Structured Behavioral Programming Idioms

Adiel Ashrov[1], Michal Gordon[2], Assaf Marron[3], Arnon Sturm[1](✉), and Gera Weiss[1]

[1] Ben Gurion University of the Negev, Beer Sheva, Israel
{ashrov,geraw}@cs.bgu.ac.il, sturm@bgu.ac.il
[2] Holon Institute of Technology, Holon, Israel
michaligordon@gmail.com
[3] The Weizmann Institute of Science, Rehovot, Israel
assaf.marron@weizmann.ac.il

Abstract. Behavioral Programming (BP) is a modelling and programming technique proposed for specifying and for implementing complex reactive systems. While effective, we report on a weakness that stems from the verbosity and from the complexity of the programming constructs in BP. Our analysis, described in this paper, shows that developers who work with BP use specific patterns that allow them to control the complexity of their specification. Thus, the main contribution of this paper is a set of specification constructs that represent those patterns. We report on the design of the new idioms, termed *structured constructs for behavioral programming* and on an empirical evaluation in a controlled experiment that proved their effectiveness. In particular, the experiment examined the comprehensibility differences between behavioral specifications with non-structured BP programming idioms and with the structured ones. The results indicate that the new structures improve the comprehension of the behavioral specification.

Keywords: Behavioral modeling · Behavioral specification · Behavioral programming · Experimentation · Abstraction · Comprehension

1 Introduction

Behavioral specifications play a crucial role in systems' specifications, in particular, reactive ones. Such specifications are rich and complex and both the research and industry communities are in continuous search for balancing the expressiveness and ease of use of languages and methods that support behavioral specification. A major tool for addressing this challenge is the use of abstraction.

While abstractions, as provided, e.g., by structured idioms, may help developers, they may also be counterproductive in some cases. In [1], for example, the authors show, via empirical evaluation, that the abstraction provided by software visualizations can provide measurable benefits in program comprehension tasks. In [2], however, the authors found that the abstraction provided by object-oriented design documents may be less effective than traditional design documents. The challenge is, of course, to identify those abstractions and structures that simplify the specification without sacrificing the naturalness of the semantics. See, for example, [3] where implementations of a specific set of solutions to problems in several languages are compared.

I. Reinhartz-Berger et al. (Eds.): BPMDS/EMMSAD 2017, LNBIP 287, pp. 319–333, 2017.
DOI: 10.1007/978-3-319-59466-8_20

In a sense, introducing modelling idioms that encapsulate existing idioms but subject them to certain design patterns, facilitates, and encourages the use of, such patterns [4]. This has been demonstrated by many studies on design and programming patterns including proposals for programming structures, for teaching methodologies, for supporting tools, etc. [5–8]. Usually, reports on the effects of patterns are available in anecdotal form from various practitioners [5], yet there are also cases that contain quantitative assessments and include empirical studies [9, 10].

The focus of this paper is on extracting patterns and structures in a particular behavioral specification approach called behavioral programming (BP) [11]. BP is an approach to software development, designed to allow developers to align their implementation with how people often describe a system's behavior. BP is an extension and generalization of scenario-based programming originally introduced with the visual language of live sequence charts (LSC) [12, 13], which has been implemented in several textual/procedural/imperative programming languages including Java [14], C++, Erlang, C, Javascript, and Blockly [15].

The structured idioms that we propose in this paper are in the context of behavior threads in an imperative language as proposed in [14]. In this context, developers specify short concurrent procedures, using standard imperative programming idioms that jointly specify how the system should respond to external events. In this paper, we propose a set of structured idioms for BP and evaluate them in an experiment. For the experiment, we use the Blockly library [15] that allows behavioral specification with a graphical interface through which subjects (software engineering students) construct imperative programs by dragging and dropping blocks (which combine text and graphic cues) on a canvas. Beyond shortening the learning phase and allowing us the flexibility to introduce the structures in a systematic manner, the use of an environment that the subjects were unfamiliar with contributes to the generality of our conclusion (that structures help developers) as the experiment by passes prejudiced preferences or habits.

The contribution of the paper is twofold: (1) A new set of idioms for BP specification is proposed. (2) An empirical evaluation that validates the improvement in comprehension of the new idioms in comparison to the existing idioms is reported upon.

The rest of the paper is organized as follows: In Sect. 2, we introduce the general principles of BP and their application to Blockly as well as the new structured idioms we devised. In Sect. 3 we described the empirical evaluation of the structured idioms in comparison to the previous, direct idioms. Finally, in Sect. 4 we conclude and outline a plan for future research.

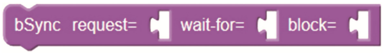
2 Behavioral Programs and Specification Idioms

A behavioral program consists of a set of independent components called behavior threads (b-threads for short) that control the flow of events and synchronize via an enhanced publish/subscribe protocol, as follows. Each b-thread is a procedure running in parallel to the other b-threads. When a b-thread reaches a synchronization point, it waits until all other b-threads also reach synchronization points in their own flow. When entering a synchronization points, each b-thread refers to three sets of events:

1. ***Requested*** events - the thread proposes that these events be considered for triggering, and asks to be notified when any of them occurs;
2. ***Waited-for*** events - the thread does not request these events, but asks to be notified when any of them is triggered; and
3. ***Blocked*** events - the thread forbids triggering of these events.

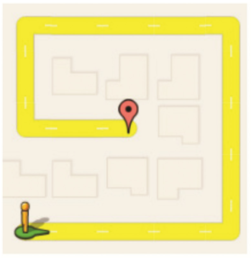
When all b-threads are at a synchronization point, an event that is requested by at least one b-thread and not blocked by any other b-thread is chosen. The selected event is then triggered by resuming all the b-threads that either requested or waited for it. Each of these resumed b-threads then proceeds with its execution, all the way to the next synchronization point, where it again presents new sets of requested, waited-for, and blocked events. The other b-threads remain at their last synchronization points, oblivious to the triggered event, until an event that they have requested or are waiting for is selected. When all b-threads are again at a synchronization point, the event selection process repeats. The translation between inputs, and the events that represent them, and between program-driven events and program outputs are handled by separate sensor and actuator code which of secondary importance in the present discussion.

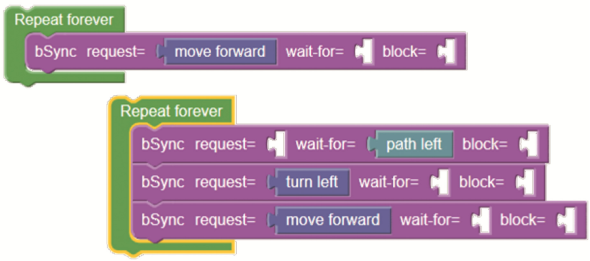
In this work, to enable experimenting with subjects who have minimal background in BP, we used the Blockly library [15] to implement a BP workbench and developed a tutorial that introduces this environment. Blockly is a library for building Scratch-like [16] visual programming environments. In the Blockly environment, we initially introduced the most basic BP idiom, bSync, shown below, very similarly to the way it is used by b-threads in the textual language implementations of BP (Java, C++, etc.) to declare the sets of requested, waited-for and blocked events:



```
bSync request= wait-for= block=
```

With this construct, developers can set the b-threads as ordinary software procedures, as follows. Each b-thread procedure consists of internal computations and bSync blocks. In the internal computation, the b-thread has access to the all events in the model and, when ready, it can declare requested, waited-for, and blocked events by plugging them into the empty slots in the bSync construct. For example, assume that we want to program a robot (drawn as a pegman) to reach a target (drawn as an inverted-drop-shaped icon) moving along a desired path in a maze:





```
Repeat forever
    bSync request= move forward wait-for= block=

Repeat forever
    bSync request= wait-for= path left block=
    bSync request= turn left wait-for= block=
    bSync request= move forward wait-for= block=
```

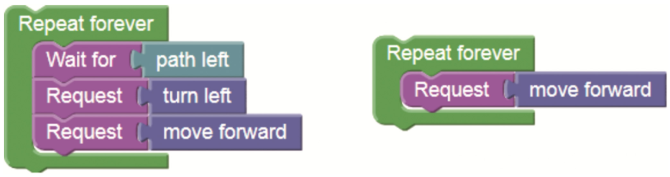
Each of the two disconnected groups of blocks represents a function that acts as a b-thread. The first (top) function always requests a '**`move forward`**' actuation event.

The second (bottom) b-thread, waits for a '**path left**' sensor event and, when it is triggered, requests a '**turn left**' followed by a '**move forward**' actuation events. The joint execution of these two b-threads is as follows:

1. Both b-threads run in parallel until both reach their first **bSync**. The first b-thread requests the event '**move forward**' and the second b-thread waits for the event '**path left**'.
2. The event '**move forward**' is triggered
3. The first b-thread is resumed and returns to the same **bSync**. The second b-thread is not resumed because the triggered event is neither requested nor is waited-for.
4. The above two steps are repeated while a hidden actuator causes the pegman to step forward every time the '**move forward**' event is triggered. After several such rounds, a hidden sensor causes the triggering of the event '**path left**' when the pegman arrives at a location where there is a path to its left.
5. The triggering of the '**path left**' event causes the second b-thread to advance to its second state where it requests the event '**turn left**'. At this point, two events, namely '**turn left**' and '**move forward**', are requested and none of them is blocked. When there are several possible events, the present mechanism triggers the first event in a predetermined order: sensing events come first, then turn events, and then the '**move forward**' event. In our case, the '**turn left**' event is selected. (In other implementations, event selection priorities are replaced, e.g., by additional b-threads that block the selection of events that would be undesired at that state).
6. In the next step, both b-threads request the '**move forward**' event so it is selected. Note that subject to the event selection process described above, the two concurrent requests are satisfied by one triggering of the event, a mechanism termed *unification* of events (see [13] for a detailed discussion on the power of event unification).
7. The run continues similarly until the pegman is at its goal, at which point a corresponding sensor and actuator display an appropriate message and stop the run.

This demonstrates the coding style before the idioms proposed in this paper were introduced, only with the, so-called, bSync idiom. The motivation for designing a new set of idioms for BP came from an examination of how the bSync idiom set is used in practice. We noticed that developers, ourselves included, utilized several patterns while developing applications (see [14, 15]). We then proceeded to extract these patterns into a new set of idioms. The development of the idioms is inspired also by the structures that existed already in the LSC language (see [12, 13, 17]).

The first pattern we noticed was that developers often use bSync with one parameter, which is either a requested or a waited-for event. Our initial simplification, then, was to add an idiom for only requesting events and an idiom for only waiting-for events; these are the Request and Wait-for idioms:

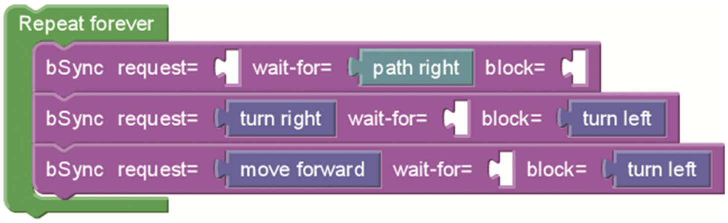


```
Repeat forever
    Wait for   path left
    Request    turn left
    Request    move forward

Repeat forever
    Request    move forward
```

These two idioms are similar to *executed* and *monitored* events in LSC [17].

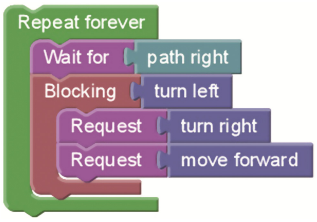
While the use of only `Wait-for` and `Request` idioms may suffice for specifying simple b-threads, we did find examples where these alone are not enough:



```
Repeat forever
    bSync request=      wait-for= path right  block=
    bSync request= turn right  wait-for=  block= turn left
    bSync request= move forward  wait-for=  block= turn left
```

Specifically, we identified cases, as seen in the above example, where developers used more than one parameter of a **bSync**. We then tried to categorize the needs for such usage, and came with two specific patterns as follows. The desired behavior, implemented above, is to complete a right turn while forbidding left turns. A right turn maneuver is composed of two requested events, '**`turn right`**' and '**`move forward`**'. The desired behavior is implemented by blocking the event '**`turn left`**' in every synchronization point that takes place while the right turn process is happening. More generally, the pattern is a sequence of **bSync** declarations with an unwanted event being blocked in each declaration.

Once such a pattern was identified, we designed a language structure to facilitate its use. Specifically, we propose the `Blocking` structure. The `Blocking` structure can be specified with an event and be associated with a scope of a sequence of blocks stacked together. The idiom specifies that the unwanted event is implicitly declared as blocked in each synchronization point under its scope. In the following, we see an equivalent to the b-thread shown above where the **bSync** was transformed to wait-for and request and event blocking declarations are specified using the new structure:



```
Repeat forever
    Wait for path right
    Blocking turn left
        Request turn right
        Request move forward
```

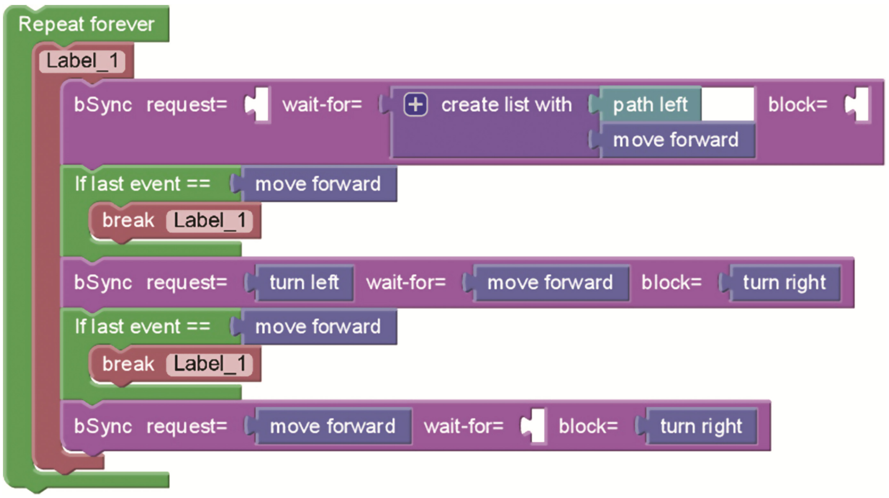
The structure demonstrated in the above specification forces discipline in the sense that it does not allow for developers to specify arbitrary blocking declarations, except for the form of a construct that specifies a scope for each blocking specification.

The second construct proposed in this paper originated from the following issue: We saw in various behavioral specifications that developers deal with situation where an event triggers some fragment of behavior but, before the fragment of the behavior is completed, some other event indicates that the fragment needs to be aborted. For example, imagine two b-threads in a system:

1. A b-thread for handling a right turn by blocking '**`turn left`**' while turning right.
2. A b-thread that requests the '**`turn left`**' event. (In this example, this b-thread is purposely "naïve", in that it does not block any event, and, in particular, does not cause the appearance of a potential deadlock).

When the pegman is faced with a T-junction, both '**`path left`**' and '**`path right`**' will be triggered and both b-threads will be notified about the appropriate event and will progress to the next synchronization point. The b-thread responsible for making a left turn will be blocked by the first b-thread and, as a result, the pegman will turn right. When the first b-thread finishes the turn, it will no longer block the '**`turn left`**' and the second b-thread will be able to proceed. As a result, the pegman will turn left and try to move forward. However, it is unknown if there is a path to the left at the updated location of the pegman. Clearly, this delayed left turn was not intended.

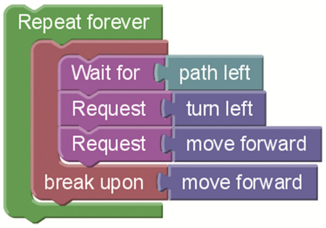
There is a need for stopping and breaking from a block of specification if there is a change in the assumed state for the operation. An example that uses standard Blockly control-flow constructs for the pattern that we want to enforce is the following:



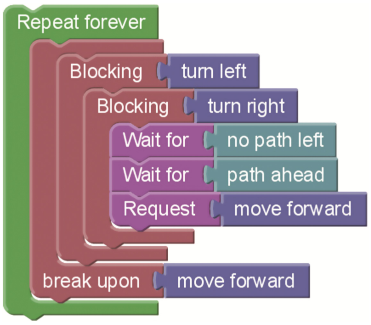
```
Repeat forever
  Label_1
    bSync request= [] wait-for= create list with [path left, move forward] block= []
    If last event == move forward
      break Label_1
    bSync request= turn left wait-for= move forward block= turn right
    If last event == move forward
      break Label_1
    bSync request= move forward wait-for= [] block= turn right
```

Here, the "`if last_event==`" construct is used together with the "`break`" command to test the last event and skip the rest of the sequence of requests if needed.

In general, the pattern we identified is a fragment of the specification in which the developers wish to abort preemptively when a certain event arrives. To allow this, we propose the idiom of '`break-upon`'. The specification below has the same logic presented above. We wait-for a '`path left`' event and then perform two requests in order to turn left. These instructions are under the scope of the break-upon idiom specified with '**`move forward`**'. This induces a `wait-for` '**`move forward`**' declaration in each of the nested synchronization points. If this event is triggered, the flow breaks from the `break-upon` scope and go back to the beginning of the loop.



Note that the semantics of the two idioms allow nesting:



Formally, the semantics (and the implementation) of the new idioms is as follows: `Request` and `Wait for` are defined as a bSync call with two of the event sets being empty; the `blocking` idiom adds the blocked events to the blocked-event set in each synchronization point within its scope; and, the `break-upon` idiom adds to the waited-for event set in each of the synchronization points in its scope the referenced event, and adds after each such synchronization point the statement "`if lastEvent==<the break-upon event> then break`". A synchronization point can be nested in several scopes, as shown in the above example, in which case multiple such actions may apply to it.

To summarize, in this paper we are proposing new idioms for behavioral programming specification. These idioms where identified by an examination of behavioral specification and of the needs of developers that work with the language. We have proved that the new set of idioms, called structured idioms below, is equivalent in expressive power to the base language, called **bSync** (see the Appendix).

3 Evaluating BP Structured Idioms

To evaluate the benefits of using the BP structured idioms introduced before, we conducted a controlled experiment to compare the usage of the **bSync** and the **Structured** idioms with respect to the comprehension of a BP specification. In particular, we were interested in comparing two issues related to comprehension. The first is the execution semantics, i.e., the order of handled events and the second is the intention behind the developed specification. Our initial conjecture was that with respect to execution semantics comprehension, the results would be in favor of the **bSync** idioms since the statements are explicated in one block, in particular, in simple cases. With respect to the comprehension of the specification intention, our hypothesis was that the results would be in favor of the **Structured** idioms as the specification is more organized. We followed the experiment model that appears in Fig. 1.

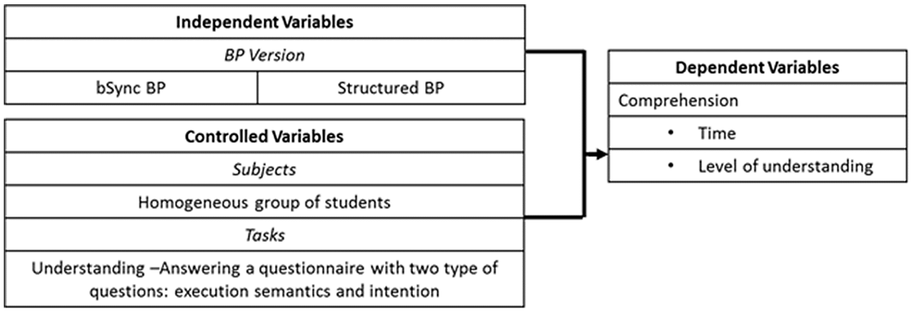


Fig. 1. The experiment model

3.1 The Experiment Settings

The Independent Variable. As the objective of the experiment was to evaluate the comprehension of the two BP versions over a set of dependent variables, the independent variable is the BP version used, i.e., **bSync** versus **Structured**.

The Dependent Variables. Following the experiment goal, the dependent variables are the level of understanding a behavioral specification and the time it takes to reach that understanding. The level of understanding was determined by a set of multiple-choice questions and the time was self-measured by the subjects.

Subjects. In the experiment, 49 software engineering students in their third year of studies participated; 22 were assigned to the **Structured** based specifications and 27 were assigned to the **bSync** based specification. They were randomly assigned to the specific set of idioms. Examining both the students' GPA and their grade in the "Topics in Software Engineering" course (from which the students were volunteered), we found no statistically significance differences (T-test: $p > 0.51$, $p > 0.69$). All subjects were at an advanced stage in their studies and already took courses related to programming and modeling. The participation in the experiment was on a voluntary basis. Yet, in order to motivate and encourage the subjects to participate (and in an effective way), they were told in advance that they would get a bonus in the aforementioned course based on their performance in the experiment. In addition, all participants signed a consent form on which they were explicitly informed that they could withdraw from the experiment at any time.[1]

[1] The experiment design and execution was also approved by an ethical committee.

Training. In the beginning of the semester, the students were provided with a lecture about behavioral specification. In addition, the first part of the experiment was devoted to introduce the subjects with their assigned behavioral specification idioms set. It was an online tutorial to Behavioral Programming using a Maze[2] application, which we mentioned in the previous section. The main purpose of the tutorial was to teach BP principles using the assigned set of idioms. We believe that the fact that Blockly and BP are not common development environments that students interacted with during their studies, freed them from existing knowledge and beliefs, and allowed them to experience the tutorial from a fresh point of view. The tutorial consisted of nine challenges (levels) in solving mazes where every challenge introduced new concepts/idioms. In every challenge, the goal was to develop a behavioral specification, which will lead a pegman to a destination.

Task. Upon completing the introductory tutorial, the subjects were requested to answer a questionnaire consisting of questions about their perceptions over BP, in general, and to its specific concepts and mechanisms, followed by 16 BP specification questions[3]. The latter were classified as appears in Table 1.

Table 1. BP specification question classification

	Execution	Intention
Request + Wait (RW)	Q1, Q9, Q4, Q12	
Request + Wait + Blocking (RWBL)	Q3, Q11	Q5, Q13
Request + Wait + Breaking (RWBR)	Q2, Q10	Q6, Q14
Request + Wait + Blocking + Breaking (RWBLBR)		Q7, Q8, Q15, Q16

Execution. The experiment took place in several sessions within a lab, in which the subjects started with a tutorial demonstrating and teaching the BP concepts and then answered the questionnaire. Each session (including the tutorial) lasted one and half-hour long.

3.2 Experimental Results

We first review the students' perception over the knowledge and understanding of BP. Table 2 summarizes the results, where the average of the scores given by the students is in the scale of [0-4], are shown for each question and for each version of BP. It seems that from the lecture on BP the students who were assigned the **bSync** version perceived their understanding of BP a bit better than those who were assigned with the **Structured** version (yet, this was not statistically significant). This might be because the **bSync** version is related to the code version shown in class. Nevertheless, based on the tutorial, it seems that the students that were trained with the **Structured** version perceived their BP understanding higher than those who were trained with the **bSync** version. Applying

[2] https://bp-new-blockly-exp-2.appspot.com/static/apps/mazeBP/index.html?version=1(version=2). Designed to work with the Firefox web browser.

[3] The questionnaires can be found in https://tinyurl.com/h3bzphs, https://tinyurl.com/zah4ap3.

the Mann-Whitney test, we found that the differences were statistically significant. In general, in both groups, the students perceived their understanding of the BP concepts in the order for high to low comprehension level: blocking, breaking out, multiple events, and the execution mechanism.

Table 2. Students Perecptions over BP

Question	Structured	bSync	Sig. (M-W)
To what extent did you understand the concept of BP from that lecture?	1.64	1.78	0.497
Do you perceive the BP tutorial helpful?	2.55	2.04	**0.004**
Do you understand the execution mechanism of BP (i.e., how events are chosen)?	2.59	2.00	**0.000**
Do you understand the concept of having the same event requested by multiple threads/scenarios?	2.77	2.30	**0.012**
Do you understand the concept of event blocking?	3.14	2.41	**0.001**
Do you understand the concept of "breaking out" from a scope in the program?	2.86	2.41	**0.002**

Next, we review the results of the questions that objectively examined the students' understanding of BP. As shown in Table 3, in general, the level of understanding of BP (measured by the number of correct answers) both with respect to the execution semantics and with respect to the intention was in favor of the students who were trained and questioned with the **Structured** version. Those differences were also statistically significant. Nevertheless, no statistical significant differences were found with respect to the time it took to answer the various questions.

Table 3. Comprehension average scores (and standard deviation) on BP specification questions

	Level of understanding			Time (min)		
	bSync	Structured	Sig. (T-test)	bSync	Structured	Sig. (T-test)
Total	7.41 (2.17)	10.91 (2.83)	**<0.001**	24.30 (8.02)	23.23 (6.39)	0.61
Execution semantics	3.19 (1.39)	4.95 (2.06)	**<0.001**	9.48 (3.77)	9.86 (4.14)	0.74
Intention	4.22 (1.37)	5.95 (1.46)	**<0.001**	14.81 (4.97)	13.36 (3.66)	0.25

We further drilled down to check what concepts or mechanisms caused that differences between the comprehensions of the two idioms set.

Table 4 shows the results with respect to the categories defined in Table 1. It seems that the most significant concepts that caused the difference in understanding the specification is the breaking out concept (RWBR-E), that also appeared in the more complex specifications (RWBLBR-I).

Table 4. Comprehension average scores (and standard deviation) on BP specification questions with respect to BP concepts

	RW- E	RWBL- E	RWBR- E	RWBL-I	RWBR-I	RWBLBR-I
Structured	2.14 (1.13)	1.27 (0.77)	1.55 (0.67)	1.77 (0.53)	1.45 (0.51)	2.73 (0.88)
bSync	1.89 (0.93)	1.04 (0.81)	0.26 (0.45)	1.67 (0.48)	1.19 (0.74)	1.37 (0.93)
Sig. (T-test)	0.41	0.30	**0.00**	0.47	0.14	**0.00**

* E- Execution, I- Intention

3.3 Discussion

The results confirmed our conjecture regarding the intention comprehension but not our conjecture regarding the execution semantics. As we further look for explanations for the results, we opt for ontological analysis of language grammars [18] in which the mapping process of the language grammar (BP version, in our case) to the ontological world (the execution semantics and intention, in our case) may suffer from various deficiencies. One such deficiency is the construct overload: “Construct overload occurs when one design construct maps into two or more ontological constructs”. In bSync (a design construct) this is exactly the case in which one statement is mapped into several event types (ontological constructs). This deficiency, that affects the mapping to the actual execution of the program, seems, according to our findings, to be the root cause for the results we get in the experiment.

Another alternative for interpreting the results is to look at cognitive analysis frameworks. The COGEVAL is one such framework [19]. We find it appropriate for our analysis as it refers to various cognitive aspects of modeling and because the framework has been demonstrated to be useful in many cases.

The framework consists of several propositions among which we found the following well fitted to our case:

P1: “The greater the degree of chunking supported by a model, the greater the modeling effectiveness.”

The results suggest that the structured BP increases the degree of chunking with respect of treating events as it has a unique concept that treats each event type separately. Thus, in simple cases of request, wait-for, and blocking no chunking was required (as these were simple to understand), no statistically significant differences were found between the comprehension of the two sets of BP idioms. However, in the case of the more complex tasks, consisting the breaking out, chunking was beneficial and as the **Structured** BP increases the degree of chunking and using it was found useful in terms of effectiveness (correctness).

As for the other way around, considering that increased chunking may be of a disadvantage in some aspects, COGEVAL suggest that:

P2: “The greater the number of simultaneous items required (over seven) to create schema segments or chunks, the lower the modeling effectiveness of the model.”

In our case, the **Structured** BP uses a limited number (three) of simultaneous items to specify the same concept as in the **bSync** BP. This explains the reason that the **Structured** BP did not negatively affect the effectiveness.

3.4 Threat to Validity

Several threats to the validity may be encountered when discussing the experiment.

Conclusion validity concerns with issues that affect the ability to infer the correct conclusion regarding the relations between the treatment and the outcome of the experiment. As we believe that the only variable that was changed during the experiment was the used version of BP, we believe that our conclusions are valid and no major treat exists.

Internal validity is the degree to which conclusions can be drawn about the causal effect of the independent variables on the dependent variables. By randomly assigning the subjects to the two versions, we neutralized most of the possible effects that may have influenced the independent variable of participants, such as experience, training, and personal characteristics. An additional threat to internal validity is the quality of training, which in our case was the tutorial. It might be that such a training is not enough; yet, the scope of the training for both groups was the same.

External validity concerns with the ability to generalize experimental results outside the experimental settings. An external validity threat is always present when experimenting with students, as the issue of whether they are representative of software professionals is raised. However, in this experiment, we checked differences among similar groups, the experiment theme is not familiar to professionals, and as the students are in their third year, they have already gained some experience. Another external validity issue, which is unfortunately inherent to controlled experiments, is the size and complexity of the tasks used. The size of the various tasks is small; nevertheless, controlled experiments require that subjects complete the assigned task in a limited amount of time and without interruption to keep variables under control.

4 Conclusions

We addressed the concern of behavioral specification abstraction. In particular, we referred to a specific approach to behavioral specification called behavioral programming (BP) that was equipped with modeling capabilities of Blockly. Within the context of that language we compared existing idioms (bSync) with a new set of structured idioms. Our comparison was in terms of understanding behavioral specification. The results of the controlled experiment indicated that abstracting the specification leads to a higher level of understanding. This was evident in particular in cases when a main flow allows for breaking out.

The results of the evaluation performed in this research indicate that it is desirable to abstract specification constructs (i.e., modeling or programming idioms) without scarifying their expressiveness as their usability is better than their low-level counterparts. This is particularly relevant when complex constructs are introduced.

Clearly, such results should be taken with caution and further examination is required. Thus, in the future, we plan to further experiment the two versions (and maybe other alternatives) to further understand the conceptual differences among them so to better design the right abstractions, in BP and in other languages.

Acknowledgement. This research was partially supported by the Israel Science Foundation, the Philip M. Klutznick Research Fund, and the Dora Joachimowicz research grant.

Appendix

While the focus of this work is on empirical evaluation of the idioms, in terms of their usability to programmers, we hereby provide a proof for the expressiveness equivalence of the two sets of idioms.

Claim 1: Every b-thread developed with the structured idioms can be translated to a semantically equivalent b-thread that applies only the `bSync` idiom using `If last_event==x {}` and `break`.

Proof: By structural induction.

Induction base: if there is no `Break-upon` or `Blocking` structures in the program, replace each `request(x)` and each `wait-for(y)` with, respectively, a `bSync(x,none,none)` and a `bSync(none,y,none)`.

Induction step: Assume, by induction, that the claim is true for programs with at most n levels of nesting of the `Break-upon` or `Blocking` structures. Given a program with $n + 1$ levels of nesting we can replace each block of maximal nesting level as follows. If the nesting level of the block is less than $n + 1$, it can be replaced by a code that uses only `bSync` by the induction assumption. For a block of the form `blocking(x){P}`, where P is some code with n levels of nesting, let P' be a code, given by the induction assumption, that is semantically equivalent to P and uses only the `bSync` idiom. Let P'' be the code obtained by adding x to the list of blocked events in every `bSync` in P'. We can now replace the code `Blocking (x) {P}` with P''. Similarly, for a block of the form `{P} break-upon(x)`, where P is some code with n levels of nesting, let P' be a code, given by the induction assumption, that is semantically equivalent to P. Let P'' be the code obtained by adding x to the list of waited-for events in each bSync in P'. Let ***label*** be a name that does not appear as a label in P'' and let P''' be the code obtained by adding the commands `If last_event==x{break` ***label*** } after each `bSync` in P''. We can now replace the code `{P} break-upon(x)` with ***label***: $\{P''\}$. Clearly, the code obtained after all the above replacements contains only `bSync`, with no application of the structured idioms, and is semantically equivalent to the original code. This proves our claim. ∎

In fact, the above proof follows the way we have implemented the structured idioms.

Another issue that should be clarified in the above proof is the semantics of waiting-for or requesting an event while, at the same time, blocking it. This can be done with `bSync`, when some event is specified both in the `waited-for` or in the `requested` lists and in the `blocked` list. It can also be done with the structured idioms, when an event is requested or waited-for in the scope where it is specified as blocked or a `break upon` event. In this case, one must decide which specification takes priority – whether to ignore the blocking or to ignore the request or wait-for. It easy to verify that the above proof is correct as long as we use the same priority order in both the `bSync` and in the structured idioms.

Claim 2 Every b-thread developed with `bSync` idiom can be translated to a semantically equivalent b-thread that uses only the structured idioms.

Proof Replace each `bSync(`*x*`,`*y*`,`*z*`)` with `blocking(`*x*`){break-upon(`*y*`)` `{request(`*x*`)}}.` ∎

References

1. Hendrix, T.D., Cross II, J.H., Maghsoodloo, S., McKinney, M.L.: Do visualizations improve program comprehensibility? experiments with control structure diagrams for Java. ACM SIGCSE Bull. **32**(1), 382–386 (2000)
2. Briand, L.C., Bunse, C., Daly, J.W., Differding, C.: An experimental comparison of the maintainability of object-oriented and structured design documents. Empirical Softw. Eng. **2**(3), 291–312 (1997)
3. Feo, J.T.: A Comparative Study of Parallel Programming Languages: The Salishan Problems. Elsevier, North Holland (2014)
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, Boston (1995)
5. Beck, K., Crocker, R., Meszaros, G., Vlissides, J., Coplien, J.O., Dominick, L., Paulisch, F.: Industrial experience with design patterns. In: Proceedings of the 18th International Conference on Software Engineering (1996)
6. Budinsky, F.J., Finnie, M.A., Vlissides, J.M., Yu, P.S.: Automatic code generation from design patterns. IBM Syst. J. **35**(2), 151–171 (1996)
7. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. Wiley, New York (1996)
8. Florijn, G., Meijers, M., Winsen, P.: Tool support for object-oriented patterns. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 472–495. Springer, Heidelberg (1997). doi:10.1007/BFb0053391
9. Prechelt, L., Unger, B., Philippsen, M., Tichy, W.: Two controlled experiments assessing the usefulness of design pattern information during program maintenance. IEEE Trans. Software Eng. **28**(6), 595–606 (2002)
10. Prechelt, L., Unger, B., Tichy, W.F., Brossler, P., Votta, L.G.: A controlled experiment in maintenance: comparing design patterns to simpler solutions. IEEE Trans. Softw. Eng. **27**(12), 1134–1144 (2001)
11. Harel, D., Marron, A., Weiss, G.: Behavioral programming. Commun. ACM **55**(7), 90–100 (2012)
12. Damm, W., Harel, D.: LSCs: breathing life into message sequence charts. J. Formal Methods Syst. Des. **19**(1), 45–80 (2001)
13. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Heidelberg (2003)

14. Harel, D., Marron, A., Weiss, G.: Programming coordinated behavior in Java. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 250–274. Springer, Heidelberg (2010). doi: 10.1007/978-3-642-14107-2_12
15. Ashrov, A., Marron, A., Weiss, G., Wiener, G.: A use-case for behavioral programming: an architecture in JavaScript and Blockly for interactive applications with cross-cutting scenarios. Sci. Comput. Program. **98**(Part 2), 268–292 (2015)
16. Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., et al.: Scratch: Programming for all. Comm. ACM **52**(11), 60–67 (2009)
17. Maoz, S., Harel, D., Kleinbort, A.: A compiler for multi-modal scenarios: transforming LSCs into AspectJ. ACM Trans. Softw. Eng. Methodol. (TOSEM) **20**(4) (2011). Article 18
18. Wand, Y., Weber, R.: On the ontological expressiveness of information systems analysis and design grammars. Inform. Syst. J. **3**, 217–237 (1993)
19. Bajaj, A., Rockwell, S.: COGEVAL: applying cognitive theories to evaluate conceptual models. In: Advanced Topics in Database Research, pp. 255–282 (2005)