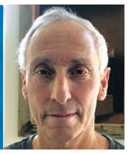
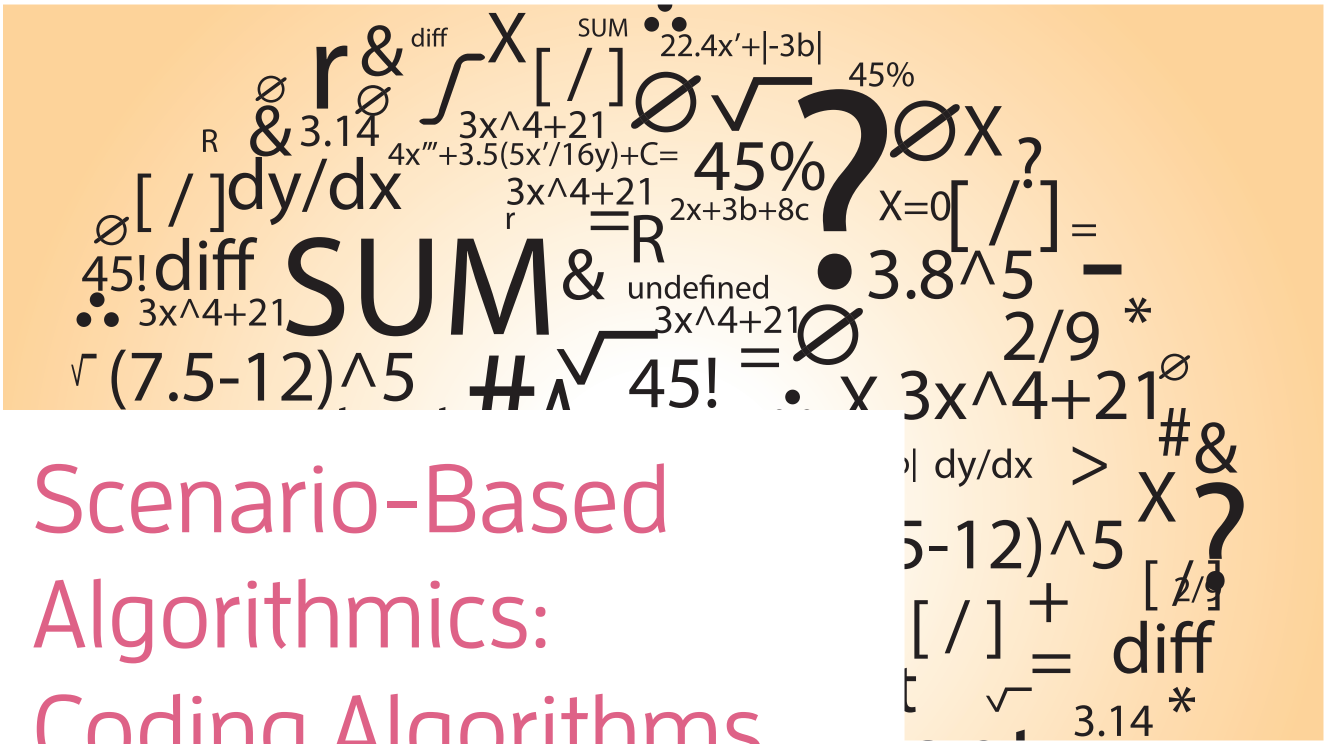
EDITOR DORON DRUSINSKY
Naval Postgraduate School ddrusins@nps.edu





Scenario-Based Algorithmics: Coding Algorithms by Automatic Composition of Separate Concerns

David Harel, Assaf Marron, and Raz Yerushalmi, Weizmann Institute of Science

A method for programming reactive systems, called scenario-based algorithmics, can have several advantages, both in programming and in computer science education. We provide new examples, experiments, and perspectives.

Digital Object Identifier 10.1109/MC.2021.3099044
Date of current version: 24 September 2021

Consider the following challenge: A manager in a car dealership wishes to rearrange the cars in the dealership's lot according to some arbitrary sort key, such as window-sticker price or engine serial number. The lot is full, and passage lanes should not be blocked. Can the manager give employees simple instructions to accomplish an efficient in-place sorting algorithm like Quicksort—not as a long sequence of steps but as an unordered set of rules? We argue that the answer is positive and discuss a new approach to specifying algorithms that embodies this answer.

Textbook specifications of procedural code for an algorithm, such as Quicksort, binary search, or two-phase commit, are usually accompanied by a description of the key tenets that distinguish this algorithm from other solutions to the problem. These explanations are needed because the code does not readily communicate all of its underlying principles: important algorithmic innovations may be hidden among steps for bookkeeping and the handling of special conditions or implied by a unique ordering of instructions and placement of parentheses, and so on. Giving functions and variables meaningful names helps but cannot fully fill this gap.

In Harel and Marron 2018,[18] we introduced *scenario-based algorithmics* (*SBA*), a new approach to specifying algorithms. Rather than providing step-by-step instructions, one codes a collection of succinct self-standing executable rules. Each rule, also termed a *scenario*, is responsible for one aspect of overall system behavior, stating what must or must not be done when certain conditions hold or following certain sequences of events. All of the scenarios are executed in parallel and in lock-step synchronization with each other. At each synchronization point, all of the rules' declarations are evaluated and reconciled, one event is triggered, all affected scenarios are notified, and, if relevant, they react to the event and step to their next synchronization point, and the process repeats. The advantages of the approach include ease of understanding, amenability to formal and compositional verification, and accommodation of incremental enhancement and refinement. We claim that, in addition, SBA may contribute to general human computational thinking skills.

The SBA specification idioms we proposed[18] are taken from a programming approach termed *scenario-based programming* (SBP)[3,8,13,14,19] as well as *behavioral programming*, which was originally intended for specifying reactive systems.[15] We believe that it is well suited for the fine separation of concerns present in the underlying idea of SBA. Here, we briefly introduce SBP, recap the proposal to use SBP for SBA, and offer new examples, methodological perspectives, and future research directions for advancing this approach for specifying algorithms.

SCENARIO-BASED PROGRAMMING

SBP aims to simplify the development of executable models of reactive systems by bringing the process closer to the way humans think about system behavior, adopting an interobject approach rather than the conventional intraobject one.[3,8,13] At the heart of SBP is the scenario, describing a desired or undesired aspect of behavior of the system, possibly involving multiple objects. Cohesive system behavior emerges from a novel concurrent synchronized execution of all scenarios.[13,19]

At every synchronization point, each scenario pauses and declares events that it requests and events that it blocks, encoding, respectively, presently desirable and forbidden actions. Scenarios can also declare events that they passively wait for, thus asking to be notified when they occur. These declarations are collected by a central event selection mechanism, which selects and triggers one of the events that is requested by at least one scenario but is blocked by none. Scenarios that requested or waited for this event are then informed and can proceed to a new synchronization point. Sensor and actuator scenarios translate between environmental occurrences and SBP events, using external programming interfaces to cameras, motors, and so on. The specification of requested, blocked, and waited-for events can include sets of events. These sets can even be infinite, in which case dynamic function invocations deliver a next concrete set element or answer a set-membership query. Figure 1 depicts an SBP specification of a simple reactive system.

SBP concepts are independent of their encoding and visualization. The first implementation was embodied in the live sequence chart language, where programs are modal sequence diagrams.[3,13] This was followed by implementations in Java method calls and in several other languages, including C, C++, Erlang, JavaScript, and Python (see, for example, Harel et al. 2012[19]). SBP was also implemented via an executable controlled natural language,[5] in the visual Scratch-like Blockly language, and in special-purpose textual languages (domain-specific languages).[6]

Selecting an event among all those that are requested and not blocked can be subjected to a policy: arbitrary, randomized, based on predefined scenario priorities and precedence, based on look-ahead for achieving certain outcomes,[11] or based on a synthesized controller that prescribes in advance the selected event for every synchronization point, among others. SBP is also amenable to model checking and compositional verification.[10,12] The result is a formal executable specification system that facilitates incremental development[2,4] and is aligned naturally with the way humans often describe complex dynamic phenomena: detailing aspects of behavior, one at a time, while relying on a common understanding of how these are composed.

SCENARIO-BASED ALGORITHMICS

SBA concepts are best explained via examples.

A basic example: The factorial function

Ask someone what "n factorial" (*n*!) is, and the verbal answer might be "one times two times three and so on, up to *n*." Standard procedural programs calculate the factorial function with a simple loop of multiplications. By contrast, the SBA specification of *n* factorial shown in Figure 2 consists of a number of scenarios, which communicate via a single event containing the string "`CALC_NEXT`" and an integer. The scenarios are implemented as coroutines/generators in Python, using the `yield` command to synchronize with each other while declaring their sets of requested, waited-for, or blocked events.

The scenario `mult_next_actuator` reacts to occurrences of "`CALC_NEXT`" and multiplies an evolving result by the provided integer. Following the definition of *n* factorial as the product of all integers between 1 and *n*, the scenario multiply_all repeatedly requests a static, randomly ordered set of CALC_NEXT events with all of these integers. This highlights the often hidden fact that the multiplications in computing the factorial function can be carried out in any order. The scenario multiply_only_once ensures that each factor is used only once in computing the product, highlighting another often-implicit trait of the function. The scenario result_monitor accesses the environment directly and prints the evolving result; had other scenarios needed the result, result_monitor could provide it by requesting another type of event containing this variable's value. Should one wish to enforce, say, a descending order of multiplication, another scenario (not shown here) can block CALC_NEXT events if there is a higher-valued one that was not hitherto invoked. For the full code for this and other examples, see our website at http://www.wisdom.weizmann.ac.il/~bprogram/sba/.

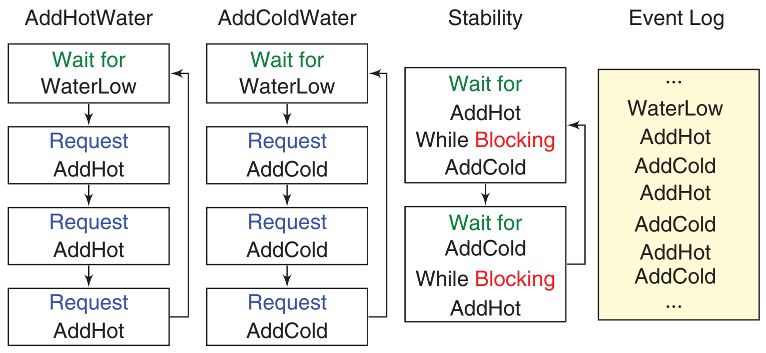


FIGURE 1. SBP specifications for controlling the temperature and fluid level in a water tank. Each scenario is given as a transition system, where the nodes represent synchronization points. The transition edges are associated with the requested or waited-for events in the preceding node. The scenarios AddHotWater and AddColdWater repeatedly wait for WaterLow events and then request three times the event AddHot or AddCold, respectively. Since, by default, these six events may be triggered in any order, a new scenario Stability is introduced, with the intent of keeping the temperature more stable. It enforces the interleaving of AddHot and AddCold events by alternately blocking them. The resulting execution trace is depicted in the event log.

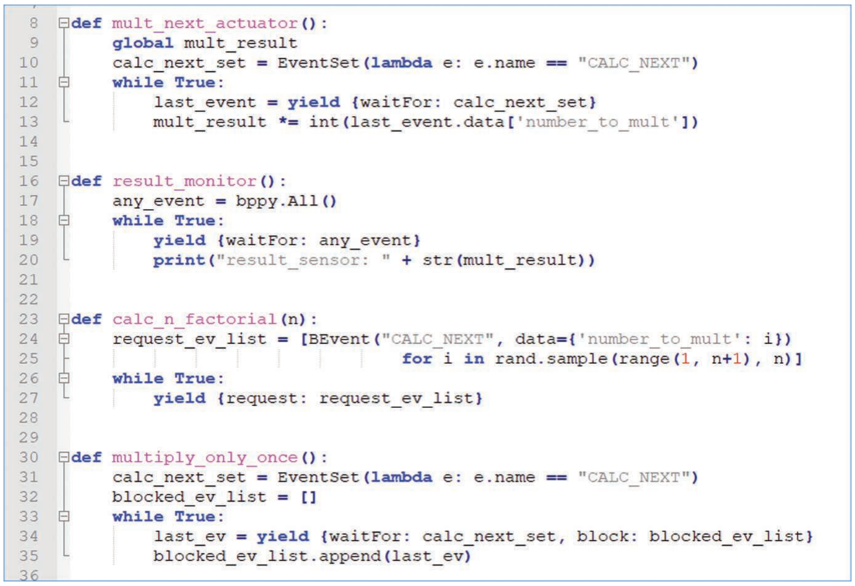
Quicksort and the separation of concerns

For a more elaborate algorithm, recall the challenge of sorting cars in a dealership lot and assume that parking spots are numbered sequentially. One might use the naive statement that lies at the heart of Bubble sort: (a) if a car in a given spot has a higher sort key than the car in the next spot, swap the locations of these two cars; (b) repeat the process until there are no such unordered pairs. In Harel and Marron 2018,[18] we describe in detail an SBP implementation of Bubble sort. The central scenario always compares and swaps adjacent elements. Bookkeeping scenarios proceed from one pair of elements to the next, start a new pass when ending a pass, skip the rechecking of already-sorted prefixes of the array, start and end the entire process, and so on. We also discuss the merits of this implementation of Bubble sort as compared with the classical procedural code containing two nested loops.[18]

We now tackle the challenge of applying SBA to Quicksort (see, for example, Aho et al.[1]). Recursive descriptions of Quicksort usually state the following: pick a pivot element and rearrange the array such that all elements smaller than or equal to the pivot reside at lower indices and those greater than the pivot are at higher indices; then, repeat the process for the two parts of the array just formed by the pivot.

Here is a suggested SBA phrasing of Quicksort in the form of rules for rearranging the cars in the dealership lot. The rules are given as a repeatable program, which should be carried out each morning, perhaps with a different sort key every time. We also assume that sort keys are accessed only physically in or on the cars.

1. Every morning, before starting this process, close the trunks of all the cars and place marks



```
def mult_next_actuator():
    global mult_result
    calc_next_set = EventSet(lambda e: e.name == "CALC_NEXT")
    while True:
        last_event = yield {waitFor: calc_next_set}
        mult_result *= int(last_event.data['number_to_mult'])


def result_monitor():
    any_event = bppy.All()
    while True:
        yield {waitFor: any_event}
        print("result_sensor: " + str(mult_result))


def calc_n_factorial(n):
    request_ev_list = [BEvent("CALC_NEXT", data={'number_to_mult': i})
                       for i in rand.sample(range(1, n+1), n)]
    while True:
        yield {request: request_ev_list}


def multiply_only_once():
    calc_next_set = EventSet(lambda e: e.name == "CALC_NEXT")
    blocked_ev_list = []
    while True:
        last_ev = yield {waitFor: calc_next_set, block: blocked_ev_list}
        blocked_ev_list.append(last_ev)

```

Figure 2. The SBA specification for the factorial function, coded in SBP in Python.

before the first car and after the last car.
2. Always, after placing a mark, look for a continuous stretch of two or more unmarked cars between two marks, which is not being processed (there is no driven car in this stretch), start processing this stretch: drive the farthest car in this stretch out of its spot and stop it next to the first spot in the stretch.
3. Always, after the driven car has just stopped by a parked car, check if the parked car's sort key is greater than that of the driven car. If so, drive that parked car to the empty spot in this stretch and park it there. (Note that there will always be one such empty spot.) If not, drive the driven car one spot up and stop it by that spot.
4. Always, after parking a car that is not "the driven car" (this is always a recently compared car being parked in a higher spot), walk down along the parked cars from that spot to smaller spot numbers and find the first parked car whose sort key is smaller than that of the driven car. If such a car is found, drive it to the spot that was just freed next to the driven car, then drive the driven car to the next higher spot and stop it next to it. If a parked car with a smaller sort key is not found before reaching the driven car, park the driven car in the empty spot by it.
5. Always, when the driven car stops by an empty spot (rather than by a parked car), park it there.
6. Always, after parking the driven car, mark it by opening its trunk. (This stretch is no longer being processed and this car is no longer a "driven car.")

In the SBA code, the ”commands” are represented by requests for events such as `DRIVE-CAR-TO-NEXT-TO-SPOT`, `PARK-CAR-IN-SPOT`, `MARK-CAR`, and so on, each with its own parameter data. Sensor scenarios retrieve sort keys from cars and announce them to other scenarios by requesting designated events. Actuator scenarios actually move the cars among locations. While leaving some room for further efficiency improvements, in these Quicksort instructions the next processed partition is always the highest one in the array, and the selected pivot is always the last item in the partition.

It is important to note that the rules can be provided in any order, as long as they are all considered all of the time. The rules can also be carried out by different people responsible for different rules if all agree on the names of events and conditions. To illustrate the incrementality property of SBP and hence also of SBA, it is quite easy to add a scenario that detects when uninvited cars are parked at spots needed for the sorting process and blocks certain car-movement events until they are removed, while temporarily suspending the processing of a stretch.

The main goal of this article is to show that many algorithms can be described textually with such an executable separation of concerns. It will then become natural to seek appropriate programming idioms, composition mechanisms, and development methodologies that simplify the creation of executable SBA code for any algorithm while maintaining such a separation of concerns.

We are not trying to address the apparent nonintuitiveness of some programming language constructs, and SBA's contribution is not in the optional usage of natural language. Instead, we want to mimic the ability of an expert engineer to explain important aspects of the algorithm to a novice, even when 1) such an aspect may involve multiple nonadjacent lines of code, 2) a given line of code serves multiple such aspects, and 3) a key aspect is "hidden" in a seemingly unimportant piece of notation, like a pair of parentheses or a –1 in an arithmetical expression.

Binary search and the fine separation of concerns

We use an SBA implementation of binary search for an integer in an array to illustrate fine-grain separation of concerns. This implementation is similar to a classical procedural program for binary search, except that `if` statements have no else clauses; the else clauses are handled as separate concerns. As will become obvious, this synthetic example is intended only to demonstrate SBA and not necessarily to produce insights into the algorithm itself.

Here is how the SBA specification works. The `B-SEARCH` event guides the search; it contains three integers: the search argument and two indices demarcating a range within the array where the next part of the search should be conducted. The `FOUND` and `NOTFOUND` events announce the results of the search. The scenarios are as follows:

1. a scenario that initiates the process
2. a scenario that waits for `FOUND` or `NOTFOUND` events and stops the process by blocking all subsequent events
3. a scenario that waits for `B-SEARCH` events and, if the search argument is smaller than the value in the middle of the range of the array specified in the event, requests `B-SEARCH`, with the low half of the range
4. a scenario that waits for `B-SEARCH` events and, if the search argument is greater than the value in the middle of the range of the array specified in the event, requests `B-SEARCH`, with the high half of the range
5. a scenario that waits for `B-SEARCH` events and, if the search argument is equal to the value in the middle of the range of the array specified in the event, requests the `FOUND` event, with the index of the middle of the range

6. a scenario that waits for B-SEARCH events and, if the range is invalid, for example, outside of the original array or the low end is greater than the high end, requests the NOTFOUND event.

See the “Programming Scenarios” and “Classical Method Calls Versus Reacting to Events” sections for discussions of design choices related to granularity of separation of concerns and the differences between a standard procedural function call and causing a scenario to react to an event.

The sieve of Eratosthenes and dynamic scenarios

SBA also allows for the dynamic creation and deletion of scenarios. Consider the sieve of Eratosthenes algorithm for printing a list of prime numbers. In a classical programming approach, one uses an array whose cells correspond to integers, where multiples of discovered primes are marked as nonprimes and unmarked cells are finally output sequentially as primes. In the SBA implementation discussed in Harel and Marron 2018,[18] one of the scenarios requests, one at a time, events that proclaim every integer as a prime, but it abandons the request if an event requested by another scenario first proclaims that integer to be nonprime. Another scenario waits for any event that proclaims an integer to be prime and dynamically creates a higher priority scenario, which sequentially proclaims all multiples of that prime to be nonprimes. In a variant of that implementation, after every proclamation of an integer as prime, a scenario is dynamically created that blocks the proclamation of all multiples of that integer as primes.

METHODOLOGICAL NOTES

The importance of specification completeness

In human-to-human specification of an algorithm, certain issues may be left unspecified, either because they are obvious or implicit, or because they can be changed without violating key requirements. Thus, in the car dealership example, we omitted instructions for how to end the sorting process or how to identify a stretch of unmarked cars. For an SBA specification to be complete and to terminate properly rather than just stall forever, such a specification must be explicit. We argue that distilling these seemingly more trivial items as separate scenarios adds to one’s understanding of the overall algorithm.

Execution semantics

Explicit execution order. In classical procedural programming, for a human reading the code, the order by which instructions are executed is implied either by their location in the text or by the semantics of constructs like `if` statements, branching, `while` and `for` loops, and so on. In SBA, such ordering must be specified explicitly: ”Always, after doing <this>, <do that>.” The specification thus tells the reader whether or not instructions that appeared consecutively in the classical procedural version of code must be executed in that order. Such explicit specification of order is also relied upon in other approaches to separation and composition of concerns. For example, in aspect-oriented programming, programmers specify whether an aspect function should be executed before, after, or instead of the runtime activation of certain other program functions.

Programming scenarios. In SBA and SBP, every scenario is coded in a programming language and can include multiple commands, conditions, flow control, and even variables, as well as rich data structures using ordinary procedural programming semantics. Thus, engineers would face the question of how granular to make the separation of concerns, for example, whether or not a scenario containing an “`if-then-else`” construct should be broken up into two separate ones. Also, when a scenario relies on memory, its number of states increases, making comprehension, testing, and verification harder, and it perhaps calls for handling the various memory states with different scenarios. In addition, when multiple scenarios are interested in the same information for different purposes, one must decide whether to collect that information in several scenarios or to have it appear just in one and communicate it to the others via events. We believe that maintaining the delicate balance among the different design and methodology goals requires a certain effort, but the accompanying thought process can enrich the design at hand as well as future ones.

Classical method calls versus reacting to events. Having one scenario request an event that, when triggered, activates actions in another, resembles ordinary function or subroutine calls. However, SBP offers additional semantics: 1) the requester (that is, the would-be caller) can sense when the system triggers another event before the requested event and can react to this situation by carrying out something else and/or withdrawing the request and 2) other scenarios can block the triggering of the requested event. For example, a scenario can be dedicated to handling some critical termination condition and can use event blocking to force the stopping of all other scenarios, without those scenarios having to constantly check for such conditions.

Composition semantics. One of the fundamental properties of SBP, and hence also of SBA, is that the specifications leave much of the semantics of scenario composition implicit; humans can properly understand the scenarios and how they operate only if they also understand and accept the manner by which these scenarios are composed. This means that the event selection and scenario composition process, for which there are many options (as mentioned earlier), must be well defined, natural, and easy to remember. We hope that the composition semantics of SBP and SBA eventually becomes as intuitive as the differences between, say, furniture assembly instructions that must be carried out sequentially, a supermarket shopping list, where item order does not matter, or the safety instructions for a work tool, all of which must be complied with all of the time.

SBA performance

Underlying the advent of SBP was the belief that the approach will turn out to be suitable for even the most demanding operational reactive systems. Indeed, various hardware and software mechanisms have been considered for dealing with the overhead of constantly synchronizing and interacting with multiple scenarios. Turning to SBA, we do not make a general claim that scenario-based specifications of classical algorithms will become a more efficient way to execute algorithms. Still, we can envision notable advantages, at least in certain cases. For example, an SBA specification of Quicksort appears to be well suited for controlling, say, an autonomous forklift for the car dealership sorting task since the real-time demands are minimal. Furthermore, even when the SBA performance is inadequate for the final system, we argue that coding, interpreting, and observing the execution of an SBA specification under various conditions can yield important insights into the algorithm and the system embedding it, including performance-related ones. Such are the insights that certain multiplications in calculating the factorial function can be carried out in parallel and that sorting the cars in the dealership lot using Quicksort can be parallelized across the different stretches.

Verification of SBA specifications

As stated earlier, the formalism of SBP specifications enables model checking and efficient compositional verification, which is often hard to accomplish in conventional programs. Alongside ordinary testing, such formal tools can be used to confirm both that the SBA-specified algorithm at hand is correct and carries out the desired functions and, when compared to a classical specification of the algorithm, that both versions produce the same sequences of actions.

FUTURE RESEARCH: EXTENDING SBA IDIOMS

The SBA specifications in the previous Python examples use the basic idioms of requesting, blocking, and waiting for events, with naive event selection. These may be enriched with priorities, look-ahead, or controller synthesis. Future research topics include exploring richer idioms and the corresponding enhanced execution semantics. Here are some possibilities.

1. Since the classical Statecharts formalism[7,9] has already been integrated with SBP in a number of ways,[17,20] the applicability of this integration to SBA should be examined.
2. In Harel et al. 2019,[16] SBP scenarios represent formal mathematical and logical constraints on models of a system and its environment. The execution mechanism applies standard constraint resolution techniques to find a model that complies with the scenarios' requirements and to infer the applicable next system event, which can itself be a rich data structure. This might be applicable to SBA too, if certain mundane steps can be relegated to easily understood constraint resolution steps.
3. Introduce new composition functions like undoing or overriding declarations of blocked events.
4. The SBP and SBA execution infrastructure and verification tools have ready access to meta and self-reflection information, such as which events are being requested, all the events that were triggered since the beginning of the run, which scenario requested the event that was last triggered, or the state of a certain scenario. We plan to explore the merits of making this information available to SBA scenarios.
5. One can encapsulate certain patterns of code in reusable methods and entire scenarios like "block all events of a certain type that have already occurred," "keep requesting a set of events until each of them has occurred at least once," and so on.

Such enhancements of SBA may bring the approach closer to succinct natural language descriptions of algorithms, while retaining the formality and executability.

We have described a scenario-based approach to algorithm specification, where different algorithm steps, special properties, and other important aspects of behavior are individually specified in a stand-alone manner. Step-by-step execution is derived from collective parallel execution of all of these specification artifacts. The main benefits of the SBA approach include clarity of the main aspects of the specification while retaining executability, amenability to compositional verification, and the facilitation of incremental development and refinement.

Clearly, more research is needed to overcome hurdles on the road to broad implementation and acceptance. First, it has been our experience that the thinking process of decomposing a well-specified algorithm into its constituent concerns is at times difficult and may seem excessively formal. For example, in fulfilling a requirement to add 1 to all elements of an array, one can readily write a program loop that does exactly that, whereas the scenario approach seems to require the tedious explicit statement that this addition must be done to all elements of the array, and that it must be done exactly once. This issue seems related to the difficulty in articulating preconditions, invariants, and postconditions in the process of proving program correctness. The SBA specification process might become more natural and intuitive when it becomes part of a set of routine tasks in software development and is supported by appropriate methodologies, languages, and tools, as discussed in the "Methodological Notes" and "Future Research: Extending SBA Idioms" sections. Finally, the development of SBA should be accompanied with iterative empirical studies to confirm its expected merits for software engineering and algorithm development. C

ACKNOWLEDGMENTS

We thank Nimrod Talmon for initial conversations that helped trigger this pursuit, Ori Koren for coding some of the initial examples, and Meir Shani for valuable comments and discussions.

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Reading, MA: Addison-Wesley, 1983.
2. G. Alexandron, M. Armoni, M. Gordon, and D. Harel. "Scenario-based programming: Reducing the cognitive load, fostering abstract thinking," in *Proc. 36th Int. Conf. Softw. Eng. (ICSE)*, 2014, pp. 311–320. doi: 10.1145/2591062.2591167.
3. W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," *J. Formal Methods Syst. Des.*, vol. 19, no. 1, pp. 45–80, 2001.
4. M. Gordon, A. Marron, and O. Meerbaum-Salant. "Spaghetti for the main course? Observations on the naturalness of scenario-based programming," in *Proc. 17th Conf. Innovation Technol. Comput. Sci. Educ. (ITICSE)*, 2012, pp. 198–203.
5. M. Gordon and D. Harel, "Generating executable scenarios from natural language," in *Proc. Int. Conf. Intell. Text Process. Computat. Linguistics*, 2009, pp. 456–467.
6. J. Greenyer, D. Gritzner, G. Katz, and A. Marron. "Scenario-based modeling and synthesis for reactive systems with dynamic system structure in ScenarioTools," in *Proc. MoDELS 2016 Demo and Poster Sessions, Co-located with ACM/IEEE 19th Int. Conf. Model Driven Eng. Languages Syst. (MoDELS)*, 2016, pp. 16–23.
7. D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Programming*, vol. 8, no. 3, pp. 231–274, 1987. doi: 10.1016/0167-6423(87)90035-9.
8. D. Harel, "Can programming be liberated, period?" *IEEE Comput.*, vol. 41, no. 1, pp. 28–37, 2008. doi: 10.1109/MC.2008.10.
9. D. Harel and E. Gery, "Executable object modeling with statecharts," *Computer*, vol. 30, no. 7, pp. 31–42, July 1997. doi: 10.1109/2.596624.
10. D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi, and G. Weiss, "On composing and proving the correctness of reactive behavior," in *Proc. Int. Conf. Embedded Softw.*, 2013, pp. 1–10.
11. D. Harel, H. Kugler, R. Marelly, and A. Pnueli. "Smart play-out of behavioral requirements," in *Proc. 4th Int. Conf. Formal Meth. Comput.-Aided Des. (FMCAD)*, 2002, pp. 378–398.
12. D. Harel, R. Lampert, A. Marron, and G. Weiss, "Model-checking behavioral programs," in *Proc. Int. Conf. Embedded Softw.*, 2011, pp. 279–288. doi: 10.1145/2038642.2038686.
13. D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Berlin: Springer-Verlag, 2003.
14. D. Harel, A. Marron, and G. Weiss. "Programming Coordinated Scenarios in Java," in *Proc. 24th European Conf. Object-Oriented Program. (ECOOP)*, 2010, pp. 250–274.
15. D. Harel and A. Pnueli, *On the Development of Reactive Systems, volume F-13 of NATO ASI Series*. New York: Springer-Verlag, 1985.
16. D. Harel, G. Katz, A. Marron, A. Sadon, and G. Weiss, "Executing scenario-based specification with dynamic generation of rich events," in *Proc. Int. Conf. Model-Driven Eng. Softw. Develop.*, 2019, pp. 246–274.
17. D. Harel, R. Marelly, A. Marron, and S. Szekely, "Integrating inter-object scenarios with intra-object statecharts for developing reactive systems," *IEEE Des. Test.*, early access, 2020. doi: 10.1109/MDAT.2020.3006805.
18. D. Harel and A. Marron, "Toward scenario-based algorithmics," in *Adventures Between Lower Bounds and Higher Altitudes*. Berlin: Springer-Verlag, 2018, pp. 549–567.
19. D. Harel, A. Marron, and G. Weiss, "Behavioral programming," *Comm. ACM*, vol. 55, no. 7, pp. 90–100, 2012.
20. A. Marron, Y. Hacohen, D. Harel, A. Mülder, and A. Terfloth. "Embedding scenario-based modeling in statecharts," in *Proc. MODELS Workshops*, 2018, pp. 443–452.

DAVID HAREL is with the Weizmann Institute of Science, Rehovot, 76100, Israel. Contact him at david.harel@weizmann.ac.il.

ASSAF MARRON is with the Weizmann Institute of Science, Rehovot, 76100, Israel. Contact him at assaf.marron@gmail.com.

RAZ YERUSHALMI is with the Weizmann Institute of Science, Rehovot, 76100, Israel. Contact him at raz.yerushalmi@weizmann.ac.il.