Semantic Navigation Strategies for Scenario-Based Programming

Michal Gordon and David Harel
Dept. of Computer Science and Applied Mathematics
The Weizmann Institute of Science
Rehovot, Israel
Email: {*michal.gordon, dharel*}*@weizmann.ac.il*

***Abstract*—The scenario-based approach to specification and programming uses powerful extensions of sequence diagrams, such as *LSCs* (live sequence charts), to model system behavior. Previous work in this area presents interesting challenges related to the scalability of the approach and to better tool support for analysis, execution, and comprehension. Here we suggest new semantic-rich ways of viewing sequence diagrams and LSCs for better comprehension of both a single large chart and a full multi-chart specification, in a variety of software engineering tasks. Our method uses weighted messages to create a semantic order that enables semantic zooming and scrolling of different parts of a chart, providing visual hints about context.**

***Keywords*-Live Sequence Charts; Sequence Diagrams; Semantic Zoom; Program Navigation; Program Comprehension;**

I. Introduction

LSCs (*live sequence charts*), a new scenario-based programming language, and sequence diagrams, a popular scenario-based specification language, are both visual formalisms that present major challenges to usability. In the case of LSCs, these challenges are made even more acute due to the multi-modal features of the language and the dependencies between different charts. These features are also found in some other programming paradigms. The challenges include scalability of the presentation, and navigation strategies for comprehension and debug. In this paper, we suggest ways to address some of these challenges by applying a new method of *semantic navigation* to LSCs. Specifically, we define new zooming and scrolling methods for LSCs based on custom weights for diagram elements. Our methods are described for LSCs but can also apply to sequence diagrams, and they can be extended to textual code.

Our custom weights are generated automatically, depending on the task at hand, or manually by the user, and they end up creating a semantic order on the elements of an LSC. This order is different from the spatial order of the elements in the chart, and we define it to assign higher weights to elements that are semantically "more relevant" to the current task. For example, in comprehension, elements that appear only once in the chart may provide more information than ones that repeat.

The order allows semantic zooming and panning on a diagram, while some additional definitions we provide maintain the context, abstracting unnecessary information to assist comprehension in the specific task. More generally, our work can be viewed as the application of the concept of *semantic zoom*, adapted from information visualization and user interface design, to the programming domain in general. We demonstrate the approach and show the new visual notations for the scenario-based visual formalism of LSCs implemented in the *play-engine* tool [1].

II. Background

Live sequence charts, LSCs, are used for specifying multi-modal scenario-based behavior [2], [1]. An LSC describes inter-object behavior, i.e., behavior between objects, capturing part of the interaction between the system's objects, or between the system and its environment. LSCs are an enrichment of message sequence charts (MSCs) [3] and the UML sequence diagrams [4], in which objects are represented by vertical lifelines and messages between the objects are visualized as horizontal arrows, with time advancing from top to bottom, as in Figure 1. Additions include subcharts, which can contain alternative statements and messages, allowing one to specify different behavior under different conditions, as well as loops and synchronization points along the lifelines.

LSCs add modalities of behavior to MSCs and sequence diagrams by, e.g., distinguishing possible (cold) from necessary (hot) behavior (the latter is where the term "live" comes from), and can also express forbidden behavior, i.e., scenarios that are not allowed to occur. A prechart fragment, represented as a blue dashed hexagon as in Figure 1, at the beginning includes cold messages, whose occurrence triggers the main part, depicted as a solid black rectangle as in Figure 1, of the LSC. To execute LSCs, the *play-out* mechanism or its more powerful variants [5], [6] monitors at all times what must be done, what may be done and what cannot be done, and proceeds accordingly. Although the execution does not result in optimal code, nor is the executed artifact deterministic (since LSC can yield an under-specification) it, nevertheless, leads to a complete consistent execution of the LSC specification, if one exists. The details of play-out are outside the scope of this paper, and are described in detail in [5], [1].

LSC is an example of a visual formalism that is sufficiently novel so as to render scalability of presentation and

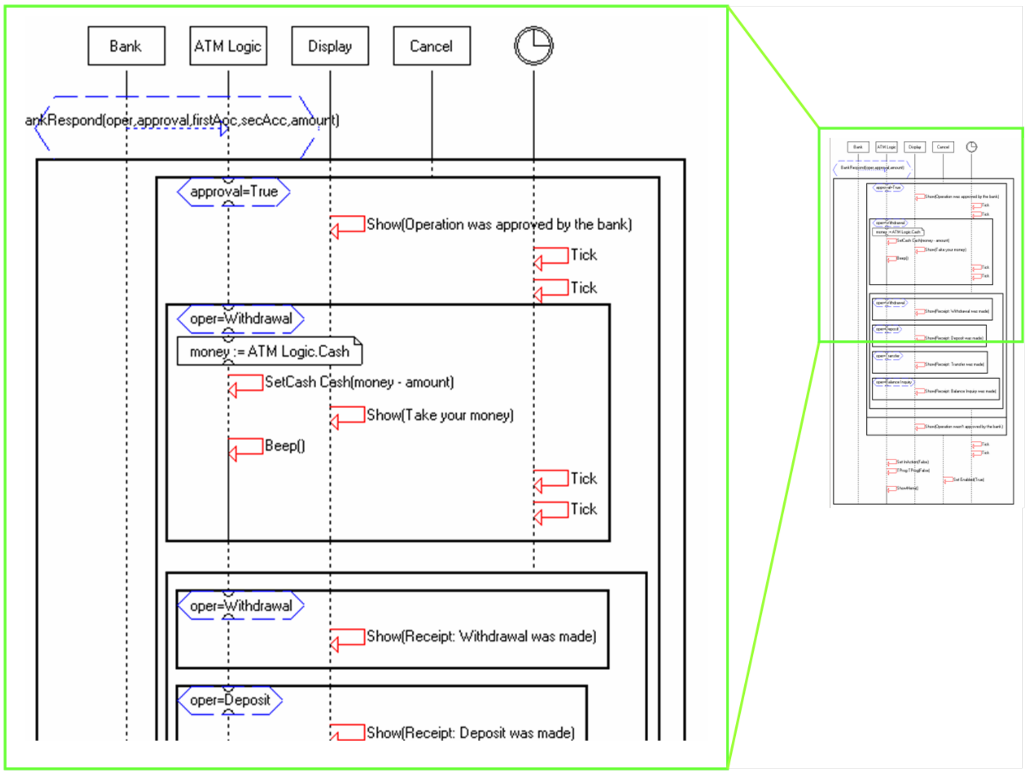
Bank
ATM Logic
Display
Cancel
ankRespond(oper,approval,firstAcc,secAcc,amount)
approval=True
Show(Operation was approved by the bank)
Tick
Tick
oper=Withdrawal
money := ATM Logic.Cash
SetCash Cash(money - amount)
Show(Take your money)
Beep()
Tick
Tick
oper=Withdrawal
Show(Receipt: Withdrawal was made)
oper=Deposit
Show(Receipt: Deposit was made)

Figure 1. The top half of the full LSC, with readable messages, part of a larger LSC

tool support for analysis a challenge that has not yet been fully addressed. Several pieces if work have addressed the issue of viewing large sequence diagrams (SDs) [7], [8], but they do not address issues of various tasks of comprehension, nor do they apply to the considerably more semantically-complex case of a *set* of diagrams, as when dealing with a full LSC specification. They also lack the ability to deal with the multi-modality of LSCs and the semantical issues these raise. These approaches also seem to require much intervention on the user's side, and little to assist him/her in filtering the information wisely, as would be expected from a truly semantic method of zooming.

Semantic zoom has been used in many fields of information processing. The idea is to balance details and context when displaying information. However, when working with LSCs, there is no clear definition of what the important details are. Here we propose a method for using custom weights, which make it possible to navigate between level of detail in LSCs. The weights are generated to correspond to different tasks, and visualization methods are suggested to allow semantic zoom and navigation.

The current paper centers around LSCs, yet many of the ideas can be used to also contribute to MSCs and UML sequence diagrams.

III. Defining semantic order

When setting up a system for carrying out semantic zoom, some form of the relevant details has to be carefully defined, to allow for navigation between different levels. Many programming languages, and LSCs among them, lack explicit definitions of such information. In our case, we found that the additional information would best come in the form of custom weights on the elements of the chart, thus creating a semantic order. When the language supports hierarchy, this information, which induces an order on element-sets, can be integrated with the custom weights. However for navigation to be continuous we require an element-wise semantic order that includes all the elements.

Most programs, including LSCs, can set as the default semantic order the textual/spatial order of the code. However, more meaningful orders, which do not necessarily coincide with the natural order in the program or diagram, can be generated by the user or computed automatically, depending on the task at hand. We shall discuss some suggestions for semantic orders in section VI, and we refer to an LSC with a semantic order as a *weighted LSC*.

More formally, a custom weight $w(m_i)$ is defined for each message m_i of the LSC. Since some elements of an LSC contain others, as in, e.g., subcharts (fragments in UML terminology), the message weight induces a weight for each subchart as the sum of the weights of the messages in the subchart, $w(s_i) = \sum\{w(m_i)|m_i \in s_i\}$, and a weight on the lifelines as the sum of the weights of messages connected to the lifeline, $w(l_i) = \sum\{w(m_i)|m_i \in cover(l_i)\}$. The custom weights define a semantic order on the elements of the LSC, and ties (equal weights) can be resolved by reverting to the spatial order in the chart.

Consider the LSC in Figure 1. First, the vertical order induces higher weights for messages higher in the vertical dimension, so that message *SetCash* has higher weight than message *Beep*. Second, the horizontal order induces higher weights for messages that are left of other messages (higher in the horizontal dimension) and on the same vertical line, hence message *SetCash*, which is to the left of message *Show(take your money)*, has higher weight. (In the figure they are not on the same vertical line due to tool limitations but they could be on the same vertical line, since there is no order between them semantics-wise.) However, message *Show(take your money)* is also higher than *Beep* and therefore we get the order $w(SetCash) > w(Show) > w(Beep)$. Section VI describes how the weight of an element can be chosen to reflect the information it provides relevent to the task at hand. Thus, the semantic order will allow a viewer to focus on more relevant elements while ignoring others.

IV. Visual Notations

A. The placeholder

Given a particular semantic order to be used (which does not necessarily depend on the "geography" of the diagram), we have to find ways to support the kinds of navigation we want, such as zoom in, zoom out, and pan/scroll at a specific zoom level. We do this by hiding some of the elements, and in LSCs this will apply directly to a message, a subchart, or a lifeline. Hiding an element can be performed anywhere, and since it means removal of information, we add a *placeholder* instead to provide context information and to hint at the fact that data has been hidden. A specific "look" for the placeholder must be devised, which should indicate the location of the missing elements and include some coding that means for capturing the sum of weights of the elements it replaces. In LSCs, the placeholder for a removed message is depicted as horizontal gray lines at the appropriate location. A vertical gray line holds the place for a removed lifeline. The weights are coded by the level of the grayscale color of the placeholder, thus hinting at the amount of information being hidden. When an element is hidden and its weight is added to a specific placeholder, we refer to the element as being consumed by the placeholder. Adjacent placeholders are placeholders that are immediately next to each other, with no interfering elements between them. If, in the process of adding a placeholder at a particular location it turns out that there is already one in an adjacent location, the two are merged into one, and the weights are summed

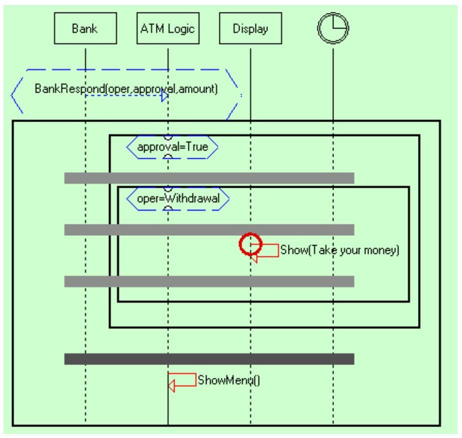

Figure 2. Placeholders not merged due to subchart separation

(and the darkened color will reflect the added weight). Structural containers impose some limitation on the placeholder merge algorithm. Subcharts in LSCs, for example, cannot be merged in a naïve way, since doing so would result in removing structural information. A placeholder inside a subchart can replace the subchart completely after it has assumed the weight of all elements in the subchart. Only then can the placeholder merge with placeholders outside the boundaries of the subchart, thus leaving the structural information intact for as long as necessary. This can be seen in the example in Figures 2 and 3 that show the LSC before and after the hiding operation of a single message that causes a merge operation cascade for four placeholders. The algorithm for hiding elements and merging placeholders is given in the next section.

B. Last change marker

Our navigation allows continuous zooming and scrolling using the scroll wheel (the former also requires holding down the ctrl button). This means that consecutive zoom / scroll steps may be taken but since the semantic order does not depend on 'geography', there is no guarantee that the elements being shown or hidden are in adjacent spatial locations. We therefore mark the last change at each step, using a special *last change marker* and focus on it at each step. In LSCs, we use a red circle; see Figure 2. When the last step added an elements, the last element added is marked as in Figure 2. When last step hid an element, the placeholder that consumed the element, is marked; see Figure 3. When scrolling we mark the added element, or the placeholder that changed due to the hidden element, depending on the direction of the scrolling.

We have also found that it is helpful to provide a clear indication to the viewer when the code/diagram is shown at some zoom level that is not the regular full-detail one. In LSCs we use a light green background for all zoomed diagrams, rather than the normal white.

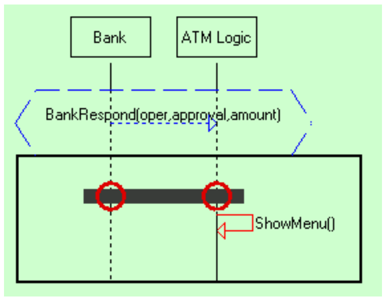

Figure 3. A merged placeholder

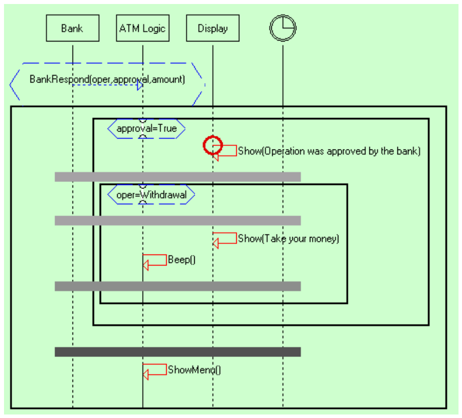

Figure 4. The full LSC zoomed

C. *Semantic navigation*

1) Zooming: During zoom-out, less information is displayed at each step and the level of detail decreases. Therefore, the element with the least weight is hidden and is replaced by a placeholder. This, in effect, leaves the most informative elements on the diagram, and they will be the last to be hidden.

During zoom-in, more information is displayed at each step and the level of detail increases. Therefore, at each step the hidden element with the largest weight is added and its placeholder is updated accordingly.

This principle can be applied to vertical elements or horizontal ones (i.e., lifelines) when appropriate. In our implementation for LSCs, the zoom is applied to the vertical elements.

In applications that have a limited area available for drawing the diagram, the principle can be used to find the right level of zoom for the area. The area can induce the maximal number of vertical or horizontal elements that can be displayed. We start by hiding all elements and then adding the most informative elements one after another, counting also the number of placeholders that are required at each step to find the optimal zoom level.

2) Scrolling: When viewing a chart at a specific zoom level, there are cases (e.g., when the chart fills the entire available "canvas") that call for allowing the user to scroll and see adjacent details presented according to the semantic order.

In such cases, we allow the user to scroll with a fixed-sized window over the ordered set of elements. For each scroll operation, the elements in the window are shown and all others are hidden and are replaced by placeholders.

3) Filtering: We can easily allow the application of a filter of a specific weight threshold. All elements with custom weight below the threshold will be hidden (and replaced by appropriate placeholders). This will allow the user to focus on the more relevant information.

The filtering operation is essentially setting an exact zoom level, but is carried out without having to go through the continuous changing of the level.

V. The Navigation Algorithm

We now describe the navigation algorithm for hiding weights and merging placeholders. It can be applied independent of the calculations for the custom weights, and has been implemented in the *play-engine* tool. The next section discusses different ways to calculate the weights.

Here we view an LSC as a structure consisting of a tree T of vertical elements taken from the following two sets: a set of *subcharts* S and a set of *messages* M listed by their 'geographical' order. The LSC also contains a list of *lifelines* L. For simplicity, no two vertical elements can be at the same vertical location and therefore each element can be replaced by a single placeholder. Subcharts can have *child* elements of type subcharts and messages, while messages cannot have child elements (thus, $child(e) = null$ for $e \in M$). Each message m or subchart s has a *parent* element in the tree. Each message m and subchart s has a non empty *cover* set of lifelines from L that it is connected to. A message is connected to at most two lifelines.

Since the basic element of an LSC is the message, the navigation is in the vertical dimension and it affects the horizontal elements; e.g. if all the elements connected to a lifeline have been removed, the lifeline will also be removed.

In the zoomed LSC, placeholders from a set P can be added as leaves in the tree. A placeholder p has a list of *contain* elements from $S \cup M$ that it consumed. Placeholders are added only in the vertical dimension but could also have been added for removed lifelines.

The input to the algorithm is an LSC $\{T, L\}$ with weights calculated as in section III, and the output is a zoomed LSC $\{T', L'\}$. Let E be a list of the messages in T sorted in increasing order by weight $(m_1, ..., m_n)$, ; thus, $w(m_i) \le w(m_{i+1})$.

For navigation, a copy of the original LSC is created, and at each step the tree structure is updated, and then drawn. The order of child elements in the tree determines their vertical order. If two elements i and j have the same parent

```
procedure: MergePlaceholders(element)
foreach c ∈ child(element) do
  MergePlaceholders(c)
for i ←0 to length(child(element)) - 2 do
  $e_0$ ←child(element,i) //gets i'th child of element
  if $e_0$ is a placeholder
    $e_1$ ←child(element,i+1)
    if $e_1$ is a placeholder  //merge placeholders
      weight($e_0$) ←weight($e_0$) + weight($e_1$)
      contain($e_0$) ←contain($e_0$) ∪ contain($e_1$)
      remove $e_1$ from element //decreases number of elements
      $i \leftarrow i - 1$ //loop increases $i$, decrease since element removed
  elseif $e_0$ is a subchart and length(child($e_0$)) = 1
      and child($e_0$,0) is a placeholder
    $e_0$ ←child($e_0$) //replace subchart $e_0$ by its child placeholder
    $i \leftarrow i - 1$
```

Table I
THE MERGE PLACEHOLDER ALGORITHM

and i is listed before j, then i will be drawn above j in the vertical dimension.

For zoom-out, we create a copy $\{T', L'\}$ of the original LSC and at each zoom-out step $i = 1, ..., n$, the procedure *RemoveMessage*(m_i) is called. In *RemoveMessage*(m_i), the message m_i is removed from T' and is replaced by a placeholder p with $parent(p) \leftarrow parent(m_i)$, $w(p) \leftarrow w(m_i)$. $w(l)$ for $l \in cover(m_i)$ is decreased by $w(m_i)$ and if $w(l) = 0$, the lifeline l is removed from L' and is not drawn. Then the procedure *MergePlacholders*(T') (see Table I) is called recursively, first on all child elements and than on their parent elements, guaranteeing that subchart elements will also be merged.

For zoom-in, if the zoom-out step is some $i > 1$ then *AddMessage*(m_i) is called for $i \leftarrow i - 1$. In *AddMessage*(m_i), let $p = \{p \in P | m_i \in contain(p)\}$, m_i is added to $parent(p)$ before p, the placeholder's weight is updated $w(p) = w(p) - w(m_i)$ and if $w(p) = 0$ then p is removed from T'. For all lifelines $l \in cover(m_i)$, if $l \notin L'$ then l is added to L', and $w(l) \leftarrow w(l) + w(m_i)$. If m_i is a child of a subchart in T that is contained in p, then m_i is not added directly; rather, the subchart is added and m_i is added as its child.

At a specific zoom-out level, for scrolling up (respectively, down), at each step the message with the smallest index m_i is removed (resp., added) and the message with the largest index m_j is added (resp., removed) using the *RemoveMessage* and *AddMessage* procedures, and then *MergePlaceholders* is called.

VI. MEANINGFUL SEMANTIC ORDERS

The semantic order is determined by weights that are calculated automatically before navigation as a consequence of the task at hand. These weights can additionally be modified manually by the user, but in most cases it is preferable to calculate them automatically. The computation may take time but it is performed offline, and therefore does not affect the user's experience of navigation. This section discusses some possible semantic orders and their applicability.

The spatial order is inherently part of the diagram. It has a clear temporal meaning, which induces a default semantic order for navigation.

Another order mentioned earlier is that created between groups of messages using lifeline composition, as suggested in [9], or class hierarchy, as described in [10]. In composition, a lifeline is composed of additional lifelines defined in a separate diagram. In hierarchy, a lifeline can also be composed of other lifelines (but the semantics is different, of course). Ties within element-sets can be resolved using the spatial order.

The other orders we suggest are heuristic, and depend on the task at hand.

A. Interdependencies for comprehension

One heuristic semantic order connected to program comprehension is related to the dependencies between messages. This order is the result of applying some function on statistics of the elements to calculate custom weights automatically. For statistics, the definition of unification is used. Roughly, when there exists two messages with the same method and connecting the same objects, but in different LSCs, they can be unified. The formal definition can be found in [1] and can be applied also to messages in the same LSC.

For a message m, we define the following local and global statistics. Local statistics depend on the single LSC: let $U_l(m)$ be the number of messages unifiable with the message local to the LSC. Let $C_l(m)$ be the *causal weight*, which is the fraction it constitutes of the prechart. A message that does not appear in the prechart has weight 0, and that of a message that is the only one in the prechart (i.e., its occurrence alone triggers the LSC's main chart) is 1.

Global statistics depend on the entire specification; that is, the full set of LSCs. Let $C_g(m)$ be the number of messages that can either be caused by m (e.g., m is in a main chart and the unifiable messages are in a prechart), or are causal to m (e.g. m is in a prechart, and the unifiable messages are in a different main chart), $U_g(m)$ the number of messages unifiable with m that are not causal or caused by m.

Our current implementation in the *play-engine* tool supports only this semantic order, and we have used equal weights for the local and global components. These weights, $w_l = 0.5$ and $w_g = 0.5$ are parameters that may be changed according to user preferences.

Using the aforementioned statistics, the final custom weight for a message m is the real number: $w(m) = w_l(\frac{1}{1+U_l(m)} + C_l(m)) + w_g(\frac{1}{1+U_g(m)} + C_g(m))$.

The $U_l(m)$ and $U_g(m)$ components make the weight of messages that appear once higher than those that repeat, both locally and globally. The causal components, both local

$C_l(m)$ and global $C_g(m)$ add to the weight of messages that cause changes in the same LSC or between LSCs. In a way this is counterintuitive to indexing methods that give higher weights to repetitive elements. However, it makes sense when the task at hand is comprehension, or, more specifically, comprehending the interdependencies between different LSCs.

Testing this heuristic on sample specifications shows that messages that are used frequently, such as clock ticks or enabling and disabling of buttons, tend to have lower weights, and are indeed less interesting to the reader. See, for example, part of a large LSC from an ATM sample specification in Figure 1 and how when zoomed, the tick messages are hidden before other messages in Figure 4.

B. *Semantic order for debug*

Another semantic order that plays a significant role in software analysis and debugging is the order of execution. In languages such as LSCs this is not identical to the textual/spatial order. LSCs are of inherent potential nondeterminism, with a partial order existing between messages in a single LSC and subtle behavioral and temporal dependencies between multiple LSCs, with their enabled and forbidden events [2]. And in smart play-out, the execution mechanism plans a series of steps ahead of time [5], so that the notion of order of execution is not a trivial matter that can be read from the text/chart.

When the task is debugging, and the focus of the user is on the recently executed message, the semantic order can change with each debug step to show previously executed messages and future enabled messages with higher weights. This will allow the user to watch a smaller window of proximal messages not necessarily in the spatial sense but in the executable sense. The ability to watch a partial LSC in a small window can clear an area for displaying other relevant LSCs, thus allowing the user to see how the LSCs interact. We have not implemented semantic zoom for debug and we leave the details for future work.

C. *Generation order*

Another order, which requires additional information that is available during diagram generation, is the order the user chose to add the elements during programming. This order has the value of displaying the user's cognitive process, and how certain elements were added later than others. In many cases, the later additions are the low-level details, or sometimes 'patches' to fix holes in the specification. This semantic order is relevant for comprehension and can be accumulated during the programming process (play-in with LSCs, for example).

D. *User selection*

A user may want to directly affect the semantic order, for example, in LSCs assigning more weights to some messages, and less to others. As in other interactive works [7], allowing the user to interact with the diagram and specify the details he/she is interested in helps navigation, this order can be combined with a default order to avoid requiring the user to assign weights to all the elements and to help the user avoid excessive interaction. Using the placeholder element for interaction, the user can also choose to expand all elements consumed by a placeholder by double-clicking it, or to extract the single highest weighted element, depending on the tool implementation.

VII. Applications to conventional programming

Reverse engineered sequence diagrams are widely available and are automatically generated by commercial and research tools. In many cases these diagrams are very large and hard to read and navigate. Applying a semantic order and using the suggested algorithms for zooming and scrolling can improve the usability of these diagrams.

Our approach may also be useful in navigating textual code. Although textual code is spatially ordered by lines, there is much information filtering that can be applied to lines of code. Most editors today allow the ability to collapse lines of code that are part of the same function or class. However, code lines have similar dependencies to those of LSC messages. If we replace unification by calls to the same function, we can create a weight function for each line that will provide information on how much this line is repetitive within a code project or a class. This information may be valuable when comprehending code and debugging.

Once the information exists, one can even debug only lines with an information level higher than a certain threshold, hiding other code lines using the suggested placeholder algorithm. For example, one can hide code lines that call a logger or deal with a database, that often repeat throughout the code, although they do not appear in consecutive lines.

VIII. Related Work

The idea of semantic zoom and zoomable user interfaces is fundamental to navigating large information spaces [11], and has been addressed also in the domain of structured textual code [12] and in model engineering [13], [7], [14]. In [12] a *degree of interest* (DOI) is defined, to allow fisheye view of information, a view that distorts information in order to allow focusing on some details rather than others. The idea is applied to textual code based on its tree structure.

Some of the ideas discussed here have been previously researched for textual code. Structure of textual code has been used in [15] for better comprehension and for navigation between components. Different indexing strategies, such as statistical measures for code parts [15] or social tagging by experts [16], have been used for better navigation in large textual code projects.

The navigation problem becomes more difficult when dealing with complex graphical models that present layout.

Challenges that do not exist when reading sequential text [14].

Many navigation solutions exist for class diagrams [13], which have been researched more extensively than LSCs or SDs, and include hierarchies that are exploited for navigation.

More recent work also address navigating and zooming for the full set of UML diagrams, [8], [14]. In [8], various diagrams are connected by special arrows for quick navigation and additionally, semantic zoom has been suggested for interrelationships between elements from different diagrams and for displaying the coarser details of a single diagram. The work in [8] also discusses sequence diagrams briefly, mentioning for example focusing on selected lifeline titles. A similar work that focuses on multiple UML diagrams [14] acknowledges that in UML multi-diagram models are loosely coupled and are therefore hard to navigate. Novel ways have been suggested to integrate different modeling aspects (such as structure, data and behavior) into a coherent model that allows definitions for navigations. In these works, sequence diagrams are treated as one among the many diagrams available in UML.

Recently, sequence diagrams have been acknowledged as important in reverse engineering [7] and novel ways have been suggested to view large diagrams using interactive zooming. They include interactively focusing on parts of the diagram while the context is displayed as small low resolution image and as collapsed fragments in the zoomed diagram. These methods can also apply to live sequence charts, yet they require extensive user interaction.

In the current paper, LSCs are considered as interconnected scenarios in an executable specification and semantic zoom is discussed for navigation and comprehension. New methods for displaying missing information and context are suggested, and less interaction is required from the user when navigating. The formulation of custom weights enables the creation of new detailed orders that are not part of the single LSC, but can provide additional information for different tasks.

IX. Conclusions and Future Work

The main contribution of the current work is to allow semantic navigation in LSCs, a form of visual programming language that does not have a trivial level of details for zooming or navigation.

Nevertheless, some of our ideas can extend beyond the domain of LSCs. Specifically, the idea of creating a semantic order that provides information not found in the original order of the artifacts might be of more general use. Also the idea of creating some form of placeholder for abstracted information, that can hint at the amount of abstracted information for non-continuous regions and merge with other placeholders depending on the 'geography', may be useful for other environments. For example, it is possible to use the notion of placeholder to abstract states in a statechart, if a different visual notation, such as a dot placeholder, is used, and the rule to merge placeholders is adjusted so that there would be a straight line connecting two placeholders, in order for them to merge.

We believe our work can also contribute to the dependency graph between LSCs, when navigating a large specification, as described in [17], and that it can assist in viewing connected LSCs side by side for simulation and debug. Although it is hard to assess how much the new method helps in navigation, we did run a cognitive walk through a large specification of an ATM that also included some large LSCs (Figure 1) to support our claims that the method can assist navigating.

When encountering a large LSC, it is necessary to scan it, sometimes completely and all the way to its end, in order to understand what it 'says'. In the context of LSCs, it is often necessary to scan multiple LSCs to understand how they interact. Our method provides zoom and scrolling that are widely used when reading information that is too large to fit on a screen. The user receives feedback that he/she is viewing parts of the full diagram from the placeholders, and also has knowledge about where the missing information is hidden so that he/she can form a mental model of the sequence of events that occurred, rather than having to scan the full document.

We have implemented the current ideas for the interdependencies semantic order. We plan to create a tool that will work also for UML SDs and will allow interaction and user defined weights. We would also like to evaluate our method in visualizing debug and simulation runs, operations that do not scale well for large LSCs or for a large specification, at the current time.

Acknowledgment

The authors would like to thank Shahar Maoz for preliminary discussions on some of the ideas in this work and for his helpful comments. We thank Liat Nakar for contributing her large LSC specification of an ATM system. We also appreciate the assistance and helpful suggestions of Itai Segall and Smadar Szekely. The first-listed author would like to thank Goren Gordon for his support, general and specific.

References

[1] D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag, 2003.

[2] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," *Formal Methods in System Design*, vol. 19, no. 1, pp. 45–80, 2001. [Online]. Available: citeseer.ist.psu.edu/damm01lscs.html

[3] ITU: International Telecommunication Union, "Recommendation Z.120: Message Sequence Chart (MSC)," Technical report, 1996.

[4] UML, “Unified Modeling Language Superstructure, v2.1.1,” Object Management Group, Tech. Rep. formal/2007-02-03, 2007.

[5] D. Harel, H. Kugler, R. Marelly, and A. Pnueli, “Smart Play-Out of Behavioral Requirements,” in *Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD’02)*. Springer-Verlag, 2002, pp. 378–398.

[6] D. Harel and I. Segall, “Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs,” in *Proc. 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’07)*, 2007, pp. 485–499.

[7] R. Sharp and A. Rountev, “Interactive exploration of uml sequence diagrams,” in *Proc. of the 3rd IEEE international Workshop on Visualizing Software For Understanding and Analysis (VISSOFT’05)*, 2005.

[8] M. Frisch, R. Dachselt, and T. Brückmann, “Towards seamless semantic zooming techniques for uml diagrams,” in *Proc. of the 4th ACM symposium on Software visualization (SoftVis’08)*, 2008, pp. 207–208.

[9] Y. Atir, D. Harel, A. Kleinbort, and S. Maoz, “Object composition in scenario-based programming,” in *Proc. Fundamental Approaches to Software Engineering, 11th International Conference, (FASE’08)*, 2008, pp. 301–316.

[10] D. Lo and S. Maoz, “Mining hierarchical scenario-based specifications,” in *24th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE’09)*, 2009.

[11] S. Pook, E. Lecolinet, G. Vaysseix, and E. Barillot, “Context and interaction in zoomable user interfaces,” in *Proc. of the working conference on Advanced visual interfaces (AVI’00)*, 2000, pp. 227–231.

[12] G. W. Furnas, “Generalized fisheye views,” *SIGCHI Bull.*, vol. 17, no. 4, pp. 16–23, 1986.

[13] A. Egyed, “Semantic abstraction rules for class diagrams,” in *Proc. of the 15th IEEE International Conference of Automated Software Engineering (ASE’00)*, 2000, pp. 301–304.

[14] T. Reinhard, S. Meier, R. Stoiber, C. Cramer, and M. Glinz, “Tool support for the navigation in graphical models,” in *Proc. of the 30th international conference on Software engineering (ICSE’08)*, 2008, pp. 823–826.

[15] J. I. Maletic and A. Marcus, “Supporting program comprehension using semantic and structural information,” in *Proc. of the 23rd International Conference on Software Engineering (ICSE’01)*, 2001, pp. 103–112.

[16] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby, “Waypointing and social tagging to support program navigation,” in *Extended abstracts on Human factors in computing systems (CHI’06)*, 2006, pp. 1367–1372.

[17] D. Harel and I. Segall, “Visualizing inter-dependencies between scenarios,” in *Proc. ACM Symposium on Software Visualization (SOFTVIS’08)*, 2008, pp. 145–153.