Toward Scenario-Based Algorithmics

David Harel and Assaf Marron

Weizmann Institute of Science, Rehovot, Israel

Abstract. We propose an alternative approach to the classical way of specifying algorithms, inspired by the scenario-based paradigm for reactive systems. Rather than being presented as a carefully ordered sequence of instructions, an algorithm is formalized as an unordered collection of rules or scenarios, specifying actions that must or must not be taken when certain conditions hold or after certain sequences of events. A successful implementation of such a methodology, which can be aligned with a natural language specification, can have many advantages, including naturalness, comprehensibility and incrementality. We believe that our approach can also accelerate the acquisition of problem-solving and analytical skills by children and students. This is because by writing (and reading) computer programs written in this way, people would have access to a broad base of instructions on how to solve problems, stated and organized in a way that can be readily understood and used in practice also by humans. We describe the principles of the approach, *scenario-based algorithmics* (SBA), provide some examples, and compare it to other techniques for algorithm specification and to human algorithmic or computational thinking.

1 Introduction

Ask a student, instructor or researcher of computer science, about a particular algorithm or protocol, be it bubble-sort, quicksort, depth-first search of a tree structure, or two-phase commit, and you are likely to get in response a short list of basic principles — the key points that distinguish this algorithm from an arbitrary or brute force or random approach to the problem. Similarly, text books often precede or summarize the actual details of such algorithms with such a list of principles. The algorithm itself is most often described in some sort of code—either pseudo-code or in some particular programming language, like C, or C++—providing the detailed step-by-step instructions for carrying out the prescribed process. While these instructions can be readily followed (and executed) by a human or a computer, the basic principles of the algorithm are often only implicit, reflected in the names of various methods and subroutines, or in the physical structure of the code. In fact, most often they appear explicitly only in text comments that explain the otherwise-arcane lines of code.

In this paper we propose an approach, termed *scenario-based algorithm specification* or *scenario-based algorithmics* (SBA), for specifying algorithms. Rather than describing the algorithm in a form that explicitly prescribes a carefully ordered sequence of step-by-step instructions, we propose to create a collection of often very brief self-standing rules (or, more precisely, scenarios), which specify actions that must or must not be taken when certain conditions hold and/or when certain sequences of events or

other actions take place. Humans and computers would execute these instructions by repeatedly considering all rules, and progressing when one of these, or several of them, instructed them to do so.

The advantages of this approach include ease of understanding by humans of the actual steps that an algorithm would call for[1], allowance of runtime recognition and tackling of conditions that were not in the specification, and amenability to incremental enhancement of the specification itself for accommodating new or refined requirements. In addition, and most importantly, the approach provides a unique opportunity to introduce human-understandable idioms for algorithmic self-reflection and meta operations, in which the scenarios and the instructions therein become part of the data visible to the algorithm.

As SBA is intended to align with how people often describe their thinking about problems and solutions, causality, planning, and design, we argue that a successful implementation and adoption of the method can have a profound effect on many facets of education in science and in human thinking skills in general.

The remainder of the paper is structured as follows. In Section 2, we present the research background that inspired the paper, and in Section 3 we provide more details on SBA and how it works. Section 4 presents technical and methodological considerations relative to the SBA approach to specification. In Section 5, we discuss the software engineering benefits we anticipate stemming from a successful implementation and adoption of SBA. In Section 6 we elaborate on why we believe that adopting SBA can contribute to education and general skill acquisition. In Section 7 we propose research and development actions that can contribute to broad usage of SBA and to empirical confirmation of the expected benefits, followed by a variety of research areas that can further serve to expand SBA concepts and their impact.

2 Rationale

Regardless of whether an algorithm is specified in code, pseudo code or natural language, working with such specifications is hard. First, points that are critical to understanding and implementing the algorithm may be hidden. These can include specific innovative steps (as distinguished from mundane bookkeeping steps), or complex, sensitive computations, where every typo in the code or error in reading could have far-reaching consequences. Furthermore, the essence of simulation results and run traces that are meant to highlight such points may be cluttered by numerous distracting and less interesting steps. Classical code would also be sensitive to hidden assumptions, such as certain properties of a data structure or of the data stored in it, and it may be difficult to identify all places in the code that should be changed when such an assumption is modified or when new requirements are introduced. In addition, much time may be spent on

[1] In [18] Juraj Hromkovič, quotes Max Planck: *"Science is an innerly compact entity. Its division into different subject areas is conditioned not only by the essence of the matter but, first and foremost, by the limited capability of human beings in the process of getting insight."*. To this profound observation by a giant, may we modestly add that perhaps such a division might also be helpful to humans' effort to understand ever-more-refined concepts and entities, algorithms included.

understanding the reason for including a line of code that is of little importance, merely handling some highly atypical or even obsolete case.

Granted, several common software development practices have been devised to alleviate some of these concerns. They include encapsulating and abstracting certain sets of steps in designated methods, giving meaningful names to program entities, writing comments in the code and in external documentation and presentations, and running selective or abstracted simulations and traces. However, the above issues in understanding and maintaining algorithms still persist.

Over the last three decades much research in computer science has been dedicated to methodologies for dealing with reactive systems [15]. A particular question is how to best describe the full behavior of a system that has to constantly react to stimuli and inputs from its environment. The meaning of the adjective *best* in this context usually reflects some balance of expressive power, intuitiveness to humans, executability, compositionality, and amenability to (formal) analysis. Of particular relevance to the present paper is the introduction of Statecharts [9] and of scenario-based programming (SBP) [4, 10, 14, 17]. This paper concentrates mainly on the latter, though in Section 7 there is a comment about the potential of the orthogonality feature of Statecharts for the present context.

2.1 Scenario-Based Specifications

In this section we provide a brief introduction to the development paradigm of *scenario-based specification* (or *modeling*, or *programming*, all of which we abbreviate as SBP). We focus on the principles and current research in SBP, emphasizing the capabilities and promise of the approach, towards our quest of using it in the context of algorithm specification. For more background and details we refer the reader to [4, 7, 8, 10, 12, 14, 17, 20] and references therein.

SBP was introduced in [4, 14], using the *live sequence charts* (*LSC*) formalism. The approach aims to streamline and simplify the development of executable models of reactive systems, by shifting the focus from describing individual objects and components into describing behaviors of the overall system. The basic building block in this approach is the *scenario*: an artifact that describes a single behavior of the system, possibly involving multiple different components thereof. Scenarios are multi-modal: they can describe desirable behaviors of the system or undesirable ones, and combinations thereof. A set of user-defined scenarios can then be interwoven and executed, yielding cohesive, potentially complex, system behavior.

In SBP, a specification, a model, or a program, is a set of scenarios, and an execution is a sequence of points in time, in which all the scenarios synchronize. At every such behavioral-synchronization point (abbreviated *bSync*) each scenario pauses and declares events that it *requests* and events that it *blocks*. Intuitively, these two sets encode desirable system behaviors (the requested events) and undesirable ones (the blocked events). Scenarios can also declare events that they passively *wait-for* — stating that they wish to be notified if and when these events occur. The scenarios do not communicate their event declarations directly to each other; rather, all event declarations are collected by a central *event selection mechanism* (*ESM*). Then, during execution, at each synchronization

point the ESM selects for execution (*triggers*) an event that is requested by some scenario and is not blocked by any scenario. Every scenario that has a pending request for, or is waiting for, the triggered event is then informed, and can update its internal state, proceeding to its next synchronization point. Fig. 1 (borrowed from [13]) demonstrates a simple behavioral model.

Sensor and actuator scenarios tie events to physical interfaces. Sensor scenarios can access the physical environment or other components of the software system through specialized devices and the software interfaces that the respective devices or components offer. They report the information to the SBP system by translating it into requested behavioral events. Actuator scenarios wait for the SBP system to trigger behavioral events that signify desired changes to the environment or to other components, and call the respective interfaces to bring about these changes.

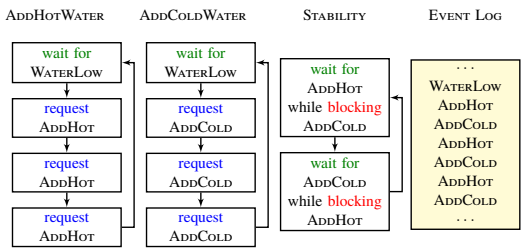


Fig. 1: Incrementally modeling a controller for the water level in a tank. The tank has hot and cold water sources, and either one may be turned on and off in order to control the temperature and quantity of water in the tank. Each scenario is given here as a transition system, where the nodes represent synchronization points. The labels on the edges are omitted here as they can be readily inferred from the context of requested or waited-for events, The scenario AddHotWater repeatedly waits for WaterLow events and requests three times the event AddHot. Scenario AddColdWater performs a similar action with the event AddCold, capturing a separate requirement, which was introduced when adding three (hot) water quantities for every sensor reading proved to be insufficient. When a model with scenarios AddHotWater and AddColdWater is executed, the three AddHot events and three AddCold events may be triggered in any order. When a new requirement is introduced, to the effect that the water temperature should be kept stable, the scenario Stability is added, enforcing the interleaving of AddHot and AddCold events by alternately blocking AddHot and AddCold events. The execution trace of the resulting enriched model is depicted in the event log.

Several facets and generalizations of scenario-based modeling and programming (also termed *behavioral programming*) have been discussed and handled in different ways. Scenarios can be specified and represented graphically, as in the original LSC approach (see, e.g., Fig 2), in natural language (see, e.g., Fig. 3), by two-dimensional blocks that are dragged-and-dropped on the programming canvas (see, e.g., Fig. 4), in specially-designed textual languages (see, e.g., 5) or in standard programming languages like Java or C++ (see Fig. 7).

Scenario-based models can be executed by naïve *play-out*, by smart playout with model-checking based lookahead, or via controller synthesis. The modeling process can be augmented by a variety of automated verification, synthesis and repair tools. However, from this and other research it seems that the basic principles at the core of the approach, shared by all flavors, are *naturalness* and *incrementality* — in the sense that scenario-based modeling is easy to learn and understand, and that it facilitates the incremental development of complex models [2, 6]. These properties stem from the fact

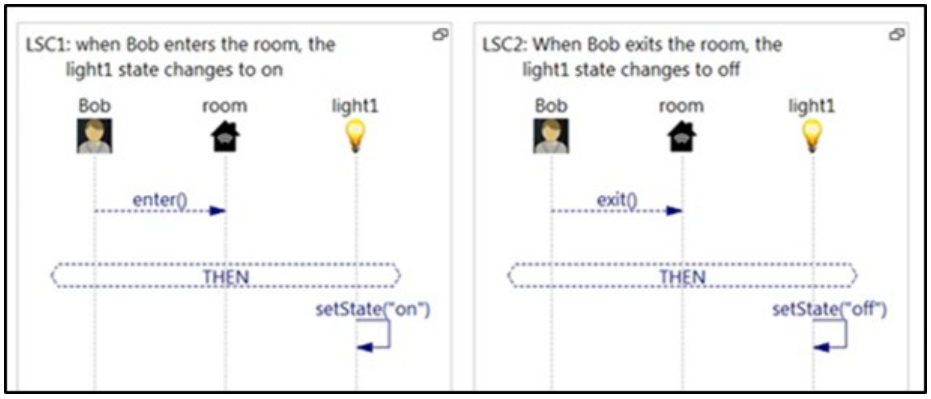


Fig. 2: LSC scenarios specifying automated light behavior in a smart home. The function of LSC1 is described by the comment *"When Bob enters the room light1 state changes to on"*. This scenario waits for the event of of Bob entering the room, and then causes the triggering of the event of the light (in this case - the one named light1) turning on. The semantics of the LSC implies that time flows from top to bottom; participating objects are represented by vertical 'lifelines'; arrows represent events or messages and are annotated as monitored or executed, mandatory or optional, and desired or forbidden.

that modeling and programming are carried out similarly to the way humans explain complex phenomena to each other, detailing the various steps, rules and behaviors one at a time.

```
"When Bob enters the room light1 state changes to on."
```

Fig. 3: A scenario specified completely in natural language. The text shown is after interactive disambiguation for specific object identification and for resolving, if needed, of whether a particular word refers to an object, a method, a property, a property value, etc. This natural-language specification is then translated automatically into an LSC (shown in Fig. 2) to be executed in the LSC development environment PlayGo.

Years ago, as part of the work in our group on SBP for reactive systems, Nimrod Talmon raised the question of whether SBP techniques could be applied to other forms of processing, such as database management or classical operations on data structures. This of course immediately points to the issue at the center of this paper — can the specification and development of general algorithms benefit from the developments in approaches to specification, execution and analysis of reactive systems. In this paper, we argue that although the algorithm itself need not be reactive in the sense of being heavily characterized by interactions with its environment, the answer is a resounding *yes*, partly due to our observation that the interaction of the algorithm's control with its data structures has the main characteristics of classical reactivity, with the data structures 'playing the role' of the environment. In addition, we may draw the following analogies:

1. The computational problem that an algorithm has to solve, and/or properties of its results align well with the concept of *requirements* (as provided by users, customers, system engineers and other stakeholders) in system specifications.

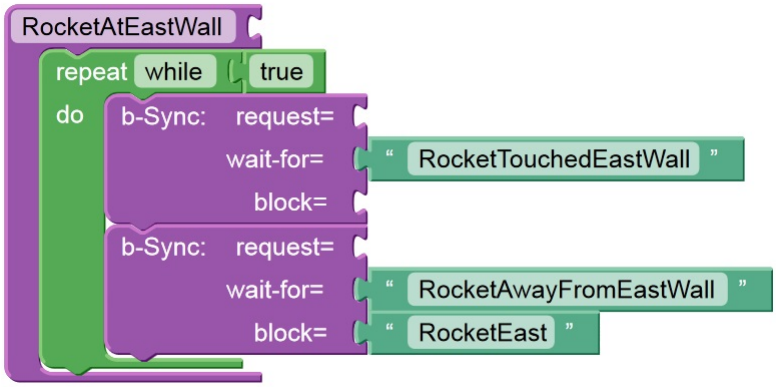


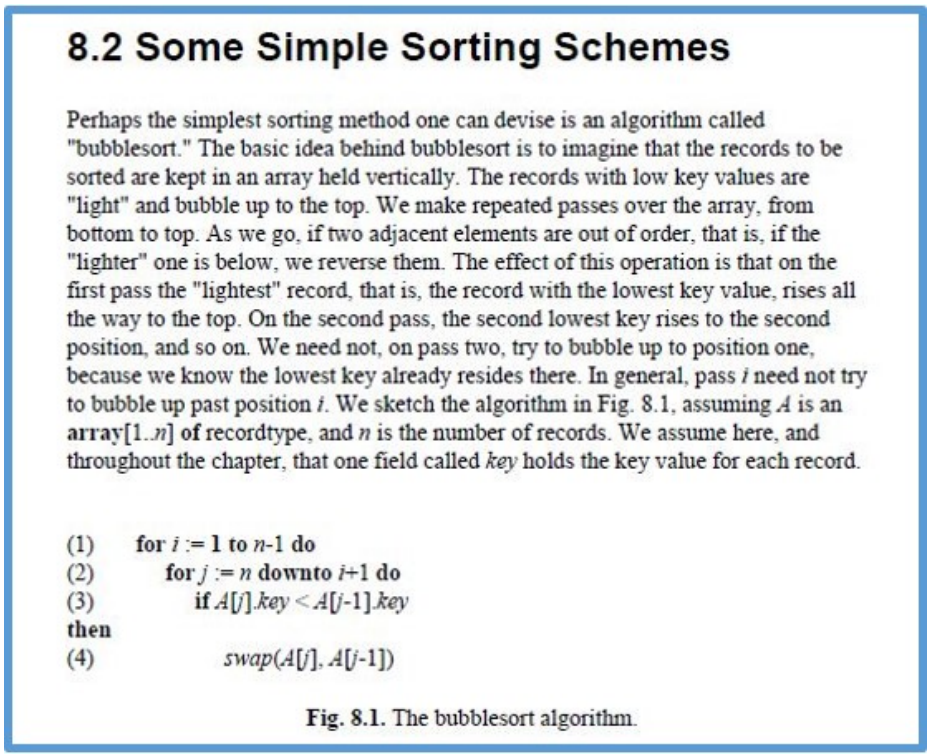
Fig. 4: A scenario in the BP-for-Blockly language. This is the code for the behavior of a wall in a 3D game, in which the user attempts to land a rocket using mouse and keyboard commands. This scenario waits for the rocket to reach the wall, and then blocks further movement in the same direction until the rocket moves back, away from the wall. In the Blockly language (from Google), commands are organized as puzzle pieces that fit together to form program modules. In BP for Blockly each scenario is a separate set of commands, as shown here.

```
assumption scenario DriverObeysStopSignal {
 car->dashboard.showStop()
 car->dashboard.showGo()
} constraints [
 forbidden env -> car.carMovesToNextArea()
 forbidden env -> car.carMovesToNextAreaOnOvertakingLane()
]
```

Fig. 5: A scenario specified in Scenario Modeling Language. The context is that of automated control of cars moving in a two-lane road, and it indicates the fact (in this case, an environment assumption rather than a system behavior), that the human driver obeys the signals of the car's dashboard and that once the dashboard displays a STOP command (which triggers this scenario in the first place), the car will not move forward and will not move to the next lane either).

2. Descriptive texts that commonly accompany algorithms, which cover design principles, operational highlights, and delicate or innovative points, also align well both with the concept of requirement specification and with detailed operational scenarios of reactive systems.
3. The incremental development and refinement of an algorithm, and the reprogramming of established algorithms, which are often narrowly scoped and are done 'one step at a time', also suggest a behavioral description of those constituent steps: "when condition C1 holds, and/or after actions A1, A2 and A3 take place, the algorithm must always carry out (also) action A4".

Thus, if indeed many aspects of traditional algorithm specification can be implemented using the methods that apply to reactive systems, one would gain the benefits of the latter methods such as intuitiveness and ease of understanding by humans, direct executability, and compositional formal analyzability.



8.2 Some Simple Sorting Schemes

Perhaps the simplest sorting method one can devise is an algorithm called "bubblesort." The basic idea behind bubblesort is to imagine that the records to be sorted are kept in an array held vertically. The records with low key values are "light" and bubble up to the top. We make repeated passes over the array, from bottom to top. As we go, if two adjacent elements are out of order, that is, if the "lighter" one is below, we reverse them. The effect of this operation is that on the first pass the "lightest" record, that is, the record with the lowest key value, rises all the way to the top. On the second pass, the second lowest key rises to the second position, and so on. We need not, on pass two, try to bubble up to position one, because we know the lowest key already resides there. In general, pass *i* need not try to bubble up past position *i*. We sketch the algorithm in Fig. 8.1, assuming *A* is an **array**[1..*n*] **of** recordtype, and *n* is the number of records. We assume here, and throughout the chapter, that one field called *key* holds the key value for each record.

```
(1)    for i := 1 to n-1 do
(2)        for j := n downto i+1 do
(3)            if A[j].key < A[j-1].key
then
(4)                swap(A[j], A[j-1])
```

Fig. 8.1. The bubblesort algorithm.

Fig. 6: Textbook presentation of the bubble-sort algorithm: introductory text and code, taken from Aho, Hopcroft and Ullman [1].

2.2 A Note on Predictability and Executability

While SBP claims to be usable for creating executable code for the final operational system, at this point we do not (yet) claim that playing out a scenario-based specification of a classical algorithm would be an efficient way to execute the algorithm. Instead, we claim that it will provide a highly understandable way to *specify* an algorithm and to follow its runs. In particular, playing out the specification with particular inputs, subject to SBP principles, can yield executable or predictable step-by-step instructions equivalent to those provided, say, by pseudo-code. We realize that this paragraph is intended to be something of a teaser, but it will become clearer as we proceed.

3 Scenario-Based Algorithmics

3.1 A motivating example: the bubble-sort algorithm

To explain the concepts of SBA we first consider an example. Fig. 6, taken from [1] contains program code for the bubble-sort algorithm, preceded by some textual introduction.
.

While the algorithm is simple and well known, we can still imagine an expert programmer explaining to a novice some delicate points and emergent concepts, which, in this short program with no additional comments, are only implicit. Such an expert explanation would attempt to be technical and precise, and to be complementary to the bubble metaphor in the introductory text but not to replace it. In fact, with a good technical explanation the metaphor might not be needed. Below are some examples of possible observations that the expert and novice would share. Note that other than the definition of the terms and concepts that should appear before they are used, the observations can be provided in any order:

1. "Generally, when we (*we* here refers to the algorithm of course) see two adjacent records in which the keys are out of order we switch the records' locations.".
2. "Swapping locations of two entries moves the one located at the higher array index to a lower index location, namely, closer to the beginning of the array."
3. "Repeating the above process in any array, moves ('bubbles-up') the record with the smallest key to the first cell of the array, possibly rearranging all other records in the array's remaining 'tail' . "
4. "In this algorithm, the resulting sorted array occupies the same place as the original."
5. "The algorithm starts by bubbling up the lowest-key record in the input array."
6. "Whenever we finish bubbling up the record with the smallest key to the first location of a given array $\mathcal{A}$, the algorithm does similar bubbling-up in the sub-array $\hat{\mathcal{A}}$, which contains all the cells of $\mathcal{A}$ from the second entry to the last one."
7. "The algorithm stops after processing the array consisting of the last two cells in the original array."
8. "For the 'bubble-up' metaphor we consider low array indices as 'up' and high array ones as 'down' "

Our goal is to establish programming idioms and development methodologies that enable the working code of the algorithm to directly reflect to humans all (and only) such relevant insights. Note that the difficulties in understanding that we are trying to address are not in the apparent non-intuitiveness of programming expressions like "`for i := 1 to 10`" or of how the scope of a loop is demarcated. In fact, many would argue that these expressions *are* quite intuitive, and if not, that they can be improved by some form of one-to-one translation. However, we believe that the core issues are that (i) many insights or guidelines about an algorithm's execution involve multiple (not necessarily adjacent) lines of code, (ii) a given line of code may serve multiple guidelines, and (iii) some central insights may be hidden in a small notation artifact like a pair of parentheses, a tagged symbol, or a -1 in an expression. Hence, we are looking for a major shift in how the program is structured in the first place, and in how the developers express their thinking. Once we discover such a shift, the individual programming idioms will be gradually refined until they are sufficiently natural and intuitive for the stakeholders who need or want to read them.

3.2 A scenario-based specification of bubble-sort

We now provide the scenario-based specification of the bubble-sort algorithm from in [1]. A snippet of the behavioral programming code (in C++) appears in Figure 7.

```
#include "IEvent.h"

class SwapPair : public BubbleSortThread
{
public:

    void entryPoint()
    {
        try
        {
        while( true ) {

        bSync( emptyEventList(),
               allSwapEvents(),
               emptyEventList(),
               "SwapPair" );

        // assert lastEvent().class_() == IEvent::SWAP
        unsigned index = lastEvent().param() ;

        // Swap
        int temp = m_array[index];
        m_array[index] = m_array[index + 1];
        m_array[index + 1] = temp;
        }

        }
        catch ( Error &e )
```

Fig. 7: Code snippet for a scenario coded in C++. This is an actuator scenario (behavior thread) that waits for all `SwapPair` events, and then performs the actual swapping. The invocation of the `bSync` method is the synchronization point where the scenario synchronizes with all other scenarios in the specification, and some event that is requested and not blocked is selected and triggered. The three parameters of the `bSync` method are the requested, waited-for and blocked events, in that order.

For brevity and clarity the full list of events and scenarios is described in natural language (English) rather than in C++. Our main focus and interest at this point is the intuitive composition of scenarios, each of which performs a function that is intuitively understood, and which often relates directly to the original requirements. As discussed in Section 7, we do not include here means for enhancing the language used to code the individual scenario. Doing so in ways that result in scenario code that is itself shorter and/or more intuitive is something we leave as a future research objective.

The events in our system are:

- `CHECK_ORDER(n)`: A request to check if the pair of array cells indexed by (i.e., at locations) n and $n+1$ are in the correct order. On the one hand, this is the backbone event of the algorithm and the order of its occurrence drives the order of most other events in each run. On the other hand, this can be viewed as a simple actuation of a sensor, which could be driven periodically (i.e., by the elapsing of some unit of time) or by other events.

- `UNSORTED(n)`: This is a sensor event that reports that the pair of array cells indexed n and $n + 1$ were compared and discovered to be out of order
- `SORTED(n)`: This is a sensor event that reports that the pair of array cells indexed n and $n + 1$ were compared and discovered to be in correct order.
- `EXTERNAL_CHANGE`: A sensor event that reports that something in the array has changed (in our case, probably due to a sort-related action).
- `START`: Triggers the entire process.
- `PREFIX_IS_SORTED(n)`: The first n array cells are sorted and their values are smaller than those in all other cells.
- `ENTIRE_ARRAY_IS_SORTED`: Reports that the process is complete

The scenarios in the specification are:

- `Sensor:` This sensor scenario is the only one that provides reports about the conditions of the environment, namely the array. It waits for any `CHECK_ORDER(n)` event, compares the values of the respective cells and then requests `SORTED(n)` if the value of the lower indexed cell in this pair is smaller than or equal to the value of the other cell, and it requests the `UNSORTED(n)` event otherwise.
- `SortPair`: This scenario is the essence of the sorting logic: it waits for any `UNSORTED(n)` event, and requests a corresponding `SWAP(n)` event. For completeness, in our implementation, this scenario also requests the `CHECK_ORDER(n)` event, to (i) make sure that the swap action completed successfully, and (ii) to facilitate the straightforward triggering of a `SORTED(n)` event from the same sensor scenario (as opposed to an insightful result report from this `SortPair` scenario or from the SwapPair scenario.
- `SwapPair`: This actuator scenario waits for all `SWAP` events, and carries out the swapping of the contents of respective cells.
- `BubblePair(n)`: This scenario controls the order of execution. When a pair of adjacent cells is discovered that needs to be sorted, it initiates the sensing and subsequent possible sorting of the next pair in the array.
- `BubbleStartNextPass`: This scenario controls the iterative processing of bubble-sort — the next bubble. Whenever an unsorted pair is sensed, it waits for the the first pair in the array to be declared as sorted, and then requests that the last pair in the array be checked again.
- `PrintArray`:This is a support scenario that prints/displays the entire array after every request for comparing any two cells.
- `LogSelected`: Another support scenarios that logs to an external file all the events that were selected for triggering.
- `BubbleSortStart`: This scenario starts the entire process by simply requesting the `CHECK_ORDER` event for the last pair in the original array.

We played out the specification using the BPC execution infrastructure for behavioral programming in C++. As expected, the trace showed the events in the same order of comparisons and swaps that we would expect from the classical algorithm, and a visualization of data movements within the array showed the expected bubble-up process.

4 Methodological Notes

4.1 Creating SBA Specifications

Creating a detailed methodology for developing scenario-based algorithm specifications, either as a transformation of a classical specification, or when starting from scratch, is a topic we leave for future research. Below, we present part of our work-in-progress efforts in this area, namely, key steps in an initial methodology for converting classical algorithm specifications into scenario-based ones. The steps can be repeated until the results are satisfactory.

- Identify the main data structures manipulated by the algorithm, and see whether they can be considered as objects with which the control part of the algorithm interacts .
- Identify main conditions or steps in the process and create behavioral events whose triggering indicates that the condition holds or the step took place.
- Describe in natural language key points of the algorithm, as one would do in an introductory or summary paragraph in a textbook (e.g., "always after event E1 we take action A2"). Attempt to translate each of these into a formal scenario.
- Encapsulate specific complex computations that are *not* a part of the essence of the algorithm as sensor and actuator scenarios.
- Examine statically the step-by-step algorithm specification, identify desired elements that are not covered yet, phrase them in self standing natural language statements (e.g. "and I have forgotten to say that after we do A3, if condition C4 holds, then before we do the action A5 as is currently prescribed, we first do action A6"), and add these (eventually by automatic translation into scenarios) as additional, stand-alone formal scenarios.
- Play out the scenario-based specification. Whenever there is a choice of multiple enabled events (say, dictated by different scenarios), add a scenario that makes a deterministic choice for this case, if so desired. If a deterministic algorithm is a must, then such choices may be arbitrary at times, as is sometimes the case in ordering algorithm steps whose order does not matter in a classical specification.
- Compare the SBP playout trace with a run of the step-by-step algorithm. Whenever there is a difference, study it, and adjust the SBP specification.
- Use formal verification tools or systematic test tools to confirm that the two implementations yield the same results for large ranges of data.

4.2 'Events vs. Method Calls' or 'Dedicated Scenarios vs. Method Parameters'

Scenario-based specifications of algorithms may at first be reminiscent of ordinary programming, with methods replaced by scenarios, and method calls replaced by broadcast events. Such an event carries information about its destination scenario or object (which corresponds to a method name) as well as its execution parameters (corresponding to the method-call parameters or return values). However, the SBA approach provides a powerful alternative view:

In the extreme case there would be no parametric information passing. Instead, for each object, field, record, or relevant set(s) thereof, and for each behavior that this entity

has to exhibit, the developer would create a dedicated scenario. This scenario stands as a sentinel, deliberately ignoring almost everything that is happening in the system, and if and only if certain conditions that apply to it are satisfied, or events that it is interested in occur, the scenario requests one or very few particular actions. Once these actions take place (and, of course, only when they are not blocked by other SBP scenarios) it reports this fact, with little or no knowledge or 'interest' in what will be done with the results, or which component will use them.

This view provides new dimensions of encapsulation, parallelization opportunities for more efficient execution, and opportunities for functional refinement, while allowing individual scenarios to be "taught" (explicitly by humans or via automated learning) how to handle yet new conditions as may be encountered within their purview.

For scalability purposes, e.g., when the number of behaving entities is large, and where some of which are dynamic in nature (i.e., are created and discarded at run time), common optimization techniques — like using a pool of worker threads — can be applied towards efficient utilization of computer resources, without diminishing the value of utilizing automated 'narrow-minded' processes working 'with their blinders on' [2].

This approach to dynamic scenario instantiation also provides an approach for recursion that is presently built into standard programming languages. This 'magic' of using the same code at different states, with the relevant context kept in the computer's or the human's mental stack, will be replaced in SBA with the explicit creation of emergent objects like ever-shorter tails of an array, ever-smaller subtrees, etc.

4.3 A Note on Feasibility

An important question regarding our approach is whether many or most interesting algorithms can be specified in SBA. In two extreme senses, the answer is trivially positive: First, one could simply encapsulate the entire algorithm in a single actuator scenario, which simply waits for one `Start` event with appropriate parameters and carries out the desired computations internally. Second, the structure of the algorithm can be left unchanged, but every command in the classical specification would be converted into a scenario, and these scenarios communicate via helper events. E.g. line 17 in some sequential code segment that is contained in a loop can be specified as the following scenario "Repeatedly, wait for the event `AnotherIterationOfThisLoopStarted`; wait for the event Line16Completed; then, forbid the event `line18Started` until further notice; request (and wait for) the event `Line17Started`; carry out as an actuator what line 17 does in the original algorithm; request (and wait for) the event `Line17Completed`; and, finally, release the blocking of `line18Started`.".

Such extreme cases are obviously not what we are after. Instead, the goal is to identify methods and principles for scenario-based creation of algorithms that will have non-trivial added value, enhancing understanding and simplifying the development of the algorithms.

[2] Those that prevent horses from being distracted or alarmed.

5 Advantages of Scenario-Based Algorithmics

As stated earlier, one of our goals is to exploit SBA to enrich the intuitiveness, clarity, alignment with the requirements, incrementality, executability, and amenability to formal verification of general algorithm specification and development. Though systematic empirical studies proving this are yet to be carried out, we believe that SBA can indeed contribute significantly to these.

An additional non-obvious advantage is succinctness. This claim may be surprising, since even in the relatively simple bubble-sort example, the natural-language scenarios were quite verbose as compared to the original pseudo code, and the C++ code was even more so. However, along the lines of formal succinctness proofs of SBP [12], we argue that *eliminating* from the specification many classical structural elements, such as intricate control flow, and the exact locations of method invocations, can serve to replace one large module, i.e., the entire algorithm, with multiple, smaller, more robust modules, i.e., the scenarios. Our measure of size here is not the number of characters or lines (which some languages can compress very efficiently), but the number of different states and conditions and flows that a module represents. Thus, we consider the difficulty of understanding the SBA specification to be proportional to the sum of the sizes of the scenarios, while understanding the original specification is proportional to its own size, which in some cases can be closer to the product, rather than the sum, of the sizes of the constituent scenarios.

Reflection in software is the ability of a programmed entity to look at itself and at other such entities and dynamically adjust its own behavior accordingly; e.g., checking at run time if the definition of a target object class includes a particular method call, and when this is the case calling this method on objects of this class. SBA enables extending such reflection significantly . Consider the following hypothetical excerpt from an email correspondence between a development manager (M) and a developer (D) on his or her team:

- (M): *"Please change all program modules where we do action A, to first check for condition C."*
- (D): *"But in program module P this change would be incorrect!"*
- (M): *"You are right, indeed, do not change program module P, and also do not change module P2, as it suffers from the same issue."*

Collectively these requests and decisions describe precisely the revised behavior of the system. Indeed, when the developer completes the task, the manager would know what the system does. However, we should note that this specification was not the same as the behavioral scenario *"Action A should occur only when condition C holds"*, but a structural one. As a proof, consider the fact that we do not know what is it about modules P and P2 that render the requirement inapplicable in those contexts. Such reflective capabilities are available in a variety of programming language, as well as in Statecharts, where the components can refer at development time and at run time to (states of) other components. SBA allows us to readily and incrementally enhance a specification with scenarios of this type.

SBA can also offer new insights into understanding data and behaviors. Consider for example the *Sieve of Eratosthenes* method for finding the prime numbers. The classical

algorithm calls for a data structure in which cells corresponding to multiples of discovered primes are marked as non-primes, and the cells remaining at the end of the process are output as primes.

In one SBA design, we modeled the method as an iterative-cumulative process. Once a prime p is discovered (starting with $p = 2$) a scenario is added that iteratively requests *events* declaring all multiples of p as non-primes. In parallel, another scenario iteratively requests events that declare an integer to be a prime but abandons that request upon the occurrence of the event declaring the said integer as a non-prime. We then used event priorities in the Behavioral-Programming-in-Java platform (BPJ) to make the "p is not a prime" event take precedence over "p is a prime", which yields the final desired results.

In another implementation (relying also on BPJ's support for infinite sets in the declaration of blocked events and waited-for events), for each discovered prime p we replaced the requesting of events that declare multiples of p as non-primes with the creation of a scenario that blocks the event "n is a prime" for all values of n that are multiples of p. In yet another possible implementation, instead of dynamically creating scenarios associated with discovered primes, one could create a scenario for each integer in the range to block declaring its multiples as primes.

The attractiveness of these SBA implementations is not just in that they replace the need for the data structure in the Sieve of Eratosthenes algorithm (note that they clearly do not save in computer memory). While loyal to producing the desired functional results, the SBA versions, using priority and event-blocking semantics, can rely on the infrastructure for some bookkeeping that in the classical versions are specified more explicitly and prominently (as storing of interim results and evaluating conditions).

6 SBA and human thinking

"Thinking like a computer scientist" [21] (or "thinking like a programmer" as it is often referred to), as well as understanding the basic principles behind algorithms and computer science at large [16], are considered to be desired mental capabilities, which are valuable in many real life contexts. Much important work has been done on endowing the population at large with such skills, even from very young ages (see, e.g., [3, 5, 19] and references therein). To these efforts we wish to add the following observation: once we are successful in explaining to humans, in human terms and in a systematic and repeatable way, what systems and computers and algorithms do, then perhaps we will be able to distill the underlying principles of such useful human terms, and start programming systems and computers and algorithms using those very terms themselves.

Once this is achieved, it may open up new opportunities for both humans and machines. E.g., a person who is often personally baffled by complex decisions and planning challenges may acquire necessary skills not just from occasional tips from friends and books, but from the systematic understanding of the principles of how powerful computer systems (like road navigation or warehouse stock management systems) accomplish the same. With SBA, these principles will stand out better. Conversely, once systems are structured in this manner (built with SBP and SBA), humans will be able to experiment with those systems, and even enhance them with an ever-growing number of desired features and capabilities. The phrasing of an idea, like " can the system do action

A1 when condition C1 holds?" will be directly translated into a component (a scenario) that can be tested independently and then be verified together with the system, in order to check whether it introduces unexpected problems or conflicts.

7 Future Research

The initial proposal described in this paper must clearly be followed by further development and research. First, existing and new integrated development environments (IDEs) should be built to support the broad application of scenario-based development in general, and SBA in particular. This should be geared to audiences in science, engineering and education, tending to the users' different needs, and with a special focus on SBA intended strengths, such as understandability and maintainability of algorithms. The expected benefits in areas such as productivity, quality and learning should then be empirically confirmed, by measuring specific results in synthetic and realistic projects carried out by many participants, and comparing the SBA-based results to the ones obtained using known techniques.

SBA also presents interesting research questions and technical challenges. Key among these is the efficient implementation of direct execution of a scenario-based specification. Work in this direction is underway along various tracks, including synthesis of an efficient monolithic automaton from a scenario-based specification, efficient approaches for distribution and parallelization, while relaxing some of the behavioral synchronization requirements, hardware circuits designed especially for assisting in the SBP event selection process, and more. Depending on the success of such solutions, especially distribution and hardware assists, we will be interested in exploring the costs and benefits of dedicating large numbers of parallel-running behavior threads (e.g., in the case of bubble-sort, one per every pair of array cells, or one per array 'tail', etc.).

For evaluating the correctness of an SBA-specified algorithm that is also available in a classic form, one should create a log for the original specification and compare the logs of the two implementations. Proving full equivalence is a separate task.

From a software engineering standpoint, it would be interesting to formalize and then discuss the syntactic and semantic differences between scenarios and method calls. In such a discussion, one would compare, e.g., the fact that each scenario can be cognizant of the global (system-wide) conditions and events under which it should be called, with the fact that in classical method calls it is the collective responsibility of all the callers to invoke the method under all relevant conditions and following all relevant event sequences. Of particular interest are the auxiliary events that are added to indicate a particular interim condition or occurrence (e.g., that a certain pair of array cells is or is not sorted). We believe that one would be able to show that while such events may seem to be merely 'helper' events, which would not be necessary in a sufficiently powerful programming language, they indeed provide a solid foundation for the specification. This would be similar to storing interim computation results (e.g., of expressions in parentheses or numerators and denominators of a division in separate well-named variables), not just for shortening and clarifying each computation, but for highlighting the existence and subsequent use of this emergent entity.

Another interesting topic for future research is the incorporation of scenario-based techniques into the Statecharts formalism. In a research project underway, we are endowing the orthogonal state components accommodated by Statecharts with the ability to declare requested, blocked and waited-for events, and are enhancing the event triggering and composition semantics of Statecharts to accommodate the SBP event selection semantics. This can allow the SBA approach to be carried out in yet another, more familiar, visual formalism.

Finally, the programming idioms at the foundation of SBP languages have in mind reactive systems operating in a real-world environment. It would be interesting to create a standard methodological mapping (or at least a consistent approach thereto) from classical entities and concepts that appear in algorithms, like complex data structures or recursion, to the scenarios and events underlying SBP. Furthermore, an SBP language may be enhanced with basic idioms geared specifically for scenario-based algorithmics. We are particularly interested in finding concise, yet explicit, idioms for cryptic or implicit notations; e.g., that in a program with an array of size n, whose indexing begins at 0, the expression $n - 1$ is the index of the last item as well as the length of the 'tail' sub-array after dropping the first array cell. These can, of course, be conveyed in meaningful names for temporary variables, but perhaps some orderly methodology could be discovered in order to balance clarity and clutter.

8 Conclusion

We have described the SBA approach to algorithm specification, where each of the algorithm's steps and special properties is specified in a stand-alone manner, in any order, and for which the step-by-step execution is derived from the collective parallel execution of all these specification artifacts. Data structures play the role of the environment of a reactive system for such steps and properties. While additional experimental work with students and engineers is still needed, SBA promises several benefits: It can facilitate better human understanding of algorithms, which can ultimately contribute to the quality of computerized systems, and it can enable, and even drive, the enhancement and improvement of algorithms, since one can more naturally focus on special features while handling bookkeeping steps and special extreme cases separately.

The approach may also bring the development of reactive systems and classical computation closer — enabling a broader or deeper application of techniques developed for reactive systems in the classical areas of algorithm design and programming. Formal verification can be one such example.

Finally, the SBA approach might also contribute to the teaching and learning processes around computer science and software engineering, and perhaps even enhance the penetration of a variety of human skills associated with computational, or algorithmic, thinking [16, 18, 21].

Acknowledgements

We thank Nimrod Talmon for initial conversations that partly triggered this pursuit. Thanks to Ori Koren for the prototype of the scenario-based implementation of bubble-

sort using behavioral programming in C++. We are also grateful to the anonymous reviewers and editors for their valuable comments. This work was supported in part by grants from the German Israeli Foundation (GIF), the Israeli Science Foundation (ISF), and the Minerva Foundation.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data structures and algorithms (1983)*. Addison-Wesley, Reading, MA.
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. on Software Engineering (ICSE)*, pages 311–320, 2014.
3. S. Bocconi, A. Chioccariello, G. Dettori, A. Ferrari, and K. Engelhardt. Developing computational thinking in compulsory education–implications for policy and practice; eur 28295 en. *URL: http://publications. jrc. ec. europa. eu/repository/bitstream/JRC104188/jrc104188_computhinkreport. pdf*, 2016.
4. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 2001.
5. P. J. Denning. Remaining trouble spots with computational thinking. *Communications of the ACM*, 60(6):33–39, 2017.
6. 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. on Innovation and Technology in Computer Science Education (ITICSE)*, pages 198–203, 2012.
7. M. Gordon and D. Harel. Generating executable scenarios from natural language. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 456–467. Springer, 2009.
8. J. Greenyer, D. Gritzner, G. Katz, and A. Marron. Scenario-Based Modeling and Synthesis for Reactive Systems with Dynamic System Structure in ScenarioTools. In *Proceedings of the MoDELS 2016 Demo and Poster Sessions, co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. CEUR, 2016.
9. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
10. D. Harel. Can Programming Be Liberated, Period? *IEEE Computer*, 41(1):28–37, 2008.
11. D. Harel and E. Gery. Executable object modeling with statecharts. *Computer*, pages 31–42, July 1997. Also in *Proc. 18th Int. Conf. Soft. Eng.*, Berlin, IEEE Press, March, 1996, pp. 246–257.
12. D. Harel, G. Katz, R. Lampert, A. Marron, and G. Weiss. On the Succinctness of Idioms for Concurrent Programming. In *CONCUR*, pages 85–99, 2015.
13. D. Harel, G. Katz, R. Marelly, and A. Marron. An Initial Wise Development Environment for Behavioral Models. In *Proc. 4th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 600–612, 2016.
14. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.
15. D. Harel and A. Pnueli. *On the Development of Reactive Systems*, volume F-13 of *NATO ASI Series*. Springer-Verlag, New York, 1985.
16. D. Harel and Y. A. Feldman. *Algorithmics: the spirit of computing*. Pearson Education, 2004.
17. D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Comm. of the ACM*, 55(7), 2012.

18. J. Hromkovič. *Algorithmic adventures: From knowledge to magic*. Springer Science & Business Media, 2009.
19. Y. B. Kafai. From computational thinking to computational participation in k–12 education. *Communications of the ACM*, 59(8):26–27, 2016.
20. A. Marron, G. Weiss, and G. Wiener. A Decentralized Approach for Programming Interactive Applications with JavaScript and Blockly. In *SPLASH Workshop on Programming Systems, Languages, and Applications based on Agents, Actors, and Decentralized Control (AGERE!)*, 2012.
21. J. M. Wing. Computational Thinking. *Comm. of the ACM*, 49(3):33–35, 2006.