServalCJ is proposed to enable COP inside the JVM. In particular, it enables the development of context-aware server and client applications in a modular fashion. To do this, ServalCJ introduces the concepts of contexts as new program entities to represent the situations from the surrounding execution environment, and layers as new modules, defined inside classes, to specify the behavioral variations associated with contexts. Furthermore, to manage the complexity of context activation, ServalCJ focuses on the unification of different scoping and activation semantics (Kamina et al., 2013b, 2016, 2017), taking into account, implicit, explicit, event-based, and reactive activation models, in which contexts (and their behavior variations) have an effect globally, locally, or in the dynamic extent of a layer. Such definitions bring the activation method of contexts to the foreground through different language-level abstractions that can be combined in a modular way. Therefore, the goal of ServalCJ is to be a COPL in which all situations that require dynamic adaption to the context are expressed in a natural way (*i.e.,* clearly modularized) and managed uniformly and transparently by the developer. This effectively leads to a more flexible adaptation model.

Criterion 2. COP concepts

Criterion 2.1. BContext. BContexts in ServalCJ represent specific situations from the surrounding execution environment and are defined in one of two categories: physical and logical contexts. Physical contexts correspond to sets of external or internal variables sensed from the surrounding environment. Such variables are used to define the conditions in which the context is active. Logical contexts are not sensed directly from the environment but are defined as combinations of BContexts, to facilitate expressing specific situation combinations. In ServalCJ, BContexts are defined in two stages. First, we must provide a specification of the contexts stating the conditions in which they are considered to be present in the environment; such conditions are described as a predicate in the function of the relevant environment variables (*i.e.,* BContexts' dynamic data type). An example of the specification of BContexts is shown in Snippet 1, first with an example of physical contexts (Line 6), and then an example of logical contexts (Lines 8–11). Second, we must provide the implementation of contexts' specification, as shown in Snippet 3. In the implementation of contexts, we first group together contexts related to the same set of environment variables, to increase modularity. Additionally, we define the activation conditions of contexts — that is, the scope in which contexts are valid once the conditions of their specification are met. In our example, when the context group is defined to be global, the BContexts therein will have an effect on the entire system.

Note that in ServalCJ, BContexts do not contain any behavior. All behavior associated with a context is defined as behavioral variations.

Criterion 2.2. Inter BContext relationship. In ServalCJ, it is possible to define dependency relations between BContexts as part of their specification. Dependencies between BContexts are defined through logic predicates, specifying the BContext as a combination of other BContexts. The BContext defined as dependent on other BContexts only becomes available if the predicates defining the corresponding BContexts are valid in accordance with the logic predicate definition. In Snippet 1, Lines 8–11 define dependencies that must be satisfied between BContexts by combining existing BContexts, as is the case for `RoomWithTaps` and the `Hot` and `Cold` BContexts. In particular, the relationship defined between the latter BContexts states that only one of these can be active at a given time. With the definition of these relationships, ServalCJ eases the management of BContexts as it is not the responsibility of programmers to ensure that the logical conditions between contexts are satisfied (*e.g.,* no accidental activation of both BContexts occurs), but rather to define the restrictions in the specification predicates.

Note that ServalCJ can define structural relationships between BContexts implicitly, as part of the specification of contexts, and the use of these BContexts in context groups (*e.g.,* defining BContexts for classes in an inheritance hierarchy).

Finally, BContexts in ServalCJ define an implicit relationship following their activation rules. That is, if two contexts defining adaptations for the same behavior are currently active, the observed behavior in the system corresponds to that associated with the BContext that was activated last. Such implicit interaction between BContexts assures that the observed system behavior will be the most appropriate to the observed environment conditions.

Criterion 2.3. The BContext life cycle. The life-cycle of ServalCJ programs is managed through the activation of layers as part of a `LayerStack`.

Whenever a context is sensed in the surrounding environment, following the activation conditions defined for BContexts in a context group (Snippet 3), the corresponding BContext is added to the top of the `LayerStack`. The way BContexts are managed within the stack depends on the activation model used for the BContexts (Kamina et al., 2013b).

Event activation In event-based activations, BContexts use the `activate` abstraction to define the conditions and the extend of its activation. The conditions are associated with system variables gathered from its surrounding environment, combined through a predicate. The activation of the context will remain so long the predicate is valid, as shown in Snippet 3. That is, the BContexts are pushed to the `LayerStack` whenever their predicate definition becomes valid, making all behavioral variations associated with the BContext immediately available.

Control flow ServalCJ introduces the `cflow` construct in the definition of BContexts to ensure the execution of a given behavior takes place in a particular context (*i.e.,* executes refined behavior). In this case, the `cflow` construct pushes the BContext into the `LayerStack` upon entrance to the construct, and pops it out of the stack, when the execution control flow exits the `cflow` construct. Moreover, when executing refined behavior using `cflow`, the corresponding BContext is assured to remain in the `LayerStack` throughout the execution of the functions inside the construct and their corresponding callbacks.

Whenever the context is no longer sensed in the environment and the BContext is deactivated, the corresponding layer is removed from the `LayerStack`, immediately deactivating the behavioral variations associated with the BContext. The deactivation of a BContext effectively changes the system behavior by reverting to the original behavior before the activation of the BContext.

ServalCJ allows users to define the effect of BContexts on the entire system, specific object instances, or given execution threads as part of the definition of context groups. Such specifications are given as part of the definition of context groups, respectively annotating the context groups with `global`, or the `perthis` and `pertarget` definitions in the object specifications to obtain the desired behavior. The combination of these definitions inside context groups offers developers the flexibility to combine different BContext activation and scoping methods as part of the same application, in response to the context requirements of the environment.

Criterion 2.4. Behavioral variation. Once BContexts are defined, ServalCJ associates them with their refined behavior, defining behavioral variations, as shown in Snippet 4.

Such refinements are defined as layered methods inside classes in ServalCJ. The base system behavior is defined as regular methods in a class as shown for the behavior of a room with taps in Snippet 4. If such methods have adaptations, then we add the definition of the adaptation inside a layer, shown in Lines 6–12 (*i.e.,* purring cold water three times). Note that a class can adapt the same behavior in different BContexts (as shown in the snippet) and that a BContext can adapt multiple methods by adding their implementation inside the corresponding layer to the BContext.

Behavioral variations refine the behavior given originally in an object, but can also be used to introduce new behavior to objects. Moreover, it is possible to introduce properties to objects inside behavioral variations. The base behavior given in an object can be extended in a behavioral variation specifying the behavior in the layer with the `before` of **`after`** annotations. New behavior or complete behavior re-definitions are annotated as **`around`** behavior in layers. The annotations for behavioral variations in a layer follow the semantics of point-cuts in aspect-oriented programming (Masuhara et al., 2006).

In ServalCJ, the relation between BContexts and behavioral variations is many-to-many. That is, one behavior can be refined in many different BContexts, and a BContext may refine many different methods for many objects. Moreover, implicitly, it is possible to define a behavioral variation for multiple BContexts at the same time, by defining a new BContext as a dependency relationship between other BContexts. Given that the same behavior can be refined within many different behavioral variations, at run time, the observed behavior will correspond to that associated with the BContext that was activated the latest (*i.e.,* pushed last to the `LayerStack`).

Criterion 2.5. The behavioral-variation life cycle. After their initial specification, behavioral variations are defined as part of a layer inside a class. Inside the layer, the behavior itself is defined as a partial method, with the refinement of the corresponding behavior. As in Snippet 4, we can define a single partial method per layer, or multiple partial methods, according to the relevance of the methods to the layer and class of the definition.

The activation and deactivation of behavioral variations in ServalCJ follow a prompt-loyal approach (Cardozo et al., 2011a; Aotani et al., 2014). Whenever a BContext is activated, all behavioral variations associated with the BContext become active as well. When the BContext is deactivated, the effect of the deactivation depends on the activation model of the BContext. Activated as an event, the behavioral variations associated with the BContext are deactivated as soon as the BContext is. Activated using the control flow, the behavioral variation is deactivated once it exits the execution control flow.

Additionally, note that the activation and deactivation of behavioral variations extend in accordance with the definition of the BContext in its context group — that is, whether the activation has an effect globally or locally, and whether it extends or redefines the behavior.

```
 1 class Kitchen {
 2   public void press() {
 3     Event("pour");
 4   }

 6   layer Cold {
 7     around public void press() {
 8       Event("cold");
 9       Event("cold");
10       Event("cold");
11     } // Requirement 5a
12   }

14   layer Hot {
15     around public void press() {
16       Event("Hot");
17       Event("Hot");
18       Event("Hot");
19     } // Requirement 5b
20   }
21 }
```

Snippet 4: Behavioral specification (logic layer) of the hot-cold system in ServalCJ.

Criterion 3. Process support

Criterion 3.1. Method and guidelines. As with most COPLs, there is no specific method to develop ServalCJ applications. Behavioral variations and their associated BContexts are defined in an ad-hoc fashion in response to the situations in the surrounding environment that require refined behavior. In the interaction between BContexts and the definition of the activation method to use, ServalCJ offers guidelines through the research articles introducing the language's characteristics. These guidelines offer examples of the different possible situations in which BContexts may interact. Such examples can be later extrapolated by developers for their own applications.

Criterion 3.2. Reasoning. The reasoning capabilities of ServalCJ are divided into two aspects. First, ServalCJ uses the logic predicate definitions to verify the consistency of transitions between BContexts (Kamina et al., 2013b). Here, the reasoning over the logic predicates and their transition is used to ensure the correct behavior of the dependency relations between BContexts, assuring the consistent combination of BContexts.

Second, alongside the development of ServalCJ, its creators have focused on the definition of the formal methods for the language. ServalCJ comes together with calculus for the upfront verification of different properties of the language (Aotani et al., 2011). In particular, the calculus is oriented toward the soundness of the unified activation mechanisms for BContexts (*i.e.,* event-based, imperative, control flow) (Kamina et al., 2013a, 2018).

Criterion 3.3. Testing. Currently, there are no specific testing techniques implemented for ServalCJ, that can validate the correct behavior of BContexts and their behavioral variations. Rather, these are verified upfront using the language's calculus and the BContext transitions.

Criterion 4. Pragmatics

Criterion 4.1. Usage. ServalCJ has been used only in academic settings with the development of proof-of-concept applications and exemplars used to present the usefulness of new language features. The language is no longer under development, although the ideas of the language are being used in a reactive language developed by the authors of ServalCJ (Kamina and Aotani, 2019).

Criterion 4.2. Resources. The main resource associated with ServalCJ comes from its research articles. Additionally, there is a GitHub repository[1] containing the language's compiler implementation and one of its recent case studies.

Criterion 4.3. Required expertise. The basic usage of ServalCJ requires knowledge of its base language, Java. However, understanding the language more in-depth requires knowledge of advanced language features such as layers, events, control flow, and advises and point-cut modifiers from AOP.

Criterion 4.4. Domain applicability. The objective of ServalCJ, as a COPL implemented on the JVM, is that the language can be used for any type of application that can run on the JVM. Typically, the application domains covered in ServalCJ's research papers cover desktop, server, and web applications.

Criterion 4.5. Scalability. Up to the date of writing this article, there is no evaluation that takes into account the scalability of ServalCJ. The scale of the exemplars used has been small to mid-size programs.

Criterion 4.6. Maintainability. There are no specific maintenance tasks related to ServalCJ programs. ServalCJ modules are ruled by the OO characteristics of its base language, Java. However, as with other COPLs, the introduction or variation of features becomes straightforward, as it is only necessary to define the corresponding BContext and behavioral variation for the desired feature (in the corresponding classes/modules). Similarly, bug fixes should be localized to a specific (refined) method, not requiring further changes than those already required for OO programs.

Criterion 4.7. Language dependability. ServalCJ has a dependency on both Java and the AspectBench compiler, abc (Allan et al., 2005). The syntax of ServalCJ resembles that of Java, for the most part, borrowing abstractions of AspectJ (*i.e.*, `before`, `after`, `around`) for the definition of the applicability of the behavioral variations associated with BContexts. Changes to the language are made directly in the ServalCJ compiler, an extension of abc.

Criterion 4.8. Evaluation. ServalCJ has not yet been used for empirical evaluation of COP features, nor is there a user study or performance benchmark of the language. Moreover, ServalCJ was not included in the existing surveys of COP, as they predate the language. All evidence of the language's benefit is part of the evaluation of the research articles about the language.

5. Applying the framework to Subjective-C

To further test our proposed framework, we next apply it to Subjective-C. Subjective-C serves as a representative for context-based COPLs, a second family of COPLs in the existing literature, characterized by behavioral variations and explicit activation. Subjective-C[2] (González et al., 2011) is a full context-oriented language extension of Objective-C, introducing the notion of contexts, behavioral variations, a context manager, a context discovery module, the current context, and context-dependency relations.

5.1. Subjective-C in a nutshell

Subjective-C realizes context-dependent adaptations by means of contexts, their associated behavioral variations, and context activation. As such, an adaptation consists of a set of methods observable in a particular (context) situation. Contexts (*i.e.*, BContexts) are defined as first-class entities of the system. BContexts are object extensions declared using the `@context(name)` language abstraction. For example, in the smart home scenario, we could define a context with the name kitchen as `@context('Kitchen')` to represent a situation in which users are in a kitchen room. BContext definitions are given in an independent Subjective-C module, called the *context discovery* module (Cardozo, 2013). Snippet 5 shows the definition of BContexts in the context discovery module of Subjective-C.

```
@context("Hot")
@context("Cold")
@context("Bathroom")
@context("RoomWithTaps")
@context("Kitchen")
```

Snippet 5: BContext definition for the smart home example.

The behavior associated with a context represents the refinement (or introduction) of some system behavior (*i.e.*, method). Similar to ServalCJ, in Subjective-C such refinements are defined as methods of the object they refine, as shown in Snippet 6. In this case, refined behavior is part of the object's implementation defined using the annotation of the BContext's name in which the behavior is applicable.

```
@implementation AdaptableObject
- (void) method() {
   //base behavior
}

@contexts BContextName
- (void) method() {
   //new behavioral variations
}
@end
```

Snippet 6: Behavior refinement definitions.

Note that any system object can define context-dependent behavior for as many contexts as different situations that require adaptation exist. At run time, the behavior associated with a BContext is dynamically composed with and withdrawn from the system, as the BContext becomes active or inactive. BContext activation and deactivation are managed by an internal system entity, the *context manager*, that keeps a representation of all BContexts in the system, and a representation of the currently active BContexts (*i.e.*, the *current context*).

[1] https://github.com/ServalCJ

[2] https://released.info.ucl.ac.be/Tools/Subjective-C

```
 1 // Environment conditions (BContext) to observe the different contexts
 2 if(Calendar.current.component(.hour, from: Date()) > 19)
 3   @activate("night")
 4 if(Calendar.current.component(.hour, from: Date()) > 6)
 5   @deactivate("night")
 6 let fetchLocationRequest = EILRequestFetchLocation(locationIdentifier: "Kitchen")
 7 fetchLocationRequest.sendRequest { (location, error) in
 8   if let location = location {
 9     self.location = location
10     @activate("Kitchen")
11   } else if let error = error
12     @deactivate("Kitchen")
13 }
14 // System context definitions
15 @context("Night")
16 @context("Room")
17 @context("RoomWithTaps")
18 @context("Kitchen")
19 @context("Bedroom")
20 @context("Bathroom")
```

Snippet 7: System and BContext definition example for the smart home example.

Behavior refinements become active in response to the situations from the surrounding environment by means of the `@activate('Kitchen')` abstraction. That is, whenever a specific situation is sensed, for example, when the user is in the kitchen room, the activation abstraction is called, and the associated behavior refinements defined in each module are activated. Whenever the situation is no longer sensed, we trigger the `@deactivate('Kitchen')` abstraction, reverting to the base behavior.

As BContexts become active, they may interact in different complex ways with each other, as we reuse the behavior associated with the currently active contexts, making sure the system behavior is the most appropriate to the situations from the surrounding execution environment.

5.2. Analysis of Subjective-C

Criterion 1. The goal of Subjective-C

The objective of Subjective-C is to enable COP for mobile devices while keeping the context-dependent behavior separated from the program's base behavior. Subjective-C is the first COP language that directly targets mobile platforms to take advantage of devices' onboard sensors to realize mobile and ubiquitous computing. The goal behind this objective is to enable the development of highly flexible and dynamic software systems, that are highly modular, adaptable, and modifiable.

Criterion 2. COP concepts

Criterion 2.1. BContext. The definition of BContexts in Subjective-C is divided into two. First, we define the entities that represent sensed (external) and monitored (internal) variables (*i.e.,* data) from the surrounding execution environment. We see the definition of six such entities for the smart home example at the end of Snippet 7. Data representations are organized structurally in Subjective-C, therefore, it is possible to reuse data from the system context. For example, the entities in Lines 16 and 17 of Snippet 7, can be reused by the subsequently defined entities by means of delegation relations. Concretely, in the smart home example, the `@context('Room')` BContext is used to abstract the behavior common to all rooms. The `@context('Bathroom')` BContext further refines the behavior of rooms for bathrooms.

Another characteristic of BContexts in Subjective-C is that they can refine objects' data. That is, it is possible to introduce, shadow, or modify the state variables of an object as part of the refined behavior associated with a context.

In Subjective-C, BContexts extend the definition of Objective-C objects. Each of the BContexts in Snippet 7, defines an object entity. However, the BContexts' states are exclusively defined by their identifying `name`; their definition is isolated from the system.

Second, together with BContexts' definition, developers must independently define the activation preconditions for each of them. Such conditions determine the situations in which a BContext is sensed to be active or not, with respect to the data from the surrounding execution environment. Whenever the conditions sensed from the environment are satisfied, the associated behavior is observed. In our example, in Snippet 7, Lines 6 to 13 specify the situation in which, whenever a user is sensed in the kitchen, then the BContext defined for that room becomes active.

Criterion 2.2. Inter BContext relationship. Requirement 6 refines the behavior for kitchen rooms. Pragmatically, we could make sure the behavior associated with 'Cold' and 'Hot' cannot take place at the same time, by ensuring the two corresponding system contexts are never active at the same time. This can be a very cumbersome task as the amount of system contexts increases, and their interactions become more complex.

Subjective-C contexts may interact in two different ways. Implicitly, BContexts interact according to their *context activation time*. That is, as BContexts are activated, their associated behavioral variations are composed with the system. If two contexts defining behavioral variations for the same base behavior are active, the observed behavior will be that of the BContext activated the latest. Such implicit relationship is defined in response to the use of explicit priorities, as hard-coded restrictions can generate undesired interactions. Moreover, implicit relationships between BContexts respond to the notion of COP to provide the most appropriate behavior to the situation at hand, in this case, the behavior associated with the BContext sensed most recently from the surrounding execution environment. If a new BContext is activated, then its refined behavior will be the observed behavior. Similarly, if a BContext is deactivated, then the observed system behavior becomes that of the currently active BContexts; excluding the BContext just deactivated (*i.e., the current context*).

Together with the implicit relationships between BContexts, Subjective-C also offers an explicit relationship method between context objects by means of *context dependency relations* (González et al., 2011; Cardozo et al., 2012b). Context dependency relations define a hierarchical structure between contexts. In the case of

Requirement 6, we can define an exclusion dependency relation between the Cold and Hot contexts, assuring the contexts are not active at the same time, and as a consequence making their behavior independent. The specification of the *exclusion dependency relation* is shown in Snippet 8.

```
[addExclusionBetween: Cold and: Hot]
```

Snippet 8: Exclusion dependency relation preventing the Cold and Hot contexts to be active at the same time.

Similarly, in the case of the structural associations between contexts explained in Snippet 5, the `@context('Room')` and `@context('Bathroom')` contexts can define an *implication dependency relation* between them to express that the behavior variations associated with the Room BContext should be available whenever the Bathroom BContext becomes active (implicitly triggering the activation of the former context). Such relationships define constraints over the activation/deactivation of a context with respect to the activation/deactivation of other contexts.

Criterion 2.3. The BContext life cycle. The life-cycle of Subjective-C programs is managed by a *context manager*. The context manager is the system entity in charge of keeping an active representation of the current context (*i.e.,* the combination of all active BContexts), activating and deactivating BContexts, and assuring the satisfaction of the context-dependency relations.

The context discovery module actively monitors the environment for changes in its sensed variables. In response to such changes, following behavior similar to that shown in Lines 2–13 in Snippet 7, a message is sent to the context manager to effectively enact the desired action (activate or deactivate a BContext). The context manager keeps a representation of all currently active contexts (a graph), activating and deactivating the requested components of the system context if and only if their corresponding data satisfies the restrictions imposed by the context dependency relations. Once a BContext is activated, the context manager is in charge of composing the behavior associated with it within the system, so that future messages use the refined behavior. The deactivation of BContexts triggers the deactivation of all behavioral variations associated with the BContext (*i.e.,* the behavior refinements) via the context manager. This behavior effectively reverts back to the behavior observed before the BContext activation. Note that the changes when activating and deactivating BContexts correspond exclusively to changes to the method implementations used at run time. Therefore, the changes are transparent to the state and only observable to users as the behavior is called.

In Subjective-C, BContext activation and deactivation are ruled by context-dependency relations. That is, a BContext is effectively activated or deactivated only if it satisfies the conditions stated by the dependency relations. After the dependency relations are verified, BContext activation is prompt (Cardozo et al., 2011a). As soon as the activation is fulfilled by the context manager, the refined behavioral variations associated with the BContext are composed into the base system. In turn, BContext deactivation is prompt-loyal (Cardozo et al., 2011a). As soon as the deactivation is fulfilled by the context manager, the behavioral variations associated with the BContext are withdrawn promptly from the base system. However, if the currently executing method corresponds to a behavioral variation, then it will finish its execution loyally, while calls to any other methods will not take into account the behavioral variations associated with the BContext.

Criterion 2.4. Behavioral variation. Behavioral variations are detached from their defining entity or associated BContext in

```
@implementation Kitchen
- (void) press() {
  Event('pour')
}

@contexts Cold
- (void) press() {
  Event('cold')
  Event('cold')
  Event('cold')
} // Requirement 5a

@contexts Hot
- (void) press() {
  Event('Hot')
  Event('Hot')
  Event('Hot')
} // Requirement 5b

```

Snippet 9: The behavioral specification (logic layer) of the hot-cold system in Subjective-C.

Subjective-C. That is, each behavior is defined independently from any other behavior and then associated with one or many BContexts. Moreover, in Subjective-C it is possible, similar to variables, to introduce, modify, or shadow any desired behavior. Subjective-C enables contexts to have an effect globally on the system as a default behavior, but can also specify local adaptations to a particular thread (Cardozo et al., 2012a). Global activations/deactivations follow the abstractions in Snippet 7 (*e.g.,* `@activate('Night')`). Local activations in Subjective-C are specific to an execution thread. Whenever local activations are used, the behavior associated with the active BContext will be available only to the specific activation thread. The language abstraction for local activation is defined as `@activate('Night' in thread)`, activating the Night BContext specifically for the thread, thread, given. A similar abstraction exists for local deactivations.

In Subjective-C, BContexts and the conditions in which they take place are always defined statically. However, the activation and deactivation of contexts respond to the dynamic conditions of the surrounding system environment, as these are sensed or monitored. The behavior observed in the system corresponds, by default, to that of the last BContext activated. For example, Lines 2 to 13 in Snippet 5 show the activation and deactivation of BContexts in response to sensed situations in our running example. In this regard, the BContext type is dynamic. Nonetheless, in Subjective-C it is also possible to in-line the `@activate` and `@deactivate` constructs within the base system behavior to ensure a context-dependent behavior takes place at a specific moment in the system execution. In that regard,BContexts can also be considered static.

Criterion 2.5. The behavioral-variation life cycle. The execution of Subjective-C programs follows an imperative execution style. However, the final behavior observed may change between different executions as the conditions of the environment may change, and as a consequence, activate different context sequences. In Snippet 9, for example, pressing the button on the kitchen under normal conditions generates a `Event('press')`. However, executing the same action in a situation in which the Hot context is active, results in observing the `Event('Hot')` event three times.

The activation and deactivation of behavioral variations are tightly coupled to that of BContexts. As a BContext is activated, all behavioral variations associated with it are made available promptly in the system. Similarly, as a BContext is deactivated, all its associated behavioral variations are withdrawn from the system execution promptly but remain loyal to the termination of the currently executing function.

Criterion 3. Process support

Criterion 3.1. Method and guidelines. The process of developing COP systems with Subjective-C is ad hoc. Developers are in charge of defining the contexts they deem more appropriate, their associated behavior variations, and the conditions in which contexts must take place. Normally, such definitions start from the base system behavior and them proceed with the desired adaptations. However, the interaction between contexts is far from straightforward and unintended interactions may lead to erroneous behavior. To counter this, Subjective-C offers the possibility of defining context-dependency relations, as a means to better manage interactions. Finally, to help programmers in the definition of all these elements, Subjective-C presents a set of guidelines for better engineering COP systems (Cardozo et al., 2011; Kasin, 2012).

Criterion 3.2. Reasoning. In its inception, Subjective-C lacked any reasoning capability about the system behavior. Given the importance of offering assurances about the system behavior, the language was extended with a formal run-time execution engine (Cardozo et al., 2012b). This execution engine focuses on the specification of context-dependency relations, the run-time verification for the constraints defined by context-dependency relations, and on the possible analyses to guarantee certain desired behavior beforehand Cardozo et al. (2015). In particular, Subjective-C assures consistent system execution with respect to the defined context-dependency relations. Moreover, the analysis and reasoning techniques incorporated in the formal Subjective-C model allow us to, for example, assert the eventual active or inactive state of BContexts from a given system state. This can be used, for example, to determine unreachable behavioral variations.

Criterion 3.3. Testing. Currently, there are no testing techniques specific for Subjective-C programs, or for the context-dependent behavior defined for COP systems.

Criterion 4. Pragmatics

Criterion 4.1. Usage. Subjective-C has been used exclusively in academic settings with the development of proof-of-concept applications demonstrating the different features of the language from the research papers associated with the language. The language is no longer under development, but the authors are working on a new version using Swift as a base language.

Criterion 4.2. Resources. Currently, the only available resources about Subjective-C are available in the different research publications about the language, and the GitHub repository containing example implementations[3] and an evaluation of the scoping mechanism and a performance mini-benchmark demonstrating the scale of contexts that can be used simultaneously in the language.[4]

Criterion 4.3. Required expertise. The use of Subjective-C relies heavily on Objective-C. As a consequence, to use the former language, it is a prerequisite to have some knowledge of the latter language, but there is no specific idiom to follow to develop Subjective-C programs. Finally, as Subjective-C is a super language on top of Objective-C, the same tools and frameworks used for Objective-C development are used for Subjective-C, together with additional visualization tools available to manage contexts (Cardozo, 2013).

[3] https://github.com/ncardozo/SmartCityGuide

[4] https://github.com/ncardozo/CopPerformanceEvaluation

Criterion 4.4. Domain applicability. Subjective-C is a full COPL, therefore, it can be used to develop any context-aware system. Nonetheless, the restrictions of the VM supporting the language makes it ideal to use in mobile app development in the iOS platform.

Criterion 4.5. Scalability. The objective of Subjective-C is to serve as a proof-of-concept language for COP. As a consequence performance is not a concern in its design. Early evaluations of the scalability of Subjective-C show that the language can effectively manage up to 50,000 contexts in some situations, although the performance of managing such a large number of contexts decreases rapidly. Additional evaluation of the execution of local and global contexts are restricted by the platform capabilities, enabling a maximum of 60 concurrent local contexts, for each of the independent threads that can be spawned in the mobile platform (Cardozo et al., 2012a; Cardozo, 2013; Cardozo and Mens, 2022).

Criterion 4.6. Maintainability. Maintenance processes of Subjective-C programs follow the same process as other software systems. The integration of new requirements or the modification of existing requirements follows the regular software process in which the functionality must be analyzed, designed, implemented, and tested in the current system. However, we note two characteristics, special to the case of Subjective-C. First, integration and modification of requirements can be done isolated from the current system as new functionality or behavior can be defined as adaptations, describing the functionality as a behavioral variation associated with a new BContext always active from this point in time onward, to be available for newer system versions, making the behavior associated with the BContext the behavior of the requirement. This can be beneficial to test the system. However, testing COP systems is not standardized and bug detection is not straightforward when behavior variations are involved.

Criterion 4.7. Language dependability. The implementation of Subjective-C is tightly coupled to that of Objective-C. Subjective-C works as a super language over Objective-C, being able to use (and modify according to the context) all library and language components from Objective-C.

Criterion 4.8. Evaluation. As before, no large-scale evaluation of different COPLs has been done. However, Subjective-C has been used in the evaluation of features offered by the language to increase the flexibility in adapting software systems (Cardozo et al., 2011). Additionally, the designed features of the language have been used to assess the specific implementation characteristics between different COPLs, to foster dynamically adaptive systems (Cardozo and Mens, 2022).

6. Applying the framework to COBPjs

As an alternative paradigm for COPLs, we continue to evaluate the framework using COBPjs, a language that implements a model-driven engineering paradigm called *Context-Oriented Behavioral Programming* (COBP).

COBP is a novel language-independent paradigm for developing context-aware systems, centered on the natural and incremental specification of context-dependent behavior (Elyasaf, 2021). The paradigm aims at creating executable specifications from context-dependent requirements. When creating a system using COBP, developers specify a set of context-dependent scenarios that may, must, or must not happen. Each scenario is a sequential thread of execution, called *b-thread*, that is normally aligned with a system requirement, such as "lock rooms at night" or "perform a sequence of opening and closing of a tap in a room". The set of b-threads in a model is called a behavioral program (*b-program*). At run-time, all b-threads participating in a b-program are combined, yielding a complex behavior that is consistent with all said b-threads. As we elaborate below, unlike other paradigms, such as functional programming, object-oriented programming, or COP, COBP does not force developers to pick a single behavior for the system to use. Rather, the system is allowed to choose any compliant behavior. This allows the run-time engine to optimize the program execution at any given moment, *e.g.*, based on available resources. The fact that all possible system behaviors comply with the b-threads (and thus with the system requirements) ensures that whichever behavior is chosen, the system will perform as specified.

BP and COBP. COBP has emerged from a scenario-based modeling paradigm, called *behavioral programming* (BP). BP allows the refinement of behavior to some degree, however, BP has no notion of global, or shared, data. B-threads are isolated and may communicate via messages only (similar to Erlang). Thus, it is not possible to bind behavior to a context. COBP extends BP by adding data, context idioms, and new abstract semantics that specify the interaction between the data model and the behavioral model.

COBP and COP. COBP shares some concepts with COP and COPLs, such as group-based behavior adaptation (Vallejos, 2001; Rein et al., 2017), asynchronous message passing as in ContextErlang (Salvaneschi et al., 2012b), and feature descriptions (Costanza and D'Hondt, 2008). Nevertheless, the paradigm has independently emerged from BP, enriching it with context idioms. There are two main differences between COBP and its origins: (1) While many COP languages aim to adapt existing programs' behavior to context, COBP aims to create executable specifications from context-dependent requirements. Thus, the development and design of COBP programs start with specifying the context of each requirement and defining its relation to the behavior (similar to the role of Object in OOP). (2) COBP has abstract semantics that is implemented by all COBP languages (see below). These semantics have rigorous mathematical characteristics that allow for executing the model. Moreover, the semantics allows for applying verification and formal method techniques on the entire model, including the BContexts, the behavioral variations, and their activation/deactivation.

COBP languages. There are currently two languages that implement COBP, one is based on a diagrammatic language called live-sequence charts (COLSC) (Elyasaf et al., 2018b,a), and the other one is based on JavaScript (COBPjs) (Elyasaf et al., 2019a; Elyasaf, 2021). The examples and the analysis below are written in COBPjs, though most are also relevant for COLSC and for any future implementations in other languages. This is due to the fact that all COBP languages implement the same abstract semantics and their code can be transformed from one language to another. The COBPjs repository includes documentation and the code for the examples below (github.com/bThink-BGU/BPjs-Context).

6.1. COBP in a nutshell

Developing with cobpjs. To make COBP concepts more concrete, we now turn to the example of the smart home system. A COBPjs program has the following components:

- The system context specifies the schema of the system entities and their initial population.
- The data access layer specifies the BContexts.
- The logic layer specifies the context-dependent behavior.
- The service layer specifies the interaction between the behavioral program and its environment (*i.e.,* sensors and actuators).

```
// Add entities
ctx.populateContext([
 ctx.Entity('r1', 'room',
     {subtype: 'kitchen'}),
 ctx.Entity('r2', 'room',
     {subtype: 'bedroom'}),
 ctx.Entity('r3', 'room',
     {subtype: 'bathroom'})
])

// Specify a BContext/layer named
// 'Night' with no encapsulated data.
ctx.registerQuery('Night',
 entity => entity.id === 'night')
// Specify a BContext named 'Room.
// WithTaps' with the subtype encapsulated.
ctx.registerQuery('Room.WithTaps',
 entity => entity.type === 'room' &&
    (entity.subtype === 'kitchen' ||
     entity.subtype === 'bathroom')

// Specify the effect of 'night begins/ends'
// events on the context. The data of the
// event can be used if needed.
ctx.registerEffect('time-21:00',
  event_data =>
      ctx.insertEntity('night', 'system')
)
ctx.registerEffect('time-08:00',
  event_data => ctx.removeEntity('night')
})
```

Snippet 10: The system context and BContext specification (data layer) of the system in COBPjs. *ctx* is the system context.

Previous demonstrations of COBP include a reactive IoT building (Elyasaf et al., 2018b) and a fully functional RoboSoccer player (Elyasaf et al., 2019b), where the specification refers to controlling a robot soccer player in a simulated environment.

6.1.1. Basic requirements

The system context. The development process of a context-aware behavioral program begins with the specification of the information model of the system context and its initial population. In Snippet 10, the call to `populateContext` defines and populates initial entities of the system of type `room`. The entities can be specified at compile time, or at run time.

The Data Access Layer (DAL). The data access layer defines the BContexts that are specified as queries that filter the entities of the system context (precondition) and return the (encapsulated) data. The *execution engine* monitors the entities and the results of the queries, activates the BContexts upon new results to the queries, and deactivates them when the results are no longer relevant.

In Snippet 10 we call the `registerQuery` method to define two queries (BContexts) — `Night` and `Room.WithTaps`. Then, we call the `registerEffect` method to specify the effect of the `time-08:00` and `time 21:00` events on the entities. For example, the `night` entity is instantiated when the event `time-21:00` is selected. Consequently, the `Night` query has a new answer, so the `Night` BContext is activated. Similarly, the `night` entity is removed upon the `time-08:00` event, deactivating the `Night` BContext.

```
bthread('no vacuuming at night', 'Night',
  entity => {
    sync({block: Event('vacuum')})
}) // Requirement 4

bthread('Add Cold Three Times',
   'Room.WithTaps', entity => {
  while(true) {
    sync({waitFor: Event('press', entity)})
    sync({request: Event('cold', entity)})
    sync({request: Event('cold', entity)})
    sync({request: Event('cold', entity)})
  }
}) // Requirement 5a

bthread('Add Hot Three Times',
   'Room.WithTaps', entity => {
  while(true) {
    sync({waitFor: Event('press', entity)})
    sync({request: Event('hot', entity)})
    sync({request: Event('hot', entity)})
    sync({request: Event('hot', entity)})
  }
}) // Requirement 5b
```

Snippet 11: The behavioral specification (logic layer) of the hot-cold system in COBPjs.

The logic layer. The logic layer in COBPjs specifies the system behavior as a self-contained unit. Like the three-tier architecture, this layer is unaware of implementation details of the environment (*i.e.,* UI, sensors, actuators) or the data tier (*i.e.,* the context).

The logic tier of our system is specified in Snippet 11. The first b-thread is bound to the `Night` BContext, defining the behavior during the night. Thus, the b-thread is not executed when the program starts; instead, it spawns a *live copy* whenever the `night` BContext is active. This is similar to Java, where a context-depended b-thread is a Runnable instance, and a live copy is the actual thread that executes the Runnable. We note that this b-thread is aligned to Requirement 4.

The next two b-threads are aligned to Requirements 5a and 5b. Since these requirements are bound to the `RoomWithTaps` BContext; the b-threads are bound to the `Room.WithTaps` query. Here, a live copy will be generated for each room with taps, with the query result passed as a local variable to the live copy. This is an example of data encapsulation within a context. When the context ends, the live copy may choose to terminate itself or do some actions before terminating (*e.g.,* release resources).

The service layer. Behavioral programs interact with the environment using a publish–subscribe mechanism. B-threads may listen to external events, such as sensor readings or UI events. Similarly, actuators can subscribe to internal/external events and act upon them. For example, the hot-water tap may subscribe to the `hot` event and pour hot water whenever it is selected. The code for this layer can be found in the COBPjs repository.

6.1.2. Executing a COBP program

Elyasaf (2021) extended the BP protocol (Harel et al., 2010) for interweaving the context-aware b-threads and executing the program. B-threads (or live copies) repeatedly execute an internal logic and then synchronize with each other by submitting a synchronization statement to a central event arbiter. Once all b-threads have submitted their statements, the central event arbiter selects a requested event that was not blocked. B-threads that either requested or waited for this event are resumed, while the rest of the b-threads remain paused for the next cycle. Back to our example, the specification in Snippet 11 does not dictate an order in which actions are performed since the b-threads do not block events of each other. Thus, any of the following runs are possible: Cold(r1)-Cold(r1)-Hot(r2)-Hot(r1)-Cold(r2)-Hot(r2)-...., or Cold(r2)-Hot(r1)-Cold(r2)-Hot(r2)-.... This contrasts with imperative programming languages that dictate the exact order. Thus, traditional programming paradigms are prone to over-specification, while BP and COBP avoid it.

```
// An additional query to the DAL:
ctx.registerQuery('Room.Kitchen',
    e => 'room' === entity.type &&
     'kitchen' === entity.subtype)

// An additional b-thread to the logic layer:
bthread('Interleave', 'Room.Kitchen',
    entity => {
  while (true) {
    sync({waitFor: Event('cold', entity),
        block: Event('hot', entity)})
    sync({waitFor: Event('hot', entity),
        block: Event('cold', entity)})
  }
}) // Requirement 6
```

Snippet 12: The `Interleave` b-thread — ensures that two actions in a kitchen of the same type cannot be executed consecutively, by blocking an additional request of 'Cold' until the 'Hot' is performed, and vice-versa.

6.1.3. Evolved requirements

Requirement 6 refines the behavior for kitchen rooms. While we may add a condition before requesting 'Cold' and 'Hot', the BP paradigm encourages us to add a new b-thread for each new requirement. Thus, we add a new query called `Room.Kitchen` and a new b-thread, `Interleave`, presented in Snippet 12.

The `Interleave` b-thread ensures that there are no repetitions by forcing an interleaved execution of the performed actions: 'Hot' is blocked until 'Cold' is executed, and then 'Cold' is blocked until 'Hot' is executed. Note that this b-thread can be added and removed without affecting other b-threads. This is an example of a *purely additive* change, where the system behavior is altered to match a new requirement without affecting the existing behaviors. While not all changes to a b-program are purely additive, many useful changes are Harel et al. (2010), Elyasaf (2021).

Note that a single change to the entities may trigger more than one answer to the queries (for example, adding a kitchen room triggers both the hot/cold b-threads and the interleave b-thread).

6.2. Analysis of COBPjs

Criterion 1. The goal of the language

COBP (and its implementing languages) aims to provide programmers with the means to focus on mapping context-dependent requirements/behavior to the codebase. COBP aims to place the context as one of the main constructs of the language to better align with the requirements and increase modularity. As shown in Section 6.1 and elaborated below, COBP introduces changes to the entire development cycle, from the program design, through the implementation, testing, and even the maintenance of the system (*i.e.,* upon the introduction of new/modified requirements).

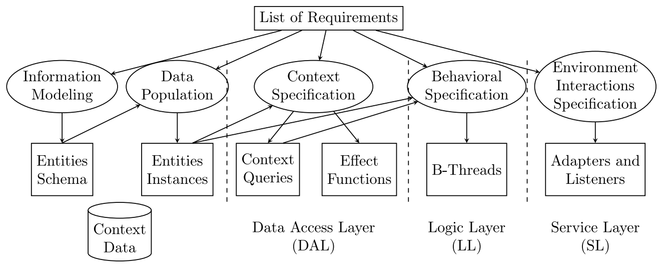

Fig. 1. The methodology for developing context-aware systems with COBPjs.

Criterion 2. Concepts

Criterion 2.1. BContext. The BContext in COBP languages is implemented as a named query, where the query name serves as the BContext name (Section 6.1.1). The queries filter the system entities, thus defining preconditions on the BContext, and may encapsulate data to be utilized by the behavior. The queries may also use the entities' states as pre-conditions. Multiple behavior definitions (*i.e.,* b-threads) can be bound to a single BContext and they may change its encapsulated data, which may trigger the activation of one or more contexts. BContext in COBP can be either static (*e.g.,* the `room` context), or dynamic (*e.g.,* the `night` context).

Criterion 2.2. Inter BContext relationship. The alignment between the code and the context-dependent requirements exceeds the need to explicitly define inter-BContext relationships unless such a relationship is specified in the requirements. To cope with an explicit specification of BContext composition, COBP languages allow for specifying compound queries and control the execution using priorities and behaviors. Additionally, all relation types can be inferred using static or dynamic analysis.

Criterion 2.3. The BContext life cycle. In Section 6.1.1 and Snippet 10, we elaborate on the BContext life cycle in COBPjs. The specification is defined by the queries, that define the preconditions and the encapsulated data. The BContext activation and deactivation are specified by the effect functions and the semantics/execution engine. The execution engine monitors the entities and the results of the queries, activates the BContexts upon new results to the queries, and deactivates them when the results are no longer relevant. As elaborated below, the deactivation of a BContext instance deactivates the bound behavioral variations, though COBPjs supports all types of reactions to a BContext deactivation.

Criterion 2.4. Behavioral variation.

1. A behavioral variation in COBP is a b-thread that is bound to a specific BContext. In many cases, requirements do not specify a default behavior. Instead, they define several of them, one for each context. Since COBP aims for an alignment between b-threads and requirements, it is possible to specify b-threads in both ways. A default behavior can be refined in certain contexts, and a behavior can be defined without a default behavior.
2. It is not possible to bind a b-thread to multiple BContexts (though it is possible in COLSC). Instead, it is possible to explicitly define a complex BContext that combines several simpler BContexts.

3-5. The b-thread is executed for each instance of the BContext and scoped to it. The b-thread may access the encapsulated data of the instance and it may also access the system context at any time.

6-7. COBP programs are aligned to the requirements. If the latter defines a specific order of execution, then the b-thread will specify that as well. Otherwise, the system is allowed to choose any compliant behavior. The rigorous mathematical semantics of COBP languages ensures that all possible system behaviors comply with the b-threads (and thus with the system requirements). Whichever behavior is chosen, the system will perform as specified.

Criterion 2.5. The behavioral-variation life cycle. In Section 6.1.1 and Snippet 10, we elaborate on the behavioral-variation life cycle in COBPjs. As noted before, the variations are defined as context-dependent b-threads.

Whenever a BContext is activated (*i.e.,* an instance of the BContext is created), a live copy is spawned for each b-thread bound to it. The live copies receive the encapsulated data of the activated BContext (elaborated in Section 6.1.1).

By default, whenever an instance of a BContext is deactivated, an exception is thrown to all of its live copies and they are immediately terminated. Nevertheless, depending on the requirements, it is possible to catch this exception and override the automatic termination for performing cleanup tasks (*e.g.,* release resources). It is also possible to ignore the termination and continue the execution.

Criterion 3. Process support

Criterion 3.1. Method and guidelines. The methodology for developing systems with COBPjs has been presented and demonstrated by Elyasaf et al. (2018b), Elyasaf (2021). The paradigm is **context-oriented**, and thus, the methodology (depicted in Fig. 1) makes emphasis on the specification of the context and its initial population (during the information modeling and data population activities) and the BContext specification (DAL). The development process continues with the b-threads (logic layer) and concludes with the service layer.

Criterion 3.2. Reasoning. One of the key advantages of BP is the amenability of the software artifacts to formal analysis and synthesis (Elyasaf, 2021; Damm and Harel, 2001). Most of the tools for BP rely on the mathematically rigorous nature of the semantics in providing tools for running formal analysis and synthesis algorithms (Elyasaf, 2021). The addition of context also improves the specification's modularity, thus contributing to each of the algorithms. Elyasaf (2021) demonstrate how reasoning techniques could directly use COBPjs programs without translating the program into a formal language or providing additional data (*e.g.,* constraints on context activation). Moreover, they were able to both identify behavior inconsistencies and verify the correctness of the queries.

Criterion 3.3. Testing. To test BP and COBP programs, users can add additional b-threads that include assertion statements. These b-threads can be used for testing the expected behavior. To cover the program behavior, there are tools for traversing the entire state space of the program and for sampling the state space, in case it is too large (Elyasaf et al., 2022). Nevertheless, there is no official methodology for using these tools for testing and it still requires some expertise in BP.

Criterion 4. Pragmatics

Criterion 4.1. Usage. Implementations of both BP and COBP are being developed in academic contexts. Nevertheless, previous demonstrations of COBP include a showcase of industry domains, specifically, a robotics controller (Elyasaf et al., 2019a) and a reactive IoT building (Elyasaf et al., 2018b). The BP showcase includes other demonstrations that are also relevant since they are fit for COBP as well, such as a fully functional nano-satellite (Bar-Sinai et al., 2019), web-servers (Harel and Katz, 2014), and black-box testing (of non-BP programs) (Elyasaf et al., 2022).

BP and COBP are mostly academic artifacts, though individuals and companies have started using COBPjs to some extent over the past year.

Criterion 4.2. Resources. The GitHub repository of COBPjs (github.com/bThink-BGU/BPjs-Context) includes code samples, explanations, a video tutorial, links to papers, a link to a Google group, and other relevant links.

COBPjs programs are written in JavaScript and can be written with any text editor. Nevertheless, to use known IDE features, such as code completion and debugging capabilities, it is best to use an IDE that supports Java and JavaScript languages.

COBPjs has libraries for verifying the program correctness, for presenting the state space, and for visually analyzing the possible execution traces.

COBPjs is still missing an API, inclusive tutorials, and non-paper-based documentation. A specialized IDE and a debugger are currently under development.

Criterion 4.3. Required expertise. Users must be familiar with the BP paradigm for programming with COBPjs. BP is well-documented and has an active group of users. Moreover, several user studies with high-school students (Alexandron et al., 2014; Harel and Gordon-Kiwkowitz, 2009) demonstrated that BP is easy and fast to learn. The authors attributed this to the fact that BP languages are high-level and declarative, allowing programmers to focus on the system's behavior rather than implementation decisions. While no user study has been made for COBPjs, these BP attributes are valid for COBP languages as well. Moreover, it is possible in COBP to align b-threads to context-dependent requirements, while in BP, it is not, thus simplifying the specification of context-aware systems (Elyasaf, 2021).

Criterion 4.4. Domain applicability. From a theoretical point of view, COBP can be used for developing any context-aware reactive system.

Criterion 4.5. Scalability. The execution engine uses a fixed number of operating system threads to advance the b-threads simultaneously. Bar-Sinai (2020) measured the performance of the execution engine with different programs with a varying number of b-threads, from 25 to 10,000. The experiments showed that the execution time is in linear proportion to the number of b-thread and was roughly affected by the number of b-threads. However, the program size does affect the ability to verify the entire program (Elyasaf, 2021). The verification is currently limited to small systems or small enough subsets of the system behaviors.

Criterion 4.6. Maintainability. The main design goal of COBP is to align the codebase organization with how requirement documents are defined (Elyasaf, 2021). In requirement documents, chapters or sections define the context of a set of different behavior (*e.g.*, "Room" or "Emergency" chapters define the behavior in the context of a `room` or an `emergency`). In contrast, the context of each requirement may refine the chapter's context. Similarly, in COBP, b-threads are bound to a query and can be grouped according to their query. In COBPjs, for example, one can group all b-threads that are bound to the "room" query in a file called `room.js`. Notably, grouping behavior in files according to their query is only one option, though other approaches are also applicable (*e.g.*, hashtags). Since each b-thread defines a single aspect of the system behavior, the grouping method does not affect the system behavior. To conclude, the modularization direction of COBP is aligned with the context-depended behavior, rather than being orthogonal to it.

New requirements, in most cases, do not affect the code maintainability, as demonstrated in Section 6. Nevertheless, when a requirement is changed (and not refined), the b-thread needs to be changed accordingly. *The binding identification* is trivial since the query name is specified at the signature of each b-thread. *BContext refactoring* requires a change to the DAL only. For example, the refactoring of the room sub-types does not change the queries' names (Snippet 12). *Identifying places to change*. As demonstrated in Snippet 12, the place to change is trivial to detect and depends on whether the change is in the requirement or in the context schema.

Criterion 4.7. Language dependability. While the abstract semantics of COBP extends the semantics of BP, the context idioms in COBPjs are implemented as syntactic sugar to BPjs idioms. Thus, all the frameworks and libraries of BPjs are compatible with COBPjs.

Criterion 4.8. Evaluation. No evaluation has been done on COBPjs. Nevertheless, the underlying programming paradigm for COBPjs, BP, offers an advantage in this regard. Much work has been done on the BP paradigm and some of its implementations. We bring here only one example of each type: user studies (Harel and Gordon-Kiwkowitz, 2009), performance benchmarks (Bar-Sinai, 2020), and controlled experiments (Ashrov et al., 2017).

7. Discussion

Applying the framework to COPLs originating from different paradigms (*i.e.,* Layered-based, Object-Oriented, Scenario-based) indicates its usefulness, and its ability to abstract important concepts and criteria for COPLs and their usage. Table 2 summarizes the analysis of the COPL representatives using the proposed framework. We observe that all languages similarly address most of the criteria. The differences emerge from the focus of the languages. ServalCJ and Subjective-C emphasize the object-oriented design, whereas COBPjs emphasize the alignment with requirements. From the analysis, we can identify a few shared drawbacks to the implementation of COPLs (some of them are also shared by other COPLs, based on our evaluation of the state-of-the-art in COPLs). Two notable drawbacks are the testing of COP systems, and the evaluation and usage of COPLs, to fully satisfy their objectives.

Applying the framework to each language also provides insights for further improvements for each of the languages. Regarding ServalCJ, through the framework, we identify that the language is positioned as a research artifact, due to the lack of material supporting its maturity, such as documentation, learning resources, or an evaluation of the language's usability.

Regarding Subjective-C, the evaluation framework helped us to identify characteristics that highlight a lack of the language's maturity, such as missing learning/reference resources, more comprehensive documentation, or tools for supporting the development process (*e.g.,* testing, methodologies). Additionally, we note our evaluation framework proved useful in making clear the underlying concepts of Subjective-C that were not defined explicitly by the language developers. In particular, we highlight the definition of the conditions to activate/deactivate contexts. In the comparison with COBPjs, we note that the underlying behavior and terms are defined in both COBPjs and Subjective-C. However,

Table 2
A criteria summary for the evaluated COPLs.

Criterion	ServalCJ	Subjective-C	COBPjs
Concepts			
BContext: Encapsulated Data	Yes	Yes	Yes
BContext: Data Pre-condition	No	Yes	Yes
BContext: Behavior Pre-condition	No	No	Yes
BContext: Type	Static and dynamic	Static and dynamic	Static and dynamic
Inter BContext Relationship	Implicit: Context activation time, control flow Explicit: dependency relations	Implicit: Context activation time Explicit: Context dependency relations	Implicit through the queries
BContext Life Cycle	BContexts can be applied globally or locally to specific instances	BContexts are global (specification, activation, deactivation); where one specification is applied to all situations	Preconditions are specified before execution. BContext instances are automatically activated and removed according to preconditions
Behavioral Variation	Bound to multiple BContexts, following the scoping rules of BContexts	Bound to multiple BContexts with global and (thread) local scoping. Execution is resolved following BContexts' dependency relations	Bound to a single BContext with global and (thread) local scoping.
Behavioral-Variation Life Cycle	Activation is prompt, deactivation is prompt or prompt-loyal depending on the model	Activation is prompt, deactivation is prompt-loyal	Follows the lifecycle of BContexts. All three types of deactivation are supported
Process			
Method and Guidelines	Ad hoc	Ad hoc	Yes
Reasoning	Formal verification and static checks	Static analysis and run-time checks	Verification and Run-time checks
Testing	None	None	Methodology that utilizes verification capabilities
Pragmatics			
Usage	Mostly academic	Mostly academic	Mostly academic with a few industrial case studies
Resources	Git repository (limited)	Git repository	Git repository
Required expertise	Java & Aspects	Objective-C	Behavioral Programming
Domain Applicability	Mainly desktop applications	Mainly for mobile applications	Mainly reactive systems
Scalability	Unknown	Highly scalable	Run-time is highly Scalable, verification scalability is limited
Maintainability	High	High	High
Language Dependability	Depends on Java and the underlying features of abc	The language implementation is tight to Objective-C	No. COBPjs idioms are implemented as syntactic sugars to BPjs idioms
Evaluation	Limited	Limited	Limited

the explicit abstractions available in COBP (*i.e.*, `registerQuery`), offer a more flexible interface for the definition of adaptations than that available in Subjective-C.

Regarding COBPjs, we identified the lack of resources, such as good documentation, tutorials, and an API. This is important as in searching for resources of other COPLs, we came to learn about the importance of each resource type. In addition, as previously mentioned, there is no need to specify inter-BContext relations in COBPjs explicitly. While they can be theoretically inferred using static and dynamic analysis, the current tools do not provide such analysis. Inspired by Duhoux et al. (2018) we wish to extract and visualize these interactions.

Using the framework, it is possible to see that even though the abstractions existing in all languages seem divergent, the underlying concepts and behavior are similar across all languages. Moreover, through the framework, such comparisons between languages are more clear. This is beneficial for language developers to define new COPLs or improve the existing ones. Also, following the results of that comparison can support language users in choosing the most appropriate language with respect to their needs.

8. Conclusion

Context-Oriented Programming is a programming language-level technique to realize dynamic adaptation of software systems with respect to their surrounding execution environment. Many different languages for COP have been proposed over the last 15 years. Such languages share the overall objective of dynamic adaptation, but present differences in their realization of such adaptations, the terminology used, and the programming interface offered to users. Moreover, the different concepts used in each language, and the lack of a standard evaluation, make the comparison between languages and their features challenging.

To ease these problems, this paper posits a framework for analyzing context-oriented programming languages. The framework focuses on language usage, and the main characteristics to realize dynamic adaptation at the programming language level. The framework consists of four parts: (1) the language goal, (2) the supported concepts, (3) the process support, and (4) the language's pragmatics. To demonstrate the use of the framework, we apply it to analyze three different COPLs: ServalCJ, Subjective-C, and COBP, each representative of the main technique used to implement COPLs. In the analysis of the three languages, we are able to evaluate the framework's capabilities to capture the relevant concepts from the three languages, and provides a baseline to discuss the differences and similarities between COPLs.

The framework is comprehensive, albeit we do not claim it is complete. In its current state, the framework can be used to start a discussion between different COPLs, but more criteria could be incorporated if the current criteria do not cover specific features of existing languages. Therefore, we plan to refine the framework by adding other relevant aspects and refining existing criteria to make the analysis of COPLs more accurate and objective. In addition, the framework can benefit from setting empirical means to evaluate some of the criteria. Following these refinements, we aim to analyze other COPLs, allowing us to understand the major concerns and challenges of designing and implementing context-oriented programming languages.

CRediT authorship contribution statement

Achiya Elyasaf: Conceptualization, Validation, Writing. **Nicolás Cardozo:** Conceptualization, Validation, Writing. **Arnon Sturm:** Conceptualization, Methodology, Writing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

References

Alexandron, Giora, Armoni, Michal, Gordon, Michal, Harel, David, 2014. Scenario-based programming, usability-oriented perception. Trans. Comput. Educ. 14 (3), 1–23.

Allan, Chris, Avgustinov, Pavel, Christensen, Aske Simon, Dufour, Bruno, Goard, Christopher, Hendren, Laurie, Kuzins, Sascha, Lhoták, Jennifer, Lhoták, Ondrej, de Moor, Oege, Sereni, Damien, Sittampalam, Ganesh, Tibble, Julian, Verbrugge, Clark, 2005. Abc the AspectBench compiler for AspectJ: A workbench for aspect-oriented programming language and compilers research. In: Companion to the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA '05, Association for Computing Machinery, New York, NY, USA, ISBN: 1595931937, pp. 88–89. http://dx.doi.org/10.1145/1094855.1094877.

Aotani, Tomoyuki, Kamina, Tetsuo, Masuhara, Hidehiko, 2011. Featherwight eventcj: A core calculus for a context-oriented language with event-based per-instance layer transition. In: International Workshop on Context-Oriented Programming, Vol. 1. COP '11, ACM, pp. 1–7.

Aotani, Tomoyuki, Kamina, Tetsuo, Masuhara, Hidehiko, 2014. Unifying multiple layer activation mechanisms using one event sequence. In: Proceedings of 6th International Workshop on Context-Oriented Programming. COP '14, ACM, New York, NY, USA, ISBN: 978-1-4503-2861-6, pp. 2:1–2:6.

Appeltauer, Malte, Hirschfeld, Robert, Haupt, Michael, Lincke, Jens, Perscheid, Michael, 2009. A comparison of context-oriented programming languages. In: International Workshop on Context-Oriented Programming. COP '09, ACM, ISBN: 9781605585383, http://dx.doi.org/10.1145/1562112.1562118.

Ashrov, Adiel, Gordon, Michal, Marron, Assaf, Sturm, Arnon, Weiss, Gera, 2017. Structured behavioral programming idioms. In: Enterprise, Business-Process and Information Systems Modeling. Springer, pp. 319–333. http://dx.doi.org/10.1007/978-3-319-59466-8_20.

Bar-Sinai, Michael, 2020. Extending Behavioral Programming for Model-Driven Engineering (Ph.D. thesis). Ben-Gurion University of the Negev, Israel.

Bar-Sinai, Michael, Elyasaf, Achiya, Sadon, Aviran, Weiss, Gera, 2019. A Scenario Based On-Board Software and Testing Environment for Satellites. In: 59th Israel Annual Conference on Aerospace Sciences Vol. 2. IACAS 2019, ISBN: 978-1-5108-8278-2, pp. 1407–1419.

Bricon-Souf, Nathalie, Newman, Conrad R., 2007. Context awareness in health care: A review. Int. J. Med. Inform. (ISSN: 1386-5056) 76 (1), 2–12. http://dx.doi.org/10.1016/j.ijmedinf.2006.01.003, URL https://www.sciencedirect.com/science/article/pii/S1386505606000098.

Cardozo, Nicolás, 2013. Identification and Management of Inconsistencies in Dynamicaly Adaptive Software Systems (Ph.D. thesis). Université catholique de Louvain - Vrije Universiteit Brussel, Louvain-la-Neuve, Belgium.

Cardozo, Nicolás, 2016. Emergent software services. In: Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. In: Onward! 2016, ACM, New York, NY, USA, ISBN: 978-1-4503-4076-2, pp. 15–28.

Cardozo, Nicolás, Dusparic, Ivana, 2021. Auto-COP: Adaptation generation in context-oriented programming using reinforcement learning options. arXiv preprint arXiv:2103.06757.

Cardozo, Nicolás, González, Sebastiá, Mens, Kim, D'Hondt, Theo, 2011a. Safer context (de)activation: Through the prompt-loyal strategy. In: International Workshop on Context-Oriented Programming. COP '11, (2), ACM, New York, NY, USA, pp. 1–6.

Cardozo, Nicolás, González, Sebastián, Mens, Kim, D'Hondt, Theo, 2012a. Uniting global and local context behavior with context Petri nets. In: International Workshop on Context-Oriented Programming, no. 3. COP '12, ACM, New York, NY, USA, pp. 1–6.

Cardozo, Nicolás, González, Sebastián, Mens, Kim, Van Der Straeten, Ragnhild, Vallejos, Jorge, D'Hondt, Theo, 2015. Semantics for consistent activation in context-oriented systems. Inf. Softw. Technol. 58, 71–94. http://dx.doi.org/10.1016/j.infsof.2014.10.002.

Cardozo, Nicolás, Günther, Sebastian, D'Hondt, Theo, Mens, Kim, 2011. Feature-oriented programming and context-oriented programming: Comparing paradigm characteristics by example implementations. In: Proceedings of the International Conference on Software Engineering Advances. ICSEA '11, IARIA, pp. 130–135.

Cardozo, Nicolás, Mens, Kim, 2022. Programming language implementations for context-oriented self-adaptive systems. Inf. Softw. Technol. (ISSN: 0950-5849) 143, 106789. http://dx.doi.org/10.1016/j.infsof.2021.106789, URL https://www.sciencedirect.com/science/article/pii/S0950584921002305.

Cardozo, Nicolás, Vallejos, Jorge, González, Sebastián, Mens, Kim, D'Hondt, Theo, 2012b. Context Petri nets: Enabling consistent composition of context-dependent behavior. In: International Workshop on Petri Nets and Software Engineering, Vol. 851. PNSE '12, CEUR, pp. 156–170.

Costanza, Pascal, D'Hondt, Theo, 2008. Feature descriptions for context-oriented programming. In: Thiel, Steffen, Pohl, Klaus (Eds.), Software Product Lines. Lero Int. Science Centre, University of Limerick, Ireland, pp. 9–14.

Costanza, Pascal, Hirschfeld, Robert, 2005. Language constructs for context-oriented programming: An overview of contextL. In: Dynamic Languages Symposium. DSL '05.

Damm, Werner, Harel, David, 2001. LSCs: Breathing Life into Message Sequence Charts. Form. Methods Syst. Des. (ISSN: 1572-8102) 19 (1), 45–80. http://dx.doi.org/10.1023/A:1011227529550.

Dey, Anind K., 2001. Understanding and using context. Pers. Ubiquitous Comput. 5 (1), 4–7.

Duhoux, Benoît, Mens, Kim, Dumas, Bruno, 2018. Feature Visualiser: An Inspection Tool for Context-Oriented Programmers. In: Proceedings of the 10th International Workshop on Context-Oriented Programming. ACM, pp. 15–22. http://dx.doi.org/10.1145/3242921.3242924.

Elyasaf, Achiya, 2021. Context-oriented behavioral programming. Inf. Softw. Technol. (ISSN: 0950-5849) 133, 106504. http://dx.doi.org/10.1016/j.infsof.2020.106504, URL http://www.sciencedirect.com/science/article/pii/S095058492030094X.

Elyasaf, Achiya, Farchi, Eitan, Margalit, Oded, Weiss, Gera, Weiss, Yeshayahu, 2022. Combinatorial sequence testing using behavioral programming and generalized coverage criteria. arXiv preprint arXiv:2201.00522.

Elyasaf, Achiya, Harel, David, Marron, Assaf, Weiss, Gera, 2018a. Towards Integration of Context-Based and Scenario-Based Development, Vol. 10748 LNCS. Springer, Cham, ISBN: 978-3-319-74729-3, http://dx.doi.org/10.1007/978-3-319-74730-9_21.

Elyasaf, Achiya, Marron, Assaf, Sturm, Arnon, Weiss, Gera, 2018b. A context-based behavioral language for IoT. In: Hebig, Regina, Berger, Thorsten (Eds.), CEUR Workshop Proceedings, Vol. 2245. CEUR-WS.org, Copenhagen, Denmark, pp. 485–494, URL http://ceur-ws.org/Vol-2245.

Elyasaf, Achiya, Sadon, Aviran, Weiss, Gera, Yaacov, Tom, 2019a. Using behavioral programming with solver, context, and deep reinforcement learning for playing a simplified RoboCup-type game. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion. MODELS-C, IEEE, pp. 243–251. http://dx.doi.org/10.1109/models-c.2019.00039.

Elyasaf, Achiya, Sadon, Aviran, Weiss, Gera, Yaacov, Tom, 2019b. Using behavioral programming with solver, context, and deep reinforcement learning for playing a simplified RoboCup-type game. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion. MODELS-C, IEEE, pp. 243–251. http://dx.doi.org/10.1109/models-c.2019.00039.

Elyasaf, Achiya, Sturm, Arnon, 2021. Towards a framework for analyzing context-oriented programming languages. In: Proceedings of the 13th ACM International Workshop on Context-Oriented Programming and Advanced Modularity. In: COP 2021, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450385428, pp. 16–23. http://dx.doi.org/10.1145/3464970.3468414.

González, Sebastián, Cardozo, Nicolás, Mens, Kim, Cádiz, Alfredo, Libbrecht, Jean-Christophe, Goffaux, Julien, 2011. Subjective-C: Bringing context to mobile platform programming. In: Malloy, Brian, Staab, Steffen, van den Brand, Mark (Eds.), International Conference on Software Language Engineering, Vol. 6563. SLE '11, Springer-Verlag, ISBN: 978-3-642-19439-9, pp. 246–265. http://dx.doi.org/10.1007/978-3-642-19440-5_15.

Harel, David, Gordon-Kiwkowitz, Michal, 2009. On teaching visual formalisms. IEEE Software 26 (3), 87–95. http://dx.doi.org/10.1109/MS.2009.76.

Harel, D., Katz, G., 2014. Scaling-Up Behavioral Programming: Steps from Basic Principles to Application Architectures. In: Proceedings of the 4th Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions. pp. 95–108.

Harel, David, Marron, Assaf, Weiss, Gera, 2010. Programming coordinated behavior in Java. In: European Conference on Object-Oriented Programming. Springer Berlin Heidelberg, (ISSN: 0302-9743) ISBN: 978-3-642-14107-2, pp. 250–274. http://dx.doi.org/10.1007/978-3-642-14107-2_12.

Hirschfeld, Robert, Costanza, Pascal, Nierstrasz, Oscar Marius, 2008. Context-oriented programming. J. Object Technol. 7 (3), 125–151.

Inoue, Hiroaki, Igarashi, Atsushi, Appeltauer, Malte, Hirschfeld, Robert, 2014. Towards type-safe JCop: A type system for layer inheritance and first-class layers. In: International Workshop on Context-Oriented Programming. COP '14, ACM, ISBN: 9781450328616, http://dx.doi.org/10.1145/2637066.2637073.

Ireri, Bonface Ngari, Wario, Ruth Diko, Mwingirwa, Irene Mukiri, 2018. Choosing and adapting a mobile learning model for teacher education. In: Handbook of Research on Digital Content, Mobile Learning, and Technology Integration Models in Teacher Education. IGI Global, pp. 132–148. http://dx.doi.org/10.4018/978-1-5225-7918-2.ch009.

ISO/IEC 25010, 2011. ISO/IEC 25010:2011, systems and software engineering — Systems and software quality requirements and evaluation (SQuaRE) — System and software quality models.

Kamina, Tetsuo, Aotani, Tomoyuki, 2019. TinyCORP: A calculus for context-oriented reactive programming. In: Proceedings of the Workshop on Context-Oriented Programming. COP '19, ACM, New York, NY, USA, ISBN: 978-1-4503-6863-6, pp. 1–8.

Kamina, Tetsuo, Aotani, Tomoyuki, Masuhara, Hidehiko, 2011. EventCJ: A context-oriented programming language with declarative event-based context transition. In: Borba, Paulo, Chiba, Shigeru (Eds.), Proceedings of the 10th International Conference on Aspect-Oriented Software Development. AOSD 2011, Porto de Galinhas, Brazil, March 21-25, 2011, ACM, pp. 253–264. http://dx.doi.org/10.1145/1960275.1960305.

Kamina, Tetsuo, Aotani, Tomoyuki, Masuhara, Hidehiko, 2013a. A core calculus of composite layers. In: Proceedings of the 12th Workshop on Foundations of Aspect-Oriented Languages. FOAL '13, ACM, New York, NY, USA, ISBN: 978-1-4503-1865-5, pp. 7–12.

Kamina, Tetsuo, Aotani, Tomoyuki, Masuhara, Hidehiko, 2013b. A unified context activation mechanism. In: Proceedings of the International Workshop on Context-Oriented Programming. COP '13, ACM, New York, NY, USA, ISBN: 978-1-4503-2040-5, pp. 2:1–2:6.

Kamina, Tetsuo, Aotani, Tomoyuki, Masuhara, Hidehiko, 2016. Generalized layer activation mechanism for context-oriented programming. Trans. Modularity Compos. 1, 123–166.

Kamina, Tetsuo, Aotani, Tomoyuki, Masuhara, Hidehiko, 2017. Push-based reactive layer activation in context-oriented programming. In: Proceedings of the International Workshop on Context-Oriented Programming. COP '17, pp. 17–21.

Kamina, Tetsuo, Aotani, Tomoyuki, Masuhara, Hidehiko, Igarashi, Atsushi, 2018. Method safety mechanism for asynchronous layer deactivation. Sci. Comput. Program. 156, 104–120.

Kamina, Tetsuo, Aotani, Tomoyuki, Masuhara, Hidehiko, Tamai, Tetsuo, 2014. Context-oriented software engineering: A modularity vision. In: Proceedings of the International Conference on Modularity. MODULARITY '14, ACM, ISBN:9781450327725.

Kasin, Guillaume, 2012. Engineering Context-Oriented Applications. Université Catholique de Louvain.

Masuhara, Hidehiko, Endoh, Yusuke, Yonezawa, Akinori, 2006. A fine-grained join point model for more reusable aspects. In: Asian Symposium on Programming Languages and Systems. APLAS '06, Springer, pp. 131–147.

Mens, Kim, Capilla, Rafael, Hartmann, Herman, Kropf, Thomas, 2017. Modeling and managing context-aware systems' variability. IEEE Softw. 34 (06), 58–63.

Pape, Tobias, Felgentreff, Tim, Hirschfeld, Robert, 2016. Optimizing sideways composition: Fast context-oriented programming in ContextPyPy. In: International Workshop on Context-Oriented Programming. ACM, pp. 13–20. http://dx.doi.org/10.1145/2951965.2951967.

Ramos, Carlos, Marreiros, Goreti, Santos, Ricardo, 2011. A survey on the use of emotions, mood, and personality in ambient intelligence and smart environments. In: Handbook of Research on Ambient Intelligence and Smart Environments: Trends and Perspectives. IGI Global, pp. 88–107.

Rein, Patrick, Ramson, Stefan, Lincke, Jens, Felgentreff, Tim, Hirschfeld, Robert, 2017. Group-based behavior adaptation mechanisms in object-oriented systems. IEEE Softw. 34 (6), 78–82. http://dx.doi.org/10.1109/MS.2017.4121224.

Salvaneschi, Guido, Ghezzi, Carlo, Pradella, Matteo, 2012a. Context-oriented programming: A software engineering perspective. J. Syst. Softw. (ISSN: 0164-1212) 85, 1801–1817. http://dx.doi.org/10.1016/j.jss.2012.03.024.

Salvaneschi, Guido, Ghezzi, Carlo, Pradella, Matteo, 2012b. ContextErlang: Introducing context-oriented programming in the actor model. In: International Conference on Aspect-Oriented Software Development. ACM Press, pp. 191–202. http://dx.doi.org/10.1145/2162049.2162072.

Sturm, Arnon, Shehory, Onn, 2003. A framework for evaluating agent-oriented methodologies. In: AOIS, Vol. 3030. Springer, pp. 94–109. http://dx.doi.org/10.1007/978-3-540-25943-5_7.

Vallejos, Jorge, 2001. Modularising Context Dependency and Group Behaviour in Ambient-oriented Programming (Ph.D. thesis). Vrije Universiteit Brussel, Brussels, Belgium.

Achiya Elyasaf is in the Software and Information Systems Engineering Department at Ben-Gurion University, Israel. Dr. Elyasaf developed the Context-Oriented Behavioral Programming paradigm. He also developed an approach for utilizing expert knowledge for evolving solvers to combinatorial games. This work was recognized as one of the most important achievements of AI in games, placed next to famous achievements, such as the win of Deep Blue over Kasparov and the win of Watson in Jeopardy! (N. Bostrom, 2014. " Superintelligence: Paths, Dangers, Strategies "). In 2021, he co-founded Provengo technologies which provides software-quality solutions based on his expertise in software engineering (SE) and artificial intelligence (AI). In his current research, Dr. Elyasaf seeks new synergies between SE and AI to improve their interoperability and contribute to both.

Nicolás Cardozo is an associate professor at Universidad de los Andes (Colombia). His research interests include the design and implementation of programming languages for distributed adaptive software systems. Nicolás has worked in implementing dynamic distributed adaptations in the smart cities domain from different perspectives, such as automated personalized assistants and evolutionary models for dynamic adaptations. Currently, he is working on the analysis and verification aspects of adaptive systems at the programming language level.

Arnon Sturm is in the Software and Information Systems Engineering department at the Ben-Gurion University of the Negev, Israel. His research interests focus on languages for various purposes, including software development ranging from end-user programming, and database applications, to complex multi-agent systems and knowledge representation and management. He is interested in developing tools to ease the work of developers throughout the development life cycle.