Systems approach to Software Engineering

Mrityunjay Kumar

ABSTRACT

We can analyze software products as a system. This note demonstrates this through two examples: one for a new system and another for an existing system. When we talk of system, we think of a set of components interacting to produce the desired behavior. We analyze such a system by modeling them as transition system. Our interest is in the dynamics of such a system because we believe behavior of a software product can be best studied through the study of the dynamics of the underlying system. The modeling language and the vocabulary introduced are intended to be very small and simple. We apply it to a Video Tab feature design to illustrate some key aspects. It is then applied to an existing system; Sleek (a todo list management app), is our running example to demonstrate the modeling approach. We also show multiple ways of modeling a system like Sleek. A specific sequence of steps is used when constructing models and we expect the students to use a similar sequence. Overall, this note is intended to familiarize you with this modeling technique to capture the dynamics of software systems, concluding sections provides multiple pointers and reading materials.

Keywords: Transition Systems, Software Modeling, Software Systems

1 TRANSITION SYSTEMS

A software system can be modeled as a Transition System. It is defined as a six tuple [1] $\{X, X^0, U, f, Y, h\}$ where

X is the set of states,

X^0 is the set of initial states,

U is the set of actions,

F is the transition relation which is a subset of $(X \times U) \times X$,

Y is the set of observables (or output space), and

h is a *display map*, mapping states to observables, $y = h(x)$.

The transition system models the dynamics of the system as depicted in Figure 1.

When there are more than a few states or actions, the transition function f is better represented as a table. T is the transition table corresponding to transition function f and t_{ij} represents the dynamics when action u_j is triggered in state x_i for the system - $f(x_i, u_j)$. We will use t_{ij} and f_{ij} interchangeably, $t_{ij} = f_{ij} = f(x_i, u_j)$.

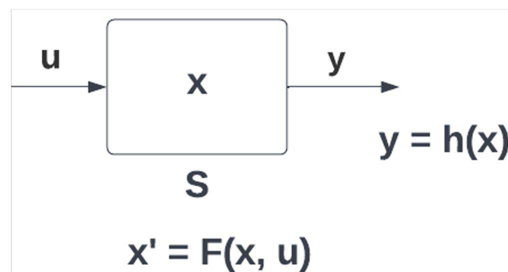

Figure 1. Transition System S (x: state, u: action, y: observable, h: display map, F: transition relation)

1.1 An Example

Let's say that we want to add a Like Icon for the product details page in an e-commerce site. When the icon is clicked, it goes from Liked to NotLiked, or vice versa.

1.1.1 Transition System Model

For the six tuple $\{X, X^0, U, f, Y, h\}$ for this system,

$X = \{Liked, NotLiked\}$,

$X^0 = \{NotLiked\}$,

$U = \{click\}$,

$Y = \{Redheart, Whiteheart\}$

Redheart and Whiteheart are the icons used to show Liked and NotLiked state respectively.

The transition function can be shown in a table, but given it is a simple system with 2 states and one event, we can describe it right here:

$f(Liked, Click) = NotLiked$,

$f(NotLiked, Click) = Liked$.

The display map h can be described easily too:

$h(Liked) = Redheart$,

$h(NotLiked) = Whiteheart$.

2 MODELING A NEW FEATURE: VIDEO PAGE

Modeling can be used as an intermediate step in building a system. In this case, we model the system using the specification provided. We will take the example of a web application where a feature needs to be added. E1 is a retailer whose flagship product is their ecommerce site from where they sell products. You are assigned a simple feature to build and will own its lifecycle going forward, which includes handling enhancements and defects reported by customers.

The product manager (PM) has written a feature specification. In many cases, a User Experience Design (UX) specification in the form of a clickable prototype or a wireframe is also created.

The goal of this feature is to allow users to see product videos and get more interested in buying the product. A new tab called 'Videos' should be available on the product details page. When user navigates to this tab, it should show a set of user videos available for this product. User can select a video to be played in the video playing area at the bottom of this tab. User should be able to stop a playing video and select any other video they like to play.

Model creation: We need to define the transition system six tuple $\{X, X^0, U, F, Y, h\}$.

2.1 Actions

A good place to start when modeling this using transition systems is to list down the actions, or inputs. In this case, there are three click actions apparent from the specification:

- *Load* - Click on the video tab (which will load the tab)
- $Select$ - Click on a stopped video (which will play the video)
- $Stop$ - Click on a playing video (which will stop the video)

Thus we can define $U = \{Load, Select, Stop\}$ to denote these three actions.

2.2 Observables

Observables, or outputs, are also apparent from the specification in most cases. We can list the observables:

- Video list page with no video playing or stopped
- Video list page with the selected video playing
- Video list page with the selected video stopped

Note that these observables align with the actions we identified above. For simple cases, this will be the case: every action changes the observable (which is how the user knows the action did something) so there is a one to one mapping. However, in a more sophisticated system, same action in different states can have different observables, and hence the number of observables can be the product of number of actions and number of states.

2.3 States

State lies at the center of a transition system, and usually the last to be specified completely in a model. Defining states is also an iterative process since different components of the transition system impact (and impacted by) it.

A state is a vector of information held by the system. While we can assign a name to the state, in most real cases, number of states can be too large to be named. In such cases, names help in defining transitions at a higher level (where the transition only manipulates the values of the information held in the state vector) and keep the description concise. This is an approach followed by Harel when describing Statecharts [2] and we use this notion in the next example.

In this example, we do start by assigning a name to the state, based on the behavior exhibited. Since actions change observables, and observables are derived from states, actions must change some state information and hence we can start with the assumption of as many states as there are actions.

We can model the system to have the following states:

- $Initial$ - No action has been performed on the product details page and the Video tab is not visible
- *LoadedTab* - The Video tab is loaded and a video can now be selected and played
- *PlayingVideo* A video has been selected and is playing

Note that each of them are states *before* the action is applied. For example, *Initial* is the state *before Load* is applied, *LoadedTab* is the state *before Select* is applied, and so on. We could have modeled them using an *after* semantic as well.

Technically, a *PlayingVideo* state when VideoId 5 is playing is a different state than when VideoId 10 is playing. However, since the broad behavior remains the same, we can define the behavior for *PlayingVideo* state and cover all related states, *PlayingVideo* acts like a superstate [2].

To support the observables above, the state needs to keep the following information:

- Video List (so that it can be shown)
- Selected Video identifier (so that currently selected video can be known)
- Video Player reference (so that selected video can be played)

In addition to this, state names will be kept. Given that this feature operates in the context of the particular product details page, it is a good idea to keep the current product identifier in the state as well. Given all this, we can surmise that the state includes the following information:

1. Product identifier $(Integer, i)$- Current product reference
2. State Name $(Enum, n)$ - $\{$Initial, LoadedTab, PlayingVideo$\}$
3. Video List $(VideoObject[], l)$- List of videos and their details
4. Selected video $(Integer, s)$- Reference to the currently selected video
5. Video Player reference $(PlayerRef, p)$ - Handle to the video player to render and play video

We also include (in square bracket) what kind of data type each of them is.

2.4 Initial States

Given that each state x is a five tuple (i,n,l,s,p), the initial state $x^0 = (-1, Initial, null, -1, null)$ denotes the case when the product details page is loaded and the Video tab is not clicked.

2.5 Display Map

The expected output (y) is a function (h) of state(x). As identified above, the observable needs to be able to show the videos, distinguish the selected video in some way, and show the player. Hence h should extract l, s, and p from the state.

So y is a three tuple (a, b, c) where $(a,b,c) = h(x), x = (i,n,l,s,p)$ and $a = l$, $b = s$, $c = p$

These values will be used appropriately to render the UX provided by the PM. The display map is defined in Table 1.

STATE	*Observable*
Initial	X
LoadedTab	Rendered Page
PlayingVideo	Selected Video Playing

Table 1. Video feature: Display map (X denotes undefined observable)

2.6 Transition relation

Using U, X and behavior description, we can define the transition relation F. The transition table that represents the relation F is shown in Table 2. In the table, F1, F2, and F3 are shorthand to refer to the logic applied by the transition function (given that the transitions are deterministic, we can treat it as a function) when the specific u and x are present: $F(Initial, Load) = F1$, $F(LoadedTab, Select) = F2$, $F(PlayingVideo, Select) = F2$, and $F(PlayingVideo, Stop) = F3$. $F1$, $F2$, and $F3$ are described below.

2.6.1 F1

When *Load* Action is applied to the State *Initial*, the system needs to initialize, which means state is set to the initial state.

State Update: $(i', n', l', s', p') = f(x), x = (i,n,l,s,p)$ where $i' = getProductId()$, $n' = LoadedTab$, $l' = getVideoList(i')$, $s' = -1$, $p = getVideoPlayerRef(i')$.

The product id is populated (i), state name is set *Loaded* (n), the list of videos for the product is fetched and added to state (l), selected video is set to -1 to denote no selection (s), a video player is initialized, and its reference is added to state (p).

	ACTION		
STATE	*Load*	*Select*	*Stop*
Initial	F1	X	X
LoadedTab	X	F2	X
PlayingVideo	X	F2	F3

Table 2. Video feature: Transition table (X denotes an undefined transition)

2.6.2 F2

When $Select$ Action is applied to the State $LoadedTab$ or $PlayingVideo$, the system needs to start playing the selected video and state variables get updated.

State Update: $(i', n', l', s', p') = f(x), x = (i, n, l, s, p)$ where $i' = i$, $n' = PlayingVideo$, $l' = l$, $s' = currentSelection()$, $p' = p$.

Other Actions: $playSelectedVideo(p', s')$ There is no change to product id, list of videos or player reference. Selected Video is updated to the new value (s') and state name is updated to $PlayingVideo$ (n'). Additionally, the newly selected video is played.

2.6.3 F3

When $Stop$ Action is applied to the State $PlayingVideo$, the system needs to stop playing the selected video and state variables get updated.

State Update: $(i', n', l', s', p') = f(x), x = (i, n, l, s, p)$ where $i' = i$, $n' = LoadedTab$, $l' = l$, $s' = s$, $p' = p$.

Other Actions: $stopVideo(p)$

There is no change to any state variable except state name which is updated to $LoadedTab$ (n'). Additionally, the currently selected video is stopped.

3 MODEL M1: MODELING AN EXISTING SYSTEM: TODO APP (SLEEK)

Now let's see how we can apply modeling technique to understand an existing system.

We take the example of a simple todo app called Sleek. Fig 2 shows the screen when there are some todos added. Fig 3 is when user wants to add a new todo, Fig 4 when they edit an existing todo, and Fig 5 shows the context menu for more operations on a todo.

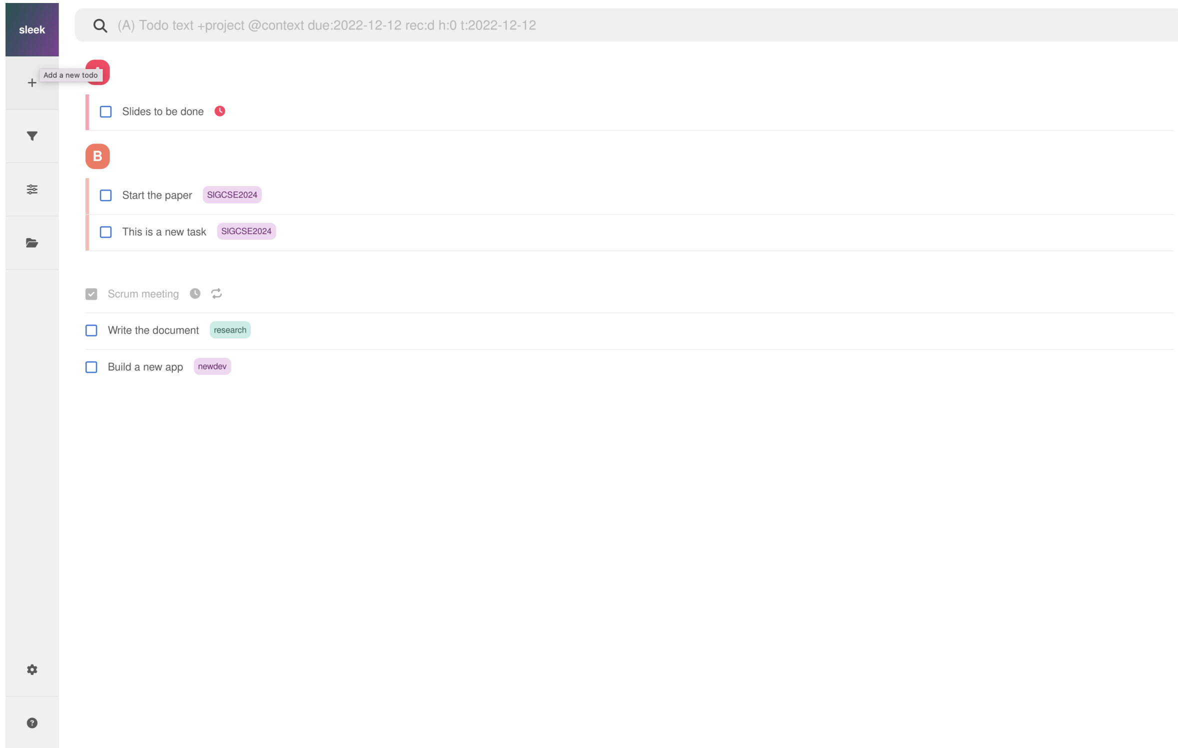

Figure 2. Sleek App: List View

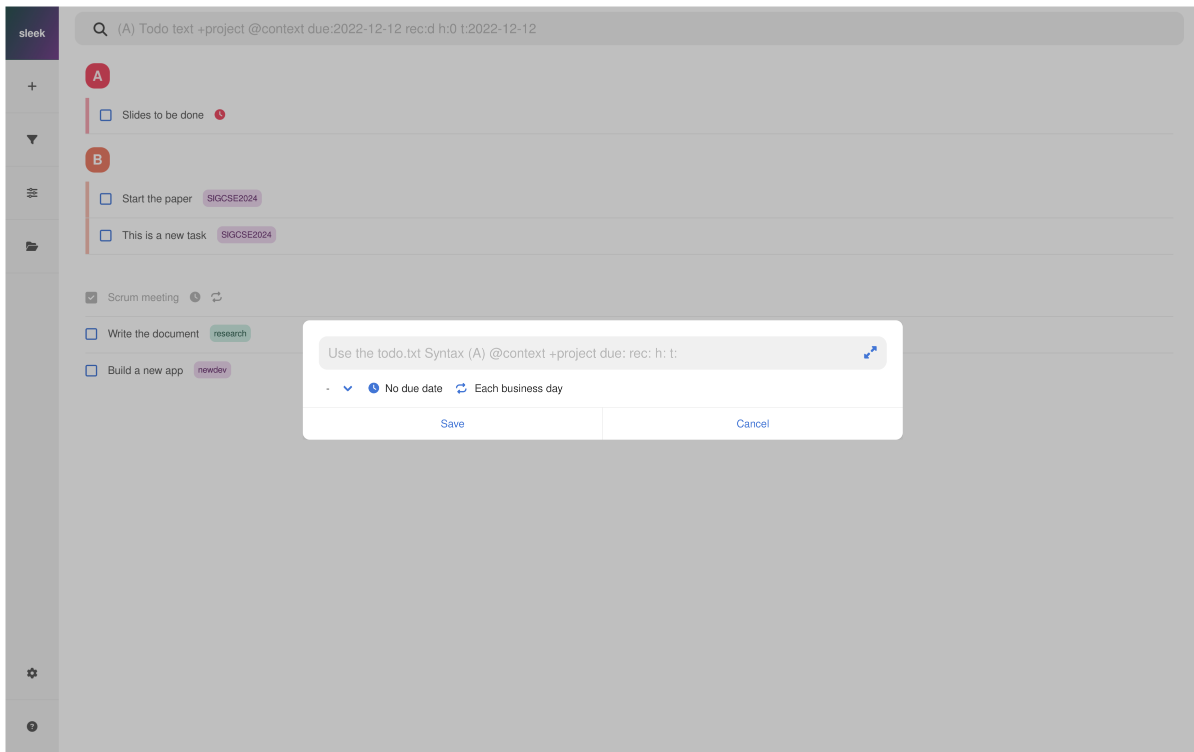

Figure 3. Sleek App: Add a Todo

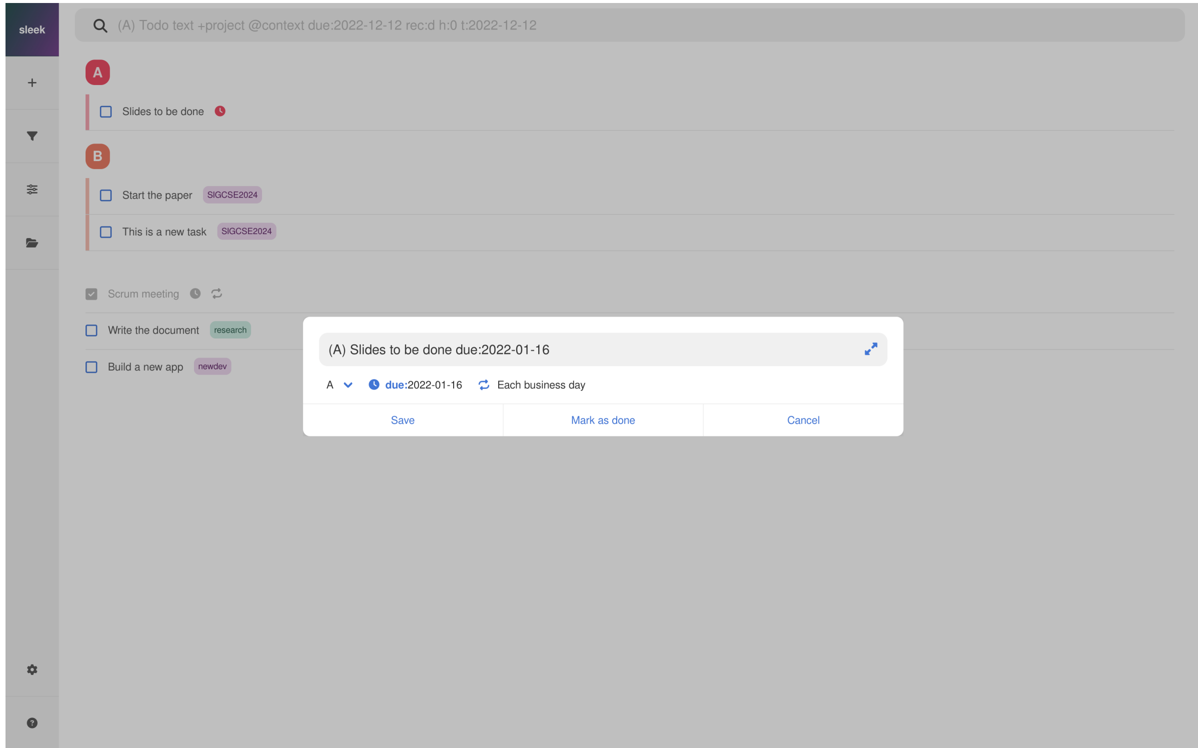

Figure 4. Sleek App: Edit a Todo

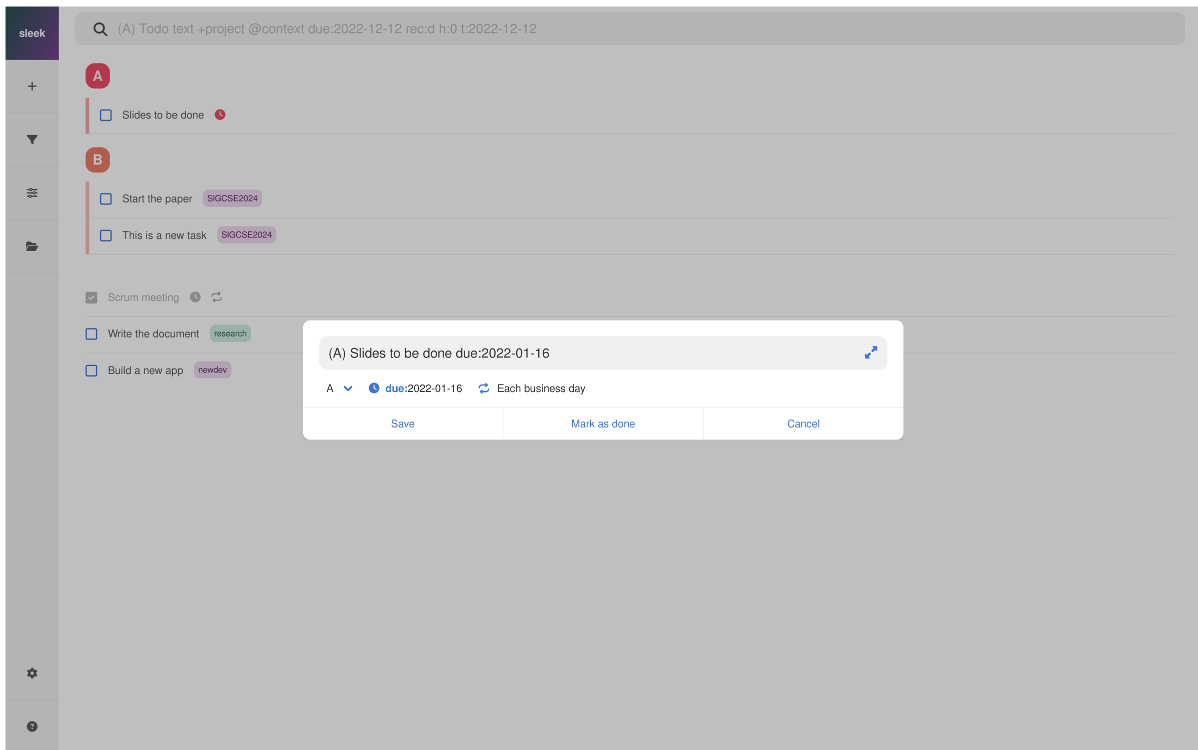

Figure 5. Sleek App: Context menu

We start by using the running system for a few key scenarios (add, delete and modify todos) and understand the key behaviors. A quick run through reveals the following:

- A new todo can be added by clicking on the blank textbox at the top and filling the details.
- A todo can have a priority (A-Z), a project name, a context and a due date.
- Clicking on a todo allows us to edit its attributes
- Todos are grouped by priority
- A todo can be marked done
- A todo can be deleted
- When a todo is past due date, clock icon against it is red, otherwise it is black. context and project are shown in specific colors as tags to the task.
- A todo can be used as a template to add a new todo

We can now proceed to describe the system by defining a transition system model for its behaviors. The approach will be similar to the case above; we review the specifications (which we write by analyzing the behavior) and start constructing the components of the system.

3.1 Actions

Reviewing various interactions performed above, we can identify the actions that change the observables (screen), these can be our starting list of actions. We also note that some actions take an existing todo (edit, mark as done, etc.) while other actions take some information that will be used to add or modify todos. Hence we define our actions to be consisting of three components:

- Action Name - name of the action being triggered
- Todo Id - Identifier for the todo that is being acted upon. This will be null for actions that do not need this id
- Todo Information - String with the details that will be used for a particular action (for ex, to create a new todo). This will be null for actions that do not need this string.

By following the actions that a user is allowed to do, we can identify the following actions (values in the brackets denote which values will be available and which will be null when the action is triggered):

- ClickPlus (null, null): Plus icon is clicked (which shows a popup window where user can add details)
- AddSave (null, todoinfo): Click on the Save button on the Add popup
- Done (todo-id, null): Checkbox on an existing todo is clicked to mark it as done
- ClickItem (todo-id, null): Click on an existing todo which opens the edit popup
- EditSave (todo-id,todoinfo): Click on the Save button on the Edit popup
- Delete(todo-id, null): Click on the delete option in the context menu
- RightClick(todo-id, null): Right click on an existing todo that opens the context menu
- UseAsTemplate (null, null): Click on the "Use as template" option in the context menu

$CommandName = \{ClickPlus, AddSave, Done, ClickItem, EditSave, Delete, RightClick, UseAsTemplate\}$ $CommandParam \subseteq (\mathbb{N} \cup \{-1\}) \times String$ to represent (todo-id, todoinfo).

Thus we can define $U = \{CommandName, CommandParam\}$.

3.2 Observables

Observables are clear from the screen as various actions are performed. Observable is primarily the screen. There are four key types of screen views, so we can think of observables as these screens:

- ListViewScreen (Fig 2): The screen that displays existing todos and their attributes
- AddViewScreen (Fig 3): The screen with add popup
- EditViewScreen (Fig 4): The screen with edit popup
- ActionViewScreen (Fig 5): The screen with context menu popup

$ScreenType \in \{ListViewScreen, AddViewScreen, EditViewScreen, ActionViewScreen\}$
$Y = ScreenType$

3.3 States

Discovering the states of an existing system is an iterative process. In some cases, you may be able to access design documents that provide insights into what the states are or can be modeled to be. However, in most cases, such documents are not available and you will need to hypothesize about it. When identifying states, we will look for states of the form ¡SuperState, SubState¿. For example, if we have two states S1 = ¡ListView, $\{1, 2, 3, 4\}$¿ and another state S2 = ¡ListView, $\{1, 2, 3, 4\}$¿, we say that S1 and S2 belong to SuperState ListView. Such a structure of states allow us to think in terms of broad behavioral changes across superstates, and small changes captured by changes in substate components of the same super state. This is an idea that statecharts [2] explores and uses, though in a slightly different way.

We will first look for super states for Sleek. In the case of Sleek, we can identify super states that are aligned with the observables we identified):

- *ListView* - 0 or more todos are available in the system
- *AddView* - System is ready to add a new todo
- *EditView* - System is ready to edit the specified todo
- *ActionView* - System is ready to apply specific actions (Delete, UseAsTemplate) on the selected todo through context menu

For substates, we look for the attributes required in a superstate to support the observables. To support the observables above, the state needs to keep the following information:

- Todo List (so that it can be shown), with the attributes for each list
- Currently selected todo
- Context List (so that they can be shown in the filter pane)
- Project List (so that they can be shown in the filter pane)
- Currently entered text in the textbox (used for Add and Edit actions)

In addition to this, state names will be kept as well as the name of the todo file in which todos are persisted. Thus, the state includes the following information:

1. Todo file name $(String, f)$
2. State Name $(Enum, n)$ - {ListView, AddView, EditView, ActionView}
3. Todo List $(Todo[], t)$, where Todo is a vector holding the attributes of a todo: $< todotext, priority, context, project, due-date >$
4. Currently selected todo $(Integer, s)$
5. Context List $(String[], c)$
6. Project List $(String[], p)$
7. Textbox input $(String, t)$

3.4 Initial States

Given that each state x is a seven tuple (f,n,t,s,c,p,i), the initial state $x^0 = (\text{"}todo.txt\text{"}, Initial, null, -1, null, null, \text{""})$ denotes the case when the app is opened with todo.txt (default name) used for todo file and no todos exist.

3.5 Display Map

The expected output (y) is a function (h) of state(x). As identified above, the observable needs to be able to show the list of todos, as well as the list of contexts and projects for filtering. It also needs to show the current todo as well as any entered text in textbox. Hence h should extract t, s, c, p and i from the state.

So y is a five tuple (y1, y2, y3, y4, y5) where $(y1,y2,y3,y4,y5) = h(x), x = (f,n,t,s,c,p,i)$ where $y1 = t$, $y2 = s$, $y3 = c$, $y4 = p$, $y5 = i$.

These values are used to create the appropriate UX. The display map is defined in Table 3. Render is assumed to be a method that can generate appropriate screen given the information from the current state. Render will use the values (the list of todos (and their attributes), currently selected todo, input text) to create the screen.

STATE	*Observable*
ListView	Render(ListViewScreen)
AddView	Render(AddViewScreen)
EditView	Render(EditViewScreen)
ActionView	Render(ActionViewScreen)

Table 3. Sleek: Display map

3.6 Transition relation

Using U, X and behavior description, we can define the transition relation f. The transition table that represents the relation f is shown in Table 4. The table only presents the transition at superstate level, substate transitions are handled through the functions F that encode the transition function logic appropriate to that particular transition.

For brevity, following have been ignored or simplified. Interested readers should extend the system and support some of these as well.

- ”View”, ”Show Filters” and Open todo.txt” options that open a separate left pane with configurations.
- ”Copy” command after Right-click
- Search bar at the top (which also includes a Add as Todo option)
- When there are no todos, the screen looks very different; but we assume the state to be same as that in which there are non-zero todos.

In the table, F1-F9 are shorthand to refer to the logic applied by the transition function (given that the transitions are deterministic, we can treat it as a function) when the specific u and x are present. Here is a brief description of their expected functionalities:

- F1: This is invoked when + is clicked in ListView state. The function doesn't have any further action to do, and the state transitions to AddView state.

	SuperState			
Action	ListView	AddView	EditView	ActionView
ClickPlus (null, null)	F1; AddView	X	X	X
AddSave (null, todoinfo)	X	F2; ListView	X	X
Done (todo-id, null)	F3; ListView	X	F4; ListView	X
ClickItem (todo-id, null)	F5; EditView	X	X	X
EditSave (todo-id,todoinfo)	X	X	F6; ListView	X
Delete(todo-id, null)	X	X	X	F7; ListView
RightClick(todo-id, null)	F8; ActionView	X	X	X
UseAsTemplate (null, null)	X	X	X	F9; AddView

Table 4. Transition Table for Sleek. X denotes no transition function.

- F2: This is invoked when the user has provided the details for a new Todo and clicks on Save. The function will create a new Todo using the information and add it to the list. If the new Todo has any new context or project, then those are added to the context list and project list respectively. The state then transitions to ListView state.
- F3: This is invoked when the user clicks on the checkbox of an existing Todo to mark it as Done. The function will update the Todo and mark it Done. The state stays in ListView state with the specific Todo marked as Done.
- F4: This is invoked when the user clicks on the ”Mark as done” button on Edit popup. The function marks the selected Todo as Done. The state transitions to ListView.
- F5: This is invoked when the user clicks on a Todo to edit it. The function doesn’t have any further action to do, and the state transitions to EditView state.
- F6: This is invoked when the user has updated the details of the Todo and clicks on Save to finalize the modification. The function updates the appropriate attributes of the Todo and persists the change in the system. The state transitions to ListView state.
- F7: This is invoked when the user selects ”Delete” option from the context menu. The function deletes the Todo and updates the list. If this was the last todo using a particular context or project, then those are removed from the context or project list respectively. The state then transitions to ListView state.
- F8: This is invoked when the user does a right click on an existing Todo to access the context menu. The function doesn’t have any further action to do, and the state transitions to ActionView state.
- F9: This is invoked when the user clicks on ”User as template” option in the context menu. The function fetches the information of the current Todo and use it to update the Textbox input value of the state (so that fields can be pre-populated for the user). The state transitions to AddView state.

Here is an example of how modeling can clarify behavior. If you focus on the EditView, you will notice that there is a way to mark the Todo as done from here (see F4 above). However, it is not evident, what happens if I modify some part of the todo and then click on Done. The way F4 has been written, it suggests that if Done is clicked, the original state of Todo is used and marked as done, the partially edited value of Todo is discarded. Is this how the system behaves today? Answer is yes. Should it behave this way? Losing user input is never a good user experience. Having model can trigger this conversation easily and earlier in the cycle.

4 MODEL M2 : MODELING SLEEK AS A WEB APPLICATION

If you are familiar with web (or UI) applications, you will notice that most of the actions and observables in the previous section essentially mapped to the UX of the system, and most of the descriptions of the transition function were simply data persistence and state transitions (though one reason for this is the simplicity of the application being analyzed).

When modeling a web application using transition systems, it is a good idea to think of it as (at least) two distinct, but connected, systems: a web system (W), and a backend system (B). W and B should be defined separately and their connections (which will be the relationship between their actions and observables) specified clearly. This also aligns well with how modern systems are built: the separation between W and B is very clear in all such systems.

In this section, we will proceed to show how such a 2-system modeling can be done for the previous example (Sleek). Such a separation is not necessarily only for a web application, it can be done for apps or any other UI based application where it makes sense to separate user interactions from system processing.

4.1 Separation of Responsibilities between the two systems

The first question that needs to be answered is this: what are the capabilities and behaviors implemented by each of these systems? If this is a new system, the modeler gets to chose the separation based on domain knowledge and best practices and patterns application in the context. if it is an existing system, this should be answered by the product architect and managers. This can be deciphered from the behavior somewhat, and such an exercise can help as a good exercise to understand the system.

The user performs following actions on Sleek:

- Add a new Todo
- Edit an existing Todo
- Delete an existing Todo
- Mark an existing Todo as done
- Add a new Todo using an existing Todo as template

We will assume the separation is done in the following way: *W is responsible for all the interactions user has with the system and B is responsible for the business logic that supports these interactions. This means that the actions and observables of the previous system become the actions and observables of W. Actions and observables of B are only used by W.*

4.2 System W

System W is the front end system that user interacts with. It supports all user interaction actions and what the user observes in response to the actions.

4.2.1 Actions U

Actions are same as those in previous model:

- ClickPlus (null, null): Plus icon is clicked (which shows a popup window where user can add details)
- AddSave (null, todoinfo): Click on the Save button on the Add popup
- Done (todo-id, null): Checkbox on an existing todo is clicked to mark it as done
- ClickItem (todo-id, null): Click on an existing todo which opens the edit popup
- EditSave (todo-id,todoinfo): Click on the Save button on the Edit popup
- Delete(todo-id, null): Click on the delete option in the context menu
- RightClick(todo-id, null): Right click on an existing todo that opens the context menu
- UseAsTemplate (null, null): Click on the ”Use as template” option in the context menu

$CommandName = \{ClickPlus, AddSave, Done, ClickItem, EditSave, Delete, RightClick, UseAsTemplate\}$ $CommandParam \subseteq (\mathbb{N} \cup \{-1\}) \times String$ to represent (todo-id, todoinfo).

Thus we can define $U = \{CommandName, CommandParam\}$.

4.2.2 Observables Y

Observables are same as those in previous model. In addition, since System W needs to communicate with System B for actions like Add, Edit, Delete and MarkDone, there are observables (not shown to the user) for this purpose.

- ListViewScreen: The screen that displays existing todos and their attributes
- AddViewScreen: The screen with add popup
- EditViewScreen: The screen with edit popup
- ActionViewScreen: The screen with context menu popup
- CommandName: {Add, Edit, Delete, MarkDone}
- CommandParam: $CommandParam \subseteq (\mathbb{N} \cup \{-1\}) \times String \times String$, for example, $c = (id, info, filename)$ where id is an index into the todolist, $info$ is a string denoting any info needed for the command, $filename$ is the todo file.

  $ScreenType \in \{ListViewScreen, AddViewScreen, EditViewScreen, ActionViewScreen\}$

  $Y \subseteq ScreenType \times CommandName \times CommandParam$

4.2.3 States X and Initial States X0

The super states defined for model M1 mapped well to the observables, so these states are relevant for System W as well, and we can continue to use them:

- $ListView$ - 0 or more todos are available in the system
- $AddView$ - System is ready to add a new todo
- $EditView$ - System is ready to edit the specified todo
- $ActionView$ - System is ready to apply specific actions (Delete, UseAsTemplate) on the selected todo through context menu

Model M1 has these attributes of the state defined:

- Todo file name $(String, f)$
- State Name $(Enum, n)$ - {ListView, AddView, EditView, ActionView}
- Todo List $(Todo[], t)$, where Todo is a vector holding the attributes of a todo: $< todotext, priority, context, project, due-date >$
- Currently selected todo $(Integer, s)$
- Context List $(String[], c)$
- Project List $(String[], p)$
- Textbox input $(String, t)$

We can remove the lists from the state since the are primarily required for processing which will be done by B. However, if we remove these, B needs to support actions which allow W to fetch these details from B as needed. This creates additional interactions with System B (causing performance issues later) which we can avoid by keeping these lists but use them as local copies of the lists that are owned and managed by B. This is a modeling choice if you are building a new system (and for existing system, you will check with a subject matter expert or read the code). For our example, we assume this model.

Creating dependency on B has an implication: there may be actions which require a request-response interaction with B. For example, when Save is called to add a new todo in AddView state, W needs to send a request to B to add it to the list, and wait for the response, before transitioning to ListView state. This means W will be in an internal state where it waits for the response. Such a waiting state is required for each state. To support this, we add two more attributes to the state - command name (updated when the request is made) and the response status (updated when the response arrives). So we use the following attributes for W:

- Todo file name $(String, f)$

- State Name $(Enum, n)$ - {ListView, AddView, EditView, ActionView}
- Todo List $(Todo[], t)$, where Todo is a vector holding the attributes of a todo: $< todotext, priority, context, project, due-date >$
- Currently selected todo $(Integer, s)$
- Context List $(String[], c)$
- Project List $(String[], p)$
- Textbox input $(String, t)$
- Command Name $(Enum, cn)$ - {Add,Delete,Edit,MarkDone}
- Command Status $(Enum, cs)$ - {InitiatedRequest, FailedRequest, FailedResponse, SucceededResponse}

$x = (f, n, t, , s, c, p, t, cn, cs)$ $x^0 = (null, null, null, , -1, null, null, null, null, null)$

4.2.4 Display Map h

Given that the state attributes haven't changed from Model M1, Display map too stays the same (Table 5)

STATE	*Observable*
ListView	Render(ListViewScreen)
AddView	Render(AddViewScreen)
EditView	Render(EditViewScreen)
ActionView	Render(ActionViewScreen)

Table 5. Sleek Model M2: Display map

4.2.5 Transition Function F

The transition table will be similar to Model 1 (Table 6), but with the following changes:

- We need to handle waiting state and internal action when response is received.
- Some of the logic (encoded in F) will be converted into command invokation to B.

Note that we have not shown four additional states we introduced: ListView_waiting, AddView_waiting, EditView_waiting, and ActionView_waiting. These states only respond to CommandResponse action that is an internal action (which we haven't shown either). No other states have any transition on CommandResponse action. Table 7 shows how this part of the transition table will look like.

F1-F9 will mostly be similar to what Model M1 define them to be. Only changes will be in terms of invoking the command instead of doing the processing directly.

- F1: This is invoked when + is clicked in ListView state. The function doesn't have any further action to do, and the state transitions to AddView state.

	SuperState			
Action	ListView	AddView	EditView	ActionView
ClickPlus (null, null)	F1; AddView	X	X	X
AddSave (null, todoinfo)	X	F2; AddView_waiting	X	X
Done (todo-id, null)	F3; ListView_waiting	X	F4; EditView_waiting	X
ClickItem (todo-id, null)	F5; ListView_waiting	X	X	X
EditSave (todo-id,todoinfo)	X	X	F6; ListView_waiting	X
Delete(todo-id, null)	X	X	X	F7; ActionView_waiting
RightClick(todo-id, null)	F8; ListView_waiting	X	X	X
UseAsTemplate (null, null)	X	X	X	F9; ActionView_waiting

Table 6. Transition Table for Sleek. X denotes no transition function.

	SuperState			
Action	Listview_waiting	AddView_waiting	EditView_waiting	ActionView_waiting
CommandResponse (Add)	X	ListView	X	X
CommandResponse (Delete)	X	X	X	ListView
CommandResponse (Edit)	X	X	ListView	X
CommandResponse (MarkDone)	ListView	X	X	X

Table 7. Wait States and Command Response Action

	SuperState			
Action	ListView	AddView	EditView	ActionView
ClickPlus (null, null)	F1; AddView	X	X	X
AddSave (null, todoinfo)	X	F2⟳; ListView	X	X
Done (todo-id, null)	F3⟳; ListView	X	F4⟳; ListView	X
ClickItem (todo-id, null)	F5⟳; EditView	X	X	X
EditSave (todo-id,todoinfo)	X	X	F6⟳; ListView	X
Delete(todo-id, null)	X	X	X	F7⟳; ListView
RightClick(todo-id, null)	F8⟳; ActionView	X	X	X
UseAsTemplate (null, null)	X	X	X	F9⟳; AddView

Table 8. Transition Table for Sleek using Async symbol. X denotes no transition function.

- F2: This is invoked when the user has provided the details for a new Todo and clicks on Save. The function will *initiate a request to system B to* create a new Todo using the information and add it to the list. The state does a transition to AddView_waiting state.
- F3: This is invoked when the user clicks on the checkbox of an existing Todo to mark it as Done. The function will *initiate a request to system B to* update the Todo and mark it Done. The state transitions to ListView_waiting state.
- F4: This is invoked when the user clicks on the ”Mark as done” button on Edit popup. The function will *initiate a request to system B to* update the Todo and mark it Done. The state transitions to EditView_waiting state.
- F5: This is invoked when the user clicks on a Todo to edit it. The function will initiate a request to system B to get the details of the current Todo, and the state transitions to ListView_waiting state.
- F6: This is invoked when the user has updated the details of the Todo and clicks on Save to finalize the modification. The function *initiates a request to system B to* update the appropriate attributes of the Todo and persists the change in the system. The state transitions to EditView_waiting state.
- F7: This is invoked when the user selects ”Delete” option from the context menu. The function *initiates a request to system B to* delete the Todo and update the list. The state then transitions to ListView_waiting state.
- F8: This is invoked when the user does a right click on an existing Todo to access the context menu. initiate a request to system B to get the details of the current Todo, and the state transitions to ListView_waiting state.
- F9: This is invoked when the user clicks on ”User as template” option in the context menu. The function initiate a request to system B to get the details of the current Todo, use it to update the Textbox input value of the state (so that fields can be pre-populated for the user). The state transitions to ActionView_waiting state.

We introduce a new method ProcessResponse which is invoked whenever CommandResponse action is triggered in any of the waiting states. When the response is ”SucceededResponse”, it refreshes the three lists (using the information from the response) and makes the transition to ListView.

Given that the handling of waiting states and CommandResponse action is so standardized, in the interest of brevity, modeler may choose to ignore the waiting states and instead just mark the transitions (in this example, the F1-F9) to be dependent on another system (which will be interpreted to mean such an asynchronous behavior and acts as a hint for the designer). We will use ⟳ as the symbol to denote such an async call in a transition table cell (see Table 8).

4.3 System B

Given the above separation of responsibilities, we can model System B easily. We made following assumptions about what the actions and observables of B:

- B will support Add, Edit, Delete, and MarkDone actions.
- Whenever B 'responds' (observable when there is a state corresponding to a command action), it includes ResponseStatus and 3 lists - Todo list, context list, and project list.

4.3.1 Actions

Primary actions are a set of commands with their parameters that are processed by B.

$U \subseteq CommandName \times CommandParam$,

$CommandName \in \{Add, Edit, Delete, MarkDone\}$ $CommandParam \subseteq (\mathbb{N} \cup \{-1\}) \times String \times String$, for ex, p = (id,info,filename) where id is an index into the todolist, $info$ is a string denoting any info needed for the command, $filename$ is the todo file.

4.3.2 Observables

As mentioned above, $Y = \{CommandName, CommandParam, CommandResult, TodoList, ContextList, ProjectList\}$:

$CommandName$ - Name of the command that caused this state transition

$CommandParam$ - The tuple that was provided with the command $(id, info, filename)$

$CommandResult$ - The result of the command, could be one of {InitiatedRequest, FailedRequest, FailedResponse, SucceededResponse}

$TodoList$ - The current Todo list after the command is executed

$ContextList$ - The current Context list after the command is executed

$ProjectList$ - The current Project list after the command is executed

4.3.3 States and Initial States

Whatever attributes are required by Y need to be in the state. All actions modify the lists we hold, there are no distinct superstates. So the state consists of following attributes:

- $CommandName$ - Name of the command that caused this state transition
- $CommandParam$ - The tuple that was provided with the command $(id, info, filename)$
- $CommandResult$ - The result of the command, could be one of {InitiatedRequest, FailedRequest, FailedResponse, SucceededResponse}
- $TodoList$ - The current Todo list after the command is executed
- $ContextList$ - The current Context list after the command is executed
- $ProjectList$ - The current Project list after the command is executed

$x = (n, p, r, tl, cl, pl)$ $x^0 = \{null, null, -1, null, null, null, null\}$

4.3.4 Display Map

Display Map is straightforward, it just uses extracts a subset of attributes from x. $y = h(x) = (n, p, r, tl, cl, pl)$

4.3.5 Transition Function

The transition function can be written like a function.

Given $u = (commandname, commandparam, filename), x = (f, n, p, r, tl, cl, pl)$,

$x' = F(x, u)$ where

$n' = commandname$

$p' = commandparam$

$r' = eval(commandname, commandparam)$

$tl' = getTodoList(commandparam.filename)$

$cl' = getContextList(commandparam.filename)$

$pl' = getProjectList(commandparam.filename)$

eval is a function that executes the given action (command) and updates the file with the updated todo values. getTodoList fetches the current state of Todo list. getContextList and getProjectList do the same for the other lists.

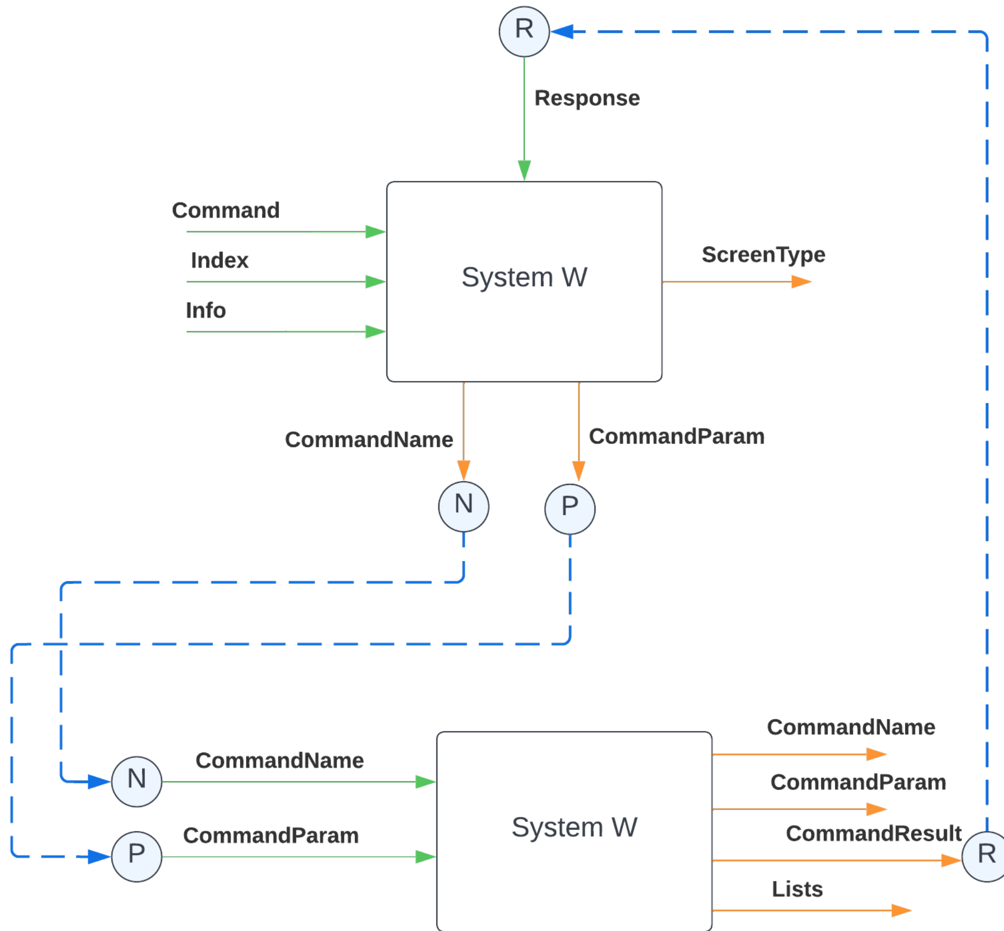

Figure 6. Composition of W and B

4.4 Composition of W and B

We can wire them based on the respective U and Y. Figure 6 shows the connections. We have simplified this by not showing a few connections like CommandName, CommandParam and Lists from B coming back to W. The idea is that these connections can be treated like electrical wiring in most cases of modeling.

5 CONCLUDING REMARKS ON MODELING USING TRANSITION SYSTEMS

As we saw above, a software can be modeled using Transition Systems. When you are trying to understand an existing system, you hypothesize about the behavior of the system, validate it by executing specific scenarios in the running system, and capturing your understanding in the form of various components of the six-tuple of the transition system. As M1 and M2 shows, there are multiple models possible for a system. When you build a new system, you get to choose them based on the tradeoffs that are usually non-functional in nature (scale, performance, availability, reliability, security, usability, etc.). When you are understanding or reverse engineering an existing system, you use partial information to reconstruct the model that captures your understanding. These models also help identify design and modeling enhancements that are possible in the system when there is time for it. A web application can be modeled along the lines of architectural patterns like Model-View-Controller.

The model can become very large very quickly. So it is important to keep the purpose in mind. When the goal is to understand the system, a formal model like this is intended to augment the mental model you hold for the system. As your mental model becomes strong, the need for the details in the formal model goes down.

The model is never done. You will always get new information about the system and you will review the model in light of new information and refine one or more components of the model. For example, when you discover that the system supports a

new scenario, you examine the state, transition function and display map to see which all need updates to accommodate this scenario in the model. You will start with a deficient, abstracted version of the model, and through multiple iterations, you will get a more precise and refined version. This iteration continues till you achieve sufficient mastery of the system. Given that software systems are always being enhanced, mastery is a moving target!

When a new enhancement is planned to an existing system, you should be able to use the existing model to figure out the changes required to support the specification of the enhancement. These are done at the model level, before proceeding to design or implement.

6 READING MATERIALS

- In a world of systems (10 min video animation)
- A philosophical look at system dynamics (Donnella Meadows, 53 minutes lecture video)
- Introduction to Transition Systems (additional notes)
- Composition of Systems (additional notes)

REFERENCES

[1] Venkatesh Choppella, Kasturi Viswanath, and Mrityunjay Kumar. Algodynamics: Algorithms as systems. In *2021 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. IEEE, 2021.

[2] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.