Reactive Animation

David Harel[1], Sol Efroni[1,2], and Irun R. Cohen[2]

[1] Dept. of Computer Science and Applied Mathematics
[2] Dept. of Immunology,
Weizmann Institute of Science,
76100 Rehovot, Israel
{sol.efroni, dharel, irun.cohen}@weizmann.ac.il

Abstract. Software engineers use system visualization mainly in two domains: algorithm visualization and system visualization, and both of these are often animated. In this paper we provide a generic link between the specification and animation of complex object-oriented reactive systems, which constitute one of the most important and difficult classes of systems. The link and its methodology form a basis for communication between standard reactive specification tools and standard animation tools. Reactive Animation can be used in a wide range of applications: computer games, navigation and traffic systems, interactive scientific visualization. Reactive Animation helps make the programming of such applications more reliable, expeditious and natural to observe and comprehend. We illustrate two examples: a complex biological model of thymic T-cell behavior and a traffic simulation. [3]

1 Introduction

We describe a generic link between two kinds of computerized tools. The first are tools that aid in the development of complex reactive systems [1], such as aerospace, automotive, communication and medical diagnostic systems. Such tools make possible to specify and execute complex behavior, and are based on visual formalisms [2]. The second tools serve for high-quality graphic animation. We call this combination Reactive Animation. We shall explain the need for a generic means of linking these two quite different kinds of tools, concentrating on complex object-oriented (OO) systems, and describe the promise of such a link. We then describe the link using two particular tools: Rhapsody from I-Logix, Inc. [3], and Flash from Macromedia [4]. We illustrate Reactive Animation with two applications: the behavior of T-cells in the thymus gland and a vehicle traffic system.

1.1 Reactive systems

One of the central issues in software and system engineering over the last decades has been to develop languages, methods and tools for the reliable construction of

[3] Some technical parts of the work described here are patent pending.

reactive systems. This term denotes systems whose complexity stems not necessarily from complicated computation but from complicated reactivity and interaction with the environment — users and/or other systems [5]. Reactive Systems have to handle discrete incoming stimuli, to which they react and with which they interact over time, and which they may also manipulate. Reactive systems are often highly concurrent and time-intensive, and exhibit hybrid behavior that is predominantly discrete in nature but has continuous aspects too. Their structure consists of many interacting, often distributed, components. Very often the structure itself is dynamic, with its components being repeatedly created and destroyed during the system's life. Thus, we are especially interested in systems modeled according to the OO paradigm. The heart of the problem is the need for good approaches to modeling and analyzing the dynamic behavior of reactive systems.

The most widely used frameworks for developing reactive systems feature visual formalisms [2], which are both graphically intuitive and mathematically rigorous, supported by powerful tools that enable full model executability and/or the automatic generation of final runnable code [6],[7],[8] (see [9] for a review). This framework enables realistic simulation prior to actual implementation. Such languages and tools are often based on the OO paradigm, and some are strengthened by verification modules, making it possible not only to execute and simulate the system models (test and observe) but also to verify dynamic properties thereof (prove) [10]. The tools are also being linked to other tools to deal with the system's continuous aspects in a full hybrid fashion. Most of the available tools are state-based, encouraging intra-object style specification [11], but there are also recent scenario-based approaches, which enable executable inter-object specification [12],[13].

The world is full of well-known types of computerized reactive systems, whose reliability and time-critical nature make the issue of good support tools crucial. Interestingly, biology supplies another kind of reactive systems, which exhibit similar characteristics and problematics (apart from the fact that we don't have to design them but, rather, to understand them).

In this paper we use the language of statecharts [14] in its OO version, as implemented in the Rhapsody tool [3],[11]. Statecharts, together with object model diagrams, as described in [11] constitute the core executable part of the UML standard [15],[16]. The specification in Rhapsody, and in other similar tools, is diagrammatic, with the different diagrams describing different aspects of the system, and is also amenable to execution. To build a model of a system in Rhapsody, one usually starts with a description of the relevant classes and possible relations between them. The end result of this systemic account is a diagrammatic representation of the structure of the system, in an object model diagram. However, the main effort is exerted on specifying the system's behavior, in this case using statecharts. This is an intricate process, during which we identify the host of possible states, events and inter-state transitions, that best describe the system's conduct, the interactions between objects that compose the system, the different attributes that are included in each class and the different functions classes own.

1.2 Model execution

Once we have a valid model, a tool like Rhapsody enables us to automatically generate code, which can be run on a desired operating system, and allows us to check and play with the model. For example, we can carry out *diagrammatic animation*, meaning that during execution we get to see the diagrams changing in a way that indicates the dynamic behavior that is being executed. To make model execution useful for a broader spectrum of less-technical people, including end-users, modelers of reactive systems often use graphical user interfaces (GUIs) to portray the look and feel of the modeled system during execution. GUI's also provide a convenient way of manipulating the model during execution. Such interfaces are sometimes created using a programming language's visual "forms", or even coded-in while creating the application's code. Other ways to create such a visual interface is to use specific tools, such as Altia [17]. Some of these tools also come with class libraries that enable the integration of such visual interfaces with the running code. The pressing of buttons, adjustments of knobs, the keying in of different parameters are therefore conveyed to the running model, where the proper operations are performed and the results are sent back to be displayed in the visual interface.

It is important to realize, however, that most GUI tools are limited, and are mainly useful for obtaining a semi-realistic rendition of the highly discrete user interface of the system in operation. They are in fact almost static in nature. For example, if we wanted to build an interface for a cellular phone, we could use GUI tools to render easy access to buttons, displays, and sounds; a button would be clickable and a display would receive values, etc. During execution, such events flow from the user manipulating the GUI to the reactive model of the cellular phone, where they are handled, and appropriate messages are sent back to the GUI to be displayed. GUI tools are thus designed to handle a well-characterized set of element that are found in systems such as watches, phones, cars, planes, electric appliances, software applications, calculators, etc. They do not supply a general programming environment that gives us full control over the different features of the components, or powerful scripting languages to perform simple manipulations of components. They have no general functional abilities or animation techniques. In short, the front end of reactive system tools is very limited, and does not provide us with what we may call *animative freedom.*

1.3 Animation

In this paper we are interested in true computerized animation, of the type one finds in computer games and animation movies. True animation holds extraordinary illustrative and explanatory power, due to its typically high-quality of realism and non-verbal mode of communication. This power is often used in technical fields too, appearing in tutorials, presentations, algorithmic animation, and more. Animation has the ability to detach objects from their specific implementation while holding on to their defining features.

One of our claims is that many reactive systems can be better represented using true animation, in which we can capture the defining features of the system in a realistic manner. By enhancing the representation with the power of animation, we can show the system changing locations, sizes, colors and shapes, switching components, rotating and shifting. And we can also show the system impact on its own structure, or that of other systems, by eliminating parts or giving birth to new ones. Running animation serves as an explanatory tool to the driving simulation. It tells a visual story that comes as close as we want to a real system, limited only by the graphical manipulative power of the animation tool itself, and, in the case of multi-agent simulation, it can tell many stories at once that merge into a comprehensive whole.

How to link up a reactive system "engine" such as Rhapsody, with a true animation tool, such as Flash — resulting in what we call here *Reactive Animation*— is the technical contribution of this paper.

1.4 Related Work

Over the past three decades there has been much work on topics related to the ideas presented here. The first is algorithm animation. Various researchers developed this area into a well-formulated and ordered sub-field of computer science.

Algorithm animation began as a visual abstraction of program operation and its data and dynamics. The main motivation for the algorithm animation tools of the early 1990's was to find different ways of representing and abstracting the text-based, mathematically rich formulation of the dynamics of algorithms with a graphically rich, visually dynamic environment that would provide an intuitive rendition of the logic behind the algorithm. Thus, tools like BALSA [18], Tango [19] and Polka [20] provide the user with the ability to construct a front-end and associate it with the behavior of the animated algorithm. Other tools harnessed the power of a diagrammatic language to animation [21]. Most approaches shared the view of the algorithm as a sequence of events occurring in time, and the most interesting events were chosen to be reflected in the animation. These systems had to confront elementary problems of animation, such as achieving smooth transitions over a discrete course of events, and developing viewers for the animation. Much of the effort in building and using such tools was associated with animation as a teaching tool for computer programming [22]. The software tool Leonardo [23], for example, is used to learn and teach through the visualization of software algorithms.

One group [24] makes a distinction between *abstract* animation - the process of changing data or relations - and *real shape*, which is directly related to how real objects appear. Thus, if we adopt this terminology, we might say that these approaches were concerned mainly with animating the abstract properties of systems. Over the years, the developers of such tools were able to handle distributed technologies [25], and three dimensional representation [26], and to solve problems of compatibility [27]. Other tools, which we will not review here, dealt with the complexity of software engineering and visualized the connection between different classes, different files or different behaviors at run-time.

The conceptual thread that runs through these previous efforts, and leads naturally to our present work, is the process of identifying attention-grabbing events at one level and showing them dynamically at another. The work presented here is not merely another, more powerful tool that claims to make the connection between algorithms and animation quicker or easier. Rather, it provides a generic methodology and a specific implementation to use state-of-the-art tools of one kind in order to build a front-end, and, in a separate effort, to use state-of-the-art tools of a completely different kind to build the specification of the dynamics.

We believe that our methodology can be easily kept up to date on both kinds of tools. Tools for the specification of reactive systems are continuously progressing to match the highest criteria of system modeling, design and deployment, while animation tools have been improving rapidly to satisfy a wide variety of relentlessly fast-growing applications and needs. By dividing the tasks, and isolating the link between these two worlds, we are able to stay updated as both of them develop, resulting in the most powerful "inner parts" of the specification of reactive system dynamics that also "look good" on the outside.

2 Reactive Animation

Most scripting languages that come with animation tools provide a set of instructions that make it possible to perform some manipulation of the animated objects. Such scripting languages, although in principle of universal computing power, are not built with any intention of matching the strength of programming languages. The underlying difference is that here we have the script, a set of commands that is run by some other language and not directly by the computer's processor (as a compiled program would). Scripting languages are common in multimedia tools in general and in animation tools in particular.

We would like to use the term Reactive Animation for the working combination between the power of reactive modeling languages and tools, such as statecharts in Rhapsody, and the power of animation tools like Flash. Reactive Animation provides a vivid representation built on a rigorous, hard core model of the system under description. Furthermore, we achieve this representation without the need to carry out difficult and tedious coding in the animation tool's scripting language, and without having to build complex and cumbersome animation capabilities on top of the reactive modeling tool.

2.1 The two components

We simulate and represent systems using two separate, detached environments. The simulation is designed without the need of any animation, and the animated

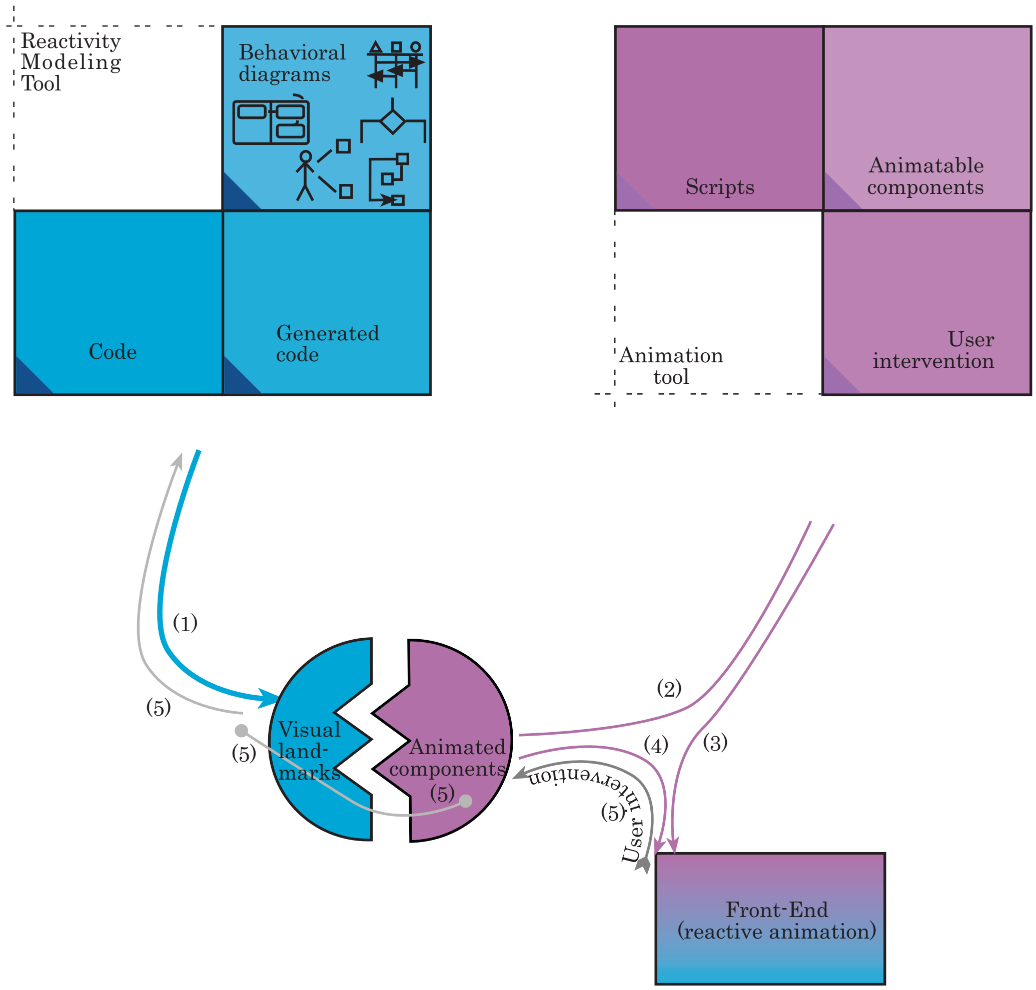

Fig. 1. A diagramtic description of Reactive Animation.

components are designed without the need of any code from the model. The final result, nevertheless, is an attachment of the two — a reactive animation (something like a sophisticated interactive movie).

We build a model of the system without necessarily taking into account the fact that we plan to later represent it in animation. The model will therefore be a regular reactive model of the system, and when building it we do not have to try to specify it according to the way we think it should ultimately be animated. We do not specify the system for the sake of animation. We animate the specification. We design a model by specifying its classes and their interconnections (e.g., with object model diagrams), and their behavior (e.g., with statecharts). We add functions and attributes. We check the model, we run it, we modify it, etc. Some information on what needs to be prepared in the animation tool is provides below (2.2).

We integrate the specification and the desired animation by building a few paths of control to coordinate directions and appearance, and to synchronize the running simulation with components from the animation. The viewer sees the result as a running movie, which spontaneously leads to the notion of a directed sequence of events. However, the movie is in fact generated on the fly by connecting the two facets of the system: our representation of how the system works and our representation of what the system looks like.

It is important to stress that we do not merely use statecharts to specify some intricate behavior of animated scenes or of animated systems, as is suggested elsewhere [28]. We are also not merely trying to incorporate reactive abilities *into* an animation tool.

2.2 How to link the two

Figure 1 is a sketch of the different parts of the scheme and their inter relationships. As the reactive modeling tool (on the left of the figure, in blue) we use Rhapsody from I-Logix. The first effort in representing the specification as animation is to identify the changes in the simulated system that would be best described by animation. We call these changes visual landmarks. Although some of these landmarks are obvious (for example, the movement of parts of the system), some can only be identified by having some pre-knowledge of the system. For example, when we simulate cells (explained in more detail below), we might want to show the appearance and disappearance of receptors on the cell's surface. However, visual landmarks do not include the decision trees incorporated into the cell's behavior.

Path (1) in Figure 1 is, therefore, the identification of visual landmarks and making them, through a communication interface, amenable for the animation tool. On the other side of the figure we see the animation tool in purple, where we build components that are able to handle each of these visual landmarks. We do not build the movie itself, i.e, the way these landmarks are combined. Rather, we only supply the running simulation with enough tools to build the movie by itself. We thus design components that would be able to collectively build, during runtime, a representation of classes from the simulation. Such components need not have a stationary non-dynamic representation, even though they are triggered by immediate changes. Scripting languages help animate these immediate changes into a smoothly running animative representation.

Path (3) in the figure may be understood as making use of the animation tool's projector abilities. Path (4) stands for the dynamic attachments between components that turn them into a representation of the system, and not merely a detached representation of components. Path (5) in the figure is the intervention the user may wish to apply to the simulation while it is running. Such interventions are mediated by components provided by the animation tool. These components are, again, linked to matching events in the simulated system.

The setup required for our two-faceted Reactive Animation therefore consists of: (i) visual landmarks identified in the system; (ii) components in the animation tool that are able to respond to alerts about visual landmarks; (iii) a medium to transfer messages back and fourth between the two tools. So far, we have not discussed the third of these, the medium that enables communication between the two tools. In principle, this medium may be any applicable form of communication that allows data transfer in real time: a communication protocol over real-time systems, Windows DLL files, network protocols, or others. We currently use TCP/IP as the medium and XML files to transfer the data between the two tools we work with.

It is not possible to provide optimal tactics on how one should animate an arbitrary reactive system. Most systems live in a pre-known environment of people and nomenclature, and come with some history of analysis, with its terms and relevant behaviors. These should be identified and sought for in the simulation and should have visual renditions in the animation.

2.3 Conceptual layout

Reactive animation builds upon the concepts and methods that were introduced earlier, in the field of algorithm animation, and the work that stemmed from this field. The conceptual layout is portrayed in Figure 2.

The general procedures in creating reactive animation are similar to algorithm animation: 1. Build the algorithm 2. Identify visual landmarks (or any other terminology for the interesting event during algorithm execution) 3. Build a bridge between the two. The novelty and the contribution of the work we present here is 1. We do not try to represent only the abstract but also real shape 2. Applicability.

3 Examples

We illustrate Reactive Animation using two examples: the development of T cells through the thymus and the simulation of traffic flow. Short videos showing parts of these are available

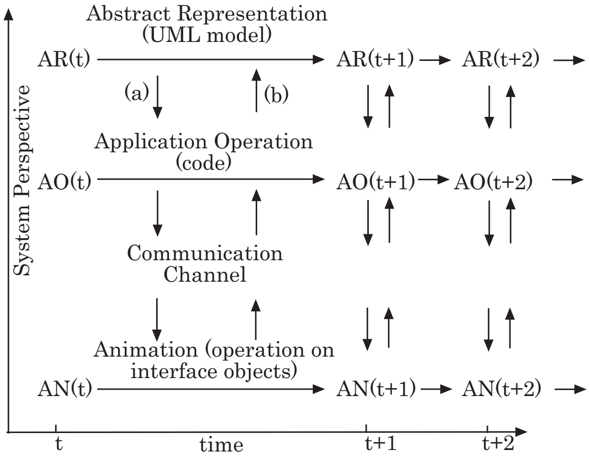

Fig. 2. The conceptual layout of reactive animation. The layout of the concept is similar to any algorithmic animation. The figure is a re-adaptation of a figure from [24]. AR - Abstract Representation, AO - Application Operation, AN - Animation, (a) - code generation. (b) Statecharts animation.

3.1 The Thymus example

This work started from an effort to understand a biological system — the thymus. This is an organ in which thymocytes develop to become mature T cells, which are an important part of the immune system [29],[30]. We have modeled the behavior of this system, and especially that of thymocytes in their journey within the thymus. Our specification of the behavior of cells that comprise the thymus is done using statecharts and object model diagrams in Rhapsody. We analyze the data about the biological objects (e.g., the cells) as they appear in literature. In this analysis, we transform current understanding into states, transitions and parameters.

We then compile the functioning Rhapsody model and run it, thus obtaining a simulation of the relevant aspects of the thymus. The problem with this Rhapsody simulation is that it is very difficult to understand. What we see when executing the model is a diagrammatic simulation, showing the generation of instances, the switching between states, the events that have been consumed, the events that are about to be consumed, and so on. This representation is detailed and rigorous, and it does help in understanding what the thymus is doing, but not nearly enough. Biologists, for example, find extremely difficult to get a feeling for the living system from such a simulation.

To alleviate this problem, we built a Flash front-end that visually resembles the way cells actually appear when viewed through a microscope (or at least the way diagrams can depict this). This front-end animates the course of events in the thymus. It shows the viewer of the animation how cells move, change their receptors, interact with other cells, proliferate, mature, die and secrete different substances. All the processes occur within the Rhapsody model, and can, in principle, be viewed in the Rhapsody simulation itself. We could have viewed the statecharts as they are animated: we could also follow text lines or watch events being consumed. This might be feasible for a system with a small number of cells, but when we have myriads of cells — as is the case here — this is infeasible, and to understand the process, we must have a visual animated front-end to the simulation.

Figure 3 illustrates a snapshot of a very small part of the diagrammatic animation and the true reactive animation. A more elaborate animation may be seen in the accompanying movie. The right hand side of part (a) of the figure shows a small portion of the statechart of a thymocyte. The full statechart is much larger and much more complicated, and cannot be shown and explained in full within the scope of this paper. On the left hand side we see the symbolic presentation. Attached to this spherical-looking rendition of a cell we can see "arms" that represent Part (b) of the figure, following the trail of a single simulated biological event as it starts with the simulation to be transferred to the animation.

Rhapsody continuously assigns a cell phase to each configuration of receptors on a cell's surface. Every such cell phase is given a matching color in the animation. When Rhapsody finds that a state of a cell has been changed, it sends a message to Flash, in which the proper cell is identified, and its phase is declared. When Flash receives this message, it first locates the appropriate cell from among the many animated cells, and then changes that cell's color according to the predefined color code.

Since the user of our simulation receives a large quantity of data via the animation, we would like him/her to be able to interact with the simulation through the animation. To that end, we have also built the animation as an interactive user interface. The user may then choose his/her current focus, send orders to the reactive engine, receive data from it, apply statistical tools, and so on. For example, the user

is able to control the different receptors on each of the cells' surfaces, find out what are some of the more important cell's attributes, or ask for the percentages of cells that are found at different developmental stage. In short, the user may perform experiments with the data and see the outcome.

It is noteworthy that the T-cell thymus model itself is very detailed and complex, and captures a tremendous amount

of information known on the behavior of the system. In fact, we have incorporated information from around 250 biology papers.

3.2 The Traffic Example[4]

Vehicle traffic is another telling example of emerging complexity on both levels. To simulate the collective behavior of moving cars in a complex context, and

[1] This part of the work was done jointly with Aron Inger.

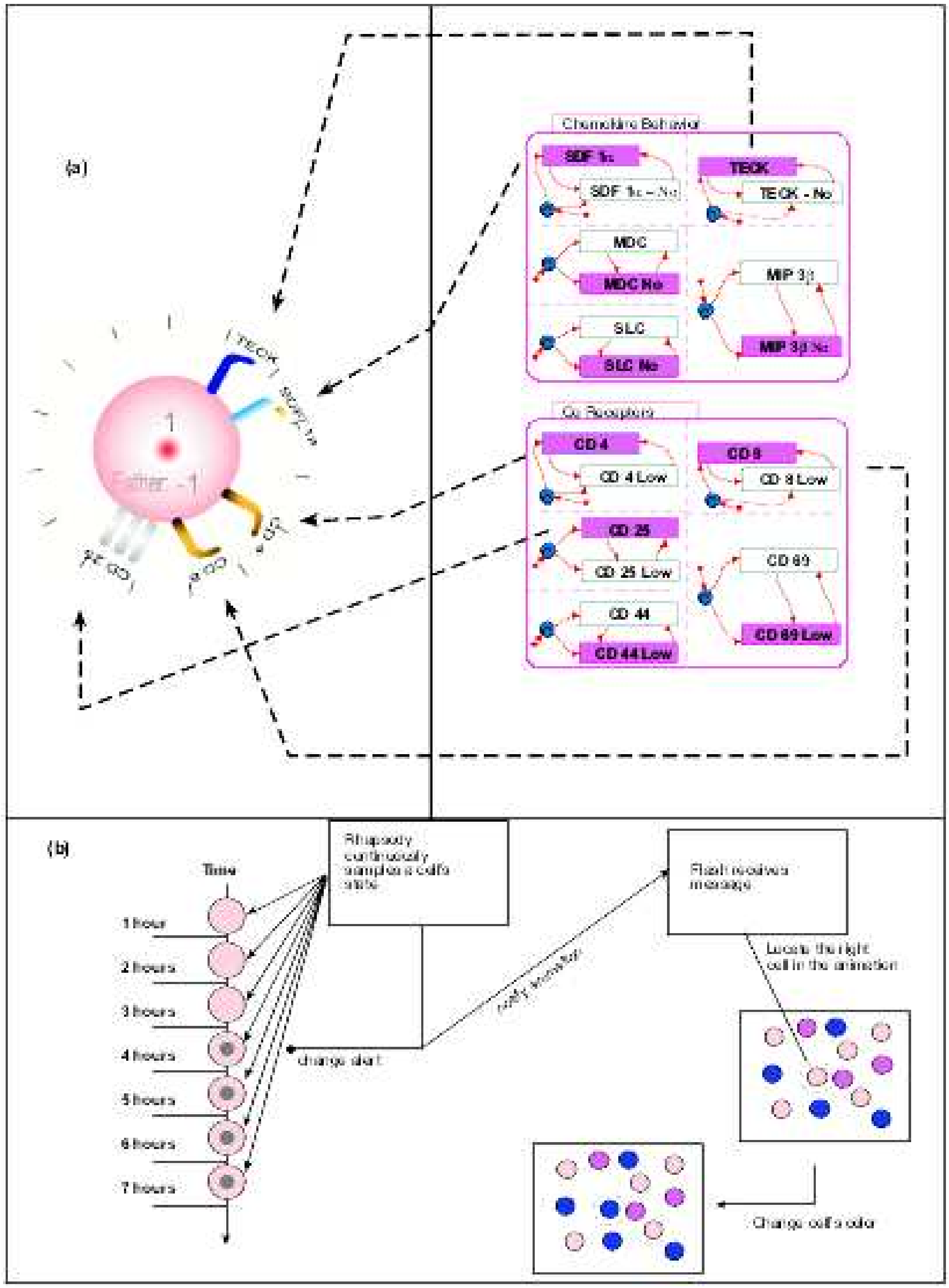

Fig. 3. a) The various features that control the appearance of an animated T cell. b) The procedure for dictating a T cell's color.

adhering to the need for real-time decision-making, we have again used the language of Statecharts in Rhapsody. Statecharts are used to specify the choices the driver makes while driving, the behavior of scene elements in the environment, such as traffic lights, the actual movement of the cars, and the overall management of the project. On the graphical side, we used Flash to build a library of animated cars, traffic lights, pedestrians, houses, lanes, and so on, in order to facilitate a real-time graphical representation of the interacting objects.

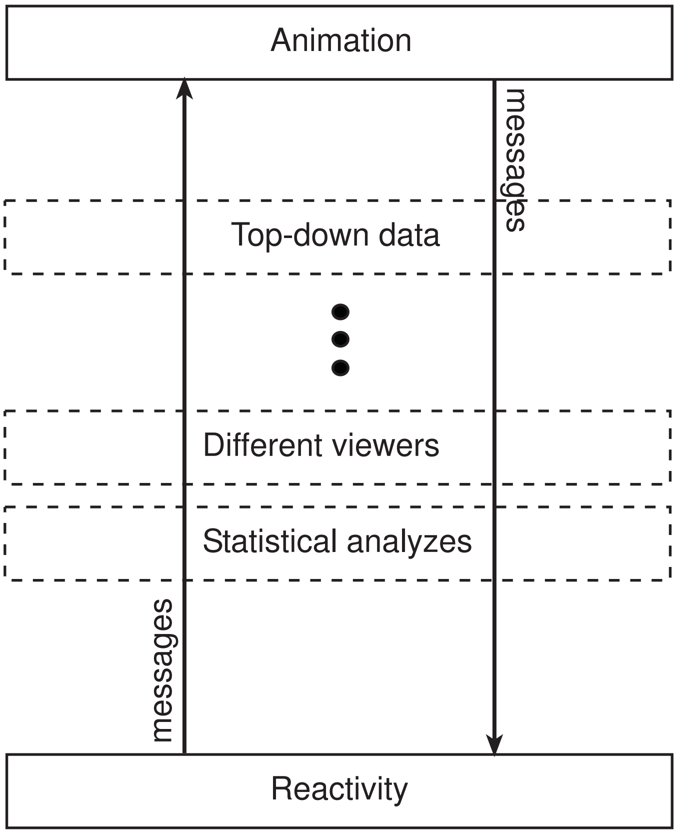

Fig. 4. A general view on layering in Reactive Animation.

The specification of the simulation serves to accomplish two primary goals. The first is the implementation of the overall management of the project. This includes such activities as the entry and removal of all the interactive objects in the environment (e.g., cars, pedestrians and obstacles), along with maintaining a map of their locations in the scene, and the handling and parsing of incoming and outgoing messages between the specification and animation sides of the simulation. The second is the specification of the reactive behavior of the objects.

This includes the cycling of traffic-light states and the movement of pedestrians. The most complex element of the specification is the implementation of the logic of driving a car. Each car must be aware of all relevant objects in its vicinity. Based on the conditions in its immediate neighborhood, a car performs a complex series of calculations in order to determine its correct behavioral response. Decisions are made to avoid obstacles, to obey the rules of the road and to maximize efficiency. In order to accomplish these objectives, cars may choose to increase or decrease their speed, to turn or to switch lanes.

This "awareness" of the car's surroundings and the calculation of its reactions are implemented with Statecharts. Nevertheless, the collective interactions of such a complex multi-agent system cannot be appreciated through the inspection of the Statecharts alone. Animating of the interactive objects, using the graphical front-end, simultaneously and in real time, facilitates a much greater understanding of the simulation. The animation displays a running movie of the current locations of all the objects, as well as their behavior. Traffic lights can be seen cycling through their various colors, while cars move, turn, change lanes, wait in traffic jams, stop for red lights, and even crash into each other. In the true spirit of reactive animation, the information required for this representation is received from Rhapsody and not directly calculated by Flash. The user may also interact with the animation by requesting to display certain kinds of information. Various details, such as the current speed or age of each car, can be displayed, in numeric format or by color-coding the cars in the scene. The awareness neighborhood of individual cars can also be reflected on the map. This sort of high-level information aids in understanding the overall behavior of the simulation in a way that would not be possible without the animation.

An example of the traffic flow simulation can be seen in Figure 5, and a movie of the animation accompanies this paper. The animation of the simulation may be seen on the left side, with some of the Statechart logic displayed on the right.

Combining specification and animation is not only helpful for understanding the behavior of the system but also facilitates the coding of the system as well. Generating the logic behind each car's decision-making process along with debugging this logic as it is coded is difficult to perform in an abstract setting. The animation provides an intuitive demonstration of the resulting behavior generated by the code. This eases the process of designing and implementing the simulation itself.

4 The implementation

This section describes the way we are currently implementing Reactive Animation in specific hardware and software. In addition to thymus and traffic simulations we are currently working on additional projects with the same implementation: cell migration and the fine details of choices made by the immunological repertoire. Somewhat different setups are used for these projects, but in all of them we use Rhapsody (either in C++ or Java) and Flash MX, but we are in

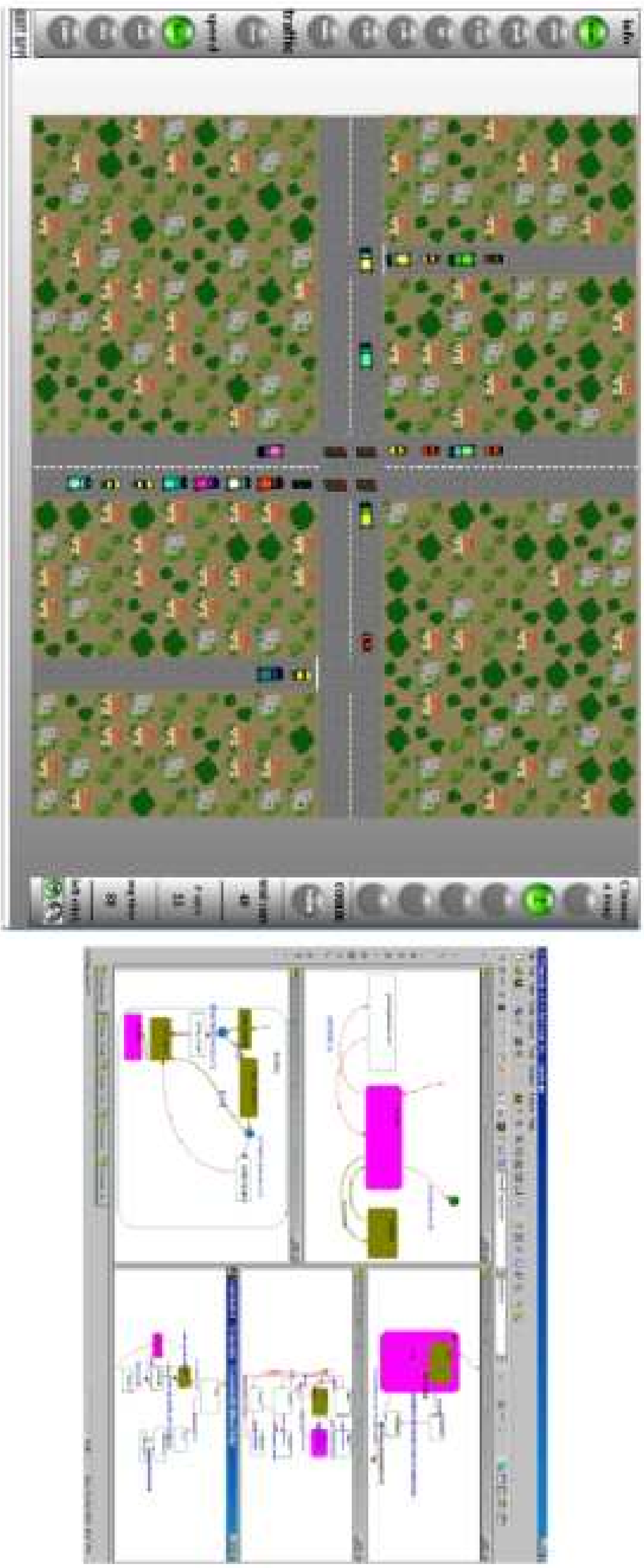

Fig. 5. Reactive animation exemplified in traffic simulation.

the process of integrating other tools and are incorporating three dimensional animation tools.

The connection between the reactive system's behavior (Rhapsody statecharts in execution) and Flash's projector (running animation) is carried out through TCP/IP. At run-time, the messages we send back and fourth between the simulation and animation consist of relatively small XML files. Flash provides built-in tools to handle XML files. We built special-purpose components into the Rhapsody model to be able to handle outgoing and incoming XML files.

The fact that we are using TCP/IP as the transfer medium makes it easy to implement the same configuration of a reactive simulation and its animation over a network. In such a network, one machine can be dedicated to all (hard) reactive simulation work and another machine is dedicated to animation display. This structure presents some advantages:

- Strong computer hardware is dedicated to performing hard tasks related to reactive simulation. CPU power and memory are not busy with tasks related to graphical display.
- Different machines with different hardware configurations may be used in optimal conditions for different computing jobs. Hardware configurations well suited for displaying graphics are not well suited for multi-threading or other computational tasks. By separating the tasks — simulation on one machine and animation on the other — we can provide an optimal hardware environment for each of these missions.
- We can assume that reactive simulation will usually be more demanding than animation. We may also assume that very powerful machines will always be rarer than conventional computing power. To accommodate the limitations of computer power, we can install the reactive simulation on a powerful machine and allow people interested in running the simulation, even if they do not own the necessary computing power, to be able to run the simulation and view its animated interface.
- There are cases where many users wish to simultaneously access the simulation. Examples are multi-participant games, the simulation of war games, traffic flow, air traffic, and more. In such cases, the various multimedia players are available for almost all operating systems, so that a variety of users can access the simulation run on a host machine. Since we can apply layering (explained below), every user may use his/hers own relevant view of the system's behavior. If we take air traffic control as an example, different users may view different areas of the country, while air field designers may wish to look at statistical data generated by some other layer in real-time or post run. Moreover, since the internet is an easy medium for TCP/IP, all this may be done over the internet.

To make the connection between the simulation and the animation, we send the simulation information, in capsules. Currently, only one entity continuously "listens" for such information. In principle, it is possible to add other "listeners", not necessarily animation tools, who may make use of this information.

For example, in our thymus simulation, we continuously release large amounts of data about a large amount of cells. It is difficult to statistically analyze so much data. Still, special tools are now available to make this kind of real-time statistical analysis possible. We could have such tools tapping the flowing data and analyzing it in real-time. Once this analysis is finished, the results may be sent to the animation tool for display (see figure 4). One can easily think of other such connections, since any large system may be viewed from many angles, each of which can be equipped with a listener, providing its own interpretation to the information that comes in.

4.1 Supporting Movies

The movies that accompany this paper provide brief illustrations of the two example applications, as they appear in our implementations. The movies are available at:

http://www.wisdom.weizmann.ac.il/˜sol/reactiveAnimation2003.zip

To view the movies, download the file “reactiveAnimation2003.zip”, extract it to some folder, and open the “main.htm” file.

The first movie, on the thymus, starts with the simplest example for the interplay between reactive specification and front-end animation. It shows the change of receptors on a T cell’s surface, first in the tool we use for specification — Rhapsody — and later in the Flash animation. Following this, we see a population of T cells, which is actually the animated representation of multiple instances of the T-cell object. All of these are created and manipulated by the underlying code that is automatically generated by Rhapsody from the statechart model in order to drive the simulation.

The second movie, on the traffic example, also shows some of the Statecharts in Rhapsody and some scenes from the animated front end. This movie does not include the kind of explanations that the first movie does, only run-time snapshots. In this movie it is particularly important to realize that in such a format we can only show very few of the statecharts that drive the simulation. The statechart portion that is shown in the background dynamically changing states represents a part of a single instance of a car, while in practice there are numerous such instances viewable on the animation.

5 Conclusions

We have presented a method for animating reactivity, or, equivalently, for driving a reactive animation by programming its dynamics using tools developed specifically for classical reactive systems. Thus, we make it possible to better understand the function of a reactive system through an animated front-end. This front-end is not a movie *about* the reactive system; rather, it is continuously generated by the underlying simulated reactive system. We do not change the methodology commonly used by tools for reactive systems, but instead introduce the well established explanatory power of animation. Further, to achieve this, we do not require that the reactive system tools be extended by specially designed add-ons to enable animation, but show how this can be done by directly linking them to available animation tools.

Reactive Animation has other benefits too, including better ways to make use of available hardware; new possibilities for multi-user simulations and games; and ways to incorporate available tools for statistical analysis on the bridge between animation and simulation. Reactive Animation also offers substantial help in handling multi-agent simulations, where dynamic representation is demanding. By using this methodology and its technical benefits, we make use of current (and probably future) state-of-the-art tools. We take advantage of two different facets of expertise in software companies: some make excellent tools for specifying reactive behavior, and others make excellent tools for producing interactive animation. By using Reactive Animation, we separate the efforts of modelers and animators and equip each with state of the art tools, only to later *join* the two.

We are currently in the process of applying Reactive Animation to different simulations and with different tools than the ones illustrated here. One application is the animated representation of models specified with tools for inter-object description [12], such as the recently developed Play-Engine, based on the language of *live sequence charts* [31] and the *play-in/play-out* methodology [13]. Other paths we are exploring include a 3-D representation of another biological environment, using Macromedia's Director [4].

Reactive Animation promises to bring, with relatively small effort, many benefits to people developing or working with reactive systems, and to people interested in enriching the reactive and interactive capabilities of animation per se.

6 Acknowledgments

We wish to thank Aron Inger for his dedicated work on the traffic example, and Yeda Research and Development, Ltd., the Weizmann Institute's technology transfer company, for financial support of that work.

References

1. D. Harel and A. Pnueli, "On the development of reactive systems," in *Logics and Models of Concurrent Systems* (K. R. Apt, ed.), vol. F-13, pp. 477–498, Springer-Verlag, New York, November 1993.
2. D. Harel, "On visual formalisms," *Comm. Assoc. Comput. Mach.*, vol. 31, no. 5, pp. 514–530, 1988.
3. I-Logix Inc. http://www.ilogix.com.
4. Macromedia Inc. http://www.macromedia.com.
5. R. J. Weiringa, *Design Methods for Reactive Systems: Yourdon, Statemate, and the UML.* Boston: Morgan Kaufmann, 2002.
6. Honeywell DOME. http://www.htc.honeywell.com/dome/.

7. Aonix. Software Through Pictures. http://www.aonix.com.
8. Rational Software. http://www.rational.com.
9. R. Mili and R. Steiner, “Software engineering - introduction,” *LNCS*, vol. 2269, pp. 129–137, 2002.
10. Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety.* New York: Springer, 1995.
11. D. Harel and E. Gery, “Executable object modeling with statecharts,” *IEEE Computer*, vol. 30, no. 7, pp. 31–42, 1997.
12. D. Harel and R. Marelly, *Come, Let’s Play: A Scenario-Based Approach to Programming.* In Preperation.
13. D. Harel and R. Marelly, “Specifying and executing behavioral requirements: The play in/play-out approach,” *Software and System Modeling*, To Appear 2003.
14. D. Harel, “Statecharts: A visual formalism for complex systems,” *Sci. Comput. Programming*, vol. 8, pp. 231–274, 1987.
15. Unified Modeling Language. http://www.uml.org.
16. J. Rumbaugh, I. Jacobson, and G. Booch, *The unified modeling language reference manual.* Reading, Mass.: Addison-Wesley, 1999.
17. Altia Inc., Embedded Systems Graphics. http://www.altia.com.
18. M. H. Brown, “Exploring algorithms using balsa-ii,” *IEEE Computer*, vol. 21, no. 15, pp. 14–36, 1988.
19. J. T. Stasko, “Tango: a framework and system for algorithm animation,” *IEEE Computer*, vol. 23, no. 9, pp. 27–39, 1990.
20. B. Topol and J. T. Stasko, “Integrating visualization support into distributed computing systems,” Tech. Rep. GIT-GVU-92-20, Georgia Institute of Technology, 1994.
21. B. Meyer, “Formalization of visual mathematical notations,” in *DR-II: AAAI Symp. on Diagrammatic Reasoning*, (Boston), 1997.
22. C. D. Hundhausen, S. A. Douglas, and J. T. Stasko, “A mets-study of algorithm visualization effectiveness,” *Journal of Visual Languages and Computing*, vol. 13, no. 3, pp. 259–290, 2002.
23. P. CRESCENZI, C. DEMETRESCU, I. FINOCCHI, and R. PETRESCHI, “Reversible execution and visualization of programs with leonardo,” *Journal of Visual Languages and Computing*, vol. 11, no. 2, pp. 125–150, 2000.
24. S. Takahashi, K. Miyashita, S. Matsuoka, and A. Yonezawa, “A framework for constructing animations via declarative mapping rules,” in *Proceedings of IEEE Symposium on Visual Languages*, (St. Louis), pp. 314–322, 1994.
25. G. F. Italiano, G. Cattaneo, U. Ferraro, and V. Scarano, “Catai: Concurrent algorithms and data types animation over the internet,” in *Proceedings of 15th IFIP World Computer Congress.*
26. J. T. Stasko and J. F. Wehrli, “Three-dimensional computation visualization,” Tech. Rep. GIT-GVU-92-20, Georgia Institute of Technology, 1992.
27. J. E. Baker, I. F. Cruz, G. Liotta, and R. Tamassia, “Algorithm animation over the world wide web,” in *Proceedings of the 1996 ACM Workshop on Advanced Visual Interfaces*, pp. 203–212, 1996.
28. J. Kaye and D. Castillo, *Flash MX for Interactive Simulation: How to Construct & Use Device Simulations.* OnWord Press, 2002.
29. I. R. Cohen, *Tending Adam’s Garden: Evolving the Cognitive Immune Self.* San Diego, CA: Academic Press, 2000.
30. R. A. Goldsby, T. J. Kindt, and B. A. Osborne, *Kuby Immunology.* New York: W. H. Freeman and Company, 2000.

31. 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.