Published in the Proceedings of the 2nd International Working Conference on Software Architecture (WICSA), Amsterdam, The Netherlands, August 2001.

Statechart Simulator for Modeling Architectural Dynamics[1]

Alexander Egyed
Teknowledge Corporation
4640 Admiralty Way, Suite 231
Marina Del Rey, CA 90292, USA
aegyed@acm.org

Dave Wile
Teknowledge Corporation
4640 Admiralty Way, Suite 231
Marina Del Rey, CA 90292, USA
dwile@teknowledge.com

Abstract

Software development is a constant endeavor to optimize qualities like performance and robustness while ensuring functional correctness. Architecture Description Languages (ADLs) form a foundation for modeling and analyzing functional and non-functional properties of software systems, but, short of programming, only the simulation of those models can ensure certain desired qualities and functionalities.

This paper presents an adaptation to statechart simulation, as pioneered by David Harel. This extension supports architectural dynamism – the creation, replacement, and destruction of components. We distinguish between design-time dynamism, where system dynamics are statically proscribed (e.g., creation of a predefined component class in response to a trigger), and run-time dynamism, where the system is modified while it is running (e.g., replacement of a faulty component without shutting down the system). Our enhanced simulation language, with over 100 commands, is tool-supported.

1. Introduction

Simulation is in common use in high-risk environments where a system's failure to function correctly may result in loss of life or massive loss of money. Simulation allows the safe exploration of a proposed solution in an environment that shields from physical harm (e.g., combat simulation) and monetary harm (e.g., investment banking) [13]. Simulation, done during the architecture and design stage, is also a low cost alternative to the actual implementation and execution of a real system.

Simulation languages have a long history in engineering; however, they are little-used in software engineering [4,13]. This is most unfortunate since "simulation can be applied in many critical areas and enable one to address issues before these issues become problems" [4]. It has become common knowledge that the early identification and resolution of potential problems can significantly reduce development time and cost, and at the same time increase the quality of the overall software product. [2]

The emergence of architecture description languages (ADLs) and design languages like the Unified Modeling Language (UML) [3] supply software architects with a new range of tools to design and test the software systems they are building. The static nature of many of those models, however, limits the role they play in the software life cycle. Although static analyses of those models result in useful insights and guarantees, for some purposes static models are simply inadequate. Architectural models, however, have the potential to attain a much more central role as participants in the software life cycle when augmented with a dynamic modeling facility; i.e. a facility for simulating certain aspects of system behavior. The architecture descriptions themselves may even be "reflected" in the running system!

Our interest is not to provide a full programming language in which to describe the functionality of systems, but rather to provide just enough structure to describe the observable effects of system activities in terms of the inputs driving their behavior. Of particular interest is the dynamic reconfiguration of the system architecture.

Fortunately, some architecture description languages have growing support for simulation. For instance, *Darwin/LTSA* [10,11] provides a simulator for executing labeled transition graphs; *Rapide* [9] can simulate events for pattern-analysis purposes; Rhapsody [7] or Statemate [8] can be used to simulate statechart models.

We adopted Statecharts [6] as our primary language for modeling behavior. Statecharts are not only used widely in components of the UML, but they are supported by a wide range of design tools. Statechart models can be used to depict the life-cycle of software components. In a graphical form they describe how components can be created, what kind of life stages (states) they can go through, and when they can be terminated. The graphical part of statechart model is re-enforced by a textual part that describes events and conditions that cause state changes as well as activities that can be performed in response to those state changes.

[1] This work was supported in part by DARPA contracts F30602-99-1-0524, F30602-00-C-0200, and F30602-00-C-0218

Our intention is to leverage people's familiarity with Statecharts and use them as an adjunct to an ADL model to simulate the behavior of component-based systems, where each component is modeled via a set of Statechart diagrams. Hence, the interactions of the components can be simulated via the interactions of their respective Statechart diagrams. To model aspects of architectural dynamism, Statecharts' extension, Rhapsody currently provides for describing simple changes components may undergo, viz. they may be created or destroyed. We call the use of Statecharts for reasoning at this level, *design-time dynamism* (e.g. understanding that component A may create zero to many components of type B over time)

Our model augments Statecharts with a more detailed language for action description and a dynamic simulation tool that can incorporate such models *after the system has already done some simulation*. Such augmentations are useful in order to model more fully activities such as upgrading components or dynamic replacement with a new version of a component, where the simulated behavior is not necessarily consistent with the previously observed behavior. Currently Statecharts can be used to simulate simple architectural changes, but not such changes in the behavioral descriptions of the components. We call these modeling aspects *run-time dynamism* because such dynamism occurs outside the scope of the design; for example, a new version of component A that was not available during design time needs to replace the old component without shutting down the system. Summarizing, modeling component-based software systems requires the modeling of design-time and run-time *component dynamism*.

Component dynamism is very important since it provides a means for dealing with the instabilities and fluctuations of today's software market, where it is often financially impossible to shut down software systems to perform upgrades or repair defects. In case of software in mission critical systems (e.g., supporting space exploration, combat, and communications) it is unreasonable to simply "stop" to make changes. This need requires software components to be adaptable to dynamic changes; even unforeseen ones. This need also requires new simulators to test the fitness of those software components to handle these dynamic changes.

To simulate statechart models, a number of very powerful commercial statechart simulators are available, Matlab [12], Statemate [8], Rhapsody [7], to name a few. We have found that those simulators are ill-equipped to handle the new challenges imposed by component dynamism. For instance, Statemate and Matlab require precise knowledge of what components exist during design-time leaving no room for component dynamism. Even tools like Rhapsody, having some support for design-time dynamism via object diagrams [3], is of little use in addressing most forms of run-time dynamism and some forms of design-time dynamism. Also, Rhapsody's dependence on object diagrams makes it less suitable for its integration with architectural description languages in general. See Related Work for a more detailed discussion of these and other tools.

In the following, we will present an adaptation to Harel's Statechart language to support component dynamism. We do this without restricting state charts to a particular modeling style or process. Our extensions are self-contained and embedded in the graphical and textual representation of Statechart models (states, transitions, guards, actions); they primary enrich the language of how guards, triggers, and actions are expressed but leave the basic notation of Statecharts intact. We refer to our language as the Statecharts for Dynamic Systems Language (SDSL). The SDSL is fully tool-supported providing software architects with a rich language of over 100 commands to model component dynamism.

2. New Television Example

In this paper, we will show how our simulator can be used to imitate various levels of component dynamism. To this end, we will make use of a so-called *New Television* example (NTV) to illustrate our work. NTV is a virtual television for PCs. This (hypothetical) NTV system has three major software component types: *client* components, *server* components, and *streamer* components. *Clients* can be downloaded over the web and started up in the users' workstations. The clients (also "*NTVClient*") have an interface for selecting available movies, which, upon the discretion of the user, will request the server to stream that movie to the client. Interacting with the client is a central *server* component. The server waits for movie requests from clients and, upon receipt of such requests, instantiates streamer components to handle the streaming of movies to their respective clients. Each client gets exactly one streamer for every movie it requests and no streamer is used for more than one client. Figure 1 represents the logical architecture of the NTV example, where client and streamer components are dynamically created, i.e. the Server creating the Streamers.

The NTV example uses design-time and run-time dynamism. During design-time it is defined that clients

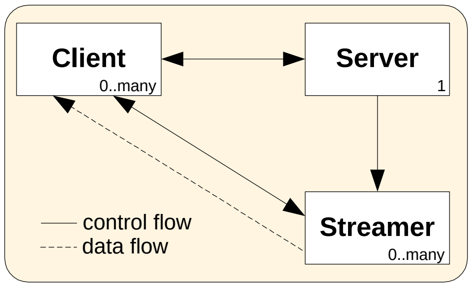


Figure 1. Logical Architecture of the NTV Example

may be instantiated at any time by any number of users. There is only one server, however, it must instantiate streamer components in response to user selections (simulating component construction and destruction). The example will also show that run-time dynamism is needed to upgrade and improve the NTV system, activities that were only partially foreseen during design time. For instance, we will postulate that the streamer component has a flaw, which can be mended through a simple fix. We will then simulate how the new version of that streamer (with the fix) can be instantiated by the server without the need to shut down the server (simulating component independence). The example will also demonstrate the ability to deal with a flaw in the server; this will require simulating the creation of a new version of the server to replace the old one without affecting currently running clients and their streamers (i.e., simulating late binding of clients to the server). We will show how simulated clients cope with the temporary lack of a simulated server to communicate with (i.e., simulation of replacement) and how simulated clients can locate the new simulated server although it is a different software component (i.e., simulation of component localization).

Although all these forms of dynamism should be able to be tested before the actual system is built, such simulations can be useful for reasoning about previously fielded systems as well. Our simulation language and its tool support provide software architects with the capability to reason about such concerns throughout the system's lifetime.

3. Dynamism in SDSL

The NTV example uses a variety of component dynamism concepts. The simplest one is simulating *component creation and destruction*. For instance, the server creates a new streamer every time a client makes a movie request. Likewise, the streamer is destroyed after the client is finished or a timeout occurs.

Simulating *component localization* is another aspect of component dynamism where one component may know about the location of a component but not have a handle on it (reference). Unique ids (e.g. GUIDs in COM, URLs on the Web) may be used to locate such components. For example, clients in our NTV example use the unique name "ntv.com" to locate the server component before making requests.

Simulating *component independence* imitates, in state charts, the unawareness of components of the existence of neighboring components as is normal in ADLs. The NTV example uses "late binding" to enable component independency. For instance, the client component searchers for the current instance of the server before making requests.

Simulating *component communication* is of particular interest for component dynamism. Since components tend to be independent, asynchronous communication methods like trigger calls need to be supported. Using triggers to communicate between state machines was already part of Harel's definition (a state machine is an simulating statechart). However, in order to support component independence, the concept of triggers had to be extended.

Simulating *component replacement* is among the most useful features of component dynamism. Due to the independence of components, one or more components may be destroyed or instantiated at any time. Naturally, such actions may have undesirable side effects on the entirety of a software system; hence, it is necessary to simulate before doing it. The NTV example will demonstrate two different types of run-time component replacement where first a faulty streamer is upgraded and second the server itself is upgraded. The replacement of the streamer component will show that only new client requests will benefit from the new steamer. Currently running streamers will not be replaced. The replacement of the server, however, replaces a currently running component. Simulating the effects of this replacement on already running clients is therefore important in evaluating the fitness of the clients and the system.

4. Using the SDSL Language

This section introduces the Statecharts for Dynamic Systems Language (SDSL) in the context of the NTV example. Special considerations will be given to those parts of the SDSL that are dynamism-specific. The SDSL, has over 100 keywords and symbols and supports an equal number of commands. In order to reduce the effort required to learn the SDSL, we modeled our language after Harel's initial Statechart definition and also adopted OCL expression language constructs [15] wherever possible (e.g., collection types and access methods).

Harel's Statechart definitions describe a set of (partially) independent states. Each state is either composite or simple; in the diagrams, composite states are named in the upper left-hand corner of rounded rectangles containing their sub-states. Additionally each composite state has a start state, indicated as a dark dot in the diagrams and an end state, indicated with a circle.

With each state *entry*, *exit*, and *during* actions may be associated. We have extended Harel's language for defining these actions by incorporating OCL constructs to allow variable declaration and assignment, as discussed within the example state descriptions below. Harel further defined that with each transition an *event* that triggers the transition may be specified, a *guard* expression that either filters the triggered events or when used on its own, is continuously evaluated and acts as a trigger itself when it

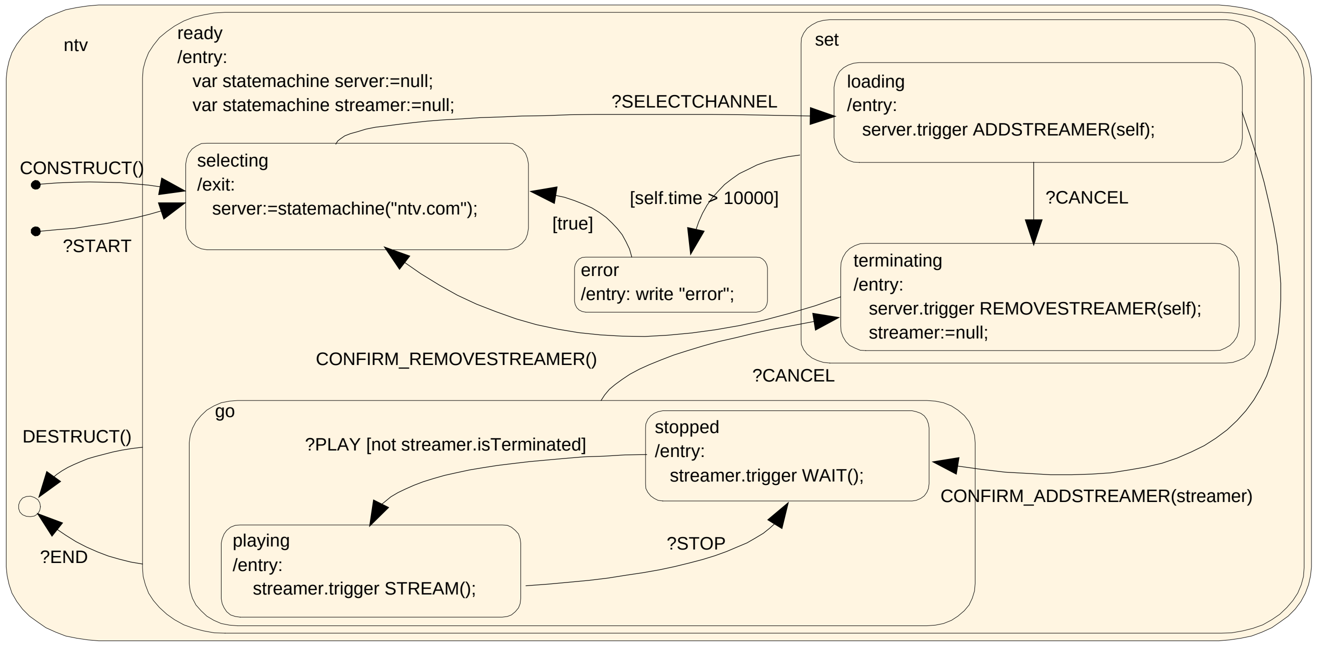


Figure 2. Statechart Model for Client Component. Uses component communication (e.g., “server.trigger ADDSTREAMER(self)”) and component localization (“statemachine(“ntv.com”)”)

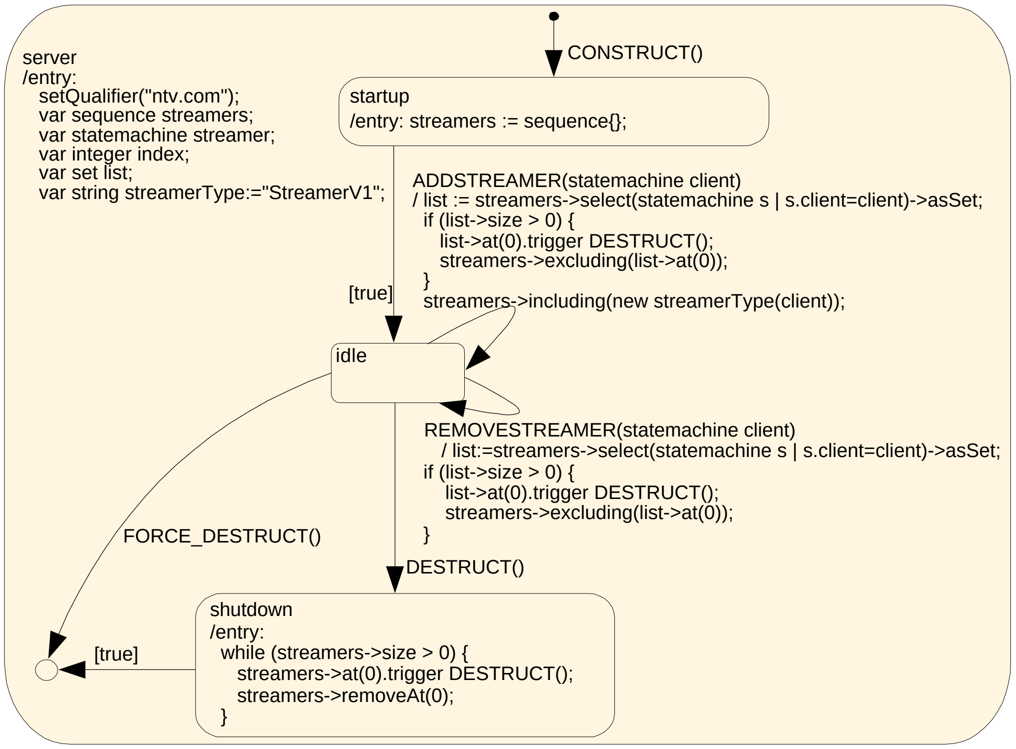
becomes true. An action may be associated with the transition as well. Triggered events are named and can be parameterized. They are explicitly triggered in actions referring to the statemachine in which the trigger should be applied, viz. *statemachine***.trigger** *event(parameters)*. Asynchronous events arising at the volition of the user are preceded by a question mark.

Ideally, the state machine execution model would mimic real-world behavior as manifested in synchronization mechanisms like Corba and DCOM, but mimic it in a reproducible way! Presently, the execution model is to sequentially consider each state machine in turn, determining whether any transition can be made. Each of its queued events is considered before any guards are evaluated. Execution of a transition entails first, the execution of any exit action from the state being left, then execution of any action associated with the transition, and finally, execution of the entry action of the state transitioned into. This sequence is all done as an atomic activity during which no other transitions occur. Moreover, any events the actions signal are queued on the relevant state machines' queues. However, the actions' effects on variables are immediate during execution (unlike in Harel's model). In the future we intend to replace this "round robin" form of synchronization of the machines with something more realistic.

4.1. The Client

Upon construction (CONSTRUCT), the client is in the simple state *selecting* of the composite state *ready* of *ntv* (Figure 2). Notice that the local variables *server* and *streamer* are defined and initialized. Once the user chooses to select a movie (?SELECTCHANNEL), the client locates the server while still in the *selecting* state via its unique name, “ntv.com.” Thereafter (after entry into the *loading* state), the client sends the event ADDSTREAMER to the server and passes (communicates) a reference to itself as a parameter. Once the client is in the *loading* state, three scenarios may happen: (1) the client receives the response event CONFIRM_ADDSTREAMER from the server side, (2) the user chooses to cancel the current selection (?CANCEL), or (3) a time out occurs after 10 seconds. If the server responds in time (option 1), the server also passes along a handle to the streamer it created. At this point the client is ready to receive a streaming movie. The user events ?PLAY and ?STOP may be used to start and stop the movie resulting is the corresponding actions STREAM() and WAIT() to be sent to the streamer component. The user may also cancel the streaming (option 2) of the movie at any time (?CANCEL) resulting in a REMOVESTREAMER(self) event to the server. The client then waits for a response from the server (CONFIRM_REMOVESTREAMER()) or, if none occurs, the termination will time out ([self.time > 10000][2]).

[2] self refers to the current state machine and self.time refers to the time elapsed since that state machine last transitioned.



```
server
/entry:
   setQualifier("ntv.com");
   var sequence streamers;
   var statemachine streamer;
   var integer index;
   var set list;
   var string streamerType:="StreamerV1";
```

```
startup
/entry: streamers := sequence{};
```

```
ADDSTREAMER(statemachine client)
/ list := streamers->select(statemachine s | s.client=client)->asSet;
  if (list->size > 0) {
     list->at(0).trigger DESTRUCT();
     streamers->excluding(list->at(0));
  }
  streamers->including(new streamerType(client));
```

```
REMOVESTREAMER(statemachine client)
   / list:=streamers->select(statemachine s | s.client=client)->asSet;
if (list->size > 0) {
    list->at(0).trigger DESTRUCT();
    streamers->excluding(list->at(0));
}
```

```
shutdown
/entry:
  while (streamers->size > 0) {
     streamers->at(0).trigger DESTRUCT();
     streamers->removeAt(0);
  }
```

Figure 3. Statechart Model for Server Component. Uses component construction and destruction (e.g., “new streamerType(terminal)”) and collection variable to handle large sets of potentially unknown types of depending state machines (e.g., “var sequence streamers”).

4.2. The Server

The server component (Figure 3) consists of only three states. Upon construction, the server defines a set of variables, initializes the *streamers* variable (of type sequence) to an empty sequence (set, bag and sequence are the most basic collection types supported by our language), and gives the server the unique name “ntv.com.” The [true] transition indicates that the server will automatically transition to the *idle* state once all initializations have been completed. In the *idle* state, the server waits for either one of two events. If it receives an ADDSTREAMER event, a series of statements are executed. The first statement searches for already existing streamer components for that client. If one is found (second statement), the existing streamer is destroyed[3] (DESTRUCT) and removed from the collection of streamers. Finally, the server creates a new streamer component of type “StreamerV1.” If the server receives a REMOVESTREAMER event, a similar set of statements is executed to destroy the current streamer supporting the given client. Upon destruction of the server, all streamers are destroyed. In order to enable a shutdown of the server component without interrupting current streaming components (e.g., for maintenance), the FORCE_DESTRUCT event is provided.

Note that the type name of the streamer is variable, viz. the value of *streamerType*. This makes it possible to support dynamic run-time upgrades of streamers where it is unknown during design-time what that streamer’s name will be. Variable component type names are another means for simulating late binding. Also note that the server uses the *setQualifier* command to define a unique name for itself. This command simulates the registration of a component in a component database (i.e., much like what COM or CORBA components do). If the *setQualifier* command is not used, an automatic ID will be generated.

The server also makes use of collection types. For instance, the variable *streamers* is defined as a collection of type *sequence*. A sequence is essentially an array where items like integers, strings, and even state machines can be stored. Collection types support design-time dynamism since they allow variable numbers of components to be defined during runtime. As such, at any given time the server may support zero, one, or many streamers. Collection types can be used to keep track of groups of statemachines. Commands available to perform operations on statemachines include tests for validity, initialization and termination. Furthermore, a series of OCL expressions, like *select*, *forAll*, or *iterate* can be used to perform collection operations. For instance, the action associated with the ADDSTREAMER event uses the *select* command to find state machines in the *streamers* collection that have a given terminal variable equal to the client.

The server is also the first component to directly construct and destruct components. The command “new streamerType(terminal)” creates a new instance of component type “StreamerV1.” Note that here a variable contained the component type. If such late binding is not desired, the command “new “ServerV1”()” could be used.

[3] There can be at most one.

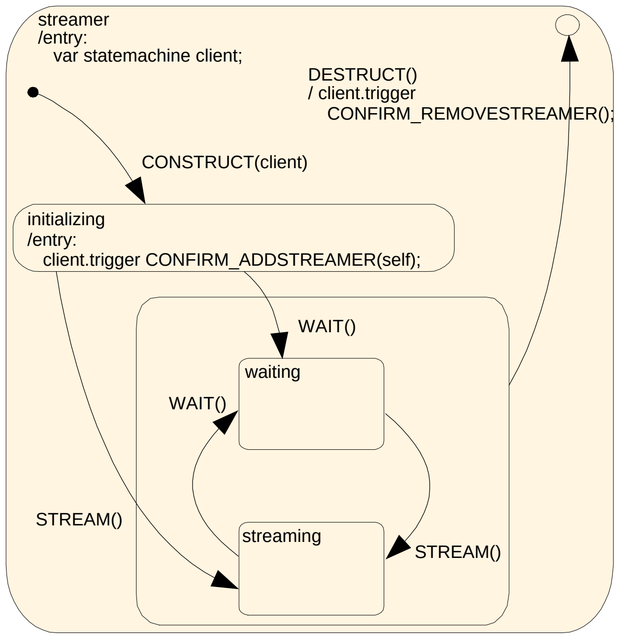


Figure 4. State chart for Streamer Component

As we shall see, this would be at the expense of flexibility should use of a new streamer type become desirable.

4.3. The Streamer

The server constructs a new streamer component (Figure 4) for every client that requests one. The CONSTRUCT event[4] passes the handle of the client, which gets stored in the variable *client* in the streamer component. In the *initializing* state, the streamer then sends the notification event CONFIRM_ADDSTREAMER(self) to the client with a handle of itself as a parameter. The streamer then uses the WAIT() and STREAM() events to send movie data to its client upon request. During destruction, the streamer sends a CONFIRM_REMOVESTREAMER notification to the client.

5. Run-Time Dynamism in SDSL

Thus, far we have primarily demonstrated how our extensions to the statechart language can handle design-time dynamism – that is the kinds of dynamism that are predicted during design time to occur while the system is running. This section discusses how more advanced run-time dynamism can be simulated via our language. Again, these entail changing the state machines under which the components operate at run time. We will demonstrate how changes to the running system can be simulated, in particular, the replacement of a faulty server component without shutting down the entire system. For the following scenarios we assume that the NTV system is operational with one server, many clients, and many streamers running.

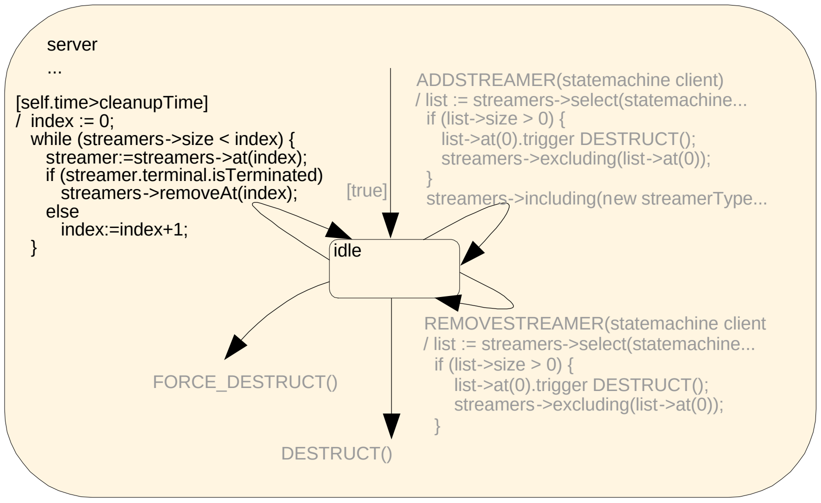
5.1 Introducing a New Streamer Version

Imagine that during operation, the designers discover a flaw that was unforeseen at design-time. When the client sends the ADDSTREAMER event to the server, the server creates a streamer and that streamer, in turn, sends a CONFIRM_ADDSTREAMER notification event back to the client. Under normal circumstances, the server can do this within a ten second period; however, in rare cases it may happen that the client times out ([time>10000] causing an error message to be displayed on the client side, followed by the termination of the selection. This may cause problems if the streamer is not aware of the timeout of the client and continues to wait for a client response. For the streamer, this constitutes a deadlock since the streamer itself never times out. A simple timeout fix is thus added to the streamer component, resulting in a new streamer version.

Fixing the timeout condition on the streamer is, however, only half the solution. The server still needs to use the new, changed version of the streamer. But since many clients are using the server, this re-configuration needs to be done during run-time without shutting down the server. One way to do this is to leave it variable on the server side what the streamer type name is. Instead of creating a pre-defined streamer, the server instead tries to locate a component definition with a given name (e.g., like COM names). Our simulator supports this via variable names for statemachines. This makes it possible to simulate how the running server, clients, and streamers react if a new streamer type is introduced. Since the server already supports a streamer type variable (as predicted during design time), updating that variable is simply accomplished by executing the command *statemachine(“ntv.com”).streamerType:=”StreamerV2.”* This command needs to be executed by the process responsible for the creation of the server. This will either be executed manually (e.g., via a server command interface supported by an equivalent command interface provided by our statechart simulator) or automatically by another component (e.g., a server user interface component). Our simulator supports both forms of access to the server.

The effect of this server re-configuration, which occurs during run time, is that new client requests will result in the server instantiating a streamer with the new

[4] Note that the construct event is implicitly sent to the new state machine with the “new” command. This guarantees that it is the first event that component receives.



```
[self.time>cleanupTime]
/  index := 0;
   while (streamers->size < index) {
      streamer:=streamers->at(index);
      if (streamer.terminal.isTerminated)
         streamers->removeAt(index);
      else
         index:=index+1;
   }
```

```
ADDSTREAMER(statemachine client)
/ list := streamers->select(statemachine...
  if (list->size > 0) {
     list->at(0).trigger DESTRUCT();
     streamers->excluding(list->at(0));
  }
  streamers->including(new streamerType...
```

```
REMOVESTREAMER(statemachine client
/ list := streamers->select(statemachine...
  if (list->size > 0) {
     list->at(0).trigger DESTRUCT();
     streamers->excluding(list->at(0));
  }
```

Figure 5. Statechart Model of New Server (upgrade) . Uses time condition to cause cleanup transition.

component type. It can then be simulated whether already running clients as well as newly created clients can handle the new streamer – a vital test before a similar streamer upgrade is performed on the real system with potentially disastrous consequences.

5.2 Replacing the Active Server

Now imagine that after the streamer fix, the designers discover another flaw that only becomes obvious after long execution times. Whenever the server adds a new streamer (after the ADDSTREAMER request), that streamer is also added to the collection variable *streamers*. Only the REMOVESTREAMER event causes that entry to be deleted from the *streamers* variable, causing problems whenever the client does not properly shut down and thus does not send a REMOVESTREAMER request. In itself this is not a problem because a new ADDSTREAMER request will automatically terminate an older running streamer for that same client. The designers, however, did not foresee that their client component would be downloaded in large numbers, resulting in users that only use it once or a few times. If a client then does not properly shut down, the server will keep a reference to its streamer indefinitely (although the streamer itself may have shut down because of our prior bug fix). This results in a gradual performance problem as time passes, causing periodic shutdowns of the server. The server therefore needs a cleanup feature that periodically removes old streamer entries. Figure 5 shows the necessary modifications required in black (grey items overlap with Figure 3). Again, we are faced with the challenge of making that change as the system is running. A potentially large number of people may be using the NTV service and it is unreasonable to assume that every client needs to be upgraded to recognize the new server. Our simulator can thus be used to simulate such a run-time swapping of components and the effect this may have on other running components. The way other clients are aware of the server component is via its unique name ("ntv.com"). That unique name is originally assigned automatically but was altered by the server state machine via the setQualifier command. To exchange the old server with the new server, we only need to issue a FORCE_DESTRUCT() command to the old server (e.g., 'statemachine("ntv.com").trigger FORCE_DESTRUCT()') followed by a startup command for the new server (e.g., 'new "NewServer"()'). The new server will register itself as the new "ntv.com" server, making it available to all currently running clients as well as old clients

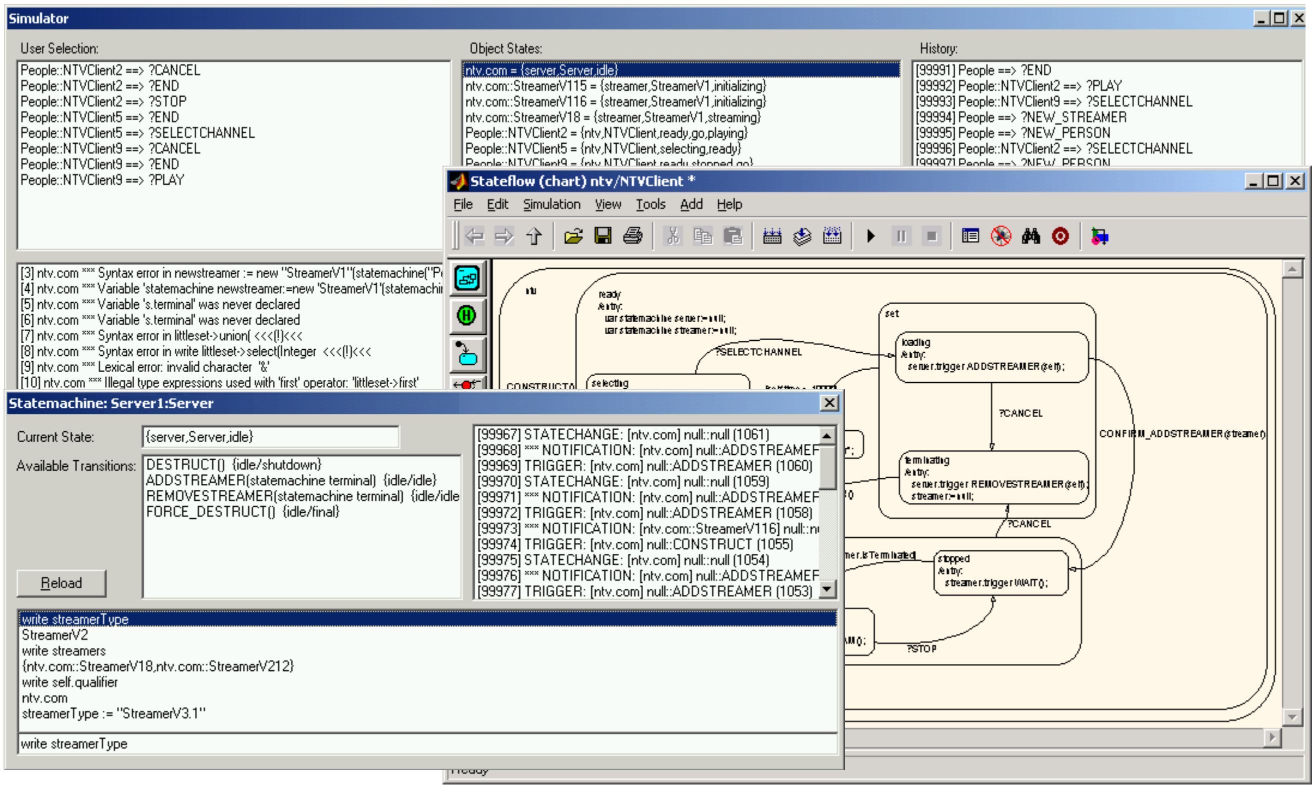
The simulated swapping of the server components can again be used to test how clients and streamers react with a brief absence of the server. For instance, prior to the "real" swapping of the servers, it is important to test whether running clients (in whatever states they may be) will continue to be able to interact with the server. Designers can also use our simulator to test whether the client timeout features work properly when the old server is down and before the new one is started up.

6. Tool Support

SDSL is supported with a simulation environment we call SDS (Simulator for Dynamic Statecharts). SDS (see Figure 6) provides for design and specification of state charts by integration with Stateflow from Mathworks and the PPDE from Teknowledge, which act as state chart drawing tools. The bottom, left pane of the figure illustrates a window onto Stateflow.

Our simulator extracts state chart descriptions from the respective drawing tools and gives the architect the option to decide which state charts to simulate (usually only a few in the beginning since state machines can instantiate needed components themselves). In the figure NTVClients 2, 5 and 9 have been created.

Our simulator provides a graphical user interface to running state machines. The graphical interface is primarily meant to support user interactions. The graphical interface thus becomes the simulated "user interface" for all running state machines (the source of events beginning with "?"). For instance, where the actual client application is expected to have a user interface that allows users to select movies and to start and stop those



```
People::NTVClient2 ==> ?CANCEL
People::NTVClient2 ==> ?END
People::NTVClient2 ==> ?STOP
People::NTVClient5 ==> ?END
People::NTVClient5 ==> ?SELECTCHANNEL
People::NTVClient9 ==> ?CANCEL
People::NTVClient9 ==> ?END
People::NTVClient9 ==> ?PLAY
```

```
ntv.com = {server,Server,idle}
ntv.com::StreamerV115 = {streamer,StreamerV1,initializing}
ntv.com::StreamerV116 = {streamer,StreamerV1,initializing}
ntv.com::StreamerV18 = {streamer,StreamerV1,streaming}
People::NTVClient2 = {ntv,NTVClient,ready,go,playing}
People::NTVClient5 = {ntv,NTVClient,selecting,ready}
People::NTVClient9 = {ntv,NTVClient,ready,stopped,go}
```

```
[99991] People ==> ?END
[99992] People::NTVClient2 ==> ?PLAY
[99993] People::NTVClient9 ==> ?SELECTCHANNEL
[99994] People ==> ?NEW_STREAMER
[99995] People ==> ?NEW_PERSON
[99996] People::NTVClient2 ==> ?SELECTCHANNEL
[99997] People ==> ?NEW_PERSON
```

```
[3] ntv.com *** Syntax error in newstreamer := new "StreamerV1"(statemachine("P
[4] ntv.com *** Variable 'statemachine newstreamer:=new 'StreamerV1'(statemachi
[5] ntv.com *** Variable 's.terminal' was never declared
[6] ntv.com *** Variable 's.terminal' was never declared
[7] ntv.com *** Syntax error in littleset->union( <<<(!)<<<
[8] ntv.com *** Syntax error in write littleset->select(Integer  <<<(!)<<<
[9] ntv.com *** Lexical error: invalid character  '&'
[10] ntv.com *** Illegal type expressions used with 'first' operator: 'littleset->first'
```

Current State: {server,Server,idle}

Available Transitions:

```
DESTRUCT()  {idle/shutdown}
ADDSTREAMER(statemachine terminal)  {idle/idle}
REMOVESTREAMER(statemachine terminal)  {idle/idle
FORCE_DESTRUCT()  {idle/final}
```

```
[99967] STATECHANGE: [ntv.com] null::null (1061)
[99968] *** NOTIFICATION: [ntv.com] null::ADDSTREAMEF
[99969] TRIGGER: [ntv.com] null::ADDSTREAMER (1060)
[99970] STATECHANGE: [ntv.com] null::null (1059)
[99971] *** NOTIFICATION: [ntv.com] null::ADDSTREAMEF
[99972] TRIGGER: [ntv.com] null::ADDSTREAMER (1058)
[99973] *** NOTIFICATION: [ntv.com::StreamerV116] null::n
[99974] TRIGGER: [ntv.com] null::CONSTRUCT (1055)
[99975] STATECHANGE: [ntv.com] null::null (1054)
[99976] *** NOTIFICATION: [ntv.com] null::ADDSTREAMEF
[99977] TRIGGER: [ntv.com] null::ADDSTREAMER (1053)
```

```
write streamerType
StreamerV2
write streamers
{ntv.com::StreamerV18,ntv.com::StreamerV212}
write self.qualifier
ntv.com
streamerType := "StreamerV3.1"
```

```
write streamerType
```

Figure 6. Screen Snapshots of SDS Tool showing an ongoing NTV Simulation, Matlab Stateflow for Drawing, and the Command Line Interface. The command line interface shows the history of recent commands that were entered manually; i.e., "write streamers" which returns the contents of the variable *streamers*.The command interface allows users to directly access the SDS interpreter to create, modify, or destruct artifacts.

movies, an interface to SDS allows the user to simulate these inputs (e.g., ?PLAY). These are entered through the "User Selection" window, the top, left window in Figure 6.

In addition to allowing users to cause events, our simulated user interface also displays the current states of all running state machines (top, middle window), their recent event history (top, right window), and error messages (middle, left window), should some illegal command have been encountered. The user interface also display messages that have been created as part of actions. For instance, our language supports the "write" command that may be used to display any text or variable contents. This information appears in a separate window (not displayed) or , if a state machine window is open, the message is displayed there.

Our simulator also provides a textual interface for running state machines. (See the bottom part of the bottom, right window in Figure 6.) A textual interface is provided for every state machine separately (the whole bottom, right window). It can be used for everything the graphical interface is used for and more. It provides direct and full access to the SDSL interpreter, allowing architects to create, modify, or delete anything they wish. Notice the history of activity in that window displayed as the largest bulk of text in the bottom, right window. These activities reflect commands that were manually entered by the user. Other state machine-specific information displayed in the textual interfaceare the current state, the available transitions from that state, and a running history of the state changes the machine has undergone. We found the textual interface to be invaluable for simulating "special cases of component dynamism," e.g., the sudden destruction of a simulated component (e.g., the server) to observe the impact this has on any other simulated component.

7. Related Work

The key idea that separates our work from other ADL and statechart simulation research is that we allow new models to be incorporated in a simulation *during the running of the simulation itself*. Nonetheless, we have certainly built on the ideas of others There are two major emphases of this research: architecture dynamism and statechart-based modeling.

In general we are interested in using other people's systems (COTS) wherever possible while avoiding over commitment to supporting formalisms. For these reasons we chose to adapt Matlab and Stateflow as our input mechanism for state charts. In addition, we tried to keep as much of the Statemate semantic framework as possible, while extracting the means for expressing simulations from the UML-based class diagrams required by Rhapsody, an extension to Statemate for expressing simulations.

We specifically wanted to avoid using knowledge about component interactions during design time because we feel it proscribes component dynamism too much. Indeed, Rhapsody incorporates design-time dynamism constructs. Its limitations for our purposes are that: it is limited to a single modeling language (object models) and it is not understandable and useable alone (one cannot understand the statechart model without also looking at the object model). Its integration with object models makes it a suitable candidate to model dynamism in the context of UML; however, architecture description language and many other design languages do not use object models.

Even in cases where ADLs have been successfully mapped to UML (e.g., C2 [14] to object model mapping) [1] this mapping also changed the meaning of those objects (that is a main reason why stereotypes were used). For instance, in C2 one component is not aware of any components next to it and thus cannot refer to it directly by name. An object model representing a C2 component model thus cannot make use of Rhapsody's statechart simulation capabilities..

Two ADLs that have stressed the ability to describe dynamism require some mention. First, the event-based model of Rapide [9] has been used to describe architectural components and the events they are exchanging. Its tool suite can then be used to analyze event patterns to identify potential problems (e.g., an invalid causality relationship).

Again, although we are unaware of any other efforts to provide run-time (model) dynamism, Rapide supports various forms of design-time dynamism, including the creation of components dynamically. In fact, the use of Rapide for dynamic modeling purposes is additionally hampered by its tight links to the rest of Rapide; this is much the same criticism as we have for Rhapsody as well.

A second ADL used to describe dynamic effects is Darwin [11]. The language is certainly of a kindred spirit in that it specifies what services are provided and what services are needed for each component. The language is unique for proscribing structural dynamism, by emphasis on lazy binding of (potentially unbounded) recursive structures and, as with both Rhapsody and Rapide, direct dynamic instantiation. Darwin is not event-based, and is incapable of modeling change to as fine a grain size as statecharts.

8. Future Work

Of course our criticisms of other work do have their flip side. It may be argued that both Rapide's and Darwin's models are more declarative than the state chart models we have adopted. This makes them more amenable to static analysis, which of course aids predictability when the system can be guaranteed to adhere to the implied design principles as it evolves. Our specifications are somewhat less general than a full programming language, but the ability to analyze them is certainly limited, especially with the potential for run-time dynamism and concurrency (the specifications are not *closed* in any sense, so there are few opportunities for preanalysis). We imagine a continuum of modeling technologies that will be useful in analyzing, simulating, and realizing systems on architecture description-based designs.

Our future work will in part involve just these concerns. We want to integrate with less dynamic languages for expressing the more static design constraints which we can rely on even during run-time changes. Hence, we want to integrate our model with a variety of concrete ADLs, by extension. We will start with Acme [5], of course, because of the potential leverage to other ADLs via interchange with it.

Along the same lines, there are analyses that could be done on the state chart specifications that extract what information we do require that a different implementation of a component must provide. Although we claim to espouse no particular object model, in fairness, there is a sense in which we build the part of the object model we need into the vars of the states, or more accurately, into other state charts' knowledge of these vars. In particular, we know which triggers it must be able to react to (including those from the user, sometimes); we know which of its state variables are relied on by other state machines. And we know what triggers it raises, although responsibility for raising these could conceivably be taken on by other component (models) after dynamic modification. All of these could be used to constrain future modification to the system.

Finally, there are situations where exactly the same abilities to change the model at run-time occur; we want to study what ways these contexts could affect the modeling constructs and tools we provide. We have already mentioned the simulation of in-place critical software systems such as space or combat missions. Another scenario involves systems that monitor their own health and status in order to determine whether to reallocate resources or raise warnings when situations change or deteriorate, respectively. Such systems are being studied in the DASADA program at DARPA. The use of such "reflective" capabilities to detect a system's deviations from its model – a simulation whose inputs are the real stimuli to which the system is reacting – is certain to modify our modeling concepts in some ways in the future.

9. Conclusions

Component-based development emphasizes the individual nature of components, the services they provide and the services they need. The increasing use of commercial off the shelf (COTS) components aggravate this. Modeling software components thus needs to take into account situations like the independence of software components, the uncertainty of the existence of other components around it and the ability of being replaceable by another component (or set of components). that provides the same services

We have presented a simulation language based of Harel's Statemate definition that supports a wide array of design-time and run-time dynamism concepts. Harel's statechart language has been extended to support component construction, destruction, localization, storing, accessing, and communication.

Current statechart modeling techniques are ill-equipped to deal with component dynamism in which the behavioral model changes. In fact, most currently available statechart simulators do not even support design-time dynamism, however, none is capable of supporting run-time dynamism in the fashion described above.

Acknowledgements

We wish to thank both Bob Balzer and Nenad Medvidovic for discussions of these ideas and tools.

References

[1] Abi-Antoun, M. and Medvidovic, N., "Enabling the Refinement of a Software Architecture into a Design," *Proceedings of the 2nd International Conference on the Unified Modeling Language (UML)*, Fort Collins, CO, Oct. 1999.

[2] B.W. Boehm. *Software Engineering Economics*, Prentice Hall, 1981.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*, Addison Wesley, 1999.

[4] Christie, A. M., Simulation: an Enabling Technology in Software Engineering *CrossTalk, The Journal of Defense Software Engineering*, vol. 12, Apr, 1999.

[5] D. Garlan, R. Monroe, and D. Wile. *Architectural Descriptions of Component-Based Systems, In Foundations of Component-Based Systems by Gary Leavens and Murali Sitaramam, eds.*, Kluwer, 2000.

[6] Harel, D., Statecharts: A Visual Formalism for Complex Systems *Science of Computer Programming*, vol. 8, 1987.

[7] Harel, D. and Gery, E., "Executable Object Modeling with Statecharts," *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, pp. 246-257, Mar. 1996.

[8] Harel, D., Lachover, H., Naamad, A., Pnuell, A., Poloti, M., Sherman, R., Shtull-Trauring, A., and Trakhtenbrot, M., STATEMATE: A Working Environment for the Development of Complex Reactive Systems *IEEE Transaction on Software Engineering*, vol. 16, Apr, 1990.

[9] Luckham, D. C. and J. Vera, J., An Event-Based Architecture Definition Language *IEEE Transactions on Software Engineering*, vol. Sep, 1995.

[10] Magee, J., "Behavioral Analysis of Software Architecture using LTSA," *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

[11] Magee, J. and Kramer, J., "Dynamic Structure in Software Architectures," *Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, San Francisco, CA, Oct. 1996.

[12] Matlab and Stateflow by Mathworks. http://www.mathworks.com.

[13] Sanders, Patricia. Study on the Effectiveness of Modeling and Simulation inthe Weapon System Acqusition Process. 96.

[14] Taylor, R. N., Medvidovic, N., Anderson, K. N., Whitehead, E. J. Jr., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L., A Component- and Message-Based Architectural Style for GUI Software *IEEE Transactions on Software Engineering*, vol. 22, pp. 390-406, 1996.

[15] J. Warmer and A. Kleppe. *The Object Constraint Language*, Reading, MA: Addison Wesley, 1999.