Visualizing Inter-Dependencies Between Scenarios *

David Harel
The Weizmann Institute of Science
Rehovot, Israel
dharel@weizmann.ac.il

Itai Segall
The Weizmann Institute of Science
Rehovot, Israel
itai.segall@weizmann.ac.il

Abstract

One of the main challenges in understanding a scenario-based specification of a reactive system is rooted in the inter-dependencies between the scenarios. These are inherently implicit in the very idea of scenario-based programming. We introduce a graph-based visualization of such inter-dependencies, and implement it in a tool we call SIV (for Scenario Inter-dependency Visualization), which supplies many options for exploration of these graphs.

1 Introduction

Scenario-based modeling appears to be a promising approach to system and software design and development, and has resulted in intensive research efforts in the last few years. One of the most widely used languages for capturing inter-object scenario-based specifications is that of *message sequence charts* (MSCs) proposed by the ITU [ITU 1996], or its UML variant, *sequence diagrams* [UML 2005]. This language has been significantly extended, resulting in a highly expressive medium for specification and programming called *live sequence charts* (LSCs) [Damm and Harel 2001]. LSCs are multi-modal charts that distinguish between behaviors that may happen (existential, cold) and those that must happen (universal, hot), and which can also easily express many other notions and constraints on scenarios, such as forbidden behavior. A variant of LSCs has also been defined, which adheres to the UML 2.0 standard; see [Harel and Maoz 2008]. The language has been extended to include, among other things, a detailed notion of time as well as symbolic instances (i.e., the ability to talk also about classes, rather than only object instances) [Harel and Marelly 2003]. An LSC is typically divided into two parts, a prechart and a main chart, the former being a precondition for the latter; i.e., in any execution (run) of the system, if the prechart of an LSC is satisfied (meaning that the run manages to get through it), then following that its main chart must be satisfied too.

One of the main problems that arise when one tries to understand an LSC specification is rooted in the complex inter-LSC dependencies. The meaning of a single LSC is evident from looking at the chart, but the way it relates to other LSCs in the system during execution is not visual in any way. The programmer or specifier might have to look through the entire LSC specification to try to figure out which LSCs relate to each other and how. Obviously, this becomes harder as the specification grows, and for very large LSC specifications/programs it becomes practically impossible.

In previous work done in our group we have made an attempt to visualize the interactions between LSCs [Maoz et al. 2007]. This work proposes a technique for visualizing and exploring execution traces of scenario-based models, particularly LSCs. The tool visualizes the activation and progress of the scenarios in the specification as they "come to life" during execution, by focussing on the activation and closure of each chart. The specification is plotted using a hierarchical Gannt chart, in which a leaf represents a single sequence chart. Each chart is plotted as a set of bars, each representing an active instance of the chart at specific time intervals.

In the present paper we focus on visualizing inter-dependencies between sequence charts, using both static and dynamic data. We use standard graph-visualization techniques for displaying a graph that represents the specification as a whole (or an interesting subset thereof). Each LSC is represented in the graph by a node, and edges represent various kinds of inter-dependencies between them. We introduce the SIV tool (for Scenario Inter-dependency Visualization), which implements the ideas and supplies various views, filters, and options for visualizing the graph and garnering information from it. We are thus able to display and emphasize aspects of the specification that would otherwise be hidden or highly unclear.

2 Preliminaries

Live Sequence Charts

LSCs [Damm and Harel 2001] constitute an extension of message sequence charts (MSCs) [ITU 1996]. LSCs, like MSCs, contain vertical lines, termed *lifelines*, which denote objects, and time flows from top to bottom. The most basic construct of the language are messages. A message is denoted by an arrow between two lifelines (or from a lifeline to itself), representing the event of the source object sending a message to the target object. More advanced constructs, like conditions, if-then-else, loops, etc. can also be expressed. A typical LSC consists of a prechart (denoted by a blue dashed hexagon), and a main chart (denoted by a solid frame). The intended semantics is that if the prechart is satisfied in a run of the system, then following its completion the main chart must also be satisfied.

Several extensions to the language were introduced in [Harel and Marelly 2003], including *symbolic lifelines*, which represent classes rather than object instances, a detailed notion of *time*, and *forbidden elements*, which forbid some event or system state from happening during a given scope in the life cycle of the LSC.

LSCs are multi-modal, in the sense that almost any construct in the language can be either *cold* (usually denoted by the color blue) or *hot* (denoted by red), with a semantics of "may happen" or "must happen", respectively. If a cold element is violated (say a condition that is not true when reached), this is considered a legal part of the specification and some appropriate action is taken. Violation of a hot element, however, is considered a violation of the specification and must be avoided. An example of an LSC can be seen in Figure 1.

*This research was supported by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science.

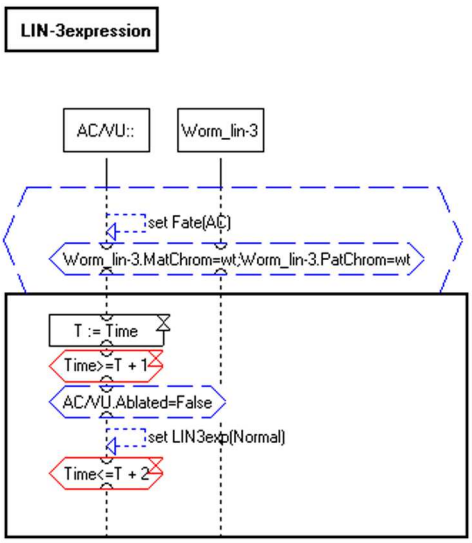

Figure 1: A sample LSC, stating that if an object of class **AC/VU** sends itself a message "**set Fate(AC)**", then it should send itself a message "**set Lin3exp(Normal)**" after 1-2 clock ticks, and only if certain cold conditions hold

An operational semantics was defined for the language in [Harel and Marelly 2003]. The result of this semantics is an execution technique for LSCs, called *play-out*. A basic notion in the operational semantics is that of *unification*, which calls for events to be unified at runtime if they represent the same message passing between the same objects. Note that such unification is not always known at design time, as it depends on bindings of symbolic lifelines, assignment to variables, the state of the relevant LSCs, etc. Nevertheless, an over-estimation of the events that might become unifiable during execution can be calculated statically, a fact that we exploit in our visualization method.

The C. elegans Model

Throughout this paper, we use as an example a rather extensive LSC model of a biological system. Specifically, it specifies the process of vulval precursor cell fate determination in the development of the *C. elegans* nematode worm [Kam et al. 2003]. (The model has been greatly extended since the publication of [Kam et al. 2003].) As argued elsewhere, we are of the opinion that understanding and analyzing biological systems, especially their dynamics, has much in common with understanding and specifying man-made reactive systems [Harel 2003]. We have chosen the *C. elegans* example here both because it is the largest LSC specification we are familiar with, but also because it comes from biology and is thus probably unfamiliar to most readers of this paper. We would like to claim that many features of the system are evident from our visualization approach, even without prior knowledge of the system and specification. This is also the reason we have chosen not to supply in this paper an in-depth description of the exact specification, nor of the system it models.

Examples of other LSC specifications can be found on the SIV website, at [SIV].

3 LSC Inter-Dependency Visualization

We represent an LSC specification as a graph, termed the inter-dependency graph (or IDG). In IDG, each LSC is represented by a node. Edges represent various kinds of inter-dependencies between the charts, as detailed below, and each edge-type is associated with a different color. Special nodes represent the objects external to the system — the user, the environment and the clock. The graph is visualized in the SIV tool using a force-directed layout algorithm [Fruchterman and Reingold 1991], as implemented in the prefuse toolkit [Heer et al. 2005]. Various views, filters, and options are supplied, to allow the user to get as much information as possible from the visualization. We describe some of these below. A video demonstrating the tool and its options can be viewed at [SIV].

The tool, along with several example input files, is available for download at [SIV].

3.1 IDG – The Inter-Dependency Graph

The vertices of the graph denote the various LSCs in the specification (or in a predefined subset thereof). Three special vertices are used for representing the three objects external to the system — the user, the environment and the clock. Edges in the graph connect two LSC nodes, or an external object node to an LSC node. LSC to LSC Edges can be of different types, according to possible LSC inter-dependencies:

- *Causal* edge: A directed edge that represents a main chart message in the source LSC being unifiable with a prechart message in the target LSC. This edge type represents the possibility that during runtime the source LSC might trigger a prechart event in the target LSC. Causal edges are denoted by directed brown lines.
- *Sync* edge: An undirected edge that represents a main chart message in one LSC being unifiable with a main chart message in the other. This edge type represents the possibility that two LSCs might have to synchronize their execution. Sync edges are denoted by undirected green lines.
- *Hot forbids* edge: A directed edge that represents the source LSC having a hot forbidden message unifiable with some message in the target LSC. This edge type represents the possibility that the source LSC might forbid the advance of the target LSC. Hot forbids edges are denoted by directed red lines.
- *Violates cold forbidden* edge: A directed edge that represents the source LSC having some message unifiable with a cold forbidden message in the target LSC. These edges represent the possibility of the source LSC causing a cold violation of a forbidden element in the target LSC. Violates cold forbidden edges are denoted by directed blue lines.
- *Shared object* edge: An undirected edge that represents the two LSCs having lifelines of the same object. Shared object edges are denoted by undirected purple lines.

Note that all edges are over-estimated, in the sense that if two messages can be unifiable in some run, they are considered unifiable in our visualization. Similarly, symbolic lifelines are considered as lifelines for all objects of the class. The visualization of actual runtime unifications is described later, in Section 4.

Edges between an LSC and an external object (user, environment or clock) can be either Causal or Sync edges, representing messages or conditions relevant to this object in this LSC, located in the prechart or main chart, respectively.

3.2 The Visualization

The inter-dependency graph is visualized using force-directed layout [Fruchterman and Reingold 1991], as implemented in the prefuse toolkit [Heer et al. 2005]. An example is shown in Figure 2.[1] The graph corresponds to the *C. elegans* specification mentioned above. In this example, a predefined subset of the LSCs is visualized, and only Causal and Sync edges are displayed.

Despite the similarities in appearance, our graphs have nothing to do with graphs for genetic pathways and other types of biological networks. It seems that these two could benefit from powerful visualization techniques such as ours. Some work on visualization of such networks can be found in [Chuang and Yang 2005; Junker et al. 2006].

This example shows some of the characteristics of the specification that are visually emphasized using our method. For example, note the **DevelopmentalTime20c** node, which has many outgoing causal edges. This is a triggering LSC that drives the execution of many other LSCs. Note also the set of LSCs named lin3* at the bottom left hand side of the graph, all specifying the handling of the LIN-3 protein – an independent process that is specified by a set of seven LSCs, and is triggered by a single event appearing in the **ACAdoptFate** LSC (as can be seen by the fact that a single edge enters this set of LSCs, from the **ACAdoptFate** LSC). Above these LSCs, notice another quite-independent process, the AC/VU decision, involving seven other LSCs (in the middle of the graph). In the bottom right corner of the graph there is a cluster of many LSCs with many edges connecting them. While the exact nature of these edges does not show up clearly in this view, it is obvious that this is a set of LSCs that are strongly inter-dependent. One can now further investigate these LSCs and the way they interact with each other.

These kind of inter-dependencies between LSCs are not visible to the user in other visualization methods we know of, and we consider it to be one of the main contributions of our work. We believe that this view gives the user an important overview of the specification, and supplies some understanding of the structure of the specification, the role various LSCs play in it, and the relations between LSCs and between different parts of the specification. Without our visualization technique, one could gain this information only by manually going over all LSCs, understanding which messages in each LSC may be unified with other messages, and how. This process is confusing, and often results in misunderstandings of the specification in hand. We believe our tool relieves much of the effort needed for understanding such inter-dependencies.

In SIV, parallel edges are unified to a single edge, colored according to the first relevant type, in the order the types were listed above. We believe that this is the order of importance for most users. A check-box allows one to weigh edges according to the number of parallel edges they represent, thus visualizing the amount of inter-dependencies between the different nodes. In order to view all inter-dependencies between two specific nodes, the user can click on an edge and select the "Details" view. In this view, all edges parallel to the one selected are listed, and the view shows extra information, such as the exact unifiable message or the name of the shared object. One can also select a node, and see a list of all edges adjacent to it. An example is given in Figure 3. Both these views can also be visualized, as we explain in Section 3.4.

[1]The examples in this paper can also be downloaded in color from [SIV].

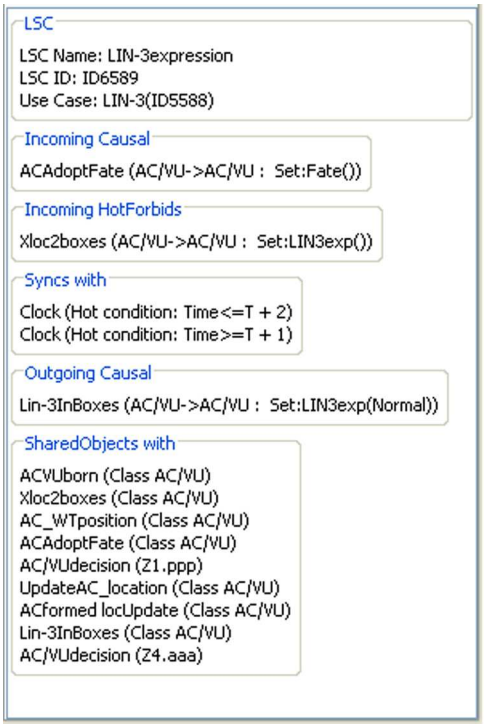

Figure 3: The details view, showing all edges adjacent to the **LIN-3expression** LSC

Figure 2 also demonstrates some of the filtering capabilities of the tool. In this view, we chose to visualize only Causal and Sync edges, and to hide the external object nodes. The user may choose which edge types she wishes to see, and which edge types she wishes to affect the layout; i.e., which edges should be assigned springs in the force-directed layout. This way, the user may separately control the layout of the graph and the visible information. Note that the set of visible and layout-able edges need not be equal — a user may wish to lay the graph out according to certain types and to simply view others. Shared-object edges can be further filtered according to the specific objects that are shared, allowing the user to further control the visible data. For example, in Figure 4, the clock object is added to the graph, and its edges were selected to be visible yet not layout-able. Therefore, the graph layout did not change from Figure 2, even though the edges connecting the clock to the relevant LSCs are now visible.

3.3 Aggregation

In SIV, one can aggregate the nodes according to one of two methods: (a) by the use-cases to which they belong in the specification, or (b) by means of Newman's community identification algorithm [Newman 2004]. The former allows one to study the relationship between the inter-dependencies evident in the graph and the specifier's choice of use-case distribution. The latter tries to group nodes into communities with strong correlations inside each community and weak correlations between groups. Since most possible behavioral

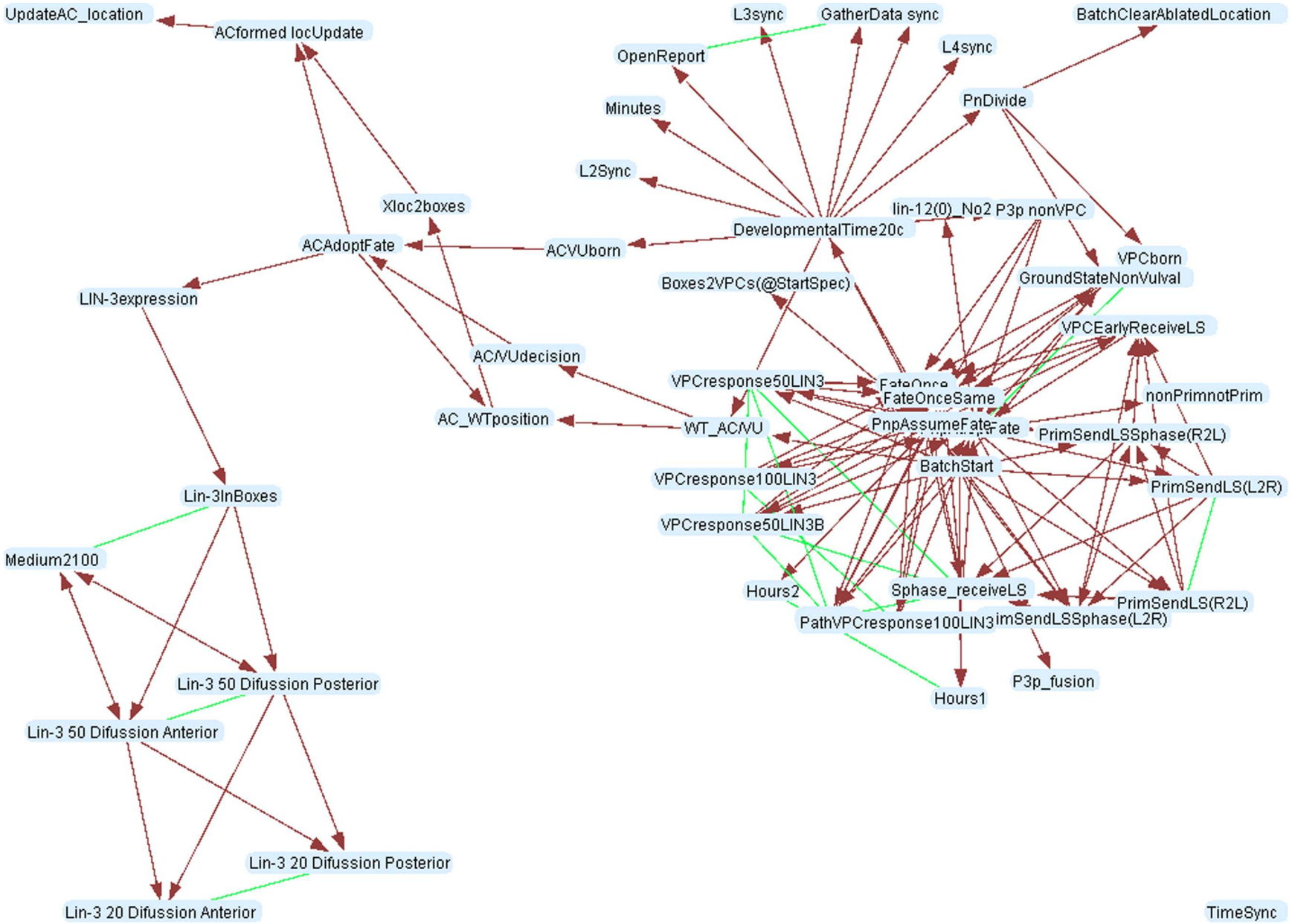

Figure 2: A visualization of the *C. elegans* LSC model

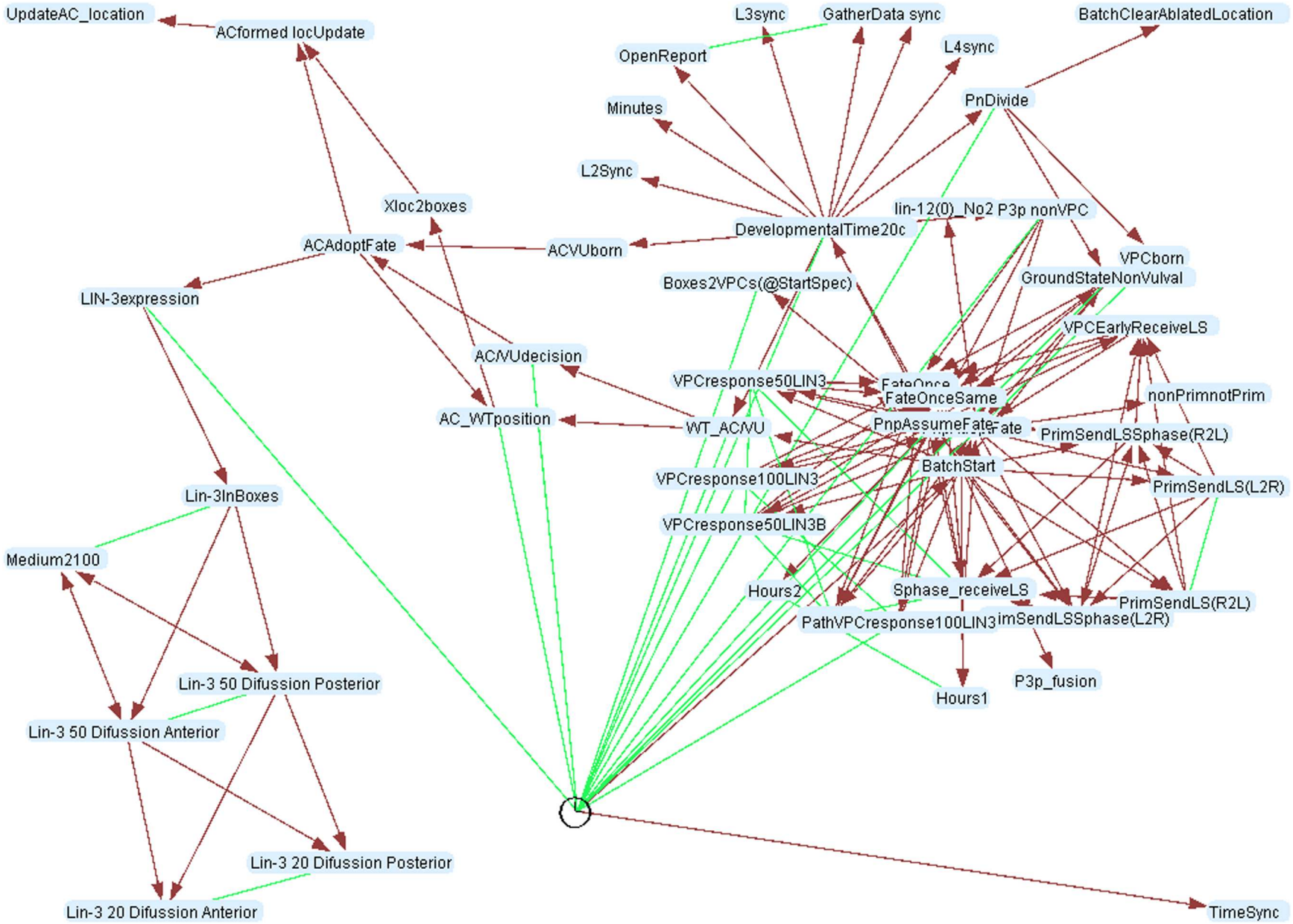

Figure 4: A visualization of the *C. elegans* specification, including the clock node. Edges adjacent to the clock were selected to be visible yet not layout-able.

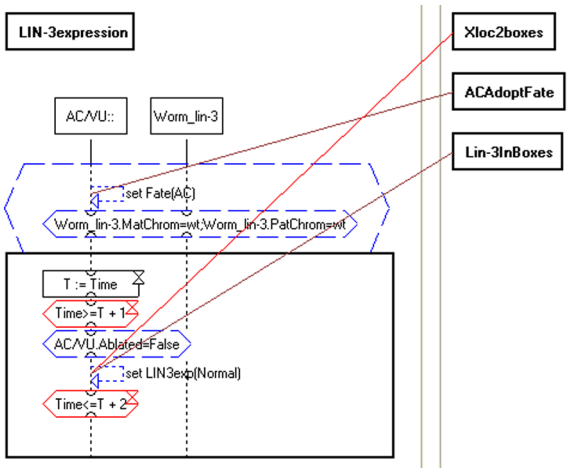

Figure 6: The one-to-many view, visualizing the **LIN-3expression** LSC, alongside a list of all LSCs inter-dependent with it

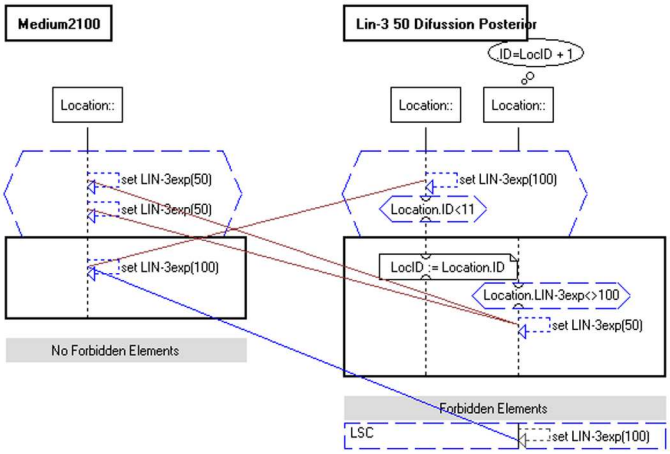

Figure 7: The one-to-one view, visualizing all inter-dependencies between two specific LSCs

inter-dependencies are represented by the graph edges, what we get is a suggested grouping of the LSCs into behavioral groups.

Figure 5 demonstrates the *C. elegans* example, aggregated according to the result of Newman's algorithm. One can see that the two processes described earlier, the lin-3 and AC/VU decision, have been clearly recognized as separate groups. The cloud on the bottom right was split into two groups, giving the developer more information about the structure and the relations therein. Note that the community structure in this example was calculated taking multiple parallel edges into account, even though these are not plotted explicitly.

The example given here consists of 54 nodes, with over 1700 inter-dependency edges (most filtered out in the figures). The nodes were automatically selected out of the 470 LSCs in the full specification, by considering only those that take part in the development of the wildtype worm.

3.4 Play-Engine Connection

The SIV tool may be connected to the Play-Engine tool. This allows the user to double-click an LSC node, and have the LSC automatically be displayed inside the Play-Engine. To accommodate this, two views were added to the Play-Engine. The first, the *one-to-many* view, allows the user to view a complete LSC alongside a list of all LSCs inter-dependent with it. Lines connect messages in this LSC to the names of other LSCs in which they appear. These lines are colored in a way consistent with the coloring in the graph visualization. In this view, the developer obtains a visual indication of all LSCs affected by a single LSC of interest. This view is updated on the fly during development of the LSC — whenever the LSC is changed, the inter-dependency lines are updated accordingly. This gives the developer a strong indication of what parts of the specification are affected by her current modification. See example in Figure 6.

The second view added to the Play-Engine is the *one-to-one* view, in which two specific LSCs are plotted alongside each other, with lines connecting pairs of unifiable messages. These lines are also colored according to the coloring scheme in the graph visualization tool. This view is reachable by double-clicking an edge in the graph, and it visualizes the exact locations, within each LSC, of all edges parallel to the selected edge. This view is also reachable by clicking on one of the LSCs listed in the one-to-many view described above. See example in Figure 7.

4 Visualizing Dynamic Information

The visualization, as described so far, uses and displays only static inter-dependencies between LSCs. Since LSCs form an executable language, in which the LSC specification can be executed directly, one may wish to also visualize dynamic data regarding inter-dependencies between LSCs during execution.

Prior to our work, there was only one way of visualizing the executed LSC specification at runtime. During execution, the Play-Engine opens all active LSCs and displays them along with their current runtime state. This works fine for very small specifications. However, in larger specifications, this view becomes quite useless — too many charts are displayed simultaneously and the developer cannot follow the course of events. Consequently, we decided to adapt our graph-based visualization to also visualize runtime information of the system.

Note that the unification-related edges described in the context of the static view above are an over-approximation of the unification scheme, in the sense that any two messages that could be unifiable in some system run are considered unifiable. In the runtime mode of the tool, however, we highlight those unifications that actually take place during execution. This way, the user gets a feeling of which LSCs drive which other LSCs, and of which LSCs get synchronized during runtime. Moreover, the nodes get colored according to the status of the LSC — blue for LSCs still monitored by the execution (i.e., those for which the prechart is not yet complete), and red for LSCs that drive the execution (i.e., those that are in the main chart). The user can choose to display only the runtime information, or the runtime and static information together. Moreover, she may choose to filter out all edges and nodes that have not yet taken part in the run, thus focusing only on the part of the specification relevant to the run so far. The user may also choose whether highlighted edges will be "sticky" or not; i.e., whether a highlighted edge should remain highlighted. Sticky highlighting allows one to get a full picture of all unifications that have occurred during a specific system run, while non-sticky highlighting gives a sense of the information flow between the LSCs, as edge

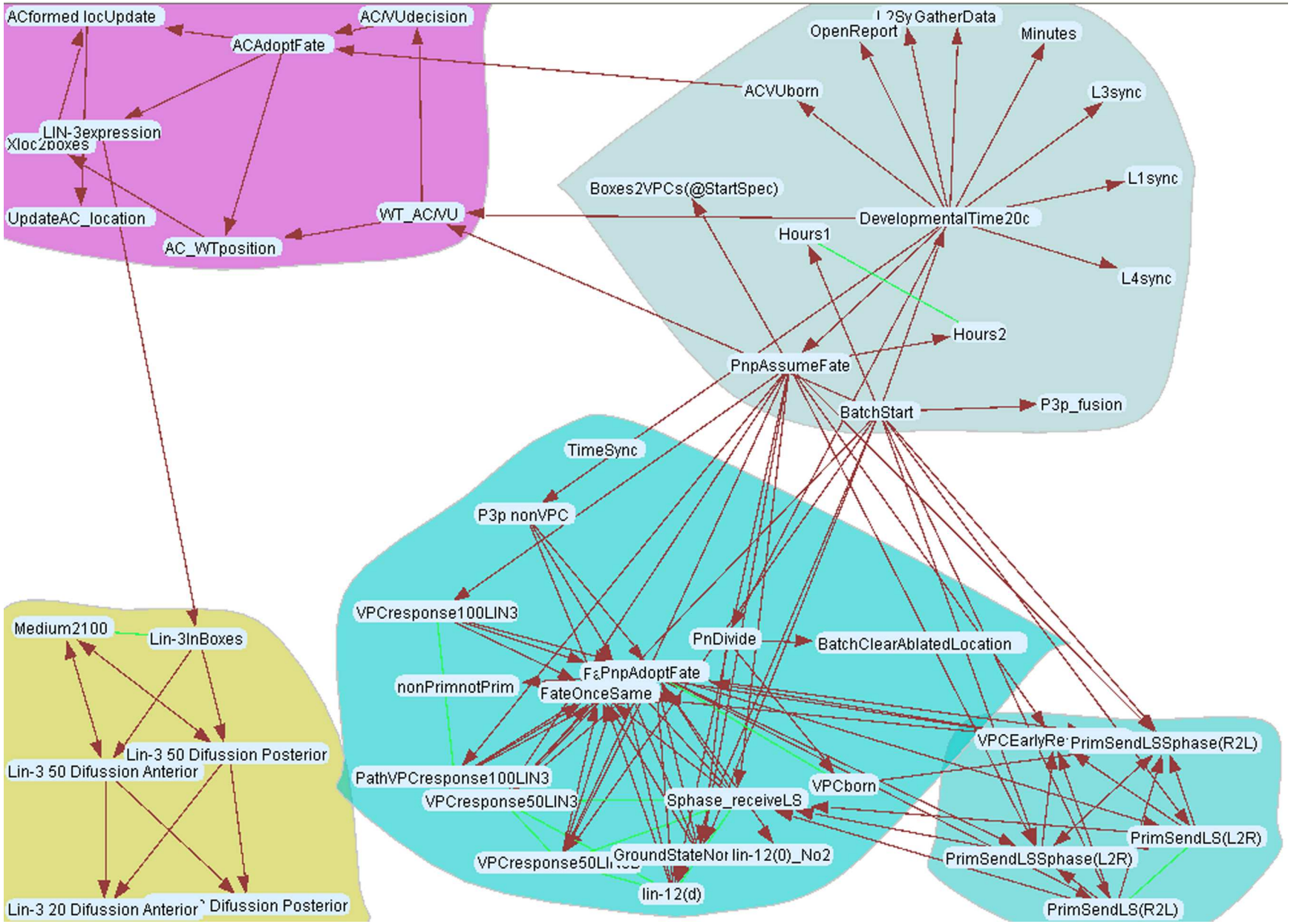

Figure 5: Aggregation of the *C. elegans* specification according to Newman's community identification algorithm

highlighting comes and goes. A video demonstrating the runtime mode in action can be downloaded from [SIV].

The connections to the Play-Engine described earlier also work during runtime. The user can double-click an active LSC, and the LSC will open in the Play-Engine, and will show its runtime information (the current cut, bindings, etc.)

The runtime mode of the tool gives the developer an overview of the specification as it drives execution. It allows one to visualize all relevant parts of the specification at once, therefore giving the user information otherwise unavailable, regarding which LSCs interact with each other when driving the system. The developer can use the Play-Engine connection to "dive" into a specific LSC, or set of LSCs, in order to better understand a specific aspect of the system run.

When used in conjunction with the Tracer tool [Maoz et al. 2007], one is able to obtain a full overview of the system run — both time-based, as visualized by the Tracer, and interaction-based, as visualized by our tool.

5 Beyond LSCs

The work described here uses LSCs for scenario-based specifications. However, most of the ideas here can be used to visualize inter-dependencies between scenarios in other scenario-based languages; e.g. MSCs [ITU 1996] and ScenarioML [Alspaugh 2005]. Some of the definitions here will clearly need to be adjusted for the particular language and its semantics, but we believe this can be done relatively easily, so that the ideas presented here become useful for such languages too.

In [Maoz and Harel 2006], the relation between scenario-based programming and aspect-oriented programming is explored. Much like scenario-based programming, one of the problems in aspect-oriented programming concerns the intricate interactions between the aspects. We believe that techniques like those presented here can also be useful for visualizing inter-dependencies between aspects.

6 Beyond Visualization

The main usage described here for the IDG graph is for visualization. However, the graph may represent deep semantic properties of the specification, and could also be used for other purposes, e.g. in execution.

In [Merom 2006], a distributed execution algorithm for LSC specifications is presented. The starting point of that work is a partitioning of the specification into small sets of LSCs that are to be distributed between different machines. These sets are required to have as few as possible inter-dependencies. We propose to use the inter-dependency graph introduced here, along with Newman's community identification algorithm [Newman 2004], in order to find a reasonable such partitioning. The number of partitions can be given as an input to the algorithm, which will then go ahead and suggest a good partitioning of the specification according to the defined LSC inter-dependencies. This would be a good initial partitioning for the distributed play-out algorithm of [Merom 2006], as the inter-dependencies defined here are exactly those that should be minimized for the algorithm.

Testing this partitioning, comparing it to other methods or hand-made partitioning, is out of scope for this paper, and is left for future work.

7 Related Work

Much research has been done towards better visualization of scenarios. Some focus on visualization of a scenario depicting an actual system run; for example [Jerding et al. 1997], which visualizes interactions between objects in a given system run, and [Reiss 2006], which visualizes program execution by following the states of a user-defined automaton on the traces. Others consider the scenarios to be the system specification, and try to better visualize each scenario independently. For example, [Alspaugh et al. 2006] uses social agents for visualization of specification scenarios. However, to our knowledge, none of these focus on the inter-dependencies between the specification scenarios, but rather on better understanding of each scenario independently.

Another set of related work concerns visualization of graphs. For example, SocialAction [Perer and Shneiderman 2006] is a social network analysis tool, aimed at a better understanding of social networks. It supplies many features for exploration of such networks, and other kinds of graphs in general. We believe that many of the ideas introduced there can be also used for exploration of the graph introduced in this paper.

Software visualization in general is a popular field of study. Graph visualization techniques have been used for many different purposes in work on comprehending software systems. For example, [Würthinger et al. 2008] visualizes the program dependence graph generated by a Java compiler, [Balmas 2004] visualizes dynamic data flow graphs computed at runtime, and [Evstiougov-Babaev 2001] visualizes call graphs and control-flow graphs of embedded systems.

8 Future Work

The tool described here, much like the Play-Engine tool [Harel and Marelly 2003], is an academic prototype, aimed at exploring novel ideas and techniques. Therefore, better evaluation of the ideas depicted here is rather difficult to gather at this point. The example used in this paper was chosen partly due to the fact that it is one of very few large specifications written using the Play-Engine tool. Once LSC tools become more widely spread, it would be important to perform a broader evaluation of the contribution of the ideas described here.

In the future, we would like to implement stronger graph exploration methods, such as those introduced in the SocialAction tool [Perer and Shneiderman 2006], to allow even better understanding of our inter-dependency graph. Using such techniques, the user could easily find LSCs with specific qualities; e.g., ones that have many outgoing edges, or ones that act as "gatekeepers" between two parts of the specification.

We would also like to explore the possibility of applying the ideas introduced here for other languages, such as ScenarioML [Alspaugh 2005] and AspectJ.

Three tools were discussed in this paper: the Play-Engine, the Tracer and the SIV tool. We believe that with better integration between these, one could enable even smarter exploration of the specification. For example, the user could mark a specific time in the Tracer, and explore the state of the specification at that time in the SIV and the Play-Engine. Similarly, SIV could be used to select several LSCs and perform different actions on this selection; e.g. viewing only them in the Tracer.

We also mentioned the possibility of using the graph aggregation for distribution of a specification into components. There is much work to be done in exploring this idea; e.g. comparison with other methods and comparison with hand-tailored distribution.

In this work, our graphs are visualized using a simple force-directed layout mechanism [Fruchterman and Reingold

1991]. Clearly, the graphs can benefit from using more advanced layout techniques. Finding the most appropriate layout techniques for the graphs, both statically and dynamically, is left as future work.

References

Alspaugh, T. A., Tomlinson, B., and Baumer, E. 2006. Using social agents to visualize software scenarios. In *Proceedings of the 2006 ACM symposium on Software visualization (SoftVis06)*, ACM, New York, NY, USA, 87–94.

Alspaugh, T. A. 2005. Temporally Expressive Scenarios in ScenarioML. Tech. Rep. UCI-ISR-05-06, Institute for Software Research, University of California, Irvine.

Balmas, F. 2004. DDFgraph: A Tool for Dynamic Data Flow Graphs Visualization. In *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*, IEEE Computer Society, 516.

Chuang, L.-Y., and Yang, C.-H. 2005. A novel biological pathways tool software. *Biomedical Engineering – Applications, Basis & Communications 17*, 1, 27–30.

Damm, W., and Harel, D. 2001. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design 19*, 1, 45–80. Preliminary version in Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99), (P. Ciancarini, A. Fantechi and R. Gorrieri, eds.), Kluwer Academic Publishers, 1999, pp. 293-312.

Evstiougov-Babaev, A. A. 2001. Call Graph and Control Flow Graph Visualization for Developers of Embedded Applications. In *Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001, Revised Lectures*, Springer, vol. 2269 of *Lecture Notes in Computer Science*, 337–346.

Fruchterman, T. M. J., and Reingold, E. M. 1991. Graph drawing by force-directed placement. *Softw., Pract. Exper. 21*, 11, 1129–1164.

Harel, D., and Maoz, S. 2008. Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. *Software and Systems Modeling 7*, 2, 237–252. (Preliminary version appeared in Proc. 5th Int. Workshop on Scenarios and State-Machines (SCESM'06), 2006).

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

Harel, D. 2003. A grand challenge for computing: Full reactive modeling of a multi-cellular animal. *Bulletin of the EATCS , European Association for Theoretical Computer Science*, 81, 226–235. (Early version prepared for the UK Workshop on Grand Challenges in Computing Research, November 2002. Reprinted in Current Trends in Theoretical Computer Science: The Challenge of the New Century, Algorithms and Complexity, Vol I (Paun, Rozenberg and Salomaa, eds.), World Scientific, pp. 559-568, 2004.).

Heer, J., Card, S. K., and Landay, J. A. 2005. prefuse: a toolkit for interactive information visualization. In *Proceedings of the 2005 Conference on Human Factors in Computing Systems, CHI 2005, Portland, Oregon, USA, April 2-7, 2005*, 421–430.

ITU. 1996. International Telecommunication Union Recommendation Z.120: Message Sequence Charts. Tech. rep.

Jerding, D. F., Stasko, J. T., and Ball, T. 1997. Visualizing interactions in program executions. In *International Conference on Software Engineering (ICSE 1997)*, 360–370.

Junker, B., Klukas, C., and Schreiber, F. 2006. VANTED: A system for advanced data analysis and visualization in the context of biological networks. *BMC Bioinformatics 7*, 109.

Kam, N., Harel, D., Kugler, H., Marelly, R., Pnueli, A., Hubbard, E. J. A., and Stern, M. J. 2003. Formal modeling of c. elegans development: A scenario-based approach. In *Computational Methods in Systems Biology, First International Workshop, CMSB 2003, Roverto, Italy, February 24-26, 2003, Proceedings*, 4–20.

Maoz, S., and Harel, D. 2006. From multi-modal scenarios to code: compiling LSCs into aspectJ. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2005, Portland, Oregon, USA, November 5-11, 2006*, 219–230.

Maoz, S., Kleinbort, A., and Harel, D. 2007. Towards trace visualization and exploration for reactive systems. In *2007 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007), 23-27 September 2007, Coeur d'Alene, Idaho, USA*, IEEE Computer Society, 153–156.

Merom, R. 2006. *Playing together: Distributed collaborative Play-Out of live sequence charts*. Master's thesis, Weizmann Institute of Science, Israel.

Newman, M. E. J. 2004. Fast algorithm for detecting community structure in networks. *Phys. Rev. E 69*, 6 (Jun), 066133.

Perer, A., and Shneiderman, B. 2006. Balancing Systematic and Flexible Exploration of Social Networks. *IEEE Transactions on Visualization and Computer Graphics 12*, 5, 693–700.

Reiss, S. P. 2006. Visualizing program execution using user abstractions. In *Proceedings of the 2006 ACM symposium on Software visualization (SoftVis06)*, ACM, New York, NY, USA, 125–134.

SIV webpage. `http://www.wisdom.weizmann.ac.il/~itais/research/SIV/index.html`.

UML. 2005. Unified Modeling Language Superstructure Specification, v2.0. OMG spec., OMG, August. Available from http://www.omg.org.

Würthinger, T., Wimmer, C., and Mössenböck, H. 2008. Visualization of Program Dependence Graphs. In *Compiler Construction, 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, Springer, vol. 4959 of *Lecture Notes in Computer Science*, 193–196.