Developing Rich, Web-Based User Interfaces with the Statecharts Interpretation and Optimization Engine

Jacob Beard

Supervised by: Hans Vangheluwe

School of Computer Science
McGill University
Montreal, Quebec, Canada

March 23^{rd}, 2013

A thesis submitted to McGill University in partial
fulfilment of the requirements of the degree of
Master of Science in Computer Science

Abstract

Today's Web browsers provide a platform for the development of complex, richly interactive user interfaces. But, with this complexity come additional challenges for Web developers. The complicated behavioural relationships between user interface components are often stateful, and are difficult to describe, encode and maintain using conventional programming techniques with ECMAScript, the general-purpose scripting language embedded in all Web browsers. Furthermore, user interfaces must be performant, reacting to user input quickly; if the system responds too slowly, the result is a visible UI "lag," degrading the user's experience.

Statecharts, a visual modelling language created in the 1980's for developing safety-critical embedded systems, can provide solutions to these problems, as it is well-suited to describing the reactive, timed, state-based behaviour that often comprises the essential complexity of a Web user interface.

The contributions of this thesis are then twofold. First, in order to use Statecharts effectively, an interpreter is required to execute the Statecharts models. Therefore, the primary contribution of this thesis is the design, description, implementation and empirical evaluation of the Statecharts Interpretation and Optimization eNgine (SCION), a Statecharts interpreter implemented in ECMAScript that can run in all Web browsers, as well as other ECMAScript environments such as Node.js and Rhino.

This thesis first describes a syntax and semantics for SCION which aims to be maximally intuitive for Web developers. Next, test-driven development is used to verify the correct implementation of this semantics. Finally, SCION is optimized and rigorously evaluated to maximize performance and minimize memory usage when run in various Web browser environments.

While SCION began as a research project toward the completion of this master thesis, it has grown into an established open source software project, and is currently being used for real work in production environments by several organizations.

The secondary contribution of this thesis is the introduction of a new Statecharts-based design pattern called Stateful Command Syntax, which can be used to guide Web user interface development with Statecharts. This design pattern can be applied to a wide class of Web user interfaces, specifically those whose behaviour comprises a command syntax that varies depending on high-level application state. Its use is illustrated through detailed case studies of two highly interactive, but very different applications.

Abrégé

Les navigateurs web modernes permettent la création d'interfaces utilisateur complexes et extrêmement interactives. Mais cela engendre également des difficultés additionnelles pour les développeurs web. Les relations complexes qui existent entre les différentes composantes des interfaces utilisateur sont souvent à états, et sont difficile à décrire, encoder et entretenir en utilisant les techniques de programmation conventionnelles qu'offre ECMAScript, un langage banalisé que l'on retrouve dans tous les navigateurs. De plus, les interfaces doivent être performantes et répondre rapidement aux actions de l'utilisateur; un système trop lent engendre un décalage perceptible qui nuit à l'expérience d'utilisation.

Statecharts, un langage de modélisation visuelle créé dans les années 80 pour assurer la sécurité des systèmes embarqués, peut offrir des solutions à ces défis: il est adapté à la description du comportement réactif, chronométré, et fondé sur l'état, qui représente en somme ce qui est le plus difficile avec les interfaces utilisateur sur le web.

L'apport de ce mémoire est double. Tout d'abord, l'exécution des modèles Statecharts, pour être efficace, nécessite un interprète. La première contribution de cet article consiste donc en le design, l'élaboration, l'implémentation et l'évaluation empirique du Statecharts Interpretation and Optimization eNgine (SCION), un interprète en ECMAScript qui peut être exécuté dans tous les navigateurs web ainsi que dans d'autres environnements ECMAScript, comme Node.js et Rhino.

Notre thèse décrit en premier lieu une syntaxe et une sémantique pour SCION qui se veut la plus intuitive possible pour les développeurs web. De nombreux tests sont ensuite effectués pour en assurer le bon fonctionnement. Enfin, SCION est rigoureusement optimisé afin de maximiser la performance et de minimiser l'utilisation de la mémoire vive dans différents navigateurs web.

Bien que SCION ait débuté comme un projet de recherche personnel en vue de l'obtention du diplôme de matrise, il est devenu un logiciel libre bien établi et utilisé par plusieurs organisations dans leurs environnements de travail.

La deuxième contribution de ce mémoire est la présentation d'un nouveau patron de conception créé à partir de Statecharts nommé Stateful Command Syntax, qui peut être utilisé pour guider la création d'interfaces utilisateur sur le web. Ce patron de conception peut être employé sur une grande variété d'interfaces, notamment celles dont le comportement inclut une syntaxe de commande qui varie selon le degré d'abstraction de l'application.

Son utilisation est illustrée à travers quelques études de cas ainsi que deux logiciels très différents mais hautement interactifs.

Acknowledgements

I would like to thank my supervisor Professor Hans Vangheluwe for providing research opportunities and setting me down an interesting path.

I would also like to thank my parents for raising me, my girlfriend Tamar Swartz for her continued love and support, my employer, INFICON, for sponsoring the development of SCION, my supervisor at INFICON, Dr. Pierre Wellner, for reading my thesis and providing useful feedback, and my colleagues Xavier Peich, who translated my thesis abstract into French, Rohan Shiloh Shah, for introducing me to Computer Science in my freshman year, and Anton Dubrau, who gave me the push I needed to start writing this thesis.

Finally, I would like to thank Linus Torvalds and Richard Stallman for their contributions to the free and open source software movement. Without the GNU/Linux operating system, I would not be a fraction as productive as I am now.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There are many challenges that software developers face during the development of complex User Interfaces (UIs). Desired behaviour may be autonomous or reactive, and possibly real-time. Each UI component may be required to exhibit a radically different behaviour from that of any other component, and the behaviour of components may be interrelated. These complex behavioural relationships are often difficult to express, and are even more difficult to encode and maintain.

A solution may be found in Statecharts[Har87], a formalism for describing complex, reactive, timed, state-based behaviour, which is highly suited to modelling richly interactive user interfaces.

At the same time, Open Web technologies, including HTML, SVG and CSS, in combination with ECMAScript, are becoming increasingly popular as a platform for application development. As ECMAScript is used to implement interactivity and dynamic behaviour in Web pages, a Statecharts interpreter optimized for ECMAScript would facilitate user interface development for the World Wide Web by allowing Web developers to specify the interactive behaviour of complex Web UIs using high-level, declarative, executable Statecharts models. Therefore, the initial contribution of this thesis is the design, careful description, implementation and empirical evaluation of the Statecharts Interpretation and Optimization eNgine (SCION), a Statecharts interpreter library implemented in ECMAScript that can run in all Web browsers.

SCION is a tool that allows web developers to use Statecharts for Web user interface development, but it is not always clear how Statecharts may best be applied to this task. Therefore, the second contribution of this thesis is the description of a technique to guide the development of Web user interfaces with Statecharts. This technique takes the form of a new Statecharts-based design pattern called Stateful Command Syntax (SCS). SCS can be applied to user interfaces whose behaviour comprises a command syntax that varies depending on high-level application state. This design pattern is illustrated through two detailed case studies.

1.1 Background

1.1.1 State Machines for User Interface Development

Statecharts and other state machine formalisms have a long history of use in Human Computer Interaction research. The notion of UI command syntax can be found in [Bux86], [Wel89], and [WN68], and in the latter, it is proposed that state machines be used to parse UI commands. In [Sam02], the "Ultimate Hook" is described as a Graphical User Interface (GUI) design pattern, whereby events bubble from inner GUI components to outer GUI components, which is directly supported by the Statecharts syntax of hierarchical states.

State machines have also been used to directly synthesize working user interfaces, thus facilitating rapid prototyping of UI behaviour during development [ABL08, JB09].

1.1.2 Interactive Web-based User Interfaces

Scripting in the Web Browser Client

A *Web browser* is a hypermedia applications designed to allow navigation of hypertext documents, written in the Hypertext Markup Language (HTML), over the Internet[BL]. The first Web browser clients, including WWW, Mosaic, and the original Netscape Web browsers, did not support UI scripting on the client. This means that the behaviour of UI components in a Web browser could not be extended by code executed in the Web browser client.

The JavaScript language (later standardized as ECMAScript) was invented in 1995 and embedded in the Netscape browser[Net95]. EMCAScript, and the associated Document Object Model (DOM) API, allowed executable content to be added to a Web page, such that developers were able to attach event listeners to individual HTML elements, and react to UI events with arbitrary behaviour written in ECMAScript. This facilitated the creation of rich, client-side Web applications with custom behaviour.

Microsoft followed Netscape with their own implementation of JavaScript, which they called JScript and embedded in their Internet Explorer web browser. Today, all modern Web browsers support scripting through ECMAScript and DOM.

Scalable Vector Graphics and Canvas

In addition to HTML, modern Web browsers support two other technologies which may be used to render rich graphical user interfaces: *Scalable Vector Graphics* (*SVG*) and *HTML5 Canvas*.

SVG is a technology for rendering vector graphics from XML documents [DFF+10]. XML nodes in an SVG document may represent primitive graphical objects, for example, the `<rect>` tag is rendered as a graphical rectangle, and the `<ellipse>` tag is rendered as an ellipse. SVG also includes support for text, images and complex paths made up of bézier curves. Web user interfaces based on ECMAScript and SVG can be developed in the same

manner as those based on HTML, such that event listeners can be attached to individual SVG elements, which can then execute arbitrary ECMAScript code in response to UI events.

HTML5 Canvas provides a low-level 2D graphics API [HH10]. Unlike HTML and SVG, Canvas is not DOM-based. Whereas SVG provides a "retained mode" API, and event listeners can be registered on individual, primitive graphical objects, Canvas simply exists as a single object in a Web page, which exposes a low-level API that JavaScript can use to "paint" vector graphics to the canvas surface.

The case studies described in Chapter 5 were both developed using SVG and ECMAScript.

1.1.3 Statecharts for Web User Interfaces

Statecharts have long been used to model behaviour of hypermedia applications [DOTM01], and after the creation of the World Wide Web, Statecharts were used to model the behaviour of Web browser clients as hypermedia applications, including for Web navigation[LHYT00], and client-side behaviour of built-in Web browser UI controls, such as hyperlinks and frames [WP03].

There are two examples where Statecharts have been embraced to model the behaviour of richly interactive Web applications based on HTML and ECMAScript, namely the *Sproutcore* and *EmberJS* libraries. Sproutcore is an open source ECMAScript framework for developing rich Web user interfaces, which includes an implementation of Statecharts called *Ki*, developed by Michael Cohen (`@frozencanuck` on Twitter)[ki]. EmberJS is another ECMAScript framework, derived from Sproutcore, which also includes a state machine implementation, called *StateManager* [emb].

1.2 Contributions

Like Ki and StateManager, SCION is an implementation of Statecharts in ECMAScript, designed to be used in the Web browser for user interface development. The first contribution of this thesis, then, is the description of the syntax and semantics of SCION, followed by a rigorous empirical evaluation of the performance of SCION's implementation of this semantics.

The second contribution of this thesis is the description of a new design pattern, Stateful Command Syntax, that illustrates how SCION can be effectively used to develop complex, interactive Web user interfaces.

SCION and the case studies in Chapter 5 were written from scratch, and have been released as open source. I am the founder and primary author of SCION. Matt Oshry at [24]7 inc., a consumer experience company, has also contributed patches. The case studies were developed entirely by me.

1.3 Thesis Outline

The first half of this thesis describes SCION, a Statecharts interpreter optimized for ECMAScript. Chapters 2 and 3 specify the syntax and semantics of SCION models, and chapter 4 describes how the implementation of this semantics has been optimized for ECMAScript and the World Wide Web.

Chapter 5 presents Stateful Command Syntax, and provides two case studies which illustrate its use.

Chapter 2

SCION Syntax and Semantics

Statecharts is a visual language invented in 1987 by David Harel as a formalism for describing complex, timed, reactive, state-based behaviour [Har87]. Since then, many variants of Statecharts have been developed. SCXML is a Statecharts variant developed by the W3C, and is described in a W3C working draft specification [BAA+10]. SCXML specifies a textual syntax for Statecharts based on *eXtensible Markup Language* (XML), a document format developed by the W3C.

StateCharts Interpretation and Optimization eNgine (SCION) is a Statecharts variant based on SCXML, and this chapter describes its syntax and semantics.

2.1 Motivation for Using SCXML

There were several pragmatic reasons why SCXML was chosen as a core technology for SCION.

First, in SCXML, Statecharts are written as human-readable and human-editable XML documents, which means that complex tooling is not needed to develop SCXML. Developers can use their preferred text editor to author SCXML documents, and do not need to rely on a complex graphical environment. Furthermore, as hand-edited XML documents are plain text files, they can be easily managed using an off-the-shelf version-control system, such as Subversion or Git.

Next, XML documents are easy to generate programmatically using a variety of well-supported tools. For example, all of the test cases described in Chapter 4 were generated programmatically using *lxml2*, an XML library for the Python programming language.

Next, there exists a suite of standards for querying and transforming XML documents, as well as excellent free and open source implementations of these standards. For example, *XPath* is a W3C standard which describes a notation for navigating the hierarchical structure of XML documents [CD+99]. In the context of SCXML, XPath can be used to analyze the static properties of a Statecharts model. Likewise, the W3C standard *XSLT* provides a language for transforming XML documents [C+99], and in the context of SCXML, can be used to implement model transformations. For example, the transition flattening opti-

mization that is described in Section 4.3.3 relies on a model transformation implemented in XSLT.

Finally, XML integrates well with other Web technologies, such as XHTML and SVG. SCXML documents can be embedded into Web pages, or parsed individually, using technology built into all modern Web browsers.

An additional reason why SCXML was chosen as a core technology for SCION, is that the SCXML specification states that ECMAScript should be used as an embedded action language. This had two main advantages. First, I felt that use of ECMAScript for this purpose would make SCXML more familiar to Web developers, who are likely already skilled users of ECMAScript. Second, the Web browser environment already includes an ECMAScript interpreter, and this made developing a browser-based SCXML implementation easier, as it was possible to delegate execution of action code to the Web browser environment.

2.2 Syntax

For completeness, this section provides an overview of the textual, XML-based syntax for SCXML, described in the W3C draft specification [BAA+10]. Furthermore, this section describes a visual syntax for SCXML, which is not described in the SCXML specification.

The visual syntax described in this section is based on the syntax that is implemented in the AToM3 DCharts modelling environment [Fen04], which was used to create all of the visual Statecharts examples in this thesis. Variations on the visual syntax presented in this section are possible, and depend on the Statecharts editor that is used.

Most examples in this chapter include figures and code listings that illustrate both the XML and the visual syntaxes. Examples in Chapter 4 primarily use the XML syntax, while examples in Chapter 5 primarily use the visual syntax.

2.2.1 States

A Statecharts model is composed of *states* and *transitions*. There are three types of states: OR states, AND states, and basic states. Each state has a name. In the visual syntax, OR and AND states are represented by dark blue rectangles, with their name in the upper left-hand corner; basic states are represented by gray circles, with a text field containing the value of the name located graphically under the circle. OR and AND states can contain other states, such that there is a hierarchical, parent-child relationship between parent states and child states, also called *sub-states*. This hierarchical, or *ancestral*, relationship is represented in the visual syntax by making the graphical rectangle associated with the parent state surround the graphical objects associated with its substates.

In the visual syntax, an AND state is differentiated from an OR state in that its child OR states, also called *orthogonal components*, have light gray rectangles.

An OR state must have exactly one child which is marked as an *initial state*. In the visual syntax, this is represented by making the stroke color of the graphical object green.

Finally, there exists a state that is the ancestor of all other states. This state is called the *root state*, and, like the OR state, one of its substates must be marked as an initial state. The root state does not have a graphical representation in the visual syntax; instead its presence is implied.

Figure 2.1 shows an example of a simple Statecharts model containing a top-level OR state, `A`, with two substates, `A1` and `A2`, and a top-level AND state, `B`, with two sub-states, `B1` and `B2`. `A` and `A1` are initial states.

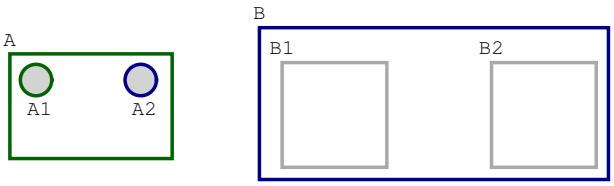

Figure 2.1: A Statecharts model with AND, OR and basic states, arranged hierarchically

In SCXML, AND, OR and basic states are all written using XML tags. State names are encoded using the `@id` attribute on the XML tag. AND states are encoded using the `<parallel>` tag, and OR and basic states are both written using the `<state>` tag. XML documents describe tree structures, so hierarchical relationships between states are directly encoded by hierarchical relationships between XML nodes. As basic states and OR states use the same tag (`<state>`), basic states are distinguished from OR states in SCXML implicitly by their lack of sub-states.

There are three ways to represent initial states in SCXML. The first option is to use an `@initial` attribute on the parent state. In this technique, the node associated with the parent state may have an `@initial` attribute with the value set to the initial state's name. The second option is to use create an `<initial>` element as a child of the parent OR state. In the third technique, the first sub-state in *document order* is implicitly used as the initial state. *Document order* is a total ordering on the nodes in an XML document, which is defined as the order in which the first character of each node occurs in the serialized XML representation of the document [CD+99]. This third technique is used in the SCXML examples throughout this chapter.

The outer root state is written using the `<scxml>` tag. The SCXML root element normally includes an XML namespace declaration, but for the sake of brevity, this is omitted from examples of SCXML documents in this thesis.

Listing 2.1 shows an XML representation of the example from Figure 2.1.

2.2.2 Transitions

Transitions represent possible flows of between states. Transitions are associated with one *source state*, zero-to-many *target states*, zero-to-many *triggers*, zero-or-one *conditions*, and zero-to-many *actions*. A transition is graphically depicted in the visual syntax as an arrow originating from the source state and targeting the target state. A transition that targets

```
<scxml>
    <state id="A">
        <state id="A1"/>
        <state id="A2"/>
    </state>
    <parallel id="B">
        <state id="B1"/>
        <state id="B2"/>
    </parallel>
</scxml>
```

Listing 2.1: An SCXML Document with AND, OR and basic states, arranged hierarchically

multiple states is represented as a hyperedge targeting multiple states, however this is only a legal syntax when the states being targeted are *orthogonal*. Two states are defined as *orthogonal* if they are not ancestrally related, and their closest mutual parent is an AND-state.

In the visual syntax, triggers, conditions, and actions are shown in a textual label that floats near the transition arrow. The visual syntax uses the following transition label format: *trigger1 trigger2[condition()]/action()*. Conditions can contain arbitrary ECMAScript expressions, and actions can contain arbitrary executable ECMAScript code. An example of this is shown in Figure 2.2.

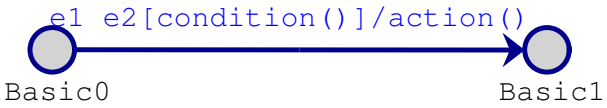

Figure 2.2: An example of a transition with two triggers, a condition and an action

In SCXML, a transition is represented using the `<transition>` tag. The source state of the transition is the parent of the `<transition>` node, and the target is written using the `@target` attribute on the node, where the attribute value is a space-separated list of ids of the target states. The transition triggers are encoded using the `@event` attribute, whose value is a space-separated list of triggers. The `@cond` attribute is used to encode the transition condition, and can contain an arbitrary ECMAScript expression as its value. SCXML also defines a special `In()` function, that returns `true` if the system is in a particular state.

The SCXML representation of the example from Figure 2.2 can be seen in Listing 2.2.

2.2.3 Actions

A transition can be associated with some action that executes when the transition is taken. In the XML syntax, actions are encoded as child nodes of the transition node. The SCXML specification describes several tags for actions. SCION currently implements the following:

- `<log>` computes a string from an ECMAScript expression, and prints it to the console.

```
<scxml>
    <state id="A">
        <transition target="B" event="e1 e2" cond="condition()">
            <script>
                action();
            </script>
        </transition>
    </state>
    <state id="B"/>
</scxml>
```

Listing 2.2: An example of a transition with two triggers, a condition and an action

```
<scxml>
    <datamodel>
        <data id="x" expr="1"/>
        <data id="y" expr="'hello'"/>
        <data id="z"/>
    </datamodel>
</scxml>
```

Listing 2.3: SCXML Datamodel

- `<script>` can contain arbitrary ECMAScript code to be executed
- `<assign>` assigns a value derived from the evaluation of an ECMAScript expression the system's *datamodel*
- `<raise>` and `<send>` are described later, in Section 2.5.4.

In SCXML, the set of variables visible to the system is explicitly declared using a `<datamodel>` tag. This is illustrated in Listing 2.3. In this example, the system would declare three variables: `x`, `y` and `z`, where `x` would be initialized to ECMAScript Number value of 1, `y` would be initialized to ECMAScript String value `"hello"`, and `z` would be undefined. The datamodel can be manipulated using the SCXML `<assign>` tag, or via ECMAScript assignment statements in the `<script>` tag.

SCXML also has a notion of a *default transition*, which is a transition without a `@target` attribute. This is similar to a *static reaction*, described in [HK04], and indicates that on an event, the actions associated with the transition may be executed, but the system will not change state.

States can also be associated with actions, of two types: *entry* actions and *exit* actions. In the visual syntax, entry and exit actions are not explicitly shown, although they can be displayed in a dialog in the AToM3 editor. In SCXML, entry and exit actions are placed in `<onentry>` and `<onexit>` tags, respectively, which are children of the XML node associated with the state. Listing 2.4 shows an example of entry and exit actions in SCXML.

```
<scxml>
    <state id="A">
        <onentry>
            <log expr="'entered A'"/>
        </onentry>
        <onexit>
            <log expr="'exited A'"/>
        </onexit>

        <transition target="B" event="t" cond="'foo' === 'bar'">
            <log expr="'taking transition from A to B'"/>
        </transition>
    </state>
    <state id="B"/>
</scxml>
```

Listing 2.4: An SCXML document with entry, exit and transition actions

2.2.4 History

Finally, a Statecharts model may contain special *history* pseudostates, which can be of two types: *deep* and *shallow*. In terms of syntax, history is treated like a basic state, in that it can be the source and target of transitions, and be the child of an OR or AND state. In DCharts, shallow history is drawn as a blue circle with an "H" character inside, and deep history is drawn as a blue circle with an "H*" string inside. In SCXML a history state is written using the `<history>` tag, and the deep/shallow property is captured using the `@type` attribute on the history node. An example of history can be seen in Figure 2.3 and Listing 2.5.

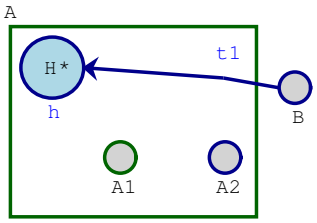

Figure 2.3: An example of a deep history state

2.3 Common Basic Semantics

At a high level, a Statecharts model describes an event machine, which executes *big-steps* in response to input events. A *big-step* is composed of a possibly infinite sequence of *small-steps*, where each small-step moves the system between *snapshots*. A snapshot is defined as tuple consisting of three elements:

```
<scxml>
    <state id="A">
        <state id="A1"/>
        <state id="A2"/>
        <history id="h" type="deep"/>
    </state>
    <state id="B">
        <transition target="h" event="t1"/>
    </state>
</scxml>
```

Listing 2.5: An example of a deep history state

1. The *basic configuration*, which is the set of basic states in which the model resides. A *full configuration*, which is the set of basic, AND and OR states in which the model resides, can be derived from a basic configuration by unioning the states in the basic configuration with their ancestors.

2. The *datamodel*, which is the set of variables visible to the Statecharts model.

3. The *present events*, which is the set of events that can be sensed by the system. Events can be generated internally (*internal events*), or received from the environment (*interface events*). The difference between internal and interface events in SCION semantics is elaborated in Section 2.5.5.

More specifically, in order to move the system between snapshots, the following tasks are performed in a small-step:

1. An event is sensed by the system and zero or more transitions are *enabled*.

2. From this set of enabled transitions, a subset of *priority enabled* transitions are selected.

3. For each state in the set of states to exit, ordered hierarchically from inner to outer, the exit actions associated with each state are executed.

4. The transition actions of each priority enabled transition are executed in document order.

5. For each state in the set of states to enter, ordered hierarchically from outer to inner, the entry actions associated with each state are executed.

6. The configuration is updated so that states in the set of states to exit are removed, and the states in the set of states to enter are added.

A big-step will continue executing small-steps until the *Maximality* constraints of its semantics are satisfied, which is described in Section 2.5.3.

The remainder of this section describes the conditions under which a transition becomes enabled, and the method of computing the set of states to enter and set of states to exit in a small-step.

2.3.1 Enabled Transitions

A transition is *enabled* if all of the following conditions are satisfied:

1. The transition's source state is in the model's full configuration.
2. The transition's trigger is satisfied, which is defined as the following disjunction:
   (a) The transition does not have a trigger (i.e., it is a default transition), or
   (b) The set of events that can be sensed is nonempty, and the transition has a *wildcard trigger*, which in SCXML is encoded as the special string "*", or
   (c) One of the transition's triggers matches an event in the set of events that can be sensed.
3. The transition's condition is satisfied, which is defined as the following disjunction:
   (a) The transition does not have a condition, or
   (b) The transition has a condition and the condition evaluates to true.

A transition is considered *priority enabled* if it is has higher priority than other transitions that can be executed instead of it. Section 2.5.7 describes the semantic aspect that determines transition priority.

2.3.2 Selecting States to Enter and States to Exit in a Small-Step

Informally, an AND state expresses an AND relationship between its sub-states, such that when the full configuration contains an AND state, it also contains all of the AND state's children. Likewise, an OR state expresses an XOR relationship between its substates, such that when the full configuration contains an OR state, it contains exactly one of the OR state's children. These constraints are captured in the following descriptions of the set of states entered and set of states exited, given a set of priority enabled transitions and the current configuration. Section 3.9 provides pseudocode implementations of these descriptions, and they are illustrated in the following section with examples.

The set of states to exit can be defined as follows. First, define `S` to be the set of states in the basic configuration that satisfy the following condition: the ancestor of the state or the state itself is the source state of a priority enabled transition. Then, the set of states to exit is defined as the union of each state in `S` with its ancestors, up to but not including the *arena* of the priority enabled transition that caused that state to be exited. The *arena* of a transition is the *least common ancestor* (LCA) of its source and destination states, which is the state lowest in the state hierarchy that is an ancestor of the source and destination states.

The set of states to enter is defined to be the set of states in the hierarchy between each priority enabled transition target and that target's ancestors, up to but not including the transition arena. If the target of a priority enabled transition is an AND or OR state, then

```
<scxml>
    <state id="A">
        <transition target="B" event="t"/>
    </state>
    <state id="B"/>
</scxml>
```

Listing 2.6: A simple SCXML document

its children must be recursively added to the set of states to enter, according to the following rules:

- If the state to be entered is an OR state, then its initial state should be recursively added to the set of states to enter.
- If the state to be added is an AND state, then its substates should be recursively added to the states to be entered.
- If the state to be added is a basic state, then it should simply be added to the states to be entered.
- If the state to be added is a history state, then its *history value* should be recursively added to the set of states to enter.

The *history value* of a history state is defined as follows:

- If history is of type *shallow*, the history value is the sub-state of history's parent state which the system was in the last time it was in history's parent.
- If history is of type *deep*, the history value is the basic configuration relative to the history state's parent which the system was in the last time it was in history's parent.

2.4 Examples of Common Basic Semantics

2.4.1 Basic States

To clarify the above definitions, consider the following example model in Listing 2.6 and Figure 2.4.

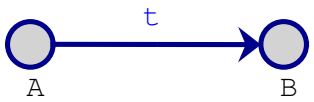

Figure 2.4: A simple Statecharts model

```
<scxml>
    <state id="A">
        <onexit>
            <log expr="'foo'"/>
        </onexit>
        <transition target="B" event="t">
            <log expr="'bar'"/>
        </transition>
    </state>
    <state id="B">
        <onentry>
            <log expr="'bat'"/>
        </onentry>
    </state>
</scxml>
```

Listing 2.7: A simple Statecharts model with entry, exit, and transition actions

The system starts in basic configuration {A}. When event t is sent to the system, the transition from state A to B is priority enabled. The arena of the transition from A to B is the root state. The set of states to exit is defined to be A's ancestors and A itself, up to but not including the transition arena (the root state), therefore the set of states to exit is {A}. Likewise, the set of states to enter is defined to be the target of the transition (B), up to but not including the transition arena (root), therefore the set of states to enter is {B}. The exit actions of A would then be fired, followed by the transition actions of the transition from A to B, followed by the entry action of B. Lastly, the basic configuration would be updated to {B}.

The example in Listing 2.7 adds state entry, exit, and transition actions to the previous example. This example would print "foo bar bat" to the console.

2.4.2 OR States

The example in Listing 2.8 and Figure 2.5 illustrates a basic scenario using OR states.

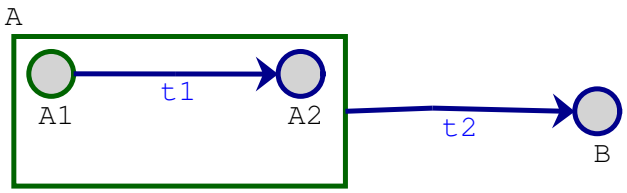

Figure 2.5: A Statecharts model illustrating OR states

The system starts in basic configuration {A1}. When event t1 is sent to the system, the transition from A1 to A2 is priority enabled. The arena of this transition is A. The set of states to exit is defined to be A1's ancestors and A1 itself, up to but not including the

```
<scxml>
    <state id="A">
        <state id="A1">
            <transition target="A2" event="t1"/>
        </state>
        <state id="A2"/>

        <transition target="B" event="t2"/>
    </state>
    <state id="B"/>
</scxml>
```

Listing 2.8: An SCXML document illustrating OR states

transition arena (A), therefore the set of states to exit is {A1}. The set of states to enter is defined to be the target of the transition (A2) and its ancestors, up to but not including the transition arena (A), therefore the set of states to enter is {A2}. The exit actions of A1 would then be fired, followed by the transition actions of the transition from A1 to A2, followed by the entry action of A2. Lastly, the basic configuration would be updated to {A2}.

When event t2 is sent to the system, the transition from A to B would be priority enabled. A2 is a basic state in the configuration whose ancestor (A) is a source state of a priority enabled transition (from A to B), and so A2 and A2's ancestors are added to the states to exit, up to but not including the transition arena (root). Therefore, the set of states to exit is {A2,A}. The set of states to enter is the target of the transition (B), up to but not including the transition arena (root), therefore the set of states to enter is {B}. The exit actions would be fired in order of increasing hierarchy, so A2's exit actions would fire followed by A's exit actions. The transition actions would then fire, followed by the entry actions for B.

2.4.3 AND States

The example in Listing 2.9 and Figure 2.6 illustrates a basic scenario for AND states.

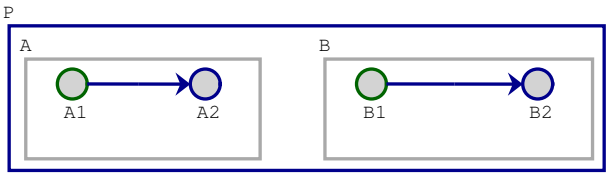

Figure 2.6: A Statecharts model illustrating AND states

The system starts in configuration {A1,B1}. When event t1 is sent to the system, the transition from A1 to A2 is priority enabled. The arena of this transition is A. The set of states to exit is defined to be A1's ancestors and A1 itself, up to but not including the arena (A), therefore the set of states to exit is {A1}. The set of states to enter is defined to be the target of the transition (A2), up to but not including the arena (A), therefore the set of

```
<scxml>
    <parallel id="P">
        <state id="A">
            <state id="A1">
                <transition target="A2" event="t1"/>
            </state>
            <state id="A2"/>
        </state>
        <state id="B">
            <state id="B1">
                <transition target="B2" event="t2"/>
            </state>
            <state id="B2"/>
        </state>
    </parallel>
</scxml>
```

Listing 2.9: An SCXML document illustrating AND states

states to enter is {A2}. The exit actions of A1 would then be fired, followed by the transition action of the transition from A1 to A2, followed by the entry action of A2. Lastly, the basic configuration would be updated to {A2,B1}.

When event t2 is sent to the statechart, the small-step would start and the transition from B1 to B2 would be priority enabled. The arena of the transition from B1 to B2 is state B. The set of states to exit is defined to be B1's ancestors and B1 itself, up to but not including the arena, which is B, therefore the set of states to exit is {B1}. The set of states to enter is defined to be the target of the transition (B2), up to but not including the arena (B), therefore the set of states to enter is {B2}. The exit actions of B1 would then be fired, followed by the transition action of the transition from B1 to B2, followed by the entry action of B2. Lastly, the basic configuration would be updated to {A2,B2}.

2.4.4 History

The example in Listing 2.10 and Figure 2.7 illustrates a basic scenario for shallow history.

The system starts in basic configuration {A1}. When event t1 is sent to the system, the transition from A1 to A2 is priority enabled. The arena of the transition from A1 to A2 is state A. The set of states to exit is defined to be A1's ancestors and A1 itself, up to but not including the arena, which is A, therefore the set of states to exit is {A1}. The set of states to enter is {A2}. The exit actions of A1 would then be fired, followed by the transition action of the transition from A1 to A2, followed by the entry action of A2, and the configuration would be updated to {A2}.

When event t2 is sent to the system, the transition from A to B would be priority enabled. A2 is a basic state in the configuration whose ancestor (A) is a source state of a priority enabled transition (from A to B), and so A2 and A2's ancestors are added to the states to exit, up to but not including the arena of the transition (root). Therefore, the set of states to exit is

```
<scxml>
    <state id="A">
        <state id="A1">
            <transition target="A2" event="t1"/>
        </state>
        <state id="A2">
            <transition target="B" event="t2"/>
        </state>

        <history id="H" type="shallow"/>
    </state>

    <state id="B">
        <transition target="h" event="t3"/>
    </state>
</scxml>
```

Listing 2.10: An SCXML document illustrating History states

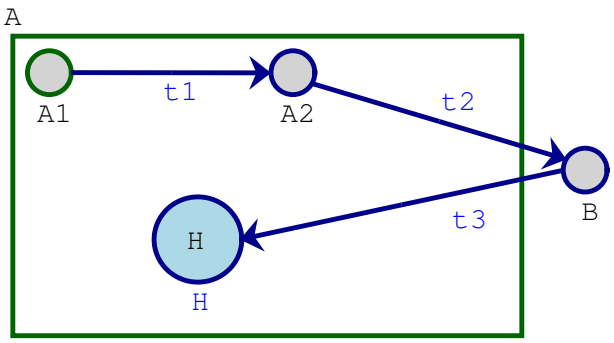

Figure 2.7: A Statecharts model illustrating History states

{A2,A}. The set of states to enter is the target of the transition (B), up to but not including the arena (root), therefore the set of states to enter is {B}. The exit actions would then be fired in order of increasing hierarchy, so A2's exit action would fire, followed by A's exit action, followed by the transition action, and entry action for B. The configuration would then be updated to {B}.

When event t3 is sent to the system, the transition from B to H would be priority enabled. The history value of shallow history state H is defined to be the sub-state of history's parent (A) which the system was in the last time it was in history's parent, therefore the history value is A2. {A2, A} are therefore added to the set of states to enter. The set of states to exit would be {B}, and the final basic configuration would be {A2}.

Deep History The example in Listing 2.11 and Figure 2.8 illustrates a basic scenario for deep history.

The system starts in basic configuration {A11}. When event t1 is sent to the system,

```
<scxml>
    <state id="A">
        <state id="A1">
            <state id="A11">
                <transition target="A12" event="t1"/>
            </state>
            <state id="A12"/>
        </state>

        <transition target="B" event="t2"/>

        <history id="H" type="deep"/>
    </state>

    <state id="B">
        <transition target="h" event="t3"/>
    </state>
</scxml>
```

Listing 2.11: An SCXML document illustrating deep History states

the transition from `A11` to `A12` is priority enabled. The arena of the transition from `A11` to `A12` is state `A1`. The set of states to exit is defined to be `A11`'s ancestors and `A11` itself, up to but not including the arena, which is `A1`, therefore the set of states to exit is {`A11`}. The set of states to enter is {`A12`}. The exit actions of `A11` would then be fired, followed by the transition action of the transition from `A11` to `A12`, followed by the entry action of `A12`, and the configuration would be updated to {`A12`}.

When event `t2` is sent to the system, the transition from `A` to `B` would be priority enabled. `A12` is a basic state in the configuration whose ancestor (`A`) is a source state of a priority enabled transition (from `A` to `B`), and so `A12` and `A12`'s ancestors are added to the states to exit, up to but not including the arena of the transition (root). Therefore, the set of states to exit is {`A12,A1,A`}. The set of states to enter is the target of the transition (`B`), up to but not including the arena (root), therefore the set of states to enter is {`B`}. The exit actions would then be fired in order of increasing hierarchy, so `A12`'s exit action would fire, followed by `A1` and `A`'s exit actions, followed by the transition action, and entry action for `B`. The configuration would then be updated to {`B`}.

When event `t3` is sent to the system, the transition from `B` to `H` would be priority enabled. The history value of deep history state `H` is defined to be the basic configuration relative to the history state's parent (`A`) which the system was in the last time it was in history's parent, therefore the history value is `A12`. {`A12, A1, A`} are therefore added to the set of states to enter. The set of states to exit would be {`B`}, and the final basic configuration would be {`A12`}.

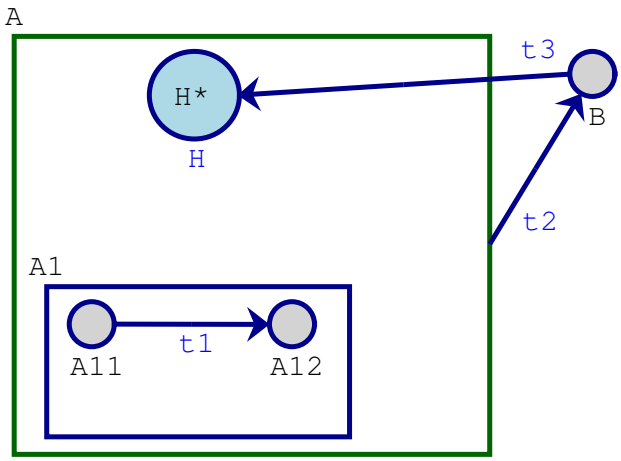

Figure 2.8: A Statecharts model illustrating History states

2.5 Semantic Aspects

The previous sections define a basic semantics for SCION, but leave many unanswered questions regarding semantic edge cases. This section enumerates those questions using the framework described in "Big-Step Semantics", by Esmaeilsabzali, Day, et. al [EDAN09]. "Big-Step Semantics" defines a set of semantic aspects that describes the executable behaviour of a range of Statecharts variants. SCION chooses one configuration of the set of possible choices for each of these semantic aspects. Each semantic aspect is summarized here and illustrated with meaningful examples.

This section simplifies some of the work presented in "Big-Step Semantics." In certain cases, some semantic choices that would be out of scope for this thesis are excluded. For example, this chapter does not discuss the notion of "combo-steps", and therefore semantic choices that rely on this concept are excluded. Additionally, the syntax and semantics presented in this section extends the work of "Big-Step Semantics" in some ways, such as adding support for history states and default transitions, which are described in the previous sections.

2.5.1 General Influences on SCION Semantics

There were several general motivating factors that influenced SCION's semantics.

First, the W3C SCXML draft specification had a large influence on the semantic choices that were made. The SCXML specification provides the "Algorithm for SCXML Interpretation," which is a canonical algorithm for executing SCXML documents. A choice was made early in SCION's development not to reuse this algorithm directly for SCION's semantics, as at the time development on SCION began, the "Algorithm for SCXML Interpretation" contained a serious bug regarding the execution of parallel states, such that the algorithm would lead to illegal system configurations for certain simple test cases [Bea]. However, cer-

tain semantic decisions made by the "Algorithm for SCXML Interpretation" were found to be useful and intuitive, and were thus incorporated into SCION's semantics. Two examples of this are the Memory Protocol and Event Lifeline semantic aspects, which are described in Sections 2.5.2 and 2.5.4.

Another way that SCXML influenced SCION semantics was through the document order property of XML, which defines a total ordering on SCXML elements. This can be used to resolve what would otherwise be nondeterministic behaviour. Two examples of semantic aspects influenced by this are Transition Priority and Order of Transitions, which are described in Sections 2.5.7 and 2.5.8.

The intended target environment of the Web browser also had an impact on SCION's semantics, particularly External Event Communication, which is described in Section 2.5.5.

Finally, several semantic choices, in particular Memory Protocol Semantics, were based on what I felt would be most intuitive for Web developers, the intended users of SCION.

Another general influence on SCION semantics was the notion that supporting the *synchrony hypothesis* was not a desirable feature for SCION. The *synchrony hypothesis* is the assumption that a big-step takes zero time, and is often used for Statecharts semantics targeting hardware, where a big-step may execute during a single clock cycle. SCION is designed such that arbitrary ECMAScript code may be executed in a single small-step, which may take non-zero time, and thus the synchrony hypothesis would not have been a good fit for SCION semantics.

The remainder of this chapter discusses various semantic aspects from "Big-Step Semantics", and describe how they apply to SCION.

2.5.2 Memory Protocol

The *Memory Protocol* semantic aspect describes when changes to the datamodel are sensed. There are two possible choices for this:

- *Next Big-Step*: variables retain the same value throughout the big-step, and are updated at the end of the big-step.
- *Next Small-Step*: variables retain the same value throughout the small-step, and are updated at the end of the small-step.

SCION supports two semantic choices: the Next Small-Step semantics, and an *ECMAScript assignment* semantics. This decision was motivated by SCXML, which embeds ECMAScript as a scripting language. ECMAScript can be used to manipulate the datamodel in action code inside of `<script>` tags. Next Small-Step Memory Protocol semantics then clash with ECMAScript assignment semantics, as in ECMAScript, like most procedural languages, the value of a variable is expected to change immediately after the execution of an assignment statement.

Furthermore, the ability to enforce Next Small-Step semantics is limited in the browser environment, as variable assignment in ECMAScript is, by default, performed in the global

scope. Therefore, a user of SCION would be able to side-step Memory Protocol semantics if they so chose simply by assigning to a global variable inside a `<script>` tag.

For these reasons, SCION supports **both** ECMAScript-style assignment semantics, such that result of an assignment action is detected in the same small-step, and optional Next Small-Step memory protocol semantics, in which assignment to variables in the datamodel is detected in the next small-step. This is managed as follows:

- In ECMAScript action code, ECMAScript assignment statements use ordinary ECMAScript semantics to perform assignments to variables in the datamodel. This means that the datamodel will be updated immediately, after the ECMAScript assignment statement finishes executing, and these changes to the datamodel will be detectable in the same small-step.
- Assignment performed by the SCXML `<assign>` tag uses ECMAScript assignment semantics as well.
- To use Next Small-Step semantics, a "getter" and "setter" API is provided to ECMAScript action code:
  - *setData(String variableName, Object value)*
  - *getData(String variableName)*

Examples

Same Small-Step Memory Protocol Semantics Consider the example in Listing 2.12.

In this example, when the system is started, datamodel variable `foo` would be initialized to value 0, and the system would begin in initial state `a`. Upon receiving event `t`, the system would transition to state `b`, and the transition `<assign>` actions would be executed in document order. `foo` would be updated so that changes are detected in the same small-step. This means that in the first `<assign>`, *foo + 1*, would assign the value $0 + 1 = 1$ to variable `foo`. In the second assign, $foo - 1$, would assign the value $1 - 1 = 0$ to `foo`. The same would be repeated in the next small-step, in the default transition from `b` to `c`. Thus, `foo` would end with value 0.

The example in Listing 2.12 is equivalent to the following example in Listing 2.13, which uses `<script>` instead of `<assign>`.

Next Small-Step Memory Protocol Semantics Listing 2.14 illustrates how Next Small-Step assignment may be used in SCION.

Like the first two examples in Listings 2.12 and 2.13, the system will begin in state `a`, variable `foo` will be initialized to value *0*, and upon receiving event `t`, the system will transition to state `b`. Because the expression "*setData('foo',getData('foo') - 1)*" is executed after the expression "*setData('foo',getData('foo') + 1)*", at the end of the first small-step `foo` will be assigned the value $foo - 1 = 0 - 1 = -1$. The system would end the small-step in state `b`. In the next small-step, the default transition to state `c` would be taken. The

```
<scxml>

  <datamodel>
    <data id="foo" expr="0"/>
  </datamode>

  <state id="a">
    <transition target="b" event="t">
      <assign location="foo" expr="foo + 1"/>
      <assign location="foo" expr="foo - 1"/>
    </transition>
  </state>

  <state id="b">
    <transition target="c">
      <assign location="foo" expr="foo + 1"/>
      <assign location="foo" expr="foo - 1"/>
    </transition>
  </state>

  <state id="c"/>
</scxml>
```

Listing 2.12: SCXML document illustrating Same Small-Step Memory Protocol

same actions would be performed, and so at the end of the small-step, variable `foo` would be updated to value $foo - 1 = -1 - 1 = -2$.

2.5.3 Maximality

A big-step consists of a possibly infinite sequence of small-steps, and the *Maximality* semantic aspect defines the conditions in which the system reaches a stable state and ends a big-step.

There are three possible semantic choices for this aspect:

- *Syntactic*: a special syntax is used to specify a state which, when entered, should end the big-step.
- *Implicit*, of which there are two varieties:
  - *Take-One*: A big-step should end after all priority enabled transitions are taken in the first small-step.
  - *Take-Many*: The system should take multiple small-steps, until it reaches a stable state in which no transitions have been enabled.

SCION uses the Take-Many semantic aspect. Listing 2.15 and Figure 2.9 provide an example of this.

```
<scxml>

  <datamodel>
    <data id="foo" expr="0"/>
  </datamode>

  <state id="a">
    <transition target="b" event="t">
      <script>
        foo = foo + 1;
        foo = foo - 1;
      </script>
    </transition>
  </state>

  <state id="b">
    <transition target="c">
      <script>
        foo = foo + 1;
        foo = foo - 1;
      </script>
    </transition>
  </state>

  <state id="c"/>
</scxml>
```

Listing 2.13: A second example illustrating Same Small-Step Memory Protocol semantics

```
<scxml>

  <datamodel>
    <data id="foo" expr="0"/>
  </datamode>

  <state id="a">
    <transition target="b" event="t">
      <script>
        setData('foo',getData('foo') + 1);
        setData('foo',getData('foo') - 1);
      </script>
    </transition>
  </state>

  <state id="b">
    <transition target="c">
      <script>
        setData('foo',getData('foo') + 1);
        setData('foo',getData('foo') - 1);
      </script>
    </transition>
  </state>

  <state id="c"/>
</scxml>
```

Listing 2.14: An SCXML document illustrating Next Small-Step Memory Protocol Semantics

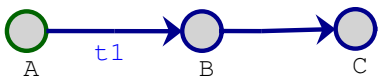

Figure 2.9: An example of Maximality semantics

In this example, after the event `t1` is sent to the system, a big-step would be initiated, and in the first small-step, the transition from `A` to `B` would be taken. Because of the Take-Many semantics, the default transition from `B` to `C` could be taken in a subsequent small-step. Finally, the system would be in a state where no transitions are selected, and the big-step would end, finishing the big-step in basic configuration {`C`}.

If Take-One semantics were used, then after the event `t1` is sent to the system, a big-step would be initiated, and in the first small-step, the transition from `A` to `B` would be taken. Take-One semantics specifies that the big-step should end after all priority enabled transitions are taken in the first small-step, and so the big-step would end, leaving the system in configuration {`B`}. In a subsequent big-step, the transition from `B` to `C` would be selected, as described above, leaving the system in configuration {`C`}.

If Syntactic semantics were used, then a special syntax would need to be invented and

```
<scxml>
    <state id="A">
        <transition target="B" event="t1"/>
    </state>
    <state id="B">
        <transition target="C"/>
    </state>
    <state id="C"/>
</scxml>
```

Listing 2.15: An SCXML document illustrating Maximality semantics

```
<scxml>
    <state id="A">
        <transition target="B" event="t1"/>
    </state>
    <state id="B">
        <transition target="B"/>
    </state>
</scxml>
```

Listing 2.16: An SCXML document illustrating a big-step that will never complete, assuming Take-Many semantics

applied to state `B` or state `C` in order to signify that the big-step should stop after entering one of these states.

The reduced example in Listing 2.16 and Figure 2.10 illustrates how, when Take-Many semantics are used, a big-step may never complete:

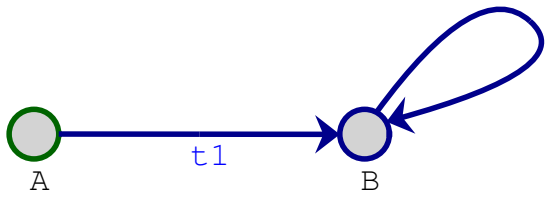

Figure 2.10: An example of a big-step that will never complete, assuming Take-Many semantics

In this case, the system starts in configuration {`A`}. When event `t1` is sent, a big-step is initiated, and in the first small-step, the transition from `A` to `B` is enabled, and the system transitions to `B`, leaving the system in configuration {`B`}. In the next small-step, the default transition from `B` to itself is enabled, and the system exits and re-enters `B`, leaving the system in configuration {`B`}. The execution of the previous small-step would continue infinitely.

2.5.4 Event Lifelines

The *Event Lifeline* semantic aspect describes when internal events can be sensed after the are generated.

There are four possible semantic choices:

- *Whole big-step*: a generated event can be sensed from the beginning of the current big-step, regardless of the small-step in which it has been generated, and persists until the end of the big-step.
- *Remainder of the current big-step*: a generated event can be sensed after the small step in which it is generated, and persists during the remainder of the big-step.
- *Next Small-Step*: a generated event can be sensed only in the small-step after it is generated.
- *Same Small-Step*: a generated event can be sensed only in the small-step in which it is generated.

SCION uses Next Small-Step Event Lifeline semantics. This decision was motivated by the SCXML step algorithm. In the SCXML step algorithm, a queue is kept for internal events, and in each small-step, a single event is dequeued and can be sensed during that small-step. I felt this would be an intuitive approach for Web developers.

Before describing an example of Event Lifeline semantics, an explanation of SCXML's `<raise>` and `<send>` elements is required. SCION uses these tags to generate internal and interface events, respectively.

SCXML `<send>` and `<raise>`

SCXML provides two tags for generating events: `<send>` and `<raise>`. `<raise>` is the simpler of the two, having a single attribute `@event` which specifies the event to be generated. `<send>` is designed to be more general, and is able to send events after a delay, for example "10 minutes", or "5 milliseconds", which is specified using the `@delay` attribute. Like `<raise>`, the event to generate is also specified for `<send>` using the `@event` attribute. In SCION semantics, `<send>` is used to generate interface events, and `<raise>` is used to generate internal events. Section 2.5.5 discusses `<send>` and interface events in more detail.

Example

For the example in Listing 2.17, assume that the Maximality semantic aspect is Take-Many, and the Memory Protocol constraint is Next Small-Step.

In this example, the system would begin in configuration {`A`}. When event `t1` is sent, a big-step would be initiated, and in the first small-step, the transition from `A` to `B` would be enabled. The system would transition to state `B`, raising internal event `t2` as its entry action, and ending the first small-step in configuration {`B`}.

```
<scxml>
    <state id="A">
        <transition target="B" event="t1"/>
    </state>
    <state id="B">
        <onentry>
            <raise event="t2"/>
        </onentry>
        <transition target="C" event="t2"/>
    </state>
    <state id="C">
        <transition target="D" event="t2"/>
    </state>
    <state id="D"/>
</scxml>
```

Listing 2.17: An SCXML document illustrating Event Lifeline semantics

Assuming Whole Big-Step or Remainder of Current Big-Step Event Lifeline semantics, event `t2` could then be sensed in the second small-step, which would enable the transition from `B` to `C`, ending the second small-step in configuration {`C`}. Event `t2` could then continue to be sensed in the third small-step, enabling the transition from `C` to `D`, ending the third small-step in configuration {`D`}. At this point, no more transitions would be enabled, thus ending the big-step in configuration {`D`}.

Like Whole Big-Step and Remainder of Current Big-Step semantics, assuming Next Small-Step Event Lifeline semantics, event `t2` would be sensed in the second small-step, which would enable the transition from `B` to `C`, ending the second small-step in configuration {`C`}. However, it would not be possible to sense `t2` in the subsequent small-step, hence, the transition from `C` to `D` would not be enabled, and the big-step would end in configuration {`C`}.

Assuming the Same Small-Step semantic option, event `t2` would be sensed in the same small-step in which it is raised, but not the one after. Hence, in the second small-step, the transition from `B` to `C` would not be enabled, and the big-step would end in configuration {`B`}.

2.5.5 External Event Communication

The External Event Communication semantic aspect describes what happens if an event is received from the environment while a big-step is executing.

There are three possible choices for this semantic aspect:

- *Strong Synchronous*: An interface event is either present throughout a big-step from the beginning, or is absent throughout the big-step.
- *Weak Synchronous*: An interface event need not be present from the beginning of a big-step, and would be sensed during the execution of the big-step.

```
package org.eclipse.swt.snippets;

import org.eclipse.swt.widgets.*;

public class Snippet1 {

    public static void main (String [] args) {
        Display display = new Display ();
        Shell shell = new Shell(display);
        shell.open ();
        //this starts the main event loop
        while (!shell.isDisposed ()) {
            //this reads and dispatches events
            if (!display.readAndDispatch ()) display.sleep ();
        }
        display.dispose ();
    }
}
```

Listing 2.18: Java SWT Event Loop

- *Asynchronous*: An interface event would be sensed in the big-step after the big-step in which it is generated.

SCION implements Asynchronous External Event Communication semantics. Furthermore, SCION treats events generated internally and externally as identical with respect to their event lifelines. This means that interface events are sensed for a single small-step in a big-step, rather than, e.g. throughout the entire big-step.

In order to provide an example of what this means for SCION, a further explanation must be provided of the execution model of the Web browser environment.

Web Browser Environment

Event Loop Abstraction An *event loop* is a programming construct which is widely used in graphical user interfaces (GUIs), where most GUI applications have a top-level event loop, known as the *main event loop*, which waits for and dispatches events, such as user interface events (e.g. mouse and keyboard events) and network events. An example of this can be seen in Listings 2.18 and 2.19, which show GUI programs containing main event loops in Java and Python for the SWT and Tkinter GUI toolkits, respectively.

The Web browser environment is also a GUI application that uses a main event loop to dispatch events. Furthermore, the Web browser environment embeds a scripting language, ECMAScript, in such a way that the main event loop is abstracted from executing ECMAScript code. This means that when a Web page is loaded, and a script tag is encountered, the ECMAScript code in the script tag is evaluated, and the thread of control is passed from the main event loop (the *embedding context*), to the executing ECMAScript code (*the embedded context*). When the ECMAScript code finishes executing, control returns to the

```
from Tkinter import *

root = Tk()

w = Label(root, text="Hello, world!")
w.pack()

#this starts the main event loop,
#which reads and dispatches events
root.mainloop()
```

Listing 2.19: Tkinter Event Loop

```
document.documentElement.addEventListener("keypress",function(event){
    console.log("received",event);
});
```

Listing 2.20: Registering a keypress event listener from ECMAScript

embedding context. This flow of control back to the embedding context happens transparently to the executing ECMAScript code.

The embedded ECMAScript code may register callback functions to listen for events asynchronously, which are passed to it from the main event loop in the embedding context. For example, Listing 2.5.5 illustrates how to register from ECMAScript an event listener so that an anonymous callback function is called on the user interface keypress event.

Time GUI toolkits that rely on an event loop also often provide a scheduling mechanism, so that a specified callback may be invoked after a delay. Examples of this include Tkinter's *Widget.after* [Lun], Adobe Flash's *flash.utils.setTimeout* [fla], and the Web browser's *window.setTimeout* [Net].

SCION uses this mechanism to delegate scheduling of delayed events to the Web browser environment. This means that SCION does not manage time internally, and instead relies on the environment to schedule sending of delayed events.

Semantic Implications SCION is designed primarily to be executed in the ECMAScript embedded context, so that the main event loop is abstracted. What this means first of all for SCION semantics is that *interface event* are events that are generated by the Web browser embedding context and passed into the ECMAScript embedded context via callback functions. Events from the embedding context are generated discretely, one at a time, such that a single event is passed to the embedded context via a callback. In order to execute a big-step, the Web browser's single thread of execution must be passed from the main event loop to the embedded context. While the embedded context has the thread, the embedding context may continue to queue events to send into the embedded context when the thread of control returns. Because of this queuing mechanism, and the single-threaded

nature of the Web browser environment, SCION can be said to implement Asynchronous External Event semantics. SCION only processes one interface event per big-step, because to do so requires passing the thread of execution from the Web browser environment into the embedded context; however, events may be queued by the environment until after the thread returns from the embedded context, at which point these queued events can be subsequently sent.

Note that while the Web browser environment can and does queue events asynchronously while the embedded context is executing a big-step, and thus SCION's Event Lifeline semantics are technically Asynchronous, a Statecharts model designed to implement user interface behaviour should not rely on this queuing mechanism, as queued UI events can result in a noticeable UI "lag", or delay between an event dispatch and its perceived results. For that reason, SCION is designed for fast execution, such that it should be able to process UI events as they are dispatched, and without causing them to be queued.

Example

An example of SCION semantics regarding External Event Communication semantics is provided in Listing 2.21.

In this example, upon entering state `A`, the system requests that the environment send interface events `t1`, `t2` and `t3` to the system after 10 milliseconds. The environment is responsible for scheduling events `t1`, `t2` and `t3`, and in this example, it is assumed that the environment is implemented in such a way that the events are sent in the order `t1`, `t2`, `t3`. This example also assumes a Next Big-Step option for the Memory Protocol semantic aspect.

Assuming an Asynchronous External Event Communication semantics, `t1`, `t2` and `t3` would each be handled in separate big-steps. As the datamodel would be updated at the end of each big-step, this means that variable `x` would be incremented three times, once for each transition taken in each big-step (from `A` to `B`, `B` to `C`, and `C` to `D`), so its resulting value would be 3.

Strong Synchronous and Weak Synchronous External Event Communication semantics would not be possible to implement in SCION given the architectural constraints mentioned above.

2.5.6 Transition Consistency

This section describes two semantics aspects concerning Transition Consistency, which refers to the conditions under which a pair of transitions can be taken together in a small-step.

Small-Step Consistency

The first semantic aspect is Small-Step Consistency. There are two choices for this:

- *Arena Orthogonal*: Two transitions may be included in the same small-step only if their arenas are orthogonal.

```
<scxml>
    <datamodel>
        <data id="x" expr="0"/>
    </datamodel>

    <state id="A">
        <onentry>
            <send delay="10ms" event="t1"/>
            <send delay="10ms" event="t2"/>
            <send delay="10ms" event="t3"/>
        </onentry>

        <transition target="B" event="t1">
            <assign location="x" expr="x + 1"/>
        </transition>
    </state>
    <state id="B">
        <transition target="C" event="t2">
            <assign location="x" expr="x + 1"/>
        </transition>
    </state>
    <state id="C">
        <transition target="D" event="t3">
            <assign location="x" expr="x + 1"/>
        </transition>
    </state>
    <state id="D"/>

</scxml>
```
Listing 2.21: An SCXML document illustrating External Event Communication semantics

```
<scxml>
    <state id="A">
        <state id="A1">
            <transition target="B" event="t1"/>
        </state>
        <transition target="C" event="t1"/>
    </state>
    <state id="B"/>
    <state id="C"/>
</scxml>
```

Listing 2.22: SCXML document illustrating Small-Step Consistency

- *Source/Destination orthogonal*: Two transitions may be included in the same small-step only if their their sources and destinations are pairwise orthogonal, which, more explicitly, entails the following three conditions: 1) their sources are orthogonal; 2) their destinations are orthogonal; and, 3) the first transition's source state is orthogonal with the second transition's destination state, and vice versa.

These semantic aspects can be illustrated in the examples in Listings 2.22 and 2.23.

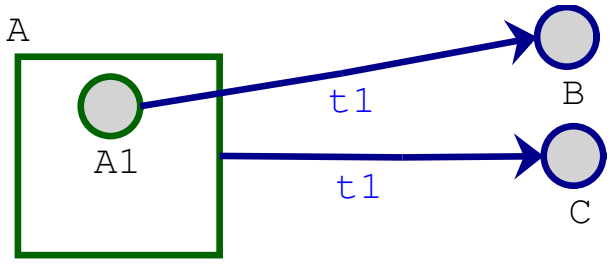

Figure 2.11: A trivial example of Small-Step Consistency semantics

The example in Listing 2.22 and Figure 2.11 presents a trivial case, because it does not involve orthogonality, and both Arena Orthogonal and Source/Destination Orthogonal semantics would produce the same outcome. However, this example illustrates a common scenario in Statecharts development, where two transitions are enabled, such that their source states are ancestrally related. Neither Arena Orthogonal nor Source/Destination Orthogonal semantics would permit the two transitions, from `A1` to `B` and `A` to `C`, to be taken together in the same small-step. For Arena Orthogonal semantics, the arena of both transitions is the root state, and a state by definition cannot be orthogonal to itself, therefore taking both transitions in the same small-step would not be permitted. For Source/Destination Orthogonal semantics, neither the sources nor destinations of the transitions are orthogonal to one another, and taking both transitions together would therefore not be permitted in the same small-step.

In the example in Listing 2.23, Arena Orthogonal and Source/Destination Orthogonal semantics would produce different outcomes. Arena Orthogonal semantics would not permit

```
<scxml>
    <parallel id="P">
        <state id="A">
            <state id="A1">
                <transition target="A2" event="t1"/>
            </state>
            <state id="A2"/>
        </state>
        <state id="B">
            <state id="B1">
                <transition target="C2" event="t1"/>
            </state>
            <state id="B2"/>
        </state>
        <state id="C">
            <state id="C1"/>
            <state id="C2"/>
        </state>
    </parallel>
</scxml>
```

Listing 2.23: A second SCXML document illustrating Small-Step Consistency

the transitions from `A1` to `A2` and `B1` to `C1` to be taken together in the same small-step, as the arena of the first transition is the state `A`, and the arena of the second transition is the state `P`, which are ancestrally related, and therefore not orthogonal. Source/Destination Orthogonal semantics, on the other hand, would permit the two transitions to be taken together in the same small-step, as the source states of the two transitions are orthogonal (`A1` and `B1`), the destination states are orthogonal (`A2` and `C1`), and the source and destination states are orthogonal (`A1` and `C1`, and `A2` and `B1`).

SCION uses Arena Orthogonal semantics, because I felt this option would be more intuitive for Web developers. More precisely, I reasoned that because developers must already think in terms of transition arena, which is used to compute the set of states to exit and the states to enter, then determining whether a pair of transitions are consistent would be a single mental operation (i.e., compare the arenas of the two transitions to determine if they are orthogonal), as opposed to four operations (compare the two transitions' source states; compare the destination states; compare the first transition's source state and second transition's destination state; and compare the first transition's destination state and second transition's source state). Ideally, a developer should be able to glance at a graphical Statecharts model, and very quickly mentally simulate it to predict how it will behave at runtime. I felt that Arena Orthogonal semantics better fulfilled the criterion of quick mental calculations, and would thus be more intuitive for developers.

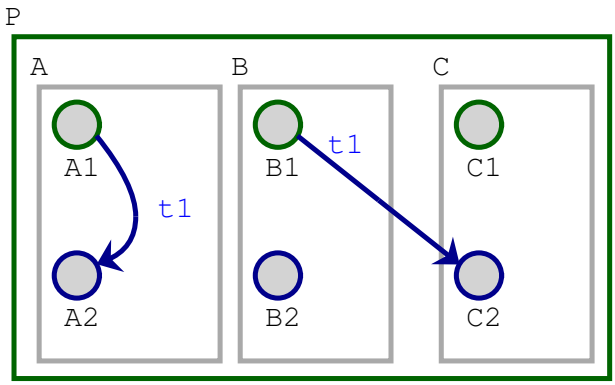

Figure 2.12: A second example of Small-Step Consistency semantics

Interrupt Transitions and Preemption

The second semantic sub-aspect concerns the case where one transition is an *interrupt* for another transition. A transition A is an interrupt for a transition B if their source states are orthogonal, and one of the following conditions holds:

- The destination of transition B is orthogonal with the source of transition A, and the destination of transition A is not orthogonal with the source of either A or B. This is illustrated in Listing 2.24 and Figure 2.13.
- The destination of transition A is not orthogonal with the source of transition A, the destination of transition B is not orthogonal with the source of transition B, and the destinations of transitions A and B are ancestrally related, such that the destination of transition A is a descendant of the destination of transition B. This is illustrated in Listing 2.25 and Figure 2.14.

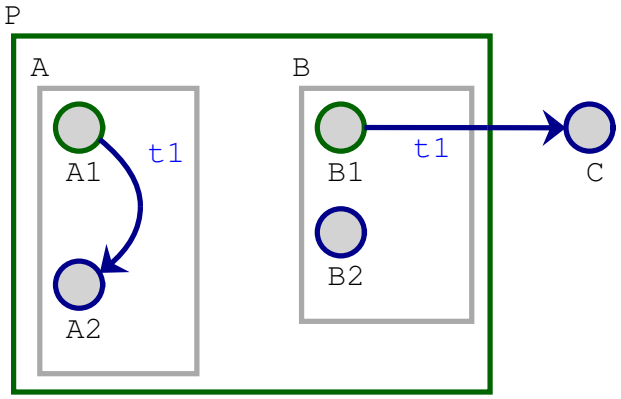

Figure 2.13: First Transition Preemption case

The semantic choices for this sub-aspect are:

```
<scxml>
    <parallel id="P">
        <state id="A">
            <state id="A1">
                <transition target="A2" event="t1"/>
            </state>
            <state id="A2"/>
        </state>
        <state id="B">
            <state id="B1">
                <transition target="C" event="t1"/>
            </state>
            <state id="B2"/>
        </state>
    </parallel>

    <state id="C"/>
</scxml>
```

Listing 2.24: First Transition Preemption case

```
<scxml>
    <parallel id="P">
        <state id="A">
            <state id="A1">
                <transition target="C" event="t1"/>
            </state>
        </state>
        <state id="B">
            <state id="B1">
                <transition target="C2" event="t1"/>
            </state>
        </state>
    </parallel>

    <state id="C">
        <state id="C1"/>
        <state id="C2"/>
    </state>
</scxml>
```

Listing 2.25: Second Transition Preemption case

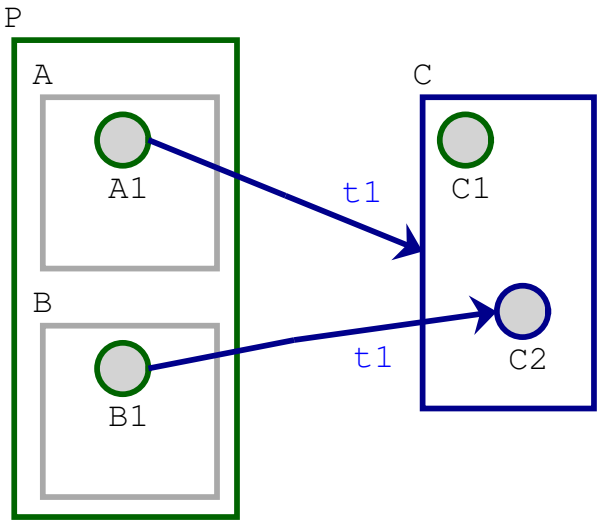

Figure 2.14: Second Transition Preemption case

- *Preemptive*: It is permissible for two transitions, where one is an interrupt for the other, to be taken together in the same small-step.
- *Non-Preemptive*: The opposite of Preemptive, such that it is not permissible for two transitions, where one is an interrupt for the other, to be taken together in the same small-step.

SCION uses Non-Preemptive semantics, again because I felt this would be a more intuitive choice for Web developers. In this case, I felt that Preemptive semantics would lead to scenarios that, while logically sound, would be surprising and difficult to visualize for many developers. For example, consider Figure 2.14. Intuitively, one would like to visualize the state machine as "flowing" along a transition from the source state to the target state. I felt it would be confusing to visualize the state machine as flowing along two transitions simulataneously to two target states that are ancestrally related.

2.5.7 Transition Priority

The previous section outlined a set of rules describing when transitions may be taken together in a small-step, but in order to describe how the examples in that section would execute in SCION, an explanation of Transition Priority semantics is required. The Transition Priority semantic aspect determines how transitions are prioritized when there are multiple enabled transitions that cannot be taken together in a small-step. Transitions which are enabled, and have priority over other transitions that are also enabled, are called *priority enabled*.

This semantic aspect is broken down into two categories of options:

- *Basis*:

  - *Source*: Priority is based on the transition source state.
  - *Destination*: Priority is based on the transition destination state.
  - *Arena*: Priority is based on transition arena.

- *Scheme*:
  - *Parent*: If the basis is higher in the state hierarchy, it has higher priority.
  - *Child*: If the basis is lower in the state hierarchy, it has higher priority.

The choice made for SCION semantics for this semantic aspect was influenced by SCXML. SCION first uses Source-Child priority to select priority enabled transitions; if there are still transitions which cannot be taken together, then SCION uses XML document order, where transitions with lower document order (i.e. specified earlier in the document) have higher priority. Because XML document order encodes a total ordering on the transitions, this entails that transition selection based on priority will always be deterministic.

It is now possible to describe how the examples from the previous section would execute in SCION.

In the example in Listing 2.22, the transition from `A1` to `B` would have priority over the transition from `A` to `C`, because the source of the first transition is lower in the state hierarchy. Therefore, on event `t1`, the system would finish the small-step in configuration {`B`}.

In the example in Listing 2.23, the transition from `A1` to `A2` would have priority over the transition from `B1` to `C1`, because, even though their source states have the same depth in the state hierarchy, the first transition occurs before the second in the XML document, therefore it has higher priority. Thus, the system would start in configuration {`A1,B1,C1`}, and on event `t1`, the system would finish the small-step in configuration {`A2,B1,C1`}.

Like the previous example, in the example in Listing 2.24, the transition from `A1` to `A2` would have priority over the transition from `B1` to `C`, because, even though the source states of both transitions have the same depth in the state hierarchy, the first transition occurs before the second in the XML document, and it therefore has higher priority. Thus, the system would start in configuration {`A1,B1`}, and on event `t1`, the system would finish the small-step in configuration {`A2,B1`}.

Likewise, in the example in Listing 2.25, the transition from `A1` to `C` would have priority over the transition from `B1` to `C2`, because, even though their source states have the same depth in the state hierarchy, the first transition occurs before the second in the XML document, therefore it has higher priority. Thus, the system would start in configuration {`A1,B1`}, and on event `t1`, the system would finish the small-step in configuration {`C1`}.

2.5.8 Concurrency

Finally, there are two semantic aspects concerning AND states.

```
<scxml>
    <parallel id="P">
        <state id="A">
            <state id="A1">
                <transition target="A2" event="t1"/>
            </state>
            </state id="A2">
        </state>
        <state id="B">
            <state id="B1">
                <transition target="B2" event="t1"/>
            </state>
            </state id="B2">
        </state>
    </parallel>
</scxml>
```

Listing 2.26: Example of multiple transitions in orthogonal components that would be enabled in a single small-step

Number of transitions

The first semantic aspect describes how many transition occurrences can be executed in a small-step, given a set of enabled transitions. There are two choices for this:

- *Single*: Only a single transition can be enabled in a small-step.
- *Multiple*: Multiple transitions can be enabled in a small-step

SCION uses Multiple semantics. This is illustrated in Listing 2.26 and Figure 2.15:

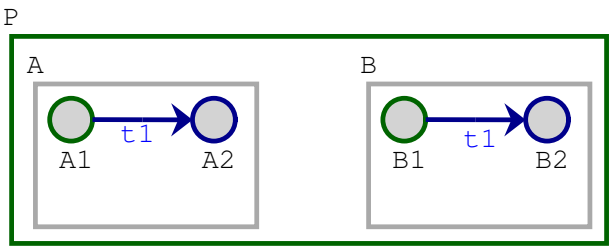

Figure 2.15: Example of multiple transitions in orthogonal components that would be enabled in a single small-step

In this example, the system would start in configuration {`A1`,`B1`}. On receiving event `t1`, in the first small-step, the transitions from `A1` to `A2` and from `B1` to `B2` would be enabled. If Multiple semantics are chosen, then both transitions would be taken in the first step. The system would then finish the first small-step in configuration {`A2`,`B2`}.

```
<scxml>
    <parallel id="P">
        <state id="A">
            <state id="A1">
                <transition target="A2" event="t1">
                    <log expr="'foo'"/>
                </transition>
            </state>
            </state id="A2">
        </state>
        <state id="B">
            <state id="B1">
                <transition target="B2" event="t1">
                    <log expr="'bar'"/>
                </transition>
            </state>
            </state id="B2">
        </state>
    </parallel>
</scxml>
```

Listing 2.27: SCXML example illustrating order of transitions

If Single semantics are chosen, then only one of the transitions could be taken in the first small-step. The transition from `A1` to `A2` would have priority, as the source states of both transitions have the same depth, and the first transition occurs earlier in the document, therefore the system's configuration at the end of the first small-step would be {`A2,B1`}. Assuming the Event Lifeline semantics are Next Small Step for interface events and Maximality semantics are Implicit, then the event `t1` would not be sensed in the following small-step, no transitions would be enabled, and the big-step would complete with the system in configuration {`A2,B1`}.

Order of transitions

The second semantic aspect describes in what order a set of enabled transitions are executed. There are two main choices for this:

- No order is specified.
- Order is explicitly defined.

The choice made for SCION semantics was again influenced by SCXML. SCION uses document order of the XML `<transition>` nodes in order to explicitly define the order in which transitions will be executed, such that transitions declared earlier in the document will be executed before those declared later in the document.

This is illustrated in Listing 2.27. In this example, the system would start in configuration {`A1,B1`}. On receiving event `t1`, in the first small-step, the transitions from `A1` to `A2` and

from `B1` to `B2` would be enabled. If Multiple Transition semantics are chosen, then both transitions would be taken in the first small-step. In SCION, as order is explicitly defined based on document order, the `<log>` action of the transition from `A1` to `A2` would be executed first, printing string "foo". The `<log>` action of the transition from `B1` to `B2` would then be executed, printing string "bar". The system would end in configuration {`A2,B2`}.

The other semantic choice for Order of Transitions would lead to non-deterministic behaviour by definition, so the system may evaluate the transitions in any order, printing "foo bar" or "bar foo".

Chapter 3

Pseudocode for SCION step algorithm

This chapter describes a canonical algorithm which implements the semantics of SCION described in the previous sections. This algorithm meant to play a similar role to the "Algorithm for SCXML Interpretation" presented in the SCXML specification. Furthermore, Chapter 4 refers to specific sections of this algorithm in order to describe strategies for optimizing its execution under ECMAScript.

Where possible, in the pseudocode algorithm, if a particular semantic option from "Big-Step Semantics" has influenced the pseudocode implementation, it is noted in the function description and in the code comments.

The pseudocode is written in a Python-inspired syntax. Indentation is used to delimit block scope. Tuples are often returned from functions, and simple destructuring assignments are used to initialize variables from the returned results. List data structures are initialized using the "`[]`" syntax.

3.1 Data Structures

The following data structures are used in the algorithm.

- *Set*: An unordered collection of objects. In particular, note that union and difference methods mutate the set, such that it is updated in-place.
- *List*: An ordered list of objects.
- *Map*: A map of string keys to object values.
- *Queue*: A First-in-First-Out (FIFO), ordered data structure.
- *Event*: A structure with the following properties:
  - *name* : String
  - *data* : Object

3.2 Constants

The following state types are enumerated constants. The terms "parallel" and "composite" are to indicate AND and OR states to avoid confusion with the boolean "and" and "or" operators.

- PARALLEL
- COMPOSITE
- HISTORY
- BASIC
- INITIAL

3.3 Global Objects

The following variables are global to the functions in the algorithm:

- *configuration*: A basic configuration, which is a Set containing only basic and initial states.
- *historyValue*: Map from a history state id string to a basic configuration.
- *innerEventQueue*: Queue containing Sets of Events.
- *datamodel*: String-to-Object Map.

3.4 Utility Functions

The following utility functions are used in the step algorithm. The implementation of these depends on the underlying representation of the Statecharts model.

- `getAncestors(s1,s2)` : Returns the ancestors of state `s1` up to, but not including state `s2`.
- `getDepth(s)` : Returns the depth of a state `s`.
- `getArena(t)` : Returns the arena state of a transition `t`.
- `isOrthogonalTo(s1,s2)` : Returns a boolean value if state `s1` is orthogonal to state `s2`.

```
procedure init():
  configuration = new Set()
  historyValue = new Map()
  innerEventQueue = []
```

Listing 3.1: procedure `init`

3.5 procedure init

Procedure `init` initializes the global data structures and is called when the interpreter is first instantiated, and before any events are sent.

3.6 procedure start

Procedure `start` begins execution of the Statecharts model. This sets the initial configuration, executes top-level scripts, initializes the datamodel, and performs the initial big-step.

```
procedure start(model)
  configuration.add(model.root.initial)

  #initialize top-level datamodel expressions. simple eval
  for script in model.scripts
    evaluateScript(script)

  for item in model.datamodel
    if item.expr then datamodel[item.location] = eval(item.expr)

  performBigStep()
```

Listing 3.2: procedure `start`

3.7 procedure performBigStep

Procedure `performBigStep` accepts an event as input. It will loop, selecting transitions and performing small-steps, until no transitions are selected, at which point the big-step is complete.

Procedure `performBigStep` implements *Maximality: Take-Many* semantics.

3.8 procedure performSmallStep

Procedure `performSmallStep` performs a single small step. First, it selects priority enabled transitions, given the datamodel and present events for the current small-step. Next, it

```
procedure performBigStep(e)
  if e then innerEventQueue.push(new Set([e]))

  keepGoing = true

  # Semantic Choice : Maximality: Take-Many
  while keepGoing
    if innerEventQueue.isEmpty()
      eventSet = innerEventQueue.shift()
    else
      eventSet = new Set()

    #create new datamodel cache for the next small step
    datamodelForNextStep = new Map()

    selectedTransitions = performSmallStep(eventSet,datamodelForNextStep)

    keepGoing = not selectedTransitions.isEmpty()
```

Listing 3.3: procedure `performBigStep`

computes states that are exited and states that are entered. Next it processes states that are exited, in order of depth, which includes updating any history states of the states exited, and performs exit actions. Next, transition actions are executed in document order. Then, enter actions are executed in reverse depth order for each state entered. Next, the configuration is updated, such that basic states which are exited are removed from the configuration, and basic states that are entered are added to the configuration. Next, the set of internal events that have been sent by a `<raise>` are added to the `innerEventQueue`, so that they will be active in the next small-step. Finally, the datamodel is updated with the values that have been set in the current small-step.

Procedure `performSmallStep` implements the following semantic options:

- *Concurrency*: Number of transitions: Multiple
- *Concurrency*: Order of transitions: Explicitly defined
- *Event Lifelines*: Next small-step

3.9 function getStatesExited

Function `getStatesExited` computes the states that have been exited, given a set of transitions. This function loops through the given set of transitions, computes the transition arena and its descendants, and adds to set `basicStatesExited` states that are both in the current configuration and descendants of the arena. Because the configuration contains only basic states, the states added to `basicStatesExited` will be basic states. To compute AND

```
procedure performSmallStep(eventSet,datamodelForNextStep)

  selectedTransitions = selectTransitions(eventSet,datamodelForNextStep)

  # if selectedTransitions is empty, we have reached a stable state,
    # and the big-step will stop, otherwise will continue
  if not selectedTransitions.isEmpty()
    [basicStatesExited,statesExited] =
            getStatesExited(selectedTransitions)
    [basicStatesEntered,statesEntered] =
            getStatesEntered(selectedTransitions)

    eventsToAddToInnerQueue = new Set()

    #update history states
    for state in statesExited

      #peform exit actions
      for action in state.onexit
        evaluateAction(action,eventSet,datamodelForNextStep,
                    eventsToAddToInnerQueue)
      #update history
      if state.history
        if state.history.isDeep
          f = lambda s0 : s0.kind is BASIC and s0 in getDescendants(state)
        else
          f = lambda s0 : s0.parent is state
        historyValue[state.history.id] = statesExited.filter(f)

    # Semantic Choice : Concurrency: Number of transitions: Multiple
    # Semantic Choice : Concurrency: Order of transitions: Explicitly defined
    sortedTransitions = selectedTransitions.sort(
            lambda (t1,t2) : t1.documentOrder - t2.documentOrder)

    for transition in sortedTransitions
      for action in transition.actions
        evaluateAction(action,eventSet,datamodelForNextStep,eventsToAddToInnerQueue)

    for state in statesEntered
      for action in state.onentry
        evaluateAction(action,eventSet,datamodelForNextStep,eventsToAddToInnerQueue)

    #update configuration by removing basic states exited,
        #and adding basic states entered
    configuration.difference(basicStatesExited)
    configuration.union(basicStatesEntered)

    #add set of generated events to the innerEventQueue
    # Semantic Choice : Event Lifelines: Next small-step
    if not eventsToAddToInnerQueue.isEmpty()
      innerEventQueue.push(eventsToAddToInnerQueue)

    #update the datamodel
    for key in datamodelForNextStep.keys
      datamodel[key] = datamodelForNextStep[key]

  return selectedTransitions
```

Listing 3.4: procedure `performSmallStep`

and OR states exited, the ancestors of the basic states exited, up to, but not including the arena, are added to `statesExited`, which contains both basic and non-basic states. The `statesExited` set is then sorted by depth and returned.

```
function getStatesExited(transitions)
  statesExited = new Set()
  basicStatesExited = new Set()

  for transition in transitions
    lca = getArena(transition)
    desc = getDescendants(lca)

    for state in configuration
      if state in desc
        basicStatesExited.add(state)
        statesExited.add(state)
        for anc in getAncestors(state,lca)
          statesExited.add(anc)

  sortedStatesExited = statesExited.sort(
        lambda (s1,s2) :  getDepth(s2) - getDepth(s1))

  return [basicStatesExited,sortedStatesExited]
```

Listing 3.5: `procedure getStatesExited`

3.10 function getStatesEntered

Function `getStatesEntered` computes the states that have been entered, given a set of transitions. First, all state targets of the given transition set are added to `statesToRecursivelyAdd`. Next, the algorithm loops so that composite states with initial sub-states are added to `statesToRecursivelyAdd`. Furthermore, at each step of the loop, the algorithm searches for children of parallel states without descendants in `statesToEnter`, as it is possible that some descendants of parallel states had been added to `statesToEnter`, but other children of those parallel states had not yet been added. This subroutine ensures that there will not exist any parallel states in `statesToEnter` whose children are not also in `statesToEnter`. If no such parallel states exist, then `statesToRecursivelyAdd` will be empty at the end of the loop, and the loop will terminate. Finally, all states in `statesEntered` are sorted by depth and returned.

3.11 procedure recursivelyAddStatesToEnter

Procedure `recursivelyAddStatesToEnter` is called by procedure `getStatesEntered`. It accepts a state `s`, and lists of states `statesToEnter` and `basicStatesToEnter`, which are

```
function getStatesEntered(transitions)
  statesToRecursivelyAdd = []
  for transition in transitions
    for state in transition.targets
      statesToRecursivelyAdd.push(state)

  statesToEnter = new Set()
  basicStatesToEnter = new Set()

  while not statesToRecursivelyAdd.isEmpty()
    for state in statesToRecursivelyAdd
      recursivelyAddStatesToEnter(state,statesToEnter,
                basicStatesToEnter)

    #add children of parallel states that are not already in
        #statesToEnter to statesToRecursivelyAdd
    statesToRecursivelyAdd  = []
    for s in statesToEnter
      if s.kind is PARALLEL
        for c in s.children
          if c.kind isnt HISTORY and not c in statesToEnter
            statesToRecursivelyAdd.push(c)

  sortedStatesEntered = statesToEnter.sort(
        lambda (s1,s2) : getDepth(s1) - getDepth(s2))

  return [basicStatesToEnter,sortedStatesEntered]
```

Listing 3.6: function getStatesEntered

mutated in each recursive call. If `s` is a history state, then if its previous history value was set, the history value is added to `statesToEnter` and `basicStatesToEnter`; otherwise, the default history state is used. If `s` is not a history state, then `s` is added to `statesToEnter`. Furthermore, if `s` is a parallel state, then the children of `s` are also added, and `s` is called recursively. If `s` is a composite state, then the initial state of `s` is added recursively. Finally, if `s` is a basic or initial state, then `s` is added to `basicStatesToEnter`.

3.12 function selectTransitions

Function `selectTransitions` accepts an eventSet and a datamodel, and returns a set of priority enabled transitions.

`selectTransitions` first computes a full configuration from the basic configuration. Transitions are then selected for each state given the set of events and the datamodel, and each transition in the returned set of enabled transitions is added to the set of all enabled transitions. Finally, `selectPriorityEnabledTransitions` is called to select transitions by their priority, and the result is returned.

```
procedure recursivelyAddStatesToEnter(s,statesToEnter,
    basicStatesToEnter)
  if s.kind is HISTORY
    if historyValue.hasKey(s.id)
      for historyState in historyValue[s.id]
        recursivelyAddStatesToEnter(historyState,statesToEnter,
                    basicStatesToEnter)
    else
      statesToEnter.add(s)
      basicStatesToEnter.add(s)
  else
    statesToEnter.add(s)

    if s.kind is PARALLEL
      for child in s.children
        if not (child.kind is HISTORY)
                    #don't enter history by default
          recursivelyAddStatesToEnter(child,statesToEnter,
                        basicStatesToEnter)

    else if s.kind is COMPOSITE

      recursivelyAddStatesToEnter(s.initial,statesToEnter,
                basicStatesToEnter)

    else if s.kind is INITIAL or s.kind is BASIC or s.kind is FINAL
      basicStatesToEnter.add(s)
```

Listing 3.7: function `recursivelyAddStatesToEnter`

3.13 function getActiveTransitions

Function `getActiveTransitions` accepts a `state` and a set of `events`, and returns a set of enabled transitions.

3.14 function selectPriorityEnabledTransitions

Function `selectPriorityEnabledTransitions` accepts a set of transitions, and returns a set of priority enabled transitions.

`selectPriorityEnabledTransitions` first gets `consistentTransitions` and `inconsistentTransitionsPairs` from function `getInconsistentTransitions`, and adds the `consistentTransitions` to the set of `priorityEnabledTransitions`. It then loops until `inconsistentTransitionsPairs` is empty, at each step creating a new set of transitions, and adding to it the transition with higher priority given each pair of transitions in `inconsistentTransitionsPairs`. `getInconsistentTransitions` is then called again to update `consistentTransitions` and `inconsistentTransitionsPairs` given the new

```
function selectTransitions(eventSet,datamodelForNextStep)

  states = new Set()

  #get full configuration, unordered
  #this means we may select transitions from parents before children
  for basicState in configuration
    state.add(basicState)

    for ancestor in getAncestors(basicState)
      state.add(ancestor)

  enabledTransitions = new Set()
  for state in states
    for transition in getActiveTransitions(state,eventSet)
      enabledTransitions.add(transition)

  return selectPriorityEnabledTransitions(enabledTransitions)
```

Listing 3.8: function `selectTransitions`

transition set. The updated `consistentTransitions` will then be added to the set of `priorityEnabledTransitions`. This will eventually reach a stable state where there are no longer any transitions pairs which conflict, at which point the set of priority enabled transitions will have been computed, and will be returned.

3.15 function getTransitionWithHigherSourceChildPriority

Function `getTransitionWithHigherSourceChildPriority` accepts a pair of transitions, and compares them based first on depth, then based on document order, and returns the one with the highest priority.

This function implements the following semantic option: *Priority: Source-Child*.

3.16 function getInconsistentTransitions

Function `getInconsistentTransitions` accepts a set of transitions, and returns a set of consistent transitions (`consistentTransitions`), and a set of inconsistent transition pairs. `getInconsistentTransitions` compares transitions pairwise to determine if the pair are *arena orthogonal*, which is to say, if their arenas are orthogonal. Those transitions that are not arena orthogonal are added to `allInconsistentTransitions`, and transition pairs that are not arena orthogonal are added to `inconsistentTransitionsPairs`. The transitions not in conflict are then computed through set difference operation with the set of all transitions,

```
function getActiveTransitions(state,events)
  transitions = new Set()

  for t in state.transitions
    if (t does not have a trigger or
            events.some(lambda e : nameMatch(e.name,t.event))) and
            (t does not have a condition or
                the cond of t evaluates to true):
      transitions.add(t)

  return transitions
```

Listing 3.9: function `getActiveTransitions`

and the result is assigned to `consistentTransitions`. Finally, `consistentTransitions` and `inconsistentTransitionsPairs` are returned.

This function implements the following semantic options:

- *Transition Consistency*: *Small-step consistency: Source/Destination Orthogonal*
- *Interrupt Transitions and Preemption*: *Non-preemptive*

3.17 function isArenaOrthogonal

Function `isArenaOrthogonal` takes two transitions and determines if they have the same arena.

3.18 procedure gen

Procedure `gen` implement's the statechart interpreter's interface to the environment. When implemented in single-threaded environments, such as the browser environment, this interface can be very simple, in essence passing the given event straight through to the `performBigStep` procedure, as `gen` is passed the thread of execution when it is called.

3.19 procedure evaluateAction

This evaluates actions. Pseudocode for the `<send>`, `<raise>`, `<assign>`, `<script>`, and `<log>` actions are shown.

```
function selectPriorityEnabledTransitions(transitions)
  priorityEnabledTransitions = new Set()

  [consistentTransitions, inconsistentTransitionsPairs] =
        getInconsistentTransitions(transitions)

  priorityEnabledTransitions.union(consistentTransitions)

  while not inconsistentTransitionsPairs.isEmpty()

    transitions = new Set()

    for transitionPair in inconsistentTransitionsPairs
      transitions.add(
                getTransitionWithHigherSourceChildPriority(
                    transitionPair))

    [consistentTransitions, inconsistentTransitionsPairs] =
            getInconsistentTransitions(transitions)

    priorityEnabledTransitions.union(consistentTransitions)


  return priorityEnabledTransitions
```

Listing 3.10: function `selectPriorityEnabledTransitions`

```
function getTransitionWithHigherSourceChildPriority(t1,t2)
  if getDepth(t1.source) < getDepth(t2.source)
    return t2
  else if getDepth(t2.source) < getDepth(t1.source)
    return t1
  else
    if t1.documentOrder < t2.documentOrder
      return t1
    else
      return t2
```

Listing 3.11: `getTransitionWithHigherSourceChildPriority`

```
function getInconsistentTransitions(transitions)

  allInconsistentTransitions = new Set()
  inconsistentTransitionsPairs = new Set()

  transitionList = transitions.toList()

  for i in range(0,transitionList.length)
    for j in range(i+1,transitionList.length)
      t1 = transitionList[i]
      t2 = transitionList[j]

      if not isArenaOrthogonal(t1,t2)
        allInconsistentTransitions.add(t1)
        allInconsistentTransitions.add(t2)
        inconsistentTransitionsPairs.add([t1,t2])

  consistentTransitions =
        transitions.difference(allInconsistentTransitions)

  return [consistentTransitions,inconsistentTransitionsPairs]
```

Listing 3.12: function `getInconsistentTransitions`

```
function isArenaOrthogonal(t1,t2)
  t1Arena = getArena(t1)
  t2Arena = getArena(t2)
  return isOrthogonalTo(t1Arena,t2Arena)
```

Listing 3.13: function `isArenaOrthogonal`

```
procedure gen(e)
  performBigStep(e)
```

Listing 3.14: procedure `gen`

```
procedure evaluateAction(action,eventSet,datamodelForNextStep,
    eventsToAddToInnerQueue)
    switch action.type
        case "script":
            evaluateAction(action,
                datamodelForNextStep,eventSet)
        case "log":
            console.log(evaluateAction(action.expr,
                datamodelForNextStep,eventSet))
        case "send":
            #this delegates sending to the environment
            setTimeout(lambda : gen(action.event),action.delay)
        case "raise"
            eventsToAddToInnerQueue.add(new Event(action.event,data)
        case "assign"
            datamodelForNextStep[action.location] =
                evaluateAction(action.expr,
                    datamodelForNextStep,eventSet)
```

Listing 3.15: procedure `evaluateAction`

Chapter 4

Optimizing SCION for ECMAScript and the World Wide Web

4.1 Problem Statement

The primary goal of SCION was to create a Statecharts interpreter that was well-suited for interactive Web-based user interface development. Because interactivity in Web-based user interfaces is implemented primarily in ECMAScript, this entailed developing a Statecharts interpreter optimized for ECMAScript.

This problem statement may be further refined, as the restrictions imposed by the Web browser environment determine the factors for which one should optimize. Specifically, event dispatch time, memory usage, and generated code size were the factors that were most likely to impact the usability of Web UIs. Fast event dispatch is required in order to handle high-fidelity UI events. For example, when dragging the mouse, mouse movement events are fired at a high rate. If a Statecharts implementation is unable to handle high-fidelity events as they occur in real time, then events may begin to queue, which may result in visible UI "lag". This would, in turn, negatively impact usability. Generated code size must also be kept to a minimum, as all code must be downloaded over a network, most often the Internet, and thus a large code payload increases the amount of time required before a UI may initially be used. Finally, reducing memory usage is also an important consideration, particularly when one considers the proliferation of ECMAScript-enabled Web browsers on mobile and embedded devices, which have more limited memory than desktop devices.

In order to optimize SCION for ECMAScript, four high-level Statecharts optimization strategies were chosen. A configuration of the available options for each of these strategies is referred to as an *optimization profile*. Every possible optimization profile was tested against a suite of benchmarks and in different Web browsers, in order to seek optimization profiles that were outliers in terms of performance and memory usage. Generated code size was ultimately not a concern due to the way SCION was architected, and this is elaborated in Section 4.4.

4.2 Project Background

Before developing SCION, the effort to develop a Statecharts implementation for ECMAScript had gone through several major iterations. First, in 2008, I developed an ECMAScript backend for Thomas Feng's SCC Statecharts compiler[Fen04]. This worked well for internal UI research projects, but had the effect of strongly coupling Statecharts development with the AToM3 DCharts environment, which I felt could be problematic for outside developers.

Next, in 2010, I developed *scxml-js*, a Statecharts-to-JavaScript compiler based on SCXML, as a course project for McGill University course COMP-621, taught by Laurie Hendren, and continued its development as a Google Summer of Code project for the Apache Software Foundation. This project had the advantage of being based on SCXML, a W3C draft specification, and of better integration with existing Web technologies. Still, scxml-js had several disadvantages, the primary being that it would generate large amounts of boilerplate code for each compiled SCXML document, and that some aspects of its semantics were not clearly defined and tested, leading sometimes to unexpected behaviour.

For these reasons, in the summer of 2011, the decision was made to rewrite scxml-js, and this new project became *SCION*. SCION aimed to improve on scxml-js in several ways. First, SCION used a "pluggable" software architecture, built around a fast interpreter core, which could be further optimized through injection of optimized data structures at runtime. Second, SCION leveraged the dynamic language features of ECMAScript in order to generate optimized data structures dynamically at runtime, so that generated code did not need to be compiled in advance and downloaded over the network. Finally, SCION started with a well-defined semantics, and used test-driven development to verify the correct implementation of this semantics. These improvements are further explained in Section 4.4.

SCION has been released under an Apache 2.0 open source license, and is available at the following URL: `https://github.com/jbeard4/SCION`

4.3 Statecharts Optimization Strategies

This section describes four strategies for optimizing the SCION step algorithm outlined in Chapter 3.

4.3.1 Transition Selection

Selecting a set of enabled transitions, given a state and a set of events, is a fundamental operation of a Statecharts interpreter. Section 3.13 illustrates a simple transition selection algorithm as a part of the SCION step algorithm. This transition selection algorithm may be further optimized to use data structures derived through static analysis of the Statecharts model. This section outlines three options for transition selection optimization. Each optimization strategy provides an example based on the SCXML document in Listing 4.1. Note that this example attaches "id" attributes to transitions, which is not normally a part of the standard SCXML syntax, and is for demonstration purpose only. These transition ids are

mapped to variables of the same name in the transition selection examples, such that they reference "transition objects" that the interpreter can use to perform the enabled transition.

```
<scxml>
    <state id="A">
        <transition target="B" event="e1" id="transition1"/>
        <transition target="B" event="e2" id="transition2"/>
        <transition target="B" event="e2" id="transition3"/>
    </state>
    <state id="B">
        <transition target="A" event="e3" id="transition4"/>
    </state>
</scxml>
```

Listing 4.1: Example SCXML document used to illustrate transition selection optimization strategies

Nested Switch Statement

In the *Nested Switch Statement* transition selection option, states and triggers are encoded as enumerations, or, if native enumerations are not available in the target language, an appropriate encoding such as integer constants. On event dispatch, each state in the configuration is passed to a switch statement, and the event is then sent to another switch statement nested in the case statement corresponding to the current state. A list of transition objects is then returned from the case statement corresponding to the input event [SZ01, Sam02]. An example in Java-like pseudocode of a transition selection function built using a nested switch statement can be seen in Listing 4.2.

This strategy has been used in the Rhapsody Statecharts implementation [SZ01].

State Table

In the *State Table* transition selection option, set of states and triggers of a Statecharts model are mapped to unique integer indexes, starting from index 0. These indexes correspond to indexes in a "State Table," which is a $|S| \times |T|$ size matrix, where S is the set of states, and T the set of triggers. Each cell in the matrix contains a possibly empty set of transitions. On event dispatch, the interpreter indexes into the State Table using the indexes of the given state and event, in order to obtain the transitions enabled by that state and event [Dou00, Sam02]. An example of a State Table data structure can be seen in Table 4.1.

The advantage of this strategy is that it has a fast, constant dispatch time. The disadvantage is that the State Table data structure tends to be large and sparse, which is wasteful in terms of memory.

This strategy has been used in the SCC Statecharts compiler [Fen04].

```
function switchSelector(stateId,eventName){
    switch(stateId){
        case "A":
            switch(eventName){
                case "e1":
                    return [transition1];
                case "e2":
                    return [transition2,transition3];
                default:
                    return [];
            }
        case "B":
            switch(eventName){
                case "e3":
                    return [transition4];
                default:
                    return [];
            }
        default:
            throw Error("State not found")
    }
}
```

Listing 4.2: Pseudocode Nested Switch Statement

states/events	e1	e2	e3
A	[transition1]	[transition2,transition3]	[]
B	[]	[]	[transition4]

Table 4.1: State Table

State Design Pattern

In the *State Design Pattern* transition selection option, each state is mapped to a unique class, and triggers are mapped to methods on each state class. The configuration is captured by a set of instances of these state classes. Hierarchy is encoded via class inheritance [Nia05, Sam02], or, in ECMAScript, via prototypal inheritance. A pseudocode example of this can be seen in Listing 4.3 using a pseudo-Java syntax.

The advantage of this approach is that it leverages the object-oriented language feature of inheritance in order to implicitly encode Source-Child transition priority semantics, described in Section 2.5.7. This is illustrated in an additional example, in Listings 4.4 and 4.5, which shows a transition originating from an inner basic state taking priority over a transition originating from an outer OR state on event `e1`.

The State Design Pattern's impact on performance and memory usage is unclear, and is generally dependent on the underlying language implementation of the Statecharts interpreter.

This strategy has been used in the SMC state machine compiler [Rap10].

```
class A {
    public e1(){
        return [transition1];
    }

    public e2(){
        return [transition2,transition3];
    }

    public e3(){
        return [];
    }
}

class B {
    public e1(){
        return [];
    }

    public e2(){
        return [];
    }

    public e3(){
        return [transition4];
    }
}
```

Listing 4.3: Pseudocode State Design Pattern

4.3.2 Set Data Structure

A *Set* is an unordered collection of unique objects. The Set interface used by the SCION step algorithm is specified in pseudo-Java syntax in Listing 4.6.

Sets are a fundamental data structure for Statecharts interpretation, as a Set is used to store the system's configuration, which is updated in each small-step. Sets are also used for a number of other purposes in the SCION step algorithm, including storing the events that can be sensed in each small-step, and the enabled transitions derived in a small-step. The performance of an implementation of the SCION step algorithm is therefore likely to be highly dependent the implementation of the Set data structure.

While many programming languages provide built-in Set data structures, either as a core part of the language, or through their standard library (for example, Java's *java.util.HashSet* implementation, included as part of the standard library [has], and python's built-in set data-type [pyt]), ECMAScript does not currently provide a native Set implementation,[1] and so

[1] A native Set data structure has been proposed as a part of the emerging ECMAScript 6 standard. However, at the time of this writing, the ECMAScript 6 standard has not been completed, and this language feature is only available in recent versions of Mozilla's implementation of ECMAScript [Int12].

```
<scxml>
  <state id="A">
    <state id="A1">
      <transition target="B" event="e1" id="transition1"/>
    </state>
    <transition target="B" event="e1" id="transition2"/>
  </state>
  <state id="B"/>
</scxml>
```

Listing 4.4: Additional example SCXML document used to illustrate State Design Pattern transition selection optimization strategy

```
class A {
    public e1(){
        return [transition2];
    }
}

class A1 extends A {
    public e1(){
        return [transition1];
    }
}

class B {
    public e1(){
        return [];
    }
}
```

Listing 4.5: Additional Pseudocode State Design Pattern example

there is variability regarding how a Set may be implemented in ECMAScript.

ECMAScript Native Data Structures

ECMAScript provides two high-level data structures upon which a Set can be based. These high-level data structures are Arrays and Objects.

An ECMAScript Object is a Map-like data structure, which stores key-value pairs [obj]. The keys must be strings (when given any other type, the string representation of that type is used as the key), and values can be of any type, including other complex types such as Object and Array. The underlying implementation of Object may be implemented as a HashTable, and so one would expect an ECMAScript Object to provide $O(1)$ expected time for operations involving key lookup, and $O(n)$ worst-case time[CSRL01]. A Set based on an ECMAScript object would therefore typically inherit these expected and worst-case times, as all Set operations rely on key lookup.

```
interface Set {
    //add an object to the set
    void add(Object x);

    //remove an object, and return true
    //if the object existed in the set and was removed
    boolean remove(Object x);

    //Union the set with another set.
    //This updates the current set in-place and returns itself.
    Set union(Set s);

    //Implements set-difference operation.
    //This updates the current set in-place and returns itself.
    Set difference(Set s);

    //returns true if x is in the set
    boolean contains(Object x);

    //returns true if the set is empty
    boolean isEmpty();

    //returns true if s2 is equal to the current set.
    boolean equals(Set s2);
}
```

Listing 4.6: Set Interface

An ECMAScript Array, on the other hand, is a list-like data structure, which has a dynamic property, `length`, that tracks the number of elements the structure contains, and maps integer indexes to ECMAScript values [arr]. In the ECMAScript object model, an Array is also an Object, but in most ECMAScript implementations, the ECMAScript Array is optimized to delegate to a native array in the underlying language implementation (usually C++), thus making operations on the ECMAScript Array have native array-like performance. For example, indexing into an ECMAScript Array should be a fast operation, and have a $O(1)$ worst-case time, as it would on a native array. Iterating through an ECMAScript Array should also be faster than iterating through the key-value pairs of an Object.

Optimization Using Static Analysis

The SCION step algorithm uses sets to contain states, transitions, and triggers. Because all objects that could possibly go into these sets (the *universe of keys*) are known at the time the Statecharts model is parsed, and are *static*, such that the keys do not change at run-time, one can utilize *perfect hashing* to optimize performance of worst-case lookup times. A hashing technique is defined to be perfect if the worst-case number of memory accesses

required to perform a search is $O(1)$, as opposed to $O(n)$ [CSRL01].

To accomplish this, each object in the universe of keys is assigned a hash value in advance, such that the integer hashes start at value 0, and increase incrementally. The universe of keys can then be used to initialize the in-memory data structure that allows $O(1)$ worst-case search time.

Perfect hashing was implemented in ECMAScript in two ways.

Boolean Array In this technique, an ECMAScript Array was used, although not in the same way as the Array-based set technique described above. In the aforementioned approach, adding an object to an Array-based set required appending that object to the Array using the ECMAScript *Array.push* method; and removing an object from an Array-based set involved retrieving that object's index in the Array, using *Array.indexOf*, deleting the object at that index and left-shifting subsequent objects by using ECMAScript's *Array.splice* method.

In the perfect hashing approach, an Array is created such that its length is initialized to the size of the universe of keys, which is passed in as an argument to the Set constructor. Each object's hash can be used directly as Array indexes. Therefore, in order to add an object to the set, that object's hash is used to index into the Array (an $O(1)$ operation), and set the value at that index to boolean *true*. In order to remove an object from the set, the object's hash is used to index into the Array and set the value at that index to boolean *false*.

Bit Vector ECMAScript has a single numeric data type, Number. Number can be used to perform bit operations, in which case it behaves as a 32-bit integer [bit].

A *bit vector* is like a Boolean Array, except that instead of using an Array to encode a list of boolean values, the bits in a 32-bit integer are used instead. Like the Boolean Array, each object's integer hash can be used directly as an index, which in this case can be used in conjunction with bit masking operations on the bit vector set.

In the case that the size of the universe of keys is less than or equal to 32, a single Number can be used to encode the Set. In the case that a universe of keys larger than 32 is used, an Array of ECMAScript Numbers can be used, such that each Number can be used to store an additional 32 objects.

4.3.3 Transition Flattening Transformation

One condition for a transition to be enabled is for the transition's source state to reside in the model's full configuration (described in Section 2.3). Section 3.12 described a basic algorithm for selecting states to pass into the transition selection function. In this algorithm, the full configuration is derived from a basic configuration, and each state in the full configuration is passed in conjunction with the event set to the transition selection function in order to derive the set of enabled transitions for that small-step.

One possible optimization to this approach would be to "flatten" the transitions, such that every basic state would be made the source state for all transitions originating from

that basic state's ancestors. This means that the algorithm would no longer be required to look up transitions hierarchically, and instead would only need to iterate over the states in the basic configuration.

This flattening transformation must be performed a way that would respect SCION semantics concerning Transition Priority (described in Section 2.5.7). This can be accomplished by appending transitions to basic states in order of state hierarchy, followed by document order. This transformation is illustrated in Listings 4.7 and 4.8, which present a simple model before and after the flattening transformation is applied. One can see in the examples that SCION semantics are respected, as in both models, the transition from `A1` to `B` would have the highest priority given event `t`.

```
<scxml>
    <state id="A">
        <state id="A1">
            <transtition target="B" event="t"/>
            <transtition target="C" event="t"/>
        </state>
        <transtition target="D" event="t"/>
        <transtition target="E" event="t"/>
    </state>

    <state id="B"/>
    <state id="C"/>
    <state id="D"/>
    <state id="E"/>
</scxml>
```

Listing 4.7: Simple model before flattening transformation

```
<scxml>
    <state id="A">
        <state id="A1">
            <transtition target="B" event="t"/>
            <transtition target="C" event="t"/>
            <transtition target="D" event="t"/>
            <transtition target="E" event="t"/>
        </state>
    </state>

    <state id="B"/>
    <state id="C"/>
    <state id="D"/>
    <state id="E"/>
</scxml>
```

Listing 4.8: Simple model after flattening transformation

4.3.4 Cached Structural Information

The `getAncestors`, `getDescendants` and `getArena` functions described in Section 3.4, and used throughout the step algorithm, involve querying the Statecharts model for a given state's ancestors and descendants, as well as a transition's arena. This information can be computed through static analysis when parsing the Statecharts model, and cached by the interpreter so that it does not need to be computed at run time. This can significantly improve the performance of the interpretation algorithm, but potentially may lead to increased memory usage as additional information must be kept by the interpreter.

4.4 SCION Architecture

Before discussing the experimental framework in which the above optimizations were evaluated, it is important to provide an overview of SCION's architecture.

SCION is is built around a core interpreter class, which is able to accept optimized data structures as optional arguments to its constructor when the interpreter is instantiated. This is illustrated in Listing 4.9 using a pseudo-Java syntax.

```
interface SCION{
    //constructor
    public SCION(
        SCXMLModel model,
        Optimizations opts
    );

    //method to start the machine
    public Configuration start();

    //method to send events
    public Configuration gen(String eventName, Object eventData);
}

interface Optimizations{
    Function transitionSelector;
    boolean isFlattend;
    Set TransitionSet;
    Set StateSet;
    Set BasicStateSe;
}
```

Listing 4.9: SCION Interpreter API

SCION's static analysis and code generation modules are also written in ECMAScript. This means optimized data structures may be generated in advance, and downloaded to the user's browser, or may be generated on the fly, in the user's browser. The time to generate these optimized data structures is negligible, taking less than a millisecond for all models

tested. The code generation modules themselves are relatively small, taking less than 1KB disk space when compressed. This means that on-the-fly code generation is cheap in terms of initial load time, and it is therefore no longer meaningful to discuss optimizing the size-on-disk of generated data structures, as they can be generated dynamically in the user's browser at runtime, without requiring the user to download the generated code over the network.

Furthermore, it should be noted that even though SCION was developed with the Web browser in mind, the interpreter core was implemented to be portable, without any dependencies on the Web browser environment. The implication of this is that SCION now works well in a number of ECMAScript environments, including server-side environments, such as node.js and Mozilla Rhino.

4.5 Experimental Framework

4.5.1 Optimization Profiles

Section 4.3 described four categories of optimizations, each with various options. The *Transition Selection* strategy has four options: *State Table*, which is simply referred to as *Table*; *Switch Statement*, which is simply referred to as *Switch*; *State Design Pattern*, which is simply referred to as *Class*, because of how it is usually implemented in object-oriented programming languages; and a default, unoptimized algorithm, referred to as *Default*, which is simply the straightforward implementation of the algorithm for transition selection described in Section 3.13. The Set category has four options: *Array Set*, *Object Set*, *Boolean Array Set*, and *Bit Vector Set*. Finally, *Transition Flattening Transformation* and *Cached Structural Information* are both optimizations that can be enabled or disabled, which is simply referred to as *True*, to describe the strategy as being enabled, and *False* to describe the strategy as being disabled. This yields $4 \times 4 \times 2 \times 2 = 64$ possible *optimization profiles*. The experimental framework sought to test the performance and memory usage of these optimization profiles on a set of tests cases divided into nine *test categories*, and executed in four different Web browsers, each using a unique ECMAScript interpreter.

4.5.2 Test Cases

While many programming languages have well-known or standard suites of tests for benchmarking performance (e.g. SunSpider for ECMAScript [sun], and SPEC for C [spe]), SCXML, as an emerging specification, does not yet have such a suite of tests.

The solution to this was to programmatically generate a suite of meaningful, reduced tests that would stress the SCION interpreter in various fundamental ways likely to be encountered in real-world usage scenarios.

A *test document* here is defined to be an SCXML document, and a *test script* is a separate document which specifies a list of events to send into the interpreter, as well as the expected resulting configuration after sending each event. A *test case* is a pair, consisting of one test document and one test script.

All test cases generated have a special property, which is that they can be *looped*, meaning that at the end of the test script, it should be possible to send the events in the test script again, resulting in the same system configurations, without resetting the interpreter.

Test cases were generated for nine separate *test categories*, where each test category was designed to test a specific and reduced aspect of the SCION interpreter. Each category generated tests according to a unique algorithm, and all algorithms were designed to accept a scalar integer that would increase the complexity of the generated test case. Each test category and its associated algorithm is described informally through examples in the following sections.

basic-states

The goal of the *basic-states* test category was to test the performance of a sparse, flat model with a variable number of basic states. The algorithm generates a basic state for each variable, and connects each state to the next state via a single transition, with a single trigger `t`. The last state in the sequence is made the source state of a transition targeting the initial state, so that a loop is formed. Examples of test SCXML documents for variable values 1 and 2 can be seen in Listings 4.10 and 4.11, respectively. The test script would simply send event `t` repeatedly.

```
<scxml>
  <state id="state-0">
    <transition target="state-0" event="t"/>
  </state>
</scxml>
```

Listing 4.10: Generated test document for *basic-states* with variable 1.

```
<scxml>
  <state id="state-0">
    <transition target="state-1" event="t"/>
  </state>
  <state id="state-1">
    <transition target="state-0" event="t"/>
  </state>
</scxml>
```

Listing 4.11: Generated test document for *basic-states* category for variable 2.

events

The *events* category is like *basic-states*, but the input variable not only specifies the number of states generated in a sequence, but also assigns a unique event to each transition. This

tests the effect of having a large set of states and events in a Statecharts model. This is illustrated in Listing 4.12.

The test script would specify that events `t-0`, `t-1`, and `t-2` be sent in a loop.

```
<scxml>
  <state id="default-state">
    <transition target="state-0" event="t-0"/>
  </state>
  <state id="state-0">
    <transition target="state-1" event="t-1"/>
  </state>
  <state id="state-1">
    <transition target="default-state" event="t-2"/>
  </state>
</scxml>
```

Listing 4.12: Generated test document for *events* for variable 3.

transitions

The *transitions* category was designed to test dense models that have few states, many transitions, and few events. The input variable determines the number of transitions which loop between two states (`default-state` and `the-other-state`), all of which have the same trigger `t`. An example test document can be seen in Listing 4.13 for input variable 3. The test script would specify that event `t` be sent in a loop, which would cause the system to transition between the two states.

```
<scxml>
  <state id="default-state">
    <transition target="the-other-state" event="t"/>
    <transition target="the-other-state" event="t"/>
    <transition target="the-other-state" event="t"/>
  </state>
  <state id="the-other-state">
    <transition target="default-state" event="t"/>
    <transition target="default-state" event="t"/>
    <transition target="default-state" event="t"/>
  </state>
</scxml>
```

Listing 4.13: Generated test document for transitions category for variable 3.

transitions2

Test category *transitions2* is like test category *transitions*, except that it generates a unique event for each transition emanating from a state. This is meant to test dense models that

also have a large number of unique events. An example of this is seen in Listing 4.14. The test script would specify that events `t-0`, `t-1`, and `t-2` be sent in a loop, causing the system to transition between the two states.

```
<scxml>
  <state id="default-state">
    <transition target="the-other-state" event="t-0"/>
    <transition target="the-other-state" event="t-1"/>
    <transition target="the-other-state" event="t-2"/>
  </state>
  <state id="the-other-state">
    <transition target="default-state" event="t-0"/>
    <transition target="default-state" event="t-1"/>
    <transition target="default-state" event="t-2"/>
  </state>
</scxml>
```

Listing 4.14: Generated test document for transitions2 category for variable 3.

depth

The *depth* test category is designed to test the role state hierarchy depth plays on performance. This test involves two main states: a single top-level state, and a basic state nested in a variable number of OR states, such that its depth is equal to the algorithm input variable. An example of this can be seen in Listing 4.15. The test script would specify that the event `t` be looped, causing the system to transition between the nested basic state `basic` and the outer default state `default-state`.

```
<scxml>
  <state id="default-state">
    <transition target="basic" event="t"/>
  </state>
  <state id="composite-0">
    <state id="composite-1">
      <state id="basic">
        <transition target="default-state" event="t"/>
      </state>
    </state>
  </state>
</scxml>
```

Listing 4.15: Generated test document for depth category for variable 2.

history-depth

The *history-depth* test category is like *depth*, but with the addition of a single top-level history state, and two basic states at the deepest level. Two basic states were used instead of one, so that history would meaningfully record the previous state that the system was in when the outermost OR state is exited. This test category is designed to test the effects of history in conjunction with deeply-nested OR states. This is illustrated in Listing 4.16.

The test script is slightly more complex for this test case, so that the history functionality can be fully exercised. This test script can be described as follows:

1. On event `in`, the system would transition to state `history`, which, the first time it is entered, would transition to state `basic1`, and subsequently would transition to the state that the system was in when state `composite-0` was last exited.
2. On event `t1`, the system would transition to state `basic2`.
3. On event `out`, the system would exit the nested OR states and transition to state `default-state`.
4. On event `in`, the system would transition back to `history`, which would return the system to `basic2`, the state that the system was in when `composite-0` was exited.
5. On event `t2`, the system would transition to state `basic1`.
6. On event `out`, the system would exit the nested OR states and transition to state `default-state`.

This sequence of events would then be looped.

concurrency

Test category *concurrency* is designed to test models with shallow AND states. The input variable determines the number of orthogonal components of the single top-level AND state. The contents of each orthogonal component are identical, where each component contains two basic states, and transitions which loop between the basic states on event `t`. An example of this is shown in Listing 4.17 for input variable 2. In this example, the event `t` would be looped, causing the system to loop between configurations {`substate-1-0,substate-1-1`} and {`substate-2-0,substate-2-1`}.

history-concurrency

Test category *history-concurrency* is like *concurrency*, but with a single top-level history state inside of the AND state. This test category is designed to test the effect of history in conjunction with shallow concurrency.

An example of this can be seen in Listing 4.18 for input variable 2. As in test category history-depth, the test script for this document is more complicated so that the history functionality can be fully exercised.

```
<scxml>
  <state id="default-state">
    <transition target="history" event="in"/>
  </state>
  <state id="composite-0">
    <state id="composite-1">
      <state id="composite-2">
        <state id="basic1">
          <transition target="basic2" event="t1"/>
        </state>
        <state id="basic2">
          <transition target="basic1" event="t2"/>
        </state>
      </state>
    </state>
    <history type="deep" id="history">
      <transition target="basic1"/>
    </history>
    <transition target="default-state" event="out"/>
  </state>
</scxml>
```

Listing 4.16: Generated test document for history-depth category for variable 3.

1. On event `in`, the system would transition to state `history`, which, the first time it is entered, would lead to configuration {`substate-1-0, substate-1-1`}, and subsequently to the configuration that the system was in when state `composite-0` was last exited.
2. On event `t1`, the system would transition to configuration {`substate-2-0, substate-2-1`}.
3. On event `out`, the system would exit the AND state and transition to `default-state`.
4. On event `in`, the system would transition back to `history`, which would return the system to configuration {`substate-2-0, substate-2-1`}, the configuration that the system was in when `parallel` was exited.
5. On event `t2`, the system would transition to configuration {`substate-1-0, substate-1-1`}.
6. On event `out`, the system would exit the outer AND state and transition to state `default-state`.

nested-parallel

Test category *nested-parallel* tests nested concurrency, where AND states are grouped inside of AND states at multiple levels. Each AND state has two orthogonal components, and the input variable determines the depth of this hierarchy.

```
<scxml>
  <parallel id="default-state">
    <state id="ortho-0">
      <state id="substate-1-0">
        <transition target="substate-2-0" event="t"/>
      </state>
      <state id="substate-2-0">
        <transition target="substate-1-0" event="t"/>
      </state>
    </state>
    <state id="ortho-1">
      <state id="substate-1-1">
        <transition target="substate-2-1" event="t"/>
      </state>
      <state id="substate-2-1">
        <transition target="substate-1-1" event="t"/>
      </state>
    </state>
  </parallel>
</scxml>
```

Listing 4.17: Generated test document for concurrency category for variable 2.

An example of this can be seen in Listing 4.19. In the test script, `t` would be sent, causing the system to loop between configurations {`substate-1-a-a, substate-1-a-b, substate-1-b-a, substate-1-b-b`} and {`substate-2-a-a, substate-2-a-b, substate-2-b-a, substate-2-b-b`}.

4.5.3 ECMAScript Interpreters

There are a variety of ECMAScript interpreters which are used in Web browsers today. Most modern ECMAScript interpreters are very sophisticated, implemented as both fast interpreters and Just-In-Time (JIT) compilers that can generate native code for multiple processor architectures. Because of this level of sophistication, the same ECMAScript code may execute very differently in terms of performance and memory usage depending on language implementation. For that reason, it was necessary benchmark SCION using multiple ECMAScript interpreters, in multiple Web browsers.

Four Web browsers were selected for testing, each using a different ECMAScript interpreter:

1. Mozilla Firefox 12.0, which includes the *Spidermonkey* ECMAScript interpreter.

2. Chromium 18, the open source version of Google's Chrome Web browser, which includes the *v8* ECMAScript interpreter.

3. Opera 12, which includes the *Presto* ECMAScript interpreter.

```
<scxml>
  <state id="default">
    <transition target="history" event="in"/>
  </state>
  <parallel id="parallel">
    <state id="ortho-0">
      <state id="substate-1-0">
        <transition target="substate-2-0" event="t1"/>
      </state>
      <state id="substate-2-0">
        <transition target="substate-1-0" event="t2"/>
      </state>
    </state>
    <state id="ortho-1">
      <state id="substate-1-1">
        <transition target="substate-2-1" event="t1"/>
      </state>
      <state id="substate-2-1">
        <transition target="substate-1-1" event="t2"/>
      </state>
    </state>
    <history type="deep" id="history">
      <transition target="substate-1-0␣substate-1-1"/>
    </history>
    <transition target="default" event="out"/>
  </parallel>
</scxml>
```

Listing 4.18: Generated test document for history-concurrency category for variable 2.

Finally, while not a full Web browser, the open source *Webkit* HTML rendering engine, which is used as the underlying engine for many browsers, including Apple's Safari Web browser and many mobile browsers, was also tested, using *GtkLauncher*, a lightweight browser GUI for Webkit included with the Webkit library package on the Ubuntu GNU/Linux operating system. Webkit includes the *JavaScriptCore* ECMAScript interpreter, and tests run in GtkLauncher should provide similar results to the Safari Web browser, which also uses the JavaScriptCore interpreter.

These Web browsers were selected because they could be run on GNU/Linux, and because they should be representative of the majority of users' Web browsing experiences today.

There were other popular ECMAScript implementations which were not tested, including JScript, which is used in Microsoft's Internet Explorer Web browser, and Mozilla Rhino. JScript was excluded because it could only be executed in Windows, and Rhino was excluded because it is not used in any modern Web browser, and is instead primarily used in sever-side ECMAScript development.

```
<scxml>
  <parallel id="parallel">
    <parallel id="ortho-a">
      <state id="ortho-a-a">
        <state id="substate-1-a-a">
          <transition target="substate-2-a-a" event="t"/>
        </state>
        <state id="substate-2-a-a">
          <transition target="substate-1-a-a" event="t"/>
        </state>
      </state>
      <state id="ortho-a-b">
        <state id="substate-1-a-b">
          <transition target="substate-2-a-b" event="t"/>
        </state>
        <state id="substate-2-a-b">
          <transition target="substate-1-a-b" event="t"/>
        </state>
      </state>
    </parallel>
    <parallel id="ortho-b">
      <state id="ortho-b-a">
        <state id="substate-1-b-a">
          <transition target="substate-2-b-a" event="t"/>
        </state>
        <state id="substate-2-b-a">
          <transition target="substate-1-b-a" event="t"/>
        </state>
      </state>
      <state id="ortho-b-b">
        <state id="substate-1-b-b">
          <transition target="substate-2-b-b" event="t"/>
        </state>
        <state id="substate-2-b-b">
          <transition target="substate-1-b-b" event="t"/>
        </state>
      </state>
    </parallel>
  </parallel>
</scxml>
```

Listing 4.19: Generated test document for nested-parallel category for variable 2.

4.5.4 Testing Methodology

The testing methodology used in this thesis was loosely based on the methodology described in the paper "On Efficient Program Synthesis from Statecharts," by Andrzej Wasowski [Was03]. In [Was03], two Statecharts optimizations were compared to an existing Statecharts implementation for execution time, memory usage, and generated code size. Six Statecharts models were used as test cases. For each test case, the time to process 10^7 random events was recorded. Furthermore, the state machine would be reset with a probability of 1% after sending each event.

The methodology used in this chapter differed from that in [Was03] in several ways. First, 88 individual test cases were used (19 *basic-states*, 6 *nested-parallel*, and 9 test cases for all other test categories), as opposed to just 6. Second, 64 optimization profiles were tested, as opposed to just 3. Third, 4 separate implementations of the underlying language runtime (the ECMAScript interpreter, in the case of SCION) were tested, as opposed to just 1. Fourth, in each test run, predefined sequences of events were looped for a minimum duration, as opposed to sending a fixed number of random events and randomly resetting the state machine. This looping methodology was feasible due to the structure of the generated test documents. Fifth, and finally, it was not necessary to record generated code size, because code was generated on-the-fly in the Web browser, as described in Section 4.4, and so only execution time and memory usage results were recorded.

To deal with this additional complexity, it is useful to first define some terminology. A *run combination* is defined to be a tuple containing an optimization profile, test case, and Web browser. A *run* is defined to be the execution of a particular run combination; this is described in more detail below. A *complete run* is defined to be the execution of all possible run combinations.

In this experiment, 22,528 unique run combinations were tested, for 88 test cases $\times$ 64 optimization profiles $\times$ 4 web browsers. Each run combination was executed 10 times, for 22,528,000 runs, and 10 complete runs.

Runs were executed as follows: a master test process would iterate asynchronously through all run combinations, in order of test category and increasing complexity. Upon initialization, the Web browser associated with the first run combination would be spawned as a separate process.

The Web browser instance would then request a test combination using the Web browser's built-in XMLHTTPRequest object, which allowed it to asynchronously communicate with the master test process. The Web browser would then execute a run by instantiating a SCION interpreter using all of the parameters of the run combination (i.e., the optimization profile and test document), and then dispatch the events listed in the test script associated with that run combination on the interpreter instance in a loop, until at least 100 milliseconds had passed.

During the run, the Web browser would measure the duration of the run, as well as the number of events that were dispatched, in order to determine the number of events sent per millisecond, which is the primary unit by which performance was measured. Duration was measured using ECMAScript's built-in `Date` object, which has millisecond precision. An

example of the way `Date` can be used to measure duration is provided in Listing 4.20.

```
var tic = new Date();
executeSomeCode();
var toc = new Date();
var duration = toc - tic;    //returns duration in milliseconds
```

Listing 4.20: Measuring duration using the ECMAScript Date Object

After the completion of a run, the Web browser would request that the master process check the virtual memory size of the browser process. This was performed by parsing the Linux process file associated with the Web browser process.

The memory and performance results of the run were then saved in a MongoDB database for further analysis.

The Web browser instance would then continue requesting run combinations and executing runs, until a run combination in a new test category was requested, at which point the master test process would kill the current Web browser process, and start a new Web browser process based on the Web browser associated with the current run combination. This was performed to ensure that memory usage of each test category would be tested in isolation from other test categories.

This process continued until all run combinations had been executed, thus finishing a complete run, at which point the master process would terminate. The master process would then be restarted by an external script. This was repeated 10 times.

All tests were performed on a Lenovo Thinkpad W520, with 2.40GHz Intel Quad-core i7-2760QM CPU and hyperthreading, running 64-bit Ubuntu 11.10 GNU/Linux.

4.6 Results

The data from each complete run was aggregated, such that the minimum and maximum values for performance and memory usage were discarded, and the mean was taken of the remaining values. In this section, these aggregate results are explored in order to determine which optimization profiles provided the best performance and memory usage for particular test categories and particular browsers.

The general approach is to first examine the results of applying each optimization strategy individually, in order to seek outlier options for each optimization strategy, and then to examine all possible combinations of optimizations profiles in order to determine which configuration of optimization options were the most performant when combined. Ultimately, it will be shown that the optimization options that are the most performant when tested individually were also the most performant when combined.

In order to examine each optimization strategy individually, the notion of a set of "default" options is required, where each optimization strategy is associated with a particular optimization option that is used when not examining that particular strategy. In other words, to examine an optimization strategy individually implies that all other optimization

options that make up an optimization profile are populated with "default" values. The default options chosen for each optimization strategy are shown in Table 4.2.

Transition Selection	Default
Set Type	Array Set
Flattening	False
Caching	True

Table 4.2: Default Options

For example, when examining the Transition Selection optimization strategy individually, the following optimization profiles would be compared:

- $(Default, ArraySet, False, True)$
- $(Class, ArraySet, False, True)$
- $(Switch, ArraySet, False, True)$
- $(State, ArraySet, False, True)$

This is because the four options of the Transition Selection optimization strategy are Default, Class, Switch, and Table, and the default options described in Table 4.2 are used to populate the other optimization strategy values of the optimization profiles to be compared.

Each optimization strategy will be examined individually for performance and memory usage. Generally, for each optimization strategy examined, one page of plots will be shown with data derived from a particular Web browser. The page will contain 9 plots, one for each test category. Note that most Web browsers actually provided similar results, so only one page of plots, with results derived from a single browser, will be shown for each individual optimization strategy. Cases where particular Web browsers deviated from the general trend will be specifically called out and examined. Furthermore, a data table will be provided for each optimization strategy, outlining results across all Web browsers.

4.6.1 Performance

Transition Selection Optimization Strategy

Figure 4.1 shows the impact of the four options of the transition selection optimization strategy (Default, Table, Switch and Class) on performance in Firefox. The horizontal axis shows an increase in complexity of the test case, and the vertical axis shows an increase in number of events that can be processed per millisecond. High numbers on the vertical axis are better, indicating increased performance.

One may make the following observations. First, for test categories *basic-states* and *events*, Default and Table options are clear winners. Second, for test category *transitions2*, Table is an outlier, followed closely by Class and Switch, and the Default option clearly performs worse than the other options. Third, for test categories *depth* and *history-depth*, Table

and Default are smaller outliers. Finally, for test categories *history-concurrency* and *concurrency*, Switch performs worse than other transition selection options, which have roughly the same performance.

In other browsers, Table continues to be significantly better for *basic-states* and *events*, but Default sometimes performs worse than other options; furthermore, Switch sometimes performs well in these categories. This can be seen in Figure 4.2 with results derived from Webkit. Finally, all transition selection options perform equally well in test categories *depth*, *history-depth*, *concurrency* and *history-concurrency* in Opera, Chromium and Webkit, so there are no transition selection outliers for these browsers and categories.

Table 4.3 illustrates these results across all browsers for test categories *basic-states*, *events*, *transitions2*, *depth*, *history-depth* and *history-concurrency*. In this table, each row corresponds to a particular test category and browser. The columns show the test category name, followed by the most performant transition selection option and its average performance, followed by the least performant transition selection option and its average performance, followed by the percentage difference between the most performant transition selection option and the next most-performant option, and finally the percentage difference between most- and least-performant transition selection options.

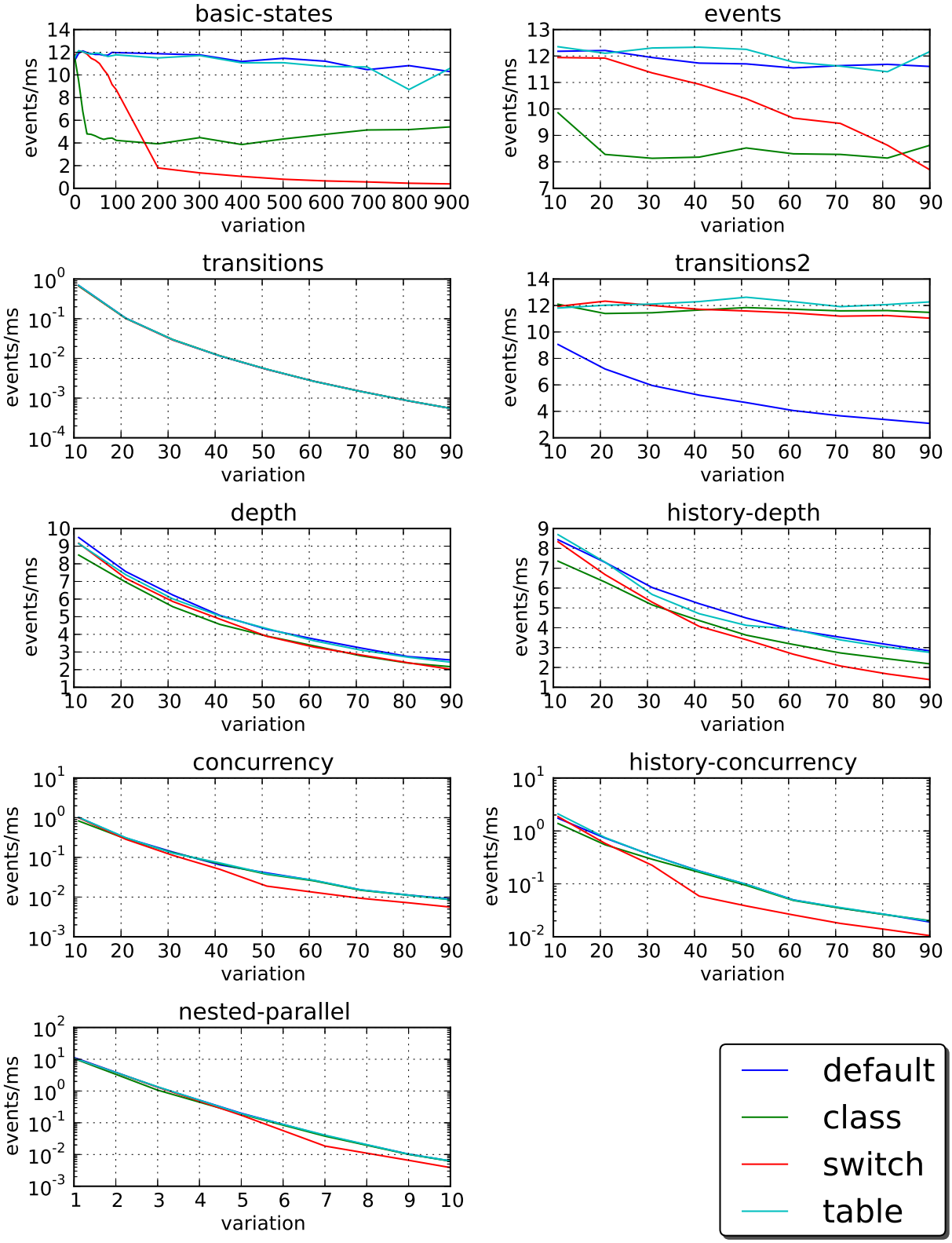

Figure 4.1: Results of Transition Selection optimization strategy in Firefox

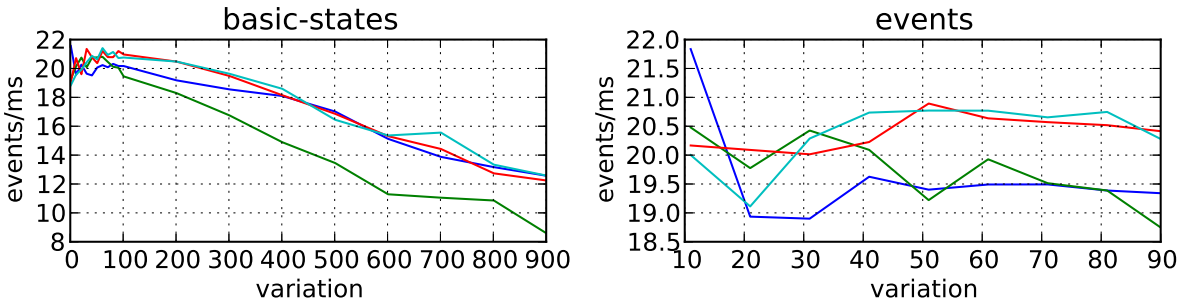

Figure 4.2: Results of Transition Selection optimization strategy in Webkit

test category	best	best ev/ms	worst	worst ev/ms	vs. next	vs. worst
firefox						
basic-states	default	11.54	class	5.32	1.37%	116.71%
events	table	12.04	class	8.49	2.03%	41.91%
transitions2	table	12.16	default	5.14	4.39%	136.65%
depth	default	4.99	class	4.46	2.56%	11.78%
history-depth	default	4.98	switch	3.95	2.90%	26.16%
history-concurrency	table	0.40	class	0.29	11.83%	37.87%
chromium						
basic-states	table	32.22	switch	27.52	2.97%	17.08%
events	table	34.56	default	32.14	3.33%	7.52%
transitions2	class	19.30	default	6.17	2.80%	212.65%
depth	table	11.11	switch	9.43	2.17%	17.81%
history-depth	table	11.52	switch	8.82	5.94%	30.61%
history-concurrency	class	1.28	default	1.17	0.25%	9.13%
opera						
basic-states	table	19.49	switch	15.53	4.92%	25.52%
events	table	20.80	switch	18.51	2.85%	12.41%
transitions2	class	19.46	default	6.26	3.96%	211.09%
depth	table	4.81	class	4.65	0.18%	3.47%
history-depth	table	5.01	class	4.74	3.09%	5.73%
history-concurrency	table	0.44	class	0.39	6.07%	11.10%
webkit						
basic-states	table	18.80	class	17.25	0.20%	8.97%
events	switch	20.39	default	19.60	0.12%	4.03%
transitions2	class	20.24	default	8.81	2.68%	129.63%
depth	class	6.75	table	6.53	0.51%	3.39%
history-depth	switch	6.73	class	6.45	2.62%	4.34%
history-concurrency	default	0.72	class	0.70	0.76%	1.63%

Table 4.3: Transition Selection performance results

Set Implementation

Figure 4.3 shows the impact of the four options of the Set Implementation optimization strategy (Array Set, Bit Vector Set, Boolean Array Set, and Object Set) on performance in Webkit.

One may observe that the Array Set implementation is significantly better for test categories *basic-states*, *events*, and *transitions2*. Furthermore, there are are no clear outliers for the other test categories.

In Chromium and Firefox, the Bit Vector set implementation also performs well for test categories *depth* and *history-depth*, performing similar to or better than the Array Set implementation. This is illustrated in Figure 4.4 with results from Chromium. One possible explanation for this boost in performance is that Chromium and Firefox's ECMAScript implementations generate native machine code on the fly, and thus, bit operations in ECMAScript may be JIT-compiled directly to bit operations in x86 assembler, which would likely provide the fast execution times that are seen here.

Table 4.4 illustrates results across all browsers for test categories *basic-states*, *events*, *transitions2*, *depth*, and *history-depth*. The format of this table is identical to the previous table, Table 4.3, which illustrated Transition Selection performance results.

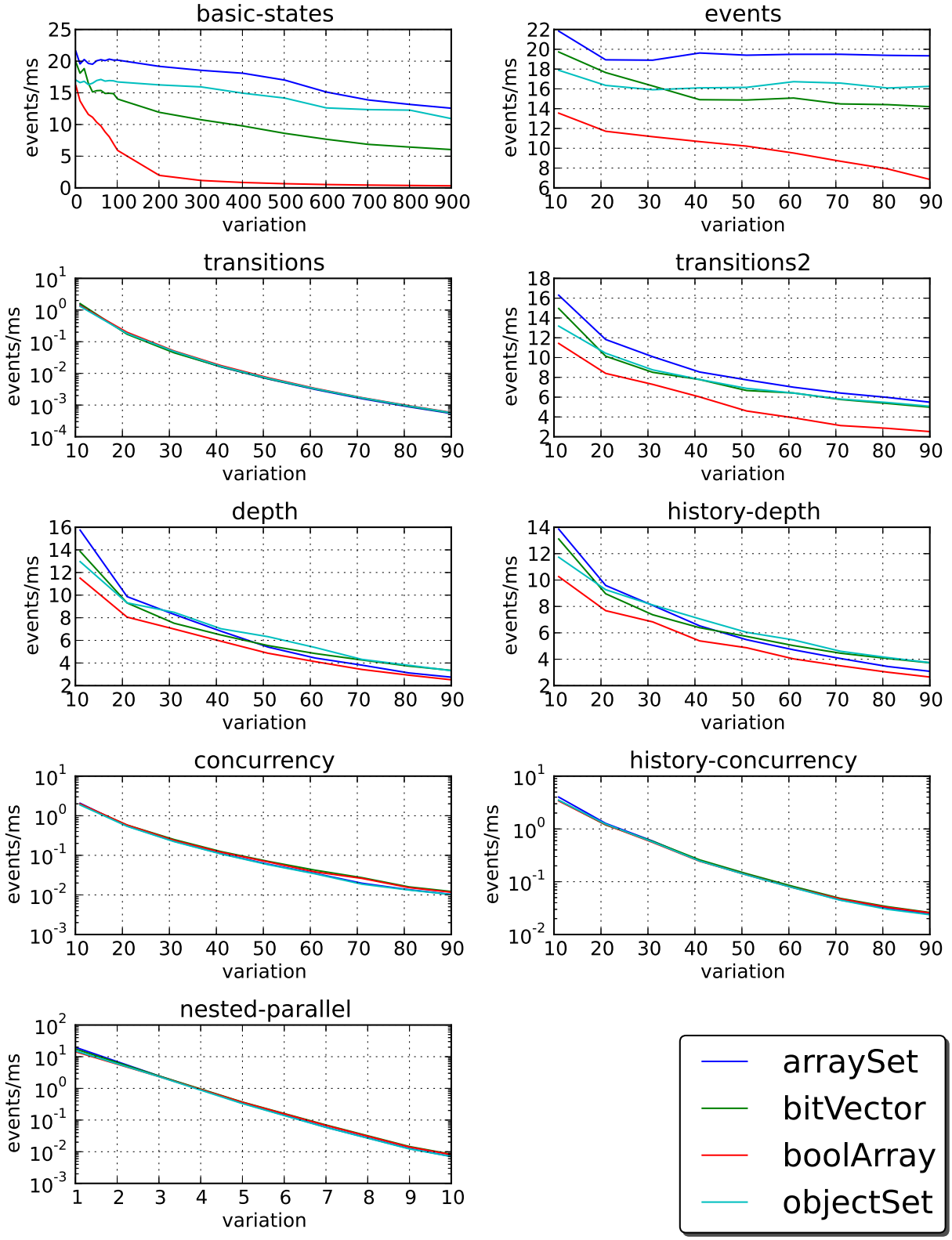

Figure 4.3: Result of Set Implementation optimization strategy in Webkit

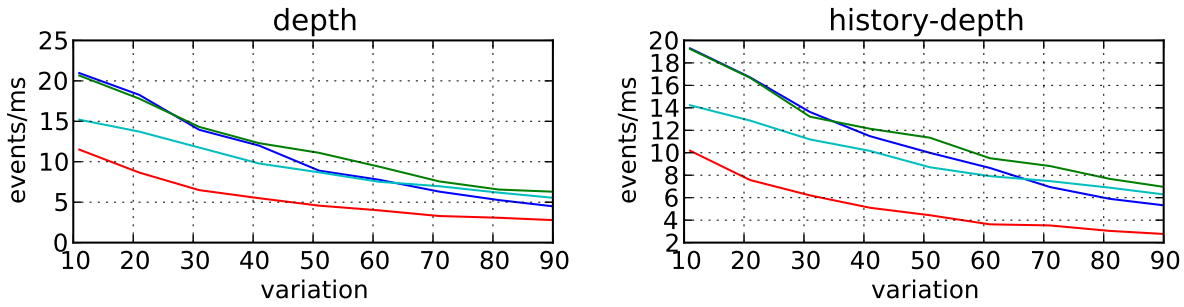

Figure 4.4: Bit Vector Set Implementation outliers in Chromium for test categories *depth* and *history-depth*

test category	best	best ev/ms	worst	worst ev/ms	vs. next	vs. worst
firefox						
basic-states	arraySet	11.54	boolArray	4.41	24.95%	161.69%
events	arraySet	11.80	boolArray	6.35	8.83%	85.83%
transitions2	arraySet	5.14	boolArray	3.77	3.52%	36.19%
depth	bitVector	5.17	boolArray	3.77	3.72%	37.20%
history-depth	bitVector	5.25	boolArray	3.71	5.32%	41.37%
chromium						
basic-states	arraySet	31.29	boolArray	5.49	40.81%	469.69%
events	arraySet	32.14	boolArray	7.63	21.65%	321.10%
transitions2	arraySet	6.17	boolArray	5.67	0.59%	8.80%
depth	bitVector	11.77	boolArray	5.54	8.31%	112.67%
history-depth	bitVector	11.72	boolArray	5.16	7.80%	127.14%
opera						
basic-states	arraySet	17.98	boolArray	12.51	25.96%	43.68%
events	arraySet	20.01	boolArray	14.97	19.31%	33.63%
transitions2	arraySet	6.26	boolArray	5.62	3.45%	11.29%
depth	objectSet	5.26	arraySet	4.80	1.41%	9.55%
history-depth	objectSet	5.33	arraySet	4.86	0.18%	9.62%
webkit						
basic-states	arraySet	18.38	boolArray	6.39	18.68%	187.53%
events	arraySet	19.60	boolArray	10.02	19.14%	95.61%
transitions2	arraySet	8.81	boolArray	5.57	12.46%	58.27%
depth	objectSet	6.76	boolArray	5.59	1.09%	21.02%
history-depth	objectSet	6.68	boolArray	5.35	2.21%	24.73%

Table 4.4: Set Type Performance Results

Flattening

Figure 4.5 shows the results of the flattening transformation on performance in Webkit.

One may observe that the flattening transformation improves performance for test categories *basic-states*, *events*, *depth*, and *history-depth*. Furthermore, one may observe that it decreased performance for test category *history-concurrency*.

These results were consistent across browsers, and are illustrated in Table 4.5. The columns in this table show the test category name, followed by the most performant option and its average performance, followed by the performance of the least performant option, followed by the percentage difference between the most performant and least performant options.

test category	best	best ev/ms	worst ev/ms	best vs. worst
firefox				
basic-states	True	11.94	11.54	3.48%
events	True	12.14	11.80	2.88%
depth	True	5.82	4.99	16.69%
history-depth	True	5.55	4.98	11.42%
history-concurrency	False	0.36	0.11	226.70%
chromium				
basic-states	True	32.41	31.29	3.59%
events	True	34.07	32.14	6.00%
depth	True	13.31	10.87	22.43%
history-depth	True	11.96	10.87	10.04%
history-concurrency	False	1.17	0.07	1693.95%
opera				
basic-states	True	21.03	17.98	16.99%
events	True	21.74	20.01	8.66%
depth	True	6.36	4.80	32.46%
history-depth	True	6.56	4.86	34.86%
history-concurrency	False	0.40	0.11	262.06%
webkit				
basic-states	True	19.75	18.38	7.46%
events	True	20.98	19.60	7.05%
depth	True	8.60	6.69	28.57%
history-depth	True	8.29	6.53	26.87%
history-concurrency	False	0.72	0.21	245.37%

Table 4.5: Flattening Transformation performance results

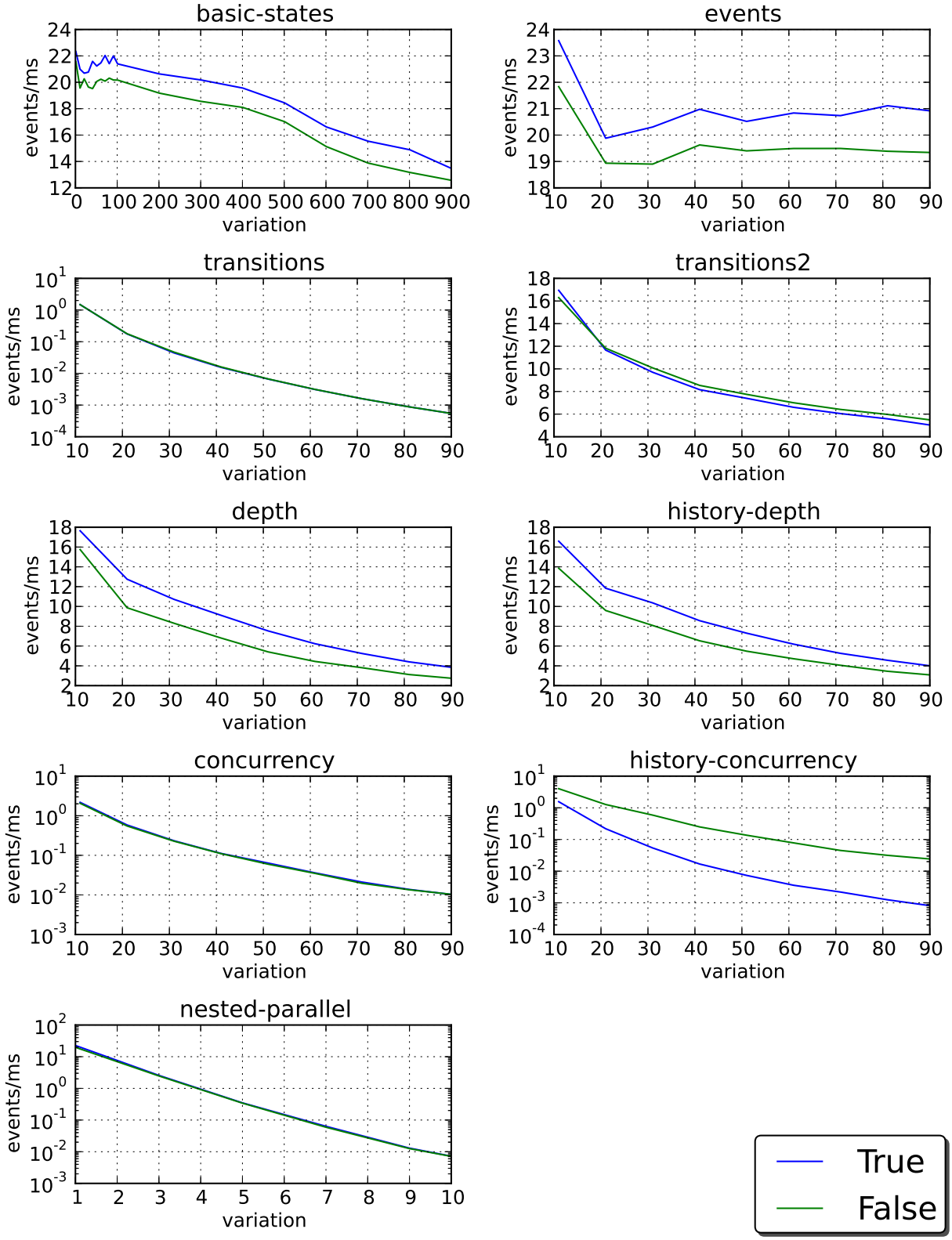

Figure 4.5: Results of flattening transformation in Webkit

Caching

Figure 4.6 shows the results of caching structural information on performance in Firefox. One may observe that caching consistently provided better performance across all test categories, except for *transitions2*, where it had no effect. These results were consistent across browsers, which is illustrated in Table 4.6. This table has the same format as the previous table, Table 4.5, which illustrated the result of the Flattening optimization strategy.

test category	best	best ev/ms	worst ev/ms	best vs. worst
firefox				
basic-states	True	11.54	5.50	109.75%
events	True	11.80	8.08	46.16%
transitions	True	0.09	0.06	64.22%
depth	True	4.99	1.42	251.59%
history-depth	True	4.98	1.51	230.47%
concurrency	True	0.18	0.01	1107.63%
history-concurrency	True	0.36	0.04	884.98%
nested-parallel	True	2.16	1.45	48.97%
chromium				
basic-states	True	31.29	13.46	132.49%
events	True	32.14	18.66	72.22%
transitions	True	0.33	0.15	124.39%
depth	True	10.87	3.70	193.76%
history-depth	True	10.87	3.87	180.67%
concurrency	True	0.57	0.05	1038.21%
history-concurrency	True	1.17	0.11	976.29%
nested-parallel	True	4.76	3.66	29.88%
opera				
basic-states	True	17.98	6.60	172.40%
events	True	20.01	9.94	101.35%
transitions	True	0.10	0.06	51.79%
depth	True	4.80	1.16	313.82%
history-depth	True	4.86	1.24	291.99%
concurrency	True	0.18	0.01	1420.48%
history-concurrency	True	0.40	0.03	1504.60%
nested-parallel	True	2.69	1.80	49.46%
webkit				
basic-states	True	18.38	7.81	135.24%
events	True	19.60	11.18	75.36%
transitions	True	0.19	0.10	95.71%
depth	True	6.69	1.98	237.56%
history-depth	True	6.53	2.05	217.95%
concurrency	True	0.34	0.03	1215.18%
history-concurrency	True	0.72	0.06	1120.22%
nested-parallel	True	3.78	2.37	59.65%

Table 4.6: Cached Structural Information performance results

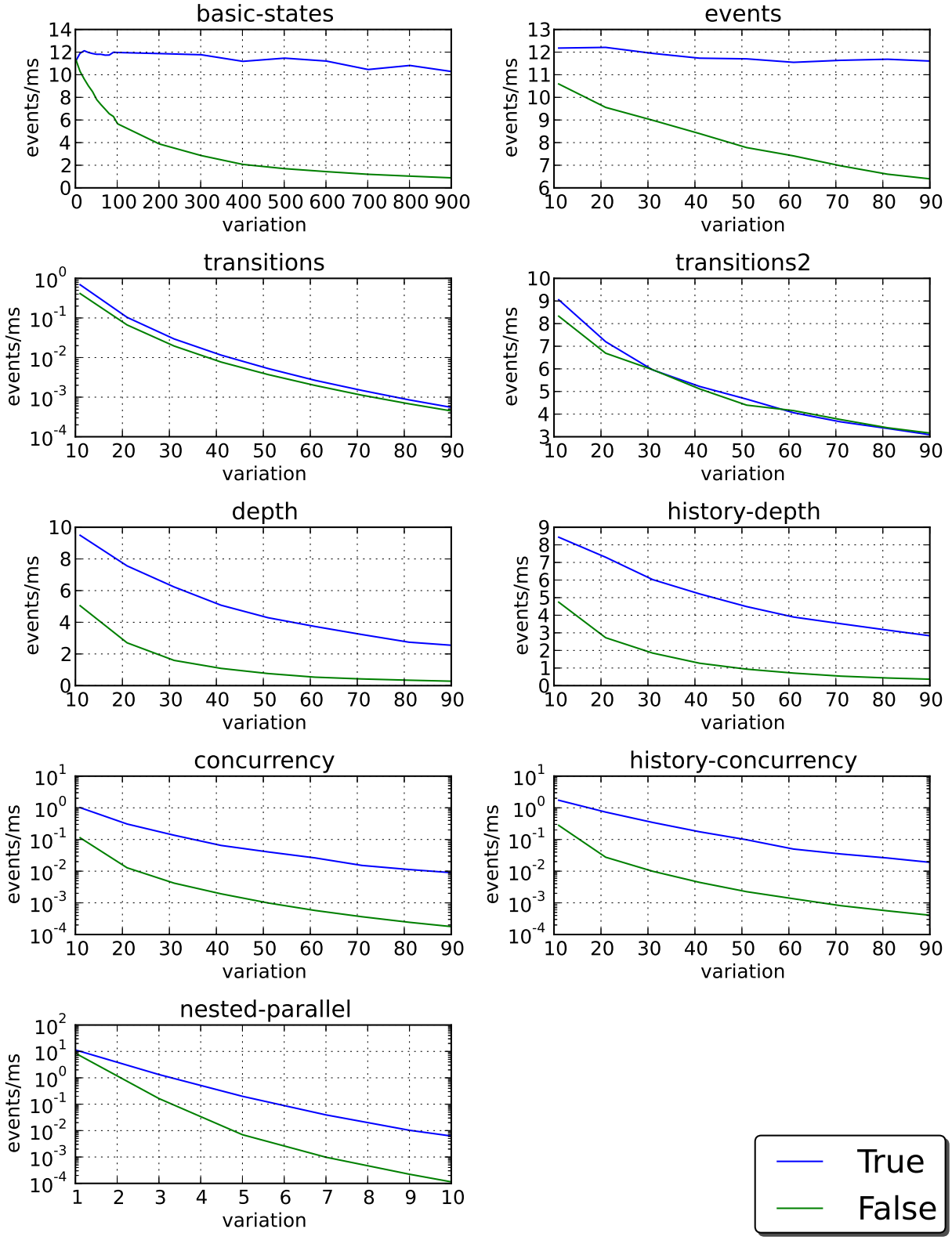

Figure 4.6: Results of caching model information in Firefox

Performance Across all Possible Optimization Profiles

One may take the intersection of the above results in order to predict the best optimization profiles when individual optimization options are combined. Table 4.8 illustrates the predicted best optimization profiles for all browsers.

This table, and the following tables and figures containing combined results, use an abbreviated notation when referring to optimization profiles. In this abbreviated notation, the optimization is expressed using four characters. The first character refers to the Transition Selection optimization strategy, the second refers to the Set Implementation, the third refers to the Flattening Transformation, and the fourth refers to the Model Caching optimization strategy. The mappings from optimization option to character are shown in Table 4.7. So, for example, the optimization profile *(Default Transition Selection, Array Set, True, True)* would be abbreviated to "daTT."

In Table 4.8, a pipe ("|") indicates an OR relationship between possible optimization options, and a question mark ("?") indicates an "undefined" or "any" relationship, such that no optimization value is predicted.

Transition Selector	
Default	d
Table	t
Class	c
Switch	s
Set Implementation	
Array Set	a
Bit Vector Set	v
Boolean Array Set	b
Object Set	o
Flattening	
True	T
False	F
Model Caching	
True	T
False	F

Table 4.7: Abbreviated notation for Optimization Profile

These predications did coincide with actual results. This can be seen in Table 4.9, which provides a detailed comparison of the most performant optimization profiles across all browsers. In this table, the columns list the test group, followed by the most performant optimization profile and its average performance, followed by a comparison to the default optimization profile *(Default, Array Set, False, True)*, followed by a comparison between the most and least performant optimization profiles.

Figures 4.7 and 4.8 provide graphical overviews of these results for the Webkit and Firefox

Test Category	Transition Selection	Set Type	Flattening	Caching
chromium				
basic-states	t | d	a	T	T
events	t	a	T	T
transitions	?	?	?	T
transitions2	t | c | s	?	?	?
depth	t | d	a | v	T	T
history-depth	t | d	a | v	T	T
concurrency	?	?	T	T
history-concurrency	?	?	F	T
nested-parallel	?	?	?	T
firefox				
basic-states	t | d	a	T	T
events	t | d	a	T	T
transitions	?	?	?	T
transitions2	t | c | s	a	?	?
depth	t | d	a | v	T	T
history-depth	t | d	a | v	T	T
concurrency	t | d | c	?	?	T
history-concurrency	t | d | c	?	F	T
nested-parallel	?	?	?	T
opera				
basic-states	t | d | c	a	T	T
events	t | d | c	a	T	T
transitions	?	?	?	T
transitions2	t | c | s	?	?	?
depth	?	?	T	T
history-depth	?	?	T	T
concurrency	?	?	?	T
history-concurrency	?	?	F	T
nested-parallel	?	?	?	T
webkit				
basic-states	t | d | s	a	T	T
events	t | s	a	T	T
transitions	?	?	?	T
transitions2	t | c | s	a	?	T
depth	?	a | o	T	T
history-depth	?	a | o	T	T
concurrency	?	?	?	T
history-concurrency	?	?	F	T
nested-parallel	?	?	?	T

Table 4.8: Predicted outlier optimization profiles for all browsers

browsers, respectively. For each graph, the best, worst and default optimization profiles are plotted.

group	best opt profile	ev/ms	vs. default	vs. worst
firefox				
basic-states	taTT	12.31	6.73%	512.74%
events	taTT	12.75	8.05%	478.83%
transitions	dvFT	0.09	0.73%	72.45%
transitions2	taTT	12.74	147.92%	249.05%
depth	dvTT	5.95	19.35%	404.67%
history-depth	taTT	6.01	20.63%	512.68%
concurrency	dvFT	0.19	2.66%	1221.26%
history-concurrency	taFT	0.40	11.83%	3310.46%
nested-parallel	saTT	2.33	7.94%	121.04%
chromium				
basic-states	taTT	33.93	8.45%	600.71%
events	taTT	37.18	15.68%	457.51%
transitions	cvTT	0.45	37.12%	608.09%
transitions2	caTT	22.08	257.65%	302.94%
depth	tvTT	14.74	35.63%	454.45%
history-depth	taTT	14.19	30.51%	535.84%
concurrency	tvTT	0.71	25.83%	1489.18%
history-concurrency	tvFT	1.35	14.87%	3877.23%
nested-parallel	taTT	6.43	35.14%	516.53%
opera				
basic-states	taTT	22.13	23.10%	301.48%
events	taTT	25.12	25.57%	200.85%
transitions	cvTT	0.10	8.22%	82.25%
transitions2	caTT	22.07	252.82%	307.50%
depth	tvTT	7.22	50.33%	557.51%
history-depth	cvTT	7.33	50.83%	753.39%
concurrency	tvTT	0.22	22.02%	1923.23%
history-concurrency	toFT	0.44	9.98%	5335.32%
nested-parallel	cvTT	3.68	36.41%	153.76%
webkit				
basic-states	taTT	20.74	12.88%	334.34%
events	taTT	22.45	14.53%	205.98%
transitions	dvFT	0.20	6.03%	148.26%
transitions2	caTT	21.45	143.37%	301.63%
depth	caTT	8.77	31.11%	395.64%
history-depth	caTT	8.67	32.69%	561.70%
concurrency	svTT	0.38	9.41%	1544.14%
history-concurrency	daFT	0.72	same	4244.46%
nested-parallel	daTT	4.27	12.90%	135.02%

Table 4.9: Most performant optimization profiles across all browsers

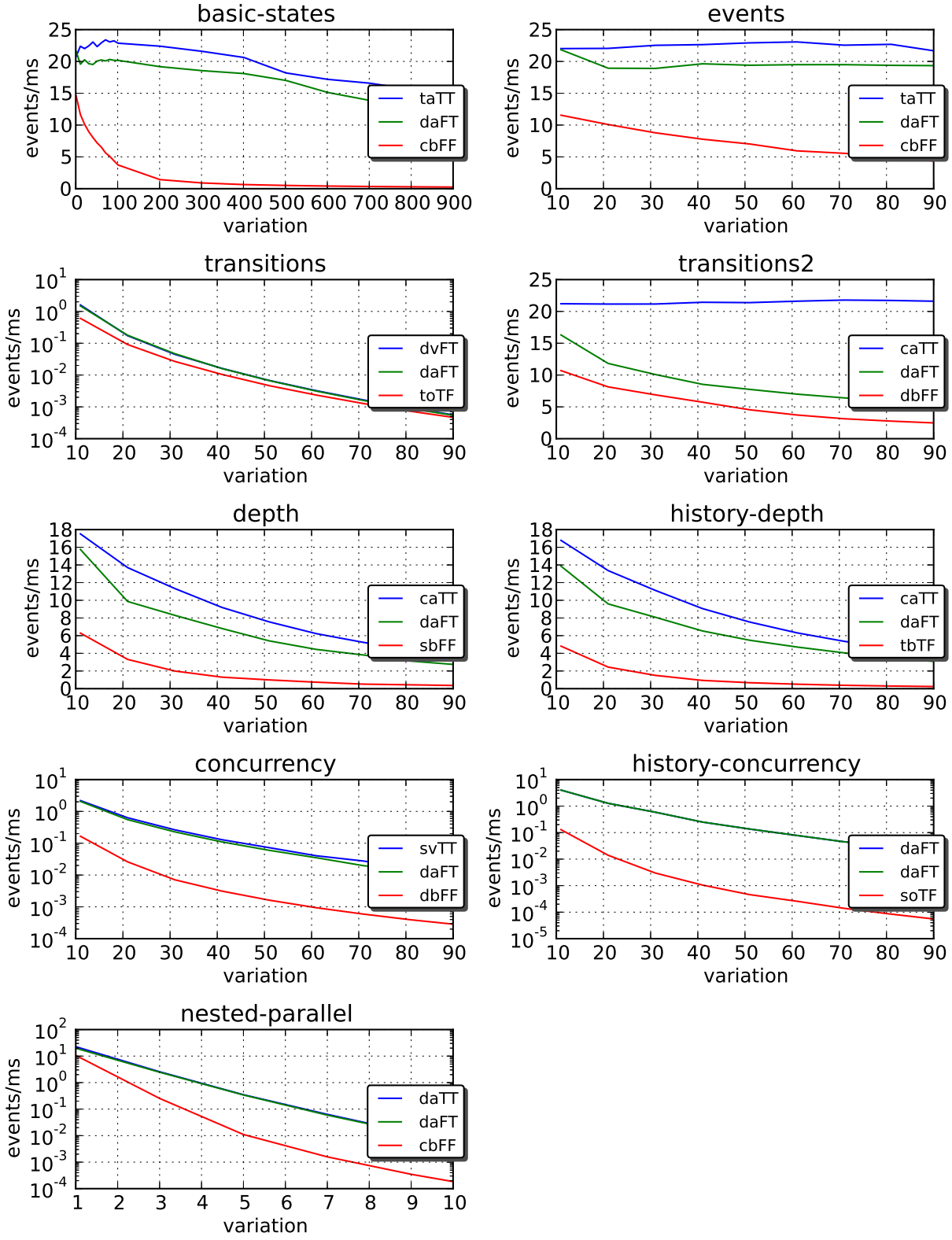

Figure 4.7: Comparison of best, worst, and default optimization profiles for performance in Webkit

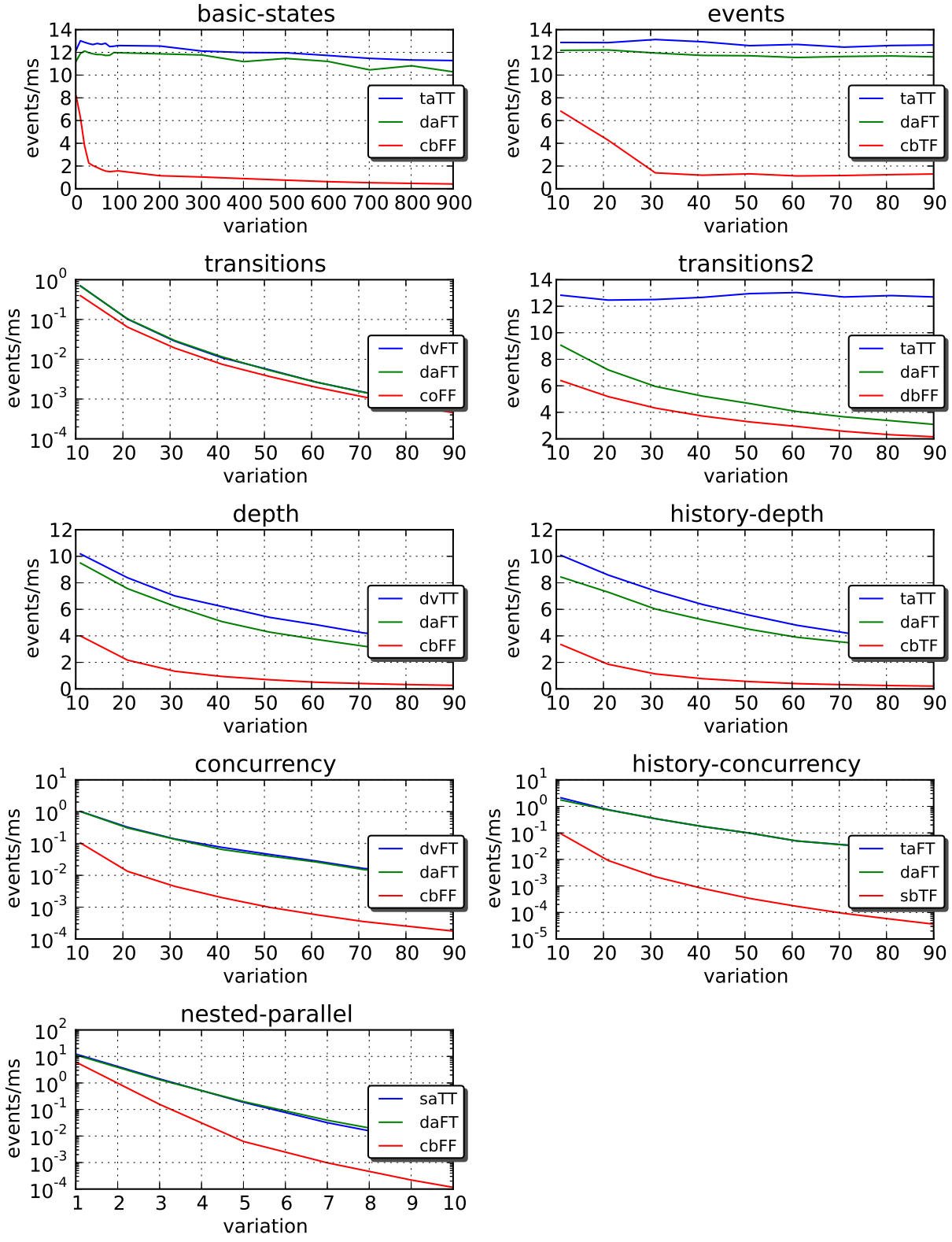

Figure 4.8: Comparison of best, worst, and default optimization profiles for performance in Firefox

4.6.2 Memory Usage

Transition Selection

Figure 4.9 shows the impact of the four options of the transition selection optimization strategy on memory usage in Firefox. All plots show virtual memory size in megabytes.

One may observe that the Table transition selection option took the most memory, and the Default transition selection option took the least. This makes sense, as the Table transition selection options relies on a large, sparse two-dimensional array, whereas the Default transition selection option can be used without loading any additional data structures.

While this observation is consistently true for Opera and Firefox, in Chromium and Webkit, the choice of transition selection had far less of an effect on memory. This is illustrated in Table 4.10. This table lists the test category, followed by the transition selection option with the lowest average virtual memory usage and its virtual memory usage in megabytes, followed by the transition selection option with the highest average virtual memory usage in megabytes, followed by the difference between the highest and lowest virtual memory usages in megabytes, and the percentage difference between the highest and lowest virtual memory usages.

One may observe in Table 4.10 that the transition selection optimization strategy had a strong effect on memory usage in Opera, a moderate effect in Firefox and Chromium, and a negligible effect in Webkit. The percentage difference between the memory usage of the best and worst transition selection strategies is at most 0.25% in Webkit. The difference between best and worst transition selections is 5.64% for test category *transition2* in Chromium, but all other test categories have less than 0.29% difference between best and worst transition selection options in Chromium. Differences in memory usage in Firefox, on the other hand, range between 2.62% and 5.39%, and Opera's memory use ranges between 4.40% and 18.54%. This can entail a difference of tens of megabytes of virtual memory usage for Firefox, and over a hundred for Opera, but, except for group *transitions2* in Chromium, there is less than 5MB difference between transition selection options across test categories in Chromium and Webkit.

Caching, Flattening, and Set Implementation

Caching, Flattening, and Set Implementation optimization strategies had a negligible effect on memory usage.

It is somewhat surprising that Caching had a negligible effect on memory usage, as it is, by definition, storing additional structural information about the Statecharts model. However, the cache stores only references to state objects, rather than copies of the objects themselves, and is thus a lightweight approach. It is therefore understandable that storing object references for model caching has a negligible impact on memory usage compared to other parts of the model. Finally, this approach scales well as the Statecharts model increases in size and complexity.

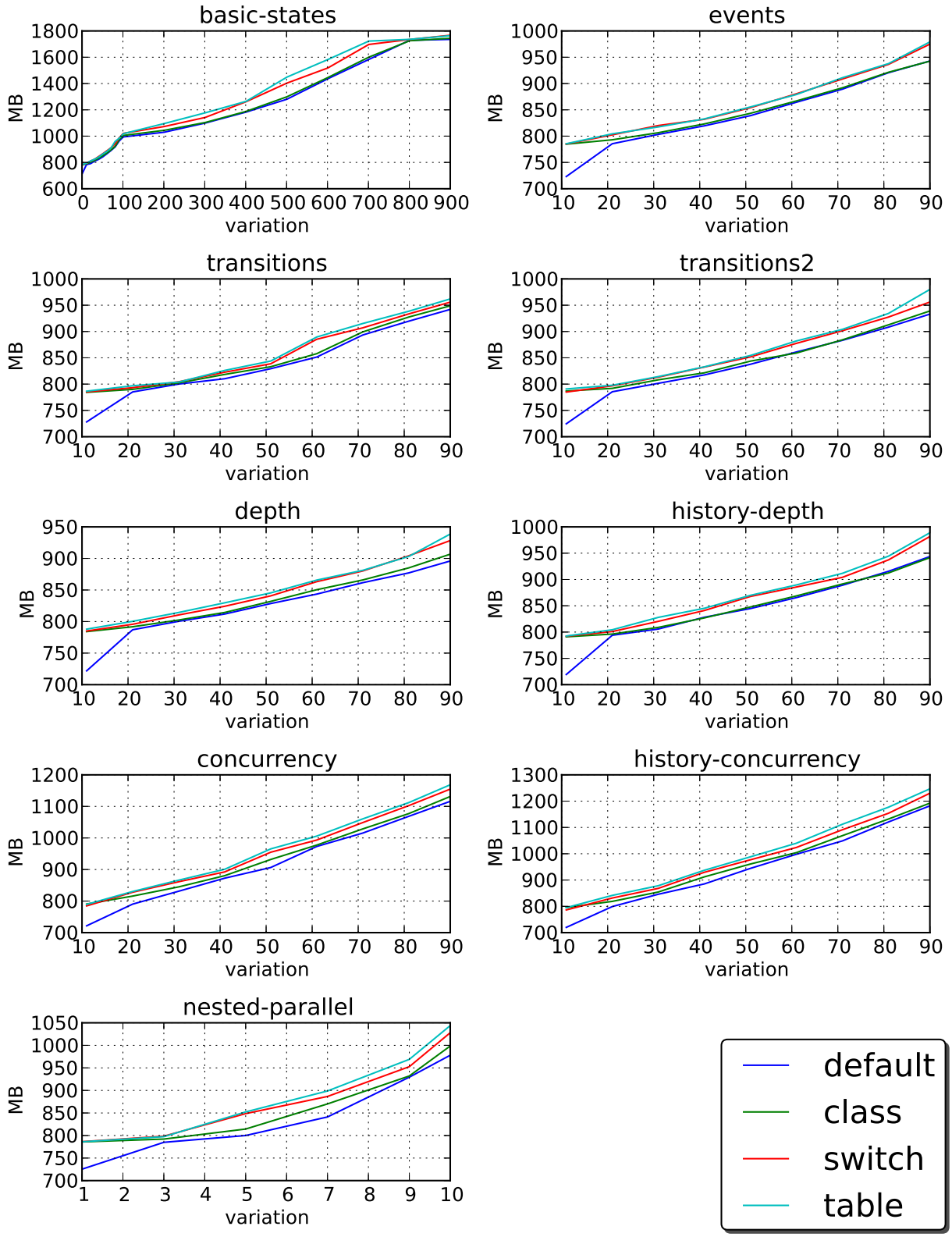

Figure 4.9: Results of transition selection optimization strategy on memory usage in Firefox

test category	best	best VSZ	worst	worst VSZ	best - worst	vs. worst
firefox						
basic-states	default	1077.88	table	1129.94	52.06	4.61%
events	default	843.36	table	867.49	24.13	2.78%
transitions	default	840.18	table	862.75	22.57	2.62%
transitions2	default	839.44	table	866.17	26.73	3.09%
depth	default	825.54	table	852.26	26.72	3.13%
history-depth	default	845.42	table	875.47	30.05	3.43%
concurrency	default	922.28	table	967.26	44.98	4.65%
history-concurrency	default	950.13	table	1002.68	52.55	5.24%
nested-parallel	default	843.16	table	891.24	48.07	5.39%
chromium						
basic-states	default	667.15	table	667.74	0.58	0.09%
events	switch	659.79	class	660.39	0.61	0.09%
transitions	default	661.97	table	663.02	1.05	0.16%
transitions2	default	456.67	table	483.97	27.30	5.64%
depth	switch	658.88	class	659.52	0.64	0.10%
history-depth	switch	659.42	class	659.78	0.36	0.06%
concurrency	switch	663.67	table	664.33	0.66	0.10%
history-concurrency	default	664.03	table	665.75	1.72	0.26%
nested-parallel	default	660.29	table	662.23	1.94	0.29%
opera						
basic-states	default	917.73	table	1042.47	124.74	11.97%
events	default	496.73	table	543.86	47.13	8.67%
transitions	default	457.90	table	478.99	21.09	4.40%
transitions2	default	450.23	table	479.94	29.72	6.19%
depth	default	495.11	table	530.99	35.88	6.76%
history-depth	default	536.87	table	586.73	49.86	8.50%
concurrency	default	742.11	table	870.27	128.16	14.73%
history-concurrency	default	831.37	table	995.87	164.50	16.52%
nested-parallel	default	515.62	table	632.99	117.37	18.54%
webkit						
basic-states	default	1681.75	table	1684.81	3.06	0.18%
events	default	1691.74	table	1693.98	2.24	0.13%
transitions	default	1711.93	table	1713.13	1.19	0.07%
transitions2	default	1715.77	table	1717.74	1.96	0.11%
depth	default	1683.97	table	1686.46	2.49	0.15%
history-depth	default	1720.45	table	1723.34	2.89	0.17%
concurrency	default	1722.61	table	1725.93	3.32	0.19%
history-concurrency	default	1703.62	table	1707.47	3.85	0.23%
nested-parallel	default	1696.13	table	1700.38	4.24	0.25%

Table 4.10: Transition Selection memory results

Memory Usage Across all Possible Optimization Profiles

To summarize previous results, the Table transition selection option proved to be the most expensive in terms of memory usage, and the Default transition selection option proved to be the least expensive, for test categories on Firefox and Opera, and for *transitions2* on Chromium. Other optimization strategies did not have a noticeable impact on memory usage.

These observations held when all possible optimization profiles were tested. An example of this can be seen in Figure 4.10, which shows results derived from the Opera browser. Table 4.11 shows the results across all browsers.

group	best opt profile	VSZ	default VSZ	vs. default	vs. worst
firefox					
basic-states	daTT	1076.22	1077.88	0.15%	4.91%
events	daTT	839.90	843.36	0.41%	3.77%
transitions	daTT	836.63	840.18	0.42%	3.44%
transitions2	daTT	836.02	839.44	0.41%	3.72%
depth	daTT	822.32	825.54	0.39%	3.51%
history-depth	daTT	843.43	845.42	0.23%	3.93%
concurrency	daTT	919.39	922.28	0.31%	5.60%
history-concurrency	daTT	947.35	950.13	0.29%	6.26%
nested-parallel	daTT	838.27	843.16	0.58%	6.60%
chromium					
basic-states	daTT	667.08	667.15	0.01%	0.12%
events	daTT	659.64	659.89	0.04%	0.12%
transitions	daFT	661.97	661.97	same	0.21%
transitions2	daFT	456.67	456.67	same	7.64%
depth	cbFT	658.86	659.35	0.07%	0.10%
history-depth	cbFT	659.11	659.62	0.08%	0.10%
concurrency	cbFF	663.39	663.79	0.06%	0.30%
history-concurrency	daFT	664.03	664.03	same	0.26%
nested-parallel	daTT	659.96	660.29	0.05%	0.42%
opera					
basic-states	daTT	917.72	917.73	same	14.42%
events	daTT	496.65	496.73	0.02%	9.39%
transitions	daTT	457.75	457.90	0.03%	4.52%
transitions2	daTT	450.15	450.23	0.02%	7.80%
depth	daTT	495.03	495.11	0.02%	7.26%
history-depth	daTT	536.83	536.87	0.01%	9.66%
concurrency	daTT	742.04	742.11	0.01%	15.76%
history-concurrency	daFT	831.37	831.37	same	16.88%
nested-parallel	daTT	515.34	515.62	0.05%	18.76%
opera					
basic-states	dbFT	1680.80	1681.75	0.06%	0.24%
events	daTT	1691.46	1691.74	0.02%	0.15%
transitions	daTT	1711.72	1711.93	0.01%	0.08%
transitions2	daTT	1715.60	1715.77	0.01%	0.13%
depth	daTT	1683.40	1683.97	0.03%	0.18%
history-depth	daTT	1720.12	1720.45	0.02%	0.19%
concurrency	daTT	1722.09	1722.61	0.03%	0.22%
history-concurrency	daTT	1702.99	1703.62	0.04%	0.26%
nested-parallel	daTT	1695.95	1696.13	0.01%	0.26%

Table 4.11: Memory results for all optimization profiles across all browsers

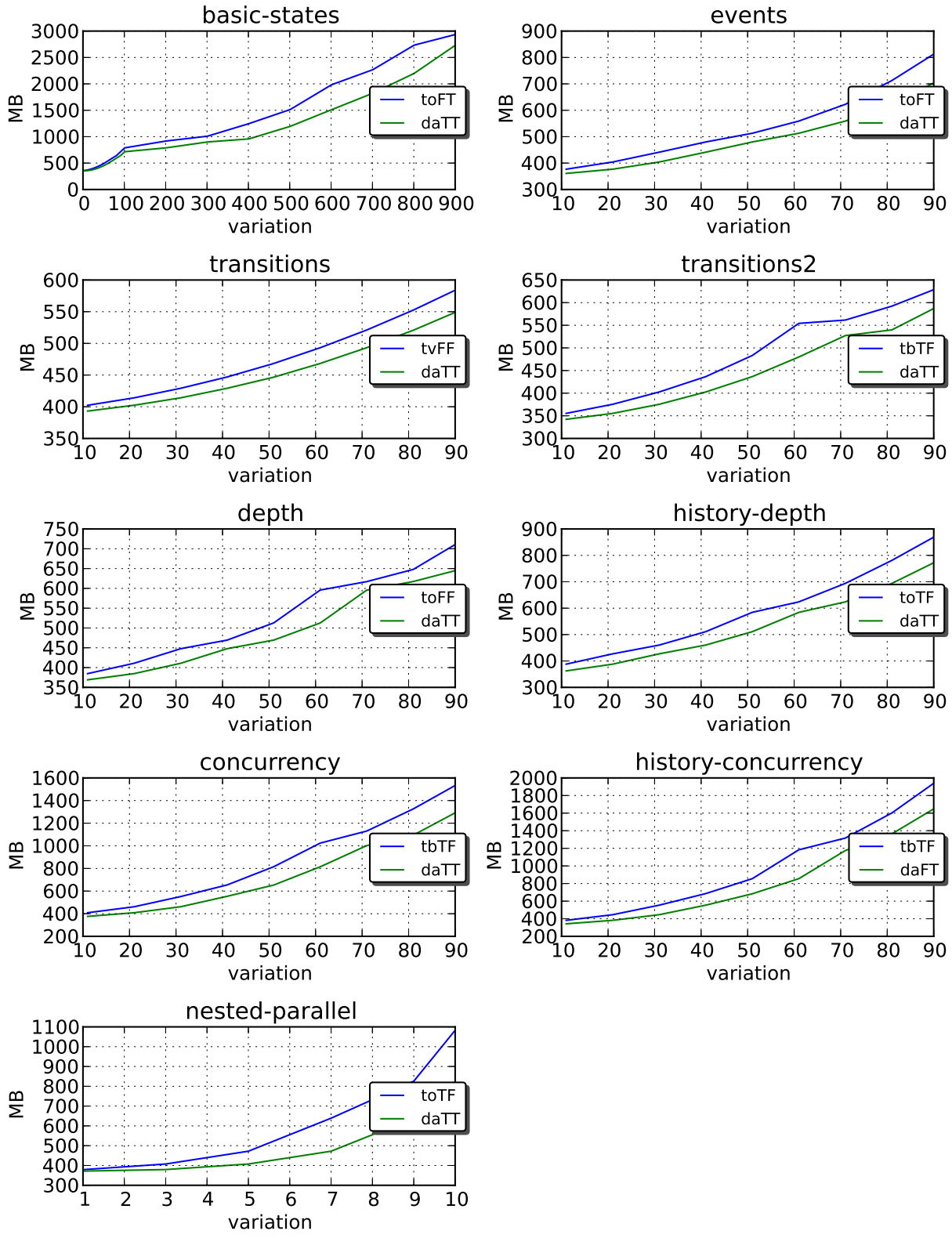

Figure 4.10: Comparison of best and worst optimization profiles for memory usage on Opera

4.6.3 Memory Usage versus Performance

Table 4.12 compares the performance and memory usage of the optimization profiles with the best performance against those with the best memory usage, for each test category and each Web browser. In this table, the optimization profile with the best performance is shown, followed by its average performance and virtual memory usage in megabytes. Next, the optimization profile with the best memory usage is shown, followed by its average performance and memory usage. The next two columns compare the performance of the profile with the best performance and the profile with the best memory usage, showing the difference and percentage difference between the two profiles. Finally, the last two columns compare the average memory usage of the two profiles, showing the difference and the percentage difference.

One may make the following observations. First, as predicted by the results of the previous section, there is very little difference in terms of memory usage between the optimization profile with the best performance and the profile with the best memory usage for Chromium and Webkit. The maximum percentage difference for Chromium is 0.42%, and the maximum for Webkit is 0.24%. There is also not much difference in memory usage for Firefox, where the maximum percentage difference is 5.52% (for test-category *history-concurrency*, and optimization profiles "taFT" and "daTT"), and the maximum difference in memory usage is 55.33MB (for test-category *basic-states*, and optimization profiles "taTT" and "daTT"). There were large differences in memory usage in Opera, however, with maximum percentage difference of 16.58% (for test-category *history-concurrency*, and optimization profiles "toFT" and "daFT"), and difference of 165.24MB (for the same test category and optimization profiles).

Second, there were large differences in performance between optimization profiles with the best performance and the optimization profiles with the best memory usage. For example, for test category *transitions2*, the optimization profile with the best performance offered speed increases of over 100% across all browsers. Another example is *basic-states*, where the optimization profile with the best performance offered a speed increase of 224.56% in Webkit, and smaller speed increases in other browsers.

group	best p.	ev/ms	VSZ	best m.	ev/ms	VSZ	p - m (ev/ms)	p / m (ev/ms)	m - p (MB)	m / p (MB)
firefox										
basic-states	taTT	12.31	1130.00	daTT	11.94	1076.22	0.37	3.14%	53.78	4.76%
events	taTT	12.75	867.21	daTT	12.14	839.90	0.61	5.03%	27.31	3.15%
transitions	dvFT	0.09	844.23	daTT	0.08	836.63	0.01	11.56%	7.60	0.90%
transitions2	taTT	12.74	865.77	daTT	5.02	836.02	7.72	153.65%	29.75	3.44%
depth	dvTT	5.95	828.91	daTT	5.82	822.32	0.13	2.28%	6.60	0.80%
history-depth	taTT	6.01	873.80	daTT	5.55	843.43	0.46	8.26%	30.36	3.47%
concurrency	dvFT	0.19	931.55	daTT	0.17	919.39	0.01	6.98%	12.16	1.30%
history-concurrency	taFT	0.40	1002.68	daTT	0.11	947.35	0.29	265.35%	55.33	5.52%
nested-parallel	saTT	2.33	883.10	daTT	2.00	838.27	0.33	16.57%	44.83	5.08%
chromium										
basic-states	taTT	33.93	667.72	daTT	32.41	667.08	1.52	4.69%	0.63	0.09%
events	taTT	37.18	660.10	daTT	34.07	659.64	3.11	9.13%	0.47	0.07%
transitions	cvTT	0.45	662.99	daFT	0.33	661.97	0.12	37.12%	1.01	0.15%
transitions2	caTT	22.08	458.60	daFT	6.17	456.67	15.90	257.65%	1.93	0.42%
depth	tvTT	14.74	659.02	cbFT	5.32	658.86	9.42	176.87%	0.16	0.02%
history-depth	taTT	14.19	659.51	cbFT	4.77	659.11	9.42	197.51%	0.40	0.06%
concurrency	tvTT	0.71	664.87	cbFF	0.05	663.39	0.66	1325.23%	1.49	0.22%
history-concurrency	tvFT	1.35	665.75	daFT	1.17	664.03	0.17	14.87%	1.72	0.26%
nested-parallel	taTT	6.43	662.21	daTT	1.04	659.96	5.39	516.53%	2.25	0.34%
opera										
basic-states	taTT	22.13	1043.06	daTT	21.03	917.72	1.10	5.22%	125.35	12.02%
events	taTT	25.12	543.84	daTT	21.74	496.65	3.38	15.56%	47.19	8.68%
transitions	cvTT	0.10	462.90	daTT	0.09	457.75	0.01	10.67%	5.15	1.11%
transitions2	caTT	22.07	452.58	daTT	6.53	450.15	15.54	238.17%	2.42	0.53%
depth	tvTT	7.22	531.05	daTT	6.36	495.03	0.86	13.49%	36.02	6.78%
history-depth	cvTT	7.33	545.36	daTT	6.56	536.83	0.78	11.85%	8.54	1.57%
concurrency	tvTT	0.22	870.87	daTT	0.19	742.04	0.03	15.91%	128.83	14.79%
history-concurrency	toFT	0.44	996.61	daFT	0.40	831.37	0.04	9.98%	165.24	16.58%
nested-parallel	cvTT	3.68	535.77	daTT	3.27	515.34	0.40	12.25%	20.43	3.81%
webkit										
basic-states	taTT	20.74	1684.81	dbFT	6.39	1680.80	14.35	224.56%	4.01	0.24%
events	taTT	22.45	1693.98	daTT	20.98	1691.46	1.47	6.98%	2.52	0.15%
transitions	dvFT	0.20	1712.65	daTT	0.19	1711.72	0.01	6.29%	0.93	0.05%
transitions2	caTT	21.45	1716.58	daTT	8.56	1715.60	12.89	150.48%	0.99	0.06%
depth	caTT	8.77	1685.37	daTT	8.60	1683.40	0.17	1.98%	1.96	0.12%
history-depth	caTT	8.67	1721.65	daTT	8.29	1720.12	0.38	4.59%	1.53	0.09%
concurrency	svTT	0.38	1725.92	daTT	0.36	1722.09	0.01	3.88%	3.83	0.22%
history-concurrency	daFT	0.72	1703.62	daTT	0.21	1702.99	0.51	245.37%	0.63	0.04%
nested-parallel	daTT	4.27	1695.95	daTT	4.27	1695.95	0.00	0.00%	0.00	0.00%

Table 4.12: Optimal Memory Profiles vs. Optimal Performance Profiles

4.6.4 Conclusion

There are several ways to apply these results. The first and most straightforward would be to recommend a general optimization profile that should provide the best performance and memory usage across as wide a range of browsers and usage scenarios as possible. The clear winner for this would be optimization profile "taTT", or Table transition selection and Array Set, with model caching and the flattening transformation enabled.

The transition selection optimization is especially important for dense models, as can be seen in the results of the *transitions2* test category, but can increase performance for large, sparse, flat models, as can be seen in test categories *basic-states* and *events*. Any transition selection option involving generated code (i.e., any option other than the Default option), performs similarly for *transitions2*, but the Table option performs well across all browsers for test categories *basic-states* and *events* as well.

Furthermore, while the Table option is most expensive in terms of memory use, this primarily affects Opera, and to a lesser extent Firefox and Chromium. Webkit is not affected at all. While there exist mobile versions of Opera, Firefox and Chromium, the Webkit browser, with the JavaScriptCore ECMAScript implementation, is shipped as a part of the default Web browsers of most mobile operating systems, including Apple iOS and Google Android. At the same time, when targeting desktop Web browsers, memory use is less of a concern. Therefore, the Table transitions selection option should be recommended for general use, as it provides the overall best performance across the widest range of browsers and usage scenarios, and should not affect memory usage at all in JavaScriptCore, the ECMAScript implementation most likely to be used in mobile, memory-restricted environments.

The Set implementation optimization can improve performance for large, sparse, flat models such as those in *basic-states* and *events*, and Array Set provides the best performance for these. The Bit Vector set provides good performance for the *depth* and *history-depth* test categories in Chromium and Firefox, but the speed increase is nominal compared to that of Array Set, and Bit Vector performs worse for test categories *basic-states* and *events*, particularly for Webkit and Opera. Therefore, Array Set is recommended for general use.

Flattening can increase performance in most scenarios and across all browsers. The one exception to this is in cases where history is combined with concurrency, as is captured in test category *history-concurrency*. As it improves performance for most other scenarios, the flattening transformation is recommended for general use.

Finally, model caching improves performance in most scenarios across all browsers, and it never leads to a decrease in performance or an increase in memory usage, therefore it is recommended for general use.

In the future, it may be possible to generalize the above results into a set of heuristics that can be applied to optimize execution on-the-fly. In this scenario, static analysis techniques could be applied when a Statecharts model is loaded into the user's web browser, in order to derive relevant properties, such as its transition density, maximum depth, use of history, and maximum number of concurrent states. The target environment could also be considered in cases where performance was found to differ among browsers. Heuristics could then be applied to these properties in order to select an optimization profile that could be used to

optimally execute the Statecharts model.

Chapter 5

Web User Interface Development with the Stateful Command Syntax Design Pattern

5.1 Overview

A "design pattern" is defined as "a solution to a problem in a context" [GHJV95]. A new design pattern called *Stateful Command Syntax*, or *SCS*, will be described and illustrated with two case studies.

The "problem" which the SCS design pattern attempts to solve is that some system behaviour comprises complex "command syntax" that varies depending on high-level application state. The "context" of SCS is the domain of User Interfaces. In many user interfaces, UI events can be entered in sequences to form complex commands, and the language of possible commands that can be entered forms a "command syntax". Furthermore, many user interfaces can change between a fixed set of possible high-level application states, or "modes," and the command syntax may change depending on the mode the application is in.

The "solution" of the SCS design pattern is to apply Statecharts in the following way. First, event listeners are registered that map UI events to Statecharts events, and dispatch them to a single Statecharts model instance.

This Statecharts instance encapsulates the behaviour of the UI, and has the following properties. First, the Statecharts model has a single top-level AND state, with two or more orthogonal components. At least one component is primarily responsible for encoding the high-level application state, or *mode*, and the other orthogonal component is primarily responsible for encoding multiple command syntaxes that vary depending on that mode.

A single UI command can often be described using a regular expression, and a regular expression has a deterministic finite state machine representation[Sip06]. The idea behind the orthogonal component that encodes the command syntax, then, is to use state hierarchy in order to elegantly combine the finite state machine representations of multiple command syntaxes, such that the common states and transitions are shared, and the overall number

of states and transitions are minimized. This will typically yield a transition-dense orthogonal component, with one "idle" or "ready" default state from which originates multiple transitions, with triggers corresponding to UI events, that target intermediate states. These intermediate states may connect to other intermediate states, until ultimately the command completes, and a loop is formed back to the "ready" state, in which case the system is ready to begin processing a new command. Furthermore, the transitions that connect the intermediate states will make heavy use of transition conditions in order to direct the flow of events through the network of intermediate states, thus encoding a command syntax that may vary depending on certain conditions. In particular, the `In()` predicate will often be used in these transition conditions in order to inspect the application mode encoded by the orthogonal component that encodes application mode. In this way, multiple command syntaxes are encoded which vary depending on mode.

The orthogonal component that encodes application mode will typically contain several *mode states* that correspond directly to high-level, named modes in the application. From these mode states will originate transitions that connect to other mode states, thus encoding possible changes in application mode given the current mode and a particular input event.

The transitions between mode states will have triggers that correspond to both UI events, as well as events sent from the orthogonal component that encodes command syntax. This is due to the fact that a common UI requirement is for a particular command to change the application mode, and therefore, the orthogonal component that encodes the command syntax may send an event at the end of a command to which the other orthogonal component must react, possibly prompting a state change, and thus a change in mode. On the other hand, some individual UI events, not part of a sequence of events that form a command, may also change the application mode, and this can be encoded as a simple transition between mode states, with a trigger corresponding to a UI event.

These techniques will be described in more detail in the following two case studies. The purpose of these case studies is to evaluate the expressiveness and generality of SCS. If SCS can capture a non-trivial subset of the user interface behaviour of two highly interactive, but very different existing applications in such a way that it seems to reduce accidental complexity, then it should be possible apply SCS effectively to other similarly complex applications.

Both case studies can be run run live in a Web browser, and can be found at the following URL: `https://github.com/jbeard4/scion-demos`

5.2 Case Study 1: Application of SCS to Vim Modal Text Editor

5.2.1 Introduction

The goal of this case study was to develop a text editor based on the Vim text editor using SVG, ECMAScript and Statecharts. This case study was developed in 2010 for course

MINF10121, "Modeling and Transformation in Software Development," at the University of Antwerp, taught by Professor Hans Vangheluwe.

Vim's User Interface (UI) behaviour is complex for a number of reasons. In simple text editors, like Microsoft Notepad, when the user enters printable characters on the keyboard, they are directly inserted into a text buffer and rendered on the screen. Typically, arrow keys and the mouse are then used to navigate within the document. This behaviour does not change very much depending on application state. In Vim, on the other hand, printable characters can be entered in sequences to form complex commands. The language of possible commands that can be entered forms a command syntax. Furthermore, Vim is a "modal" text editor, which means that it can change between a fixed set of possible application states, referred to in documentation as "modes." Vim's command syntax also changes depending on the mode that Vim is in. Finally, Vim includes other features that can also alter the command syntax, an example of which is macro recording.

From a high level, the approach employed in this project was to use the Vim Reference Manual [BM08] to obtain a natural-language specification of Vim's behavioural requirements. This behavioural specification was then described using Statecharts. Concurrently, an editor framework was developed in SVG and ECMAScript. The Statecharts model was then embedded within the editor framework and executed using SCION to produce a working Web-based user interface.

5.2.2 Case Study Goals

This case study had several high-level goals. First, it was necessary to implement a reasonable subset of Vim's features in SVG and ECMAScript. Next, it was important to minimize accidental complexity in the implementation, such that the final result would be easily understood and maintained by a developer. It was also important that the resulting implementation be robust, which is to say, reasonably stable and free of bugs.

Furthermore, it was necessary for the application to be usable. Of particular concern when developing a text editor is responsiveness: the user should not be able to perceive a delay between the sending of an event, and the corresponding visual changes in the UI. Therefore, it was necessary for common editing operations such as moving the cursor, entering text and changing modes to seem to occur instantaneously.

5.2.3 Editor Requirements

A goal of this project was to implement a reasonable subset of Vim's features. These features were chosen based upon the author's personal, day-to-day use of Vim.

These requirements can be summarized as follows:

- The editor can switch between Normal, Insert, Command, Visual, and Select modes. These are all of Vim's modes, except for Replace and Ex mode. A complete description of Vim's mode-switching behaviour can be found in [BM09]. This project implements a subset of that behaviour, which is summarized in Table 5.1.

FROM mode	TO mode				
	Normal	Visual	Select	Insert	Command
Normal		v, V, ctrl_v		i, a	:
Visual	esc		ctrl_g		
Select	esc	ctrl_g		*printable character*	
Insert	esc				
Command	esc, enter				

Table 5.1: Editor Mode-switching Behaviour

- In Normal Mode, the following movement commands are available:
  - Move the cursor left, down, up, and right (triggered by "h"/"left arrow","j"/"down arrow", "k"/"up arrow", and "l"/"right arrow", respectively).
  - Move the cursor to the beginning of the next word, end of next word, and beginning of previous word (triggered by "w", "e", and "b", respectively).
  - Move the cursor to the beginning of the current line, the end of the current line, the beginning of the document, and the end of the document (triggered by "0", "$", "gg", and "G", respectively).
- Text can be copied and pasted to and from registers (triggered by "dd"/"yy", and "p" while in Normal Mode, respectively).
- Commands can be repeated by prefixing the command with a count. For example, "3dd" while in Normal Mode would delete the next three lines.
- Text can be entered while in Insert Mode.
- Visual and Select modes support Line, Character and Block selection.
- Macros can be recorded and played back.

5.2.4 Application Architecture

Before developing the behaviour of the SVG-based text editor, it was necessary to construct a framework that would provide a programming interface to support basic text editing operations. The application's architecture is illustrated in Figures 5.1 and 5.2. The following is an overview of the classes used in the application.

Overview of Classes

Line Conceptually, a Line is a string without "newline" characters. A Line is responsible for controlling the string's representation in SVG. This is non-trivial, because SVG 1.1 does

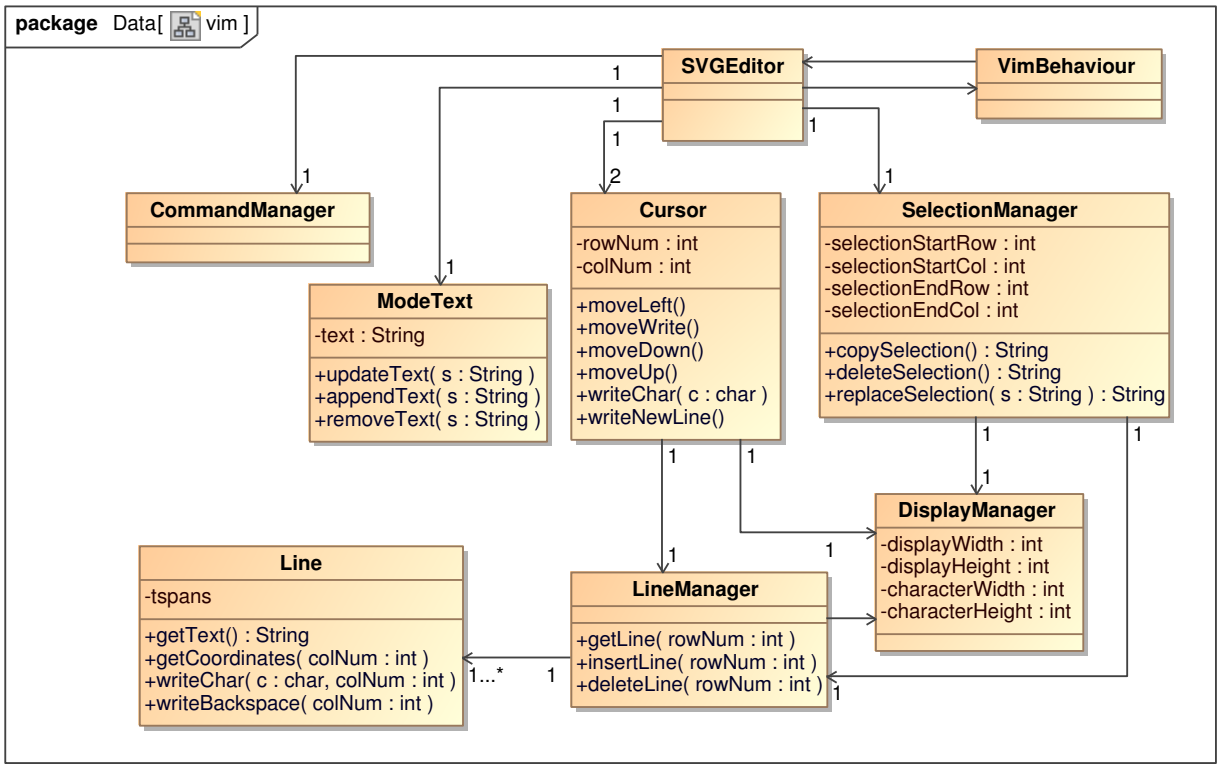

Figure 5.1: Class Diagram of Visual Editor

not provide built-in support for word or character wrapping, and the string must therefore be broken up across multiple `<tspan>` elements. Line then implements algorithms for wrapping text across multiple tspan elements, which are applied upon insertion or deletion of a new character.

Furthermore, Line provides methods to map a character's index to its coordinates on the screen.

LineManager The LineManager aggregates multiple Lines to form a text buffer. It provides methods for getting a Line given its row index, and creating and deleting Lines. The LineManager will always have at least one Line. If the last Line is deleted, it will immediately create a new empty Line to take its place.

Cursor The Cursor is a visual icon on the screen, which indicates where text will be inserted in the text buffer. The cursor has a row and column index, and is responsible for inserting and deleting text into the LineManager at its current position. Furthermore, the cursor has a visual representation, and is responsible for updating that visual representation when its row and column indexes change.

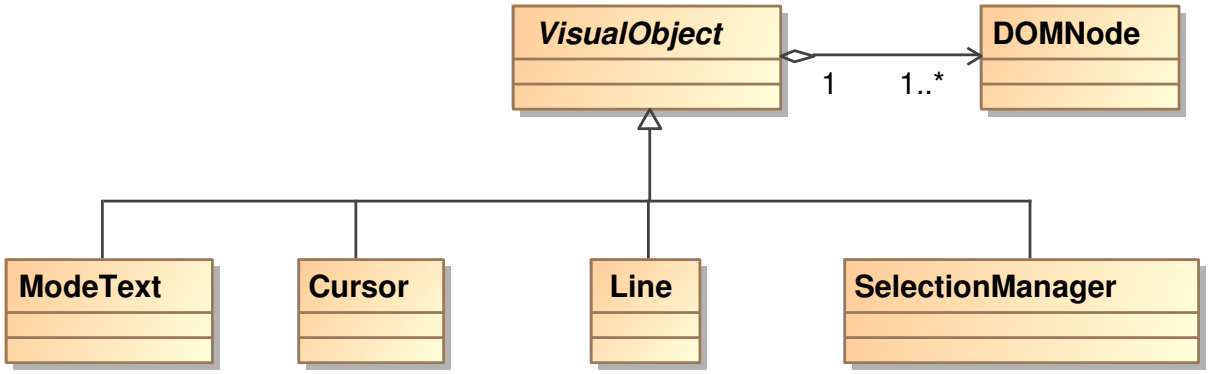

Figure 5.2: Classes extending VisualObject have a visual representation

SelectionManager The SelectionManager stores a pair of row and column indexes, which correspond to a range of selected text. It provides methods to copy the text in that range, as well as to delete it or replace it. Furthermore, it controls its visual representation as its attributes are updated.

DisplayManager The DisplayManager is responsible for providing the width and height dimensions of the display, and of the fixed-width font used by the editor. This information is used for text wrapping, and is thus associated with Line and SelectionManager.

ModeText The ModeText is responsible for controlling the text at the bottom of the screen, which indicates the editor's mode (e.g., "-- INSERT --" for Insert Mode). An example of this text is shown in the bottom, left-hand corner of Figure 5.3. ModeText provides methods to set the mode text, and append or remove a string from the mode text.

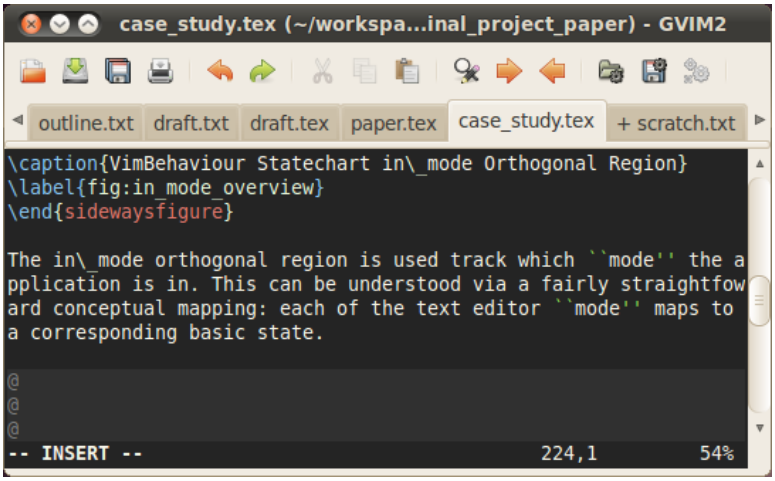

Figure 5.3: Screenshot of GVim text editor with mode text in bottom left corner

CommandManager The CommandManager stores the various commands which may be executed by the editor in Command Mode. Currently, accepted commands are "!", which evaluates an arbitrary ECMAScript string, and "help" which prints some useful help text.

VimBehaviour VimBehaviour is responsible for managing the application state. It is implemented by a Statecharts model.

SVGEditor There is one SVGEditor instance per application instance. SVGEditor acts as a controller to the editor framework for the Statecharts model instance.

5.2.5 VimBehaviour Statechart Design

High-Level Overview

Figure 5.4 provides a high-level overview of the design of the VimBehaviour statechart. There is an AND-state `main`, a final state, and a basic state `initial_default`. `initial_default` is the top-level default state, and so the application begins in the `initial_default` state. The purpose of `initial_default` is to provide the opportunity for the environment to send the system a controller object. The controller object is sent to the statechart as data on the `init` event (`_event.data`), and is assigned to variable `controller`, which is local to the statechart. The controller object will be an instance of SVGEditor, and will allow the statechart to control the text editing environment via calls to its API in the action code of the statechart.

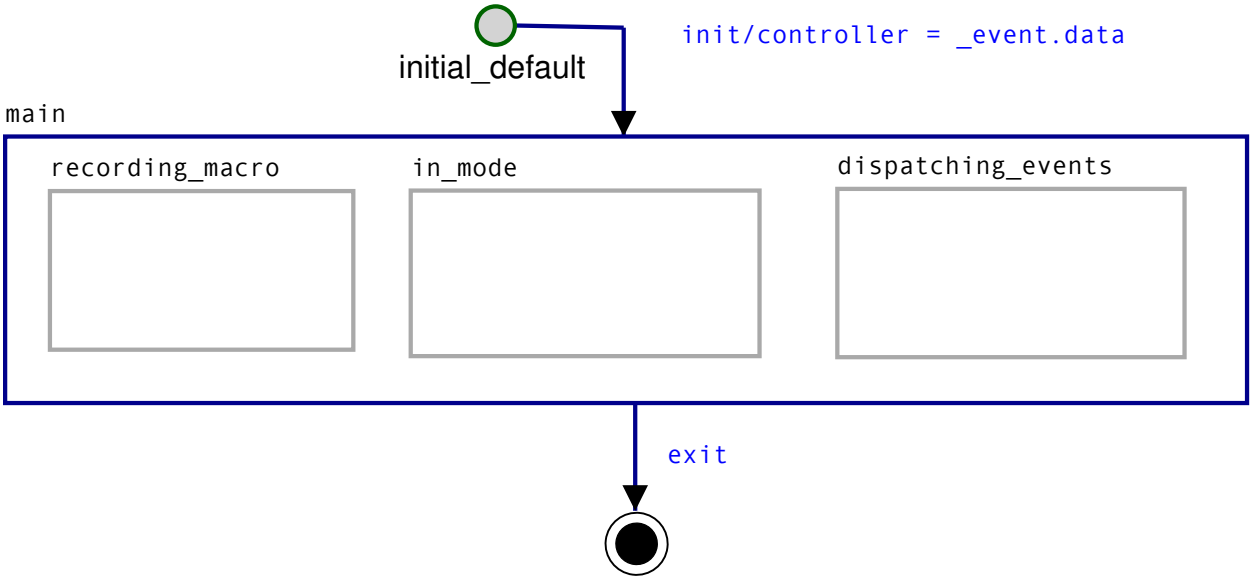

Figure 5.4: VimBehaviour Statechart Top States

Upon receiving the `init` event, the statechart transitions to state `main`, an AND-state with three orthogonal components. The statechart remains in the components until the

statechart receives `exit`, causing the statechart to enter a final state, and the application to quit. The orthogonal components of `main` will be discussed in detail in the following sections, but at a high level, one may describe them as follows. `in_mode` tracks the application's "mode". `dispatching_events` is responsible for encoding command syntax depending on mode. Finally, `recording_macro` handles the logic pertaining to macro recording.

This follows the SCS technique of using one or more top-level orthogonal components primarily to capture application mode, and one orthogonal component primarily to dispatch events based on that mode.

Mapping User Interface Events to Statecharts Events

User Interface events must first be mapped to Statecharts events. The domain of the mapping, or set of possible User Interface events, depends on the underlying platform upon which the User Interface is built. SVG uses the Document Object Model (DOM) Level 2 Events Specification for event handling [DFF+10, Ch. 16], and so the domain of the mapping is the set of DOM events. The co-domain of the mapping is the universe of possible Statecharts events, which has been described in Chapter 2 as a structure with a property "name" of type String, and property "data" of type Object. In SCION, any ECMAScript String can be used as an event name, except for the wildcard event `*`, which has a special interpretation in SCION semantics, such that it matches any event (described in Section 2.3).

Initially, a simplifying assumption was made for this case study to only deal with keyboard events, and not mouse events. However, it would not be difficult to extend the following mapping to also handle mouse events.

The mapping from DOM events to Statecharts events can be defined as follows. In the case that the user inputs a printable character, that printable character will be used as the Statecharts event name. In DOM events, printable characters are distinguished by their `charCode` attribute, which will be equal to zero if the character is not printable, and will equal a positive integer if the character is printable. A string representation is derived from the `charCode` by using the ECMAScript built-in `String.fromCharCode` method. So, for example, when the user presses the "e" key, a DOM event is dispatched to a DOM event listener. The `charCode` attribute of the dispatched DOM event would be set to 101. When this `charCode` is passed into `String.fromCharCode`, the string "e" would be returned, which would then be used as the Statecharts event name.

In the case of non-printable characters, a mapping is defined from the character's unique `keyCode` attribute to a symbolic string name, for each event that must be explicitly handled by the statechart. Here, "explicitly handled" means that the statechart contains transitions with events which correspond to that non-printable character. Examples of non-printable characters that must be explicitly handled by the VimBehaviour statechart are "esc", "left", "right", "end", and other non-printable movement keys. So, for example, if the user presses the left arrow key, a DOM event with `keyCode` 37 would be dispatched to the DOM event listener. Because `charCode` on the event object would be 0, the event would be sent to the `keyCode`-to-name map, which in this case would return the string "left". Note that, in this mapping, it is important that the `keyCode`-to-name map does not return strings that are

single characters, as this would collide with the mapping applied to printable characters.

The final step is to then map modifier keys to a symbolic string representation (e.g., "ctrl", "shift", "alt", and "meta"), and to then prepend that string to the derived symbolic event name. The order in which these modifier strings are prepended matters, as for example, Statecharts event `ctrl_shift_a` would be distinct from `shift_ctrl_a`.

The final resulting string is then used to instantiate a Statecharts event, with name property set to that string, which is is then dispatched upon the statechart instance. Furthermore, the original DOM event object is sent as Statecharts event data. The statechart therefore has access to platform-specific data (such as `keyCode` and `charCode`) from the DOM event.

Advanced UI Event Mapping

Sometimes it is desirable to specify ranges of acceptable User Interface events for an individual transition. For example, in state `before_nonzero_digit` the state machine should transition to `after_nonzero_digit` if sent an event in the range of 1 through 9.

The solution to this problem was to use a transition with the wildcard event, and a transition condition to determine if the event belonged within a specified set of characters. ECMAScript regular expressions were used to determine this. This meant that regular expressions could be used to define event ranges, and that a transition would not be selected unless the given event is a member of the set defined by that range.

Modelling Application Mode in Orthogonal Component `in_mode`

The `in_mode` orthogonal component is used to track which editing "mode" the application is in. This can be understood via a straightforward mapping of Vim's editing concepts onto the Statecharts language: each of Vim's editing "modes" maps to a corresponding basic state in the Statechart model. This is shown in Figure 5.5.

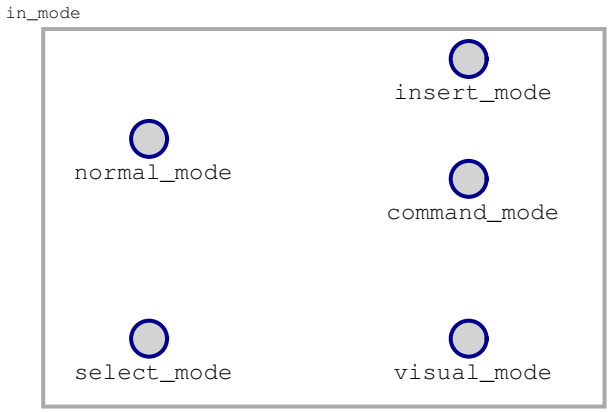

Figure 5.5: Vim modes mapped onto states

Vim changes modes when the user enters certain characters. For example, when in

Normal Mode, `i` will bring the editor into Insert Mode. Likewise, when in Insert Mode, `esc` will return the editor to Normal Mode. This can be described using Statecharts with a transition from state `normal_mode` to state `insert_mode` on event `i`, and with a transition from `insert_mode` to `normal_mode` on event `esc`. This is shown in Figure 5.6.

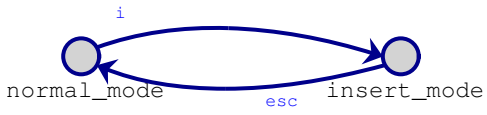

Figure 5.6: Transitions between mode states

In Vim, when a mode is exited, and a new mode is entered, the mode text is updated to correspond to the new mode. For example, changing to Insert Mode will update the mode text to "-- INSERT --". An example of this is shown in Figure 5.3. In Statecharts, this is expressed by adding an enter action to each mode state which calls the method `controller.updateModeText`, and passes in a string corresponding to the state that will be entered.

The Vim Visual and Select modes require an additional refinement to the above mapping. In Vim, for both Visual and Select modes, when the mode is entered, the editor begins selecting text, and when Visual or Select mode is exited, the editor stops selecting text, and the selection is cleared. Furthermore, `ctrl_g` toggles between Visual and Select modes without clearing the selection.

In Statecharts, this can be expressed by creating a OR state `visual_or_select_mode`, with `visual_mode` and `select_mode` as substates. When `visual_or_select_mode` is entered, the selection is started by calling the method `startSelection` on the controller, and when it is exited, the selection is cleared by calling `clearSelection`. This then allows the system to transition between `visual_mode` and `select_mode` without clearing the selection. This is shown in Figure 5.7.

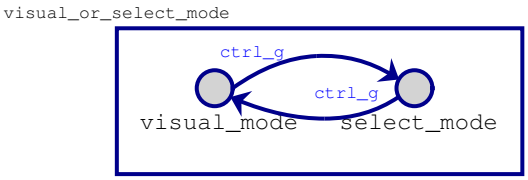

Figure 5.7: Visual and Select modes as basic states in OR state

One further refinement to the statechart representations of Visual and Select modes is needed. In Vim, there are three possible forms of Visual mode and three possible forms of Select mode: Visual Line, Visual Block, Visual Character; and Select Line, Select Block,

Select Character. These are entered into from Normal Mode in different ways. `v`, `V`, and `ctrl_v` events will bring the statechart into Visual Character, Visual Line, and Visual Block modes, respectively. Also, when toggling between Visual and Select modes, the Line, Block, and Character modes will be preserved. For example, if the editor is in Visual Block mode, and transitions to Select mode, it will then be in Select Block mode. The same is true when transitioning from Select to Visual mode, and for Line and Character modes. Finally, the mode text will be updated to "-- VISUAL BLOCK --" when entering Visual Block mode, and so on, for all other combinations.

The solution that was chosen is shown in Figure 5.8. In this solution, `visual_mode` and `select_mode` are OR states, and Visual Character, Visual Line, etc. are basic substates. On `ctrl_g`, transitions take the system from Visual Line to Select Line, and vice versa, for each Line, Block, and Character state. Finally, each Line, Character, and Block mode can switch to other modes without changing from Select to Visual mode with `v`, `V`, and `ctrl_v`. On entering each basic state, the mode text is updated appropriately.

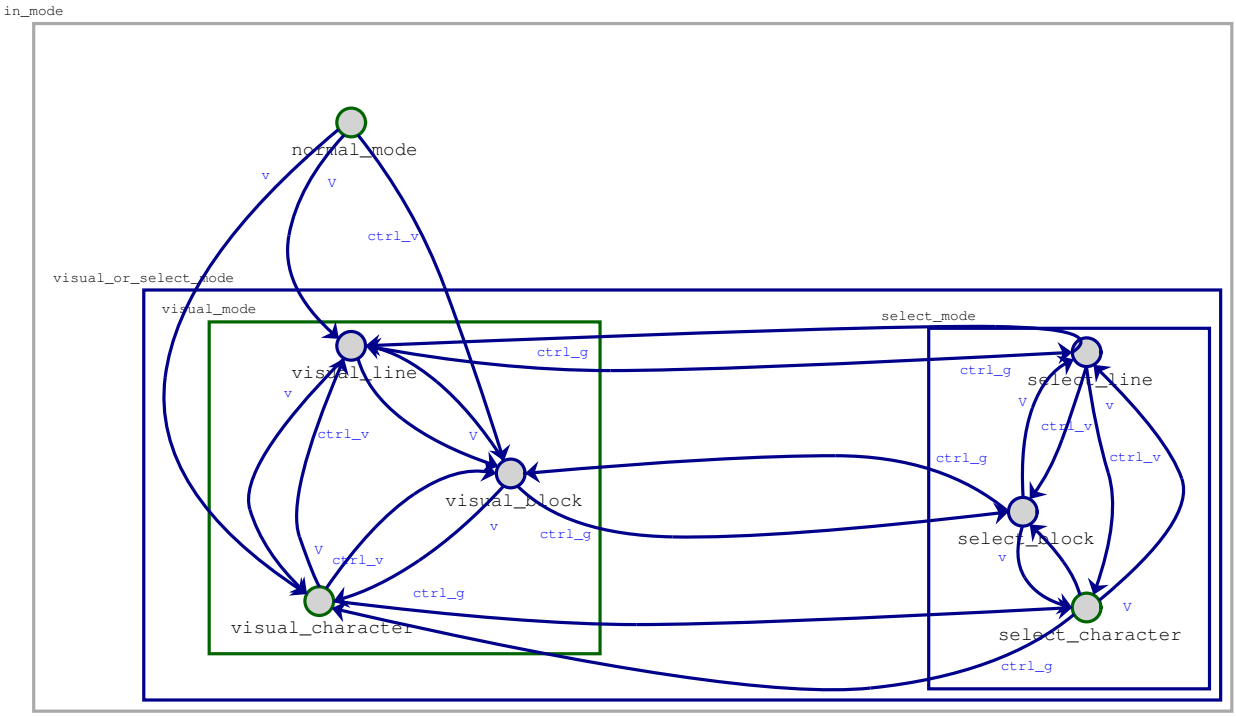

Figure 5.8: Line, Block and Character Selection modes as substates in Visual and Select OR states

The full solution to `in_mode` is shown in Figure 5.9.

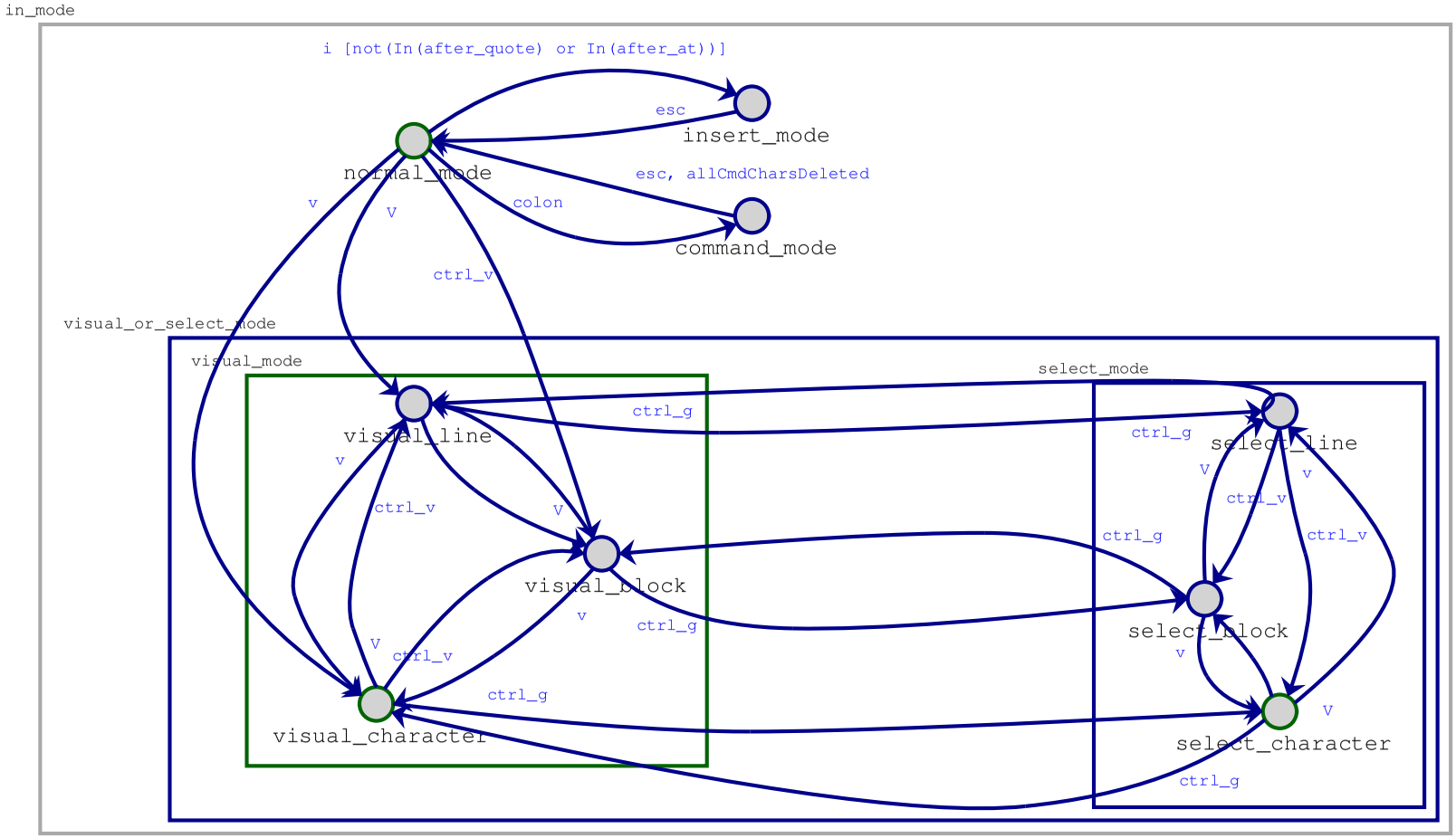

Figure 5.9: VimBehaviour Statechart `in_mode` Orthogonal Component

Modelling Mode-Dependent Command Syntax in Orthogonal Component `dispatching_events`

The main purpose of `dispatching_events` is to define a syntax for parsing commands composed of sequences of events, such that the syntax may vary depending on the mode the application is in. Furthermore, it provides a way to group common behaviours shared by different modes.

The following are some examples of Vim commands and their meanings:

- "`"`i12yy" : yank ("copy") the next twelve lines into register "i"
- "15@a" : play back the contents of macro register "a" 15 times
- "12gg" : move the cursor to line 12

As an example of how this syntax changes depending on mode, the above commands are processed in the manner described while in Normal Mode, but when in Insert Mode, the character data corresponding to each event would be inserted directly into the main text buffer.

Figure 5.11 shows a way of describing this behaviour using Statecharts.

The initial state is `before_nonzero_digit` in OR state `main_dispatching_events`. The statechart keeps a variable called `count`. If it receives an event in the set [1-9], it updates the count with this number and transitions to `after_nonzero_digit`. While in `after_nonzero_digit`, when an event in the set [0-9] is received, the count is updated. This count will be used later to determine how many times to repeat a command when it is invoked.

It is necessary to have the two states `before_nonzero_digit` and `after_nonzero_digit`, because, if a `0` is entered before a number in the range [1-9] is received, and the editor is in Normal Mode or Visual mode, then the cursor should be moved to the first column on the current line, and the count should be reset.

In Vim, simple commands can be invoked before or after any numbers have been entered. Here, "simple" commands refers to commands that are executed after entering a single character. Examples of simple commands include those triggered by movement keys, such as `h` or left arrow to move left, or `$` to move to the last column of the current line, and all printable characters when in Select, Insert, and Command Modes, in which case the editor will write the character to the main or command text buffers.

Complex commands are executed after receiving two or more keypress events. Examples of complex commands include "dd" (cut a line), "yy" (copy a line), "gg" (go to first line), "`"`*character in [a-zA-Z]*" (select the register into which text will be copied), and "@*character in [a-zA-Z]*" (invoke macro in register *character*). There are complex commands that are composed of more than two characters, but the solution presented restricts itself to the complex commands listed.

Commands, whether simple or complex, will be performed a number of times equal to the count. After performing a command, the count will be reset to 0.

Simple commands are realized using Statecharts with transitions from `main_dispatching_events` to itself. Drawing all of them would lead to a cluttered diagram,

so they are grouped here under a single transition labelled with an ellipsis. By making the transition exit the OR `main_dispatching_events` state, it is possible to elegantly fulfill the requirement that these simple commands be performed regardless of whether any digits have been entered, as the transition will be taken if the statechart is in `before_nonzero_digit` or `after_nonzero_digit`. Furthermore, by making the transition enter the `main_dispatching_events` OR state, and putting an entry action that clears the count on the `main_dispatching_events` state, it is possible to elegantly fulfill the requirement that, upon completing a command, the count should be cleared.

Complex commands are realized by transitioning to an intermediate state in the OR state `completing_two_part_command`. Once in the intermediate state, the statechart will wait to receive the second event to complete the command. Upon receiving the second event, the statechart will invoke the appropriate command, and transition back to `main_dispatching_events`.

The advantage of using a OR state to group these intermediate states is that it elegantly fulfills the requirement that, if the user enters an event which is not a legal part of the command syntax (e.g., an `x` following a `g`), then the editor should clear the count and transition back to a mode where it is ready to receive a new command. In essence, by default, if the statechart receives an input that is not part of the command syntax, then the statechart will return to a ready state. This is expressed by connecting a transition with an event `*` from `completing_two_part_command` to `main_dispatching_events`.

Using the `In` Predicate to Control Event Dispatch Based on Mode

Not every mode will have the same command syntax. For example, Normal Mode would accept "`""ayy`" to yank the current line into the register "a". But in Visual mode, the "yy" command does not exist. Instead, pressing "y" once while in Visual Mode yanks the current selection into the default register, and pressing "`""ay`" yanks the current selection into the "a" register. These command syntaxes are mutually exclusive.

In order to implement these differences using Statecharts, the built-in `In` predicate is used as a condition on most transitions. The `In` predicate accepts a reference to a state, and returns `true` if the statechart's current configuration includes that state; otherwise, it returns `false`. Using the `In` predicate in a condition on a transition expresses the notion that certain transitions are only taken while in specific modes, hence varying the command syntax depending on mode. For example, in Vim, while in Normal Mode, if "i" is pressed, Vim will enter Insert Mode. The exception to this, however, is if "i" follows "`"`" or "@", as "i" will then be used to select a register. This can be expressed using Statecharts by putting the condition "`not(In(after_quote) or In(after_at))`" on the transition from state `normal_mode` to `insert_mode` in the `in_mode` orthogonal component. The same is true for transitions to command_mode (on ":") and `visual_mode` (on `v`, `V`, or `ctrl_v`). An example of this is shown in Figure 5.10.

The technique of using the `In` predicate to control event dispatch based on mode is a standard part of SCS, and will appear again in the following case study.

To summarize, `dispatching_events` uses elements of the Statecharts language to cleanly encode several different command syntaxes, which vary depending on the application mode.

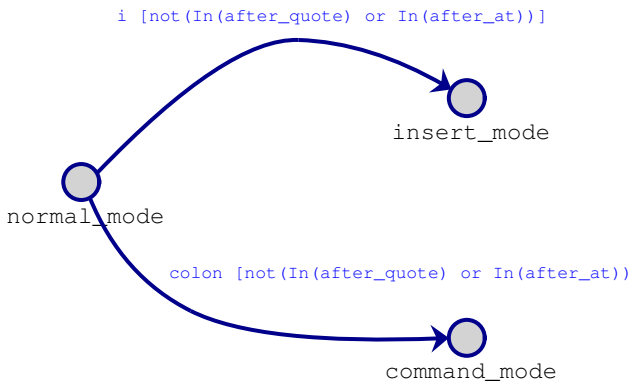

Figure 5.10: Example of `in_mode` mode affected by `dispatching_events`

Further Application Mode in Orthogonal Component `recording_macro`

Vim has the capability to record macros. These are sequences of events which are stored in a register selected by the user, and can later be played back as though the user had retyped them. Macro recording is initiated by pressing "q*character in [a-zA-Z]*" while in Normal Mode. After that, every keypress event will be stored in buffer *character*. Macro recording is stopped by pressing "q", also while in Normal Mode.

A solution to this using Statecharts is shown in Figure 5.12. By default, the statechart begins in state `before_q_keypress`, in OR state `recording_off`. When `recording_off` is entered, an empty event list data structure is initialized. If in `normal_mode`, pressing "q" transitions to state `selecting_register`. Any printable character then transitions the statechart to state `recording_on`, at which point all events will be appended to the event list. Finally, pressing "q" while in `normal_mode` will cause the statechart to transition back to OR state `recording_off`.

Pressing "@*character in [a-zA-Z]*" will then cause the events stored in register *character* to be dispatched to the statechart. This command is handled by orthogonal component `main_dispatching_events`.

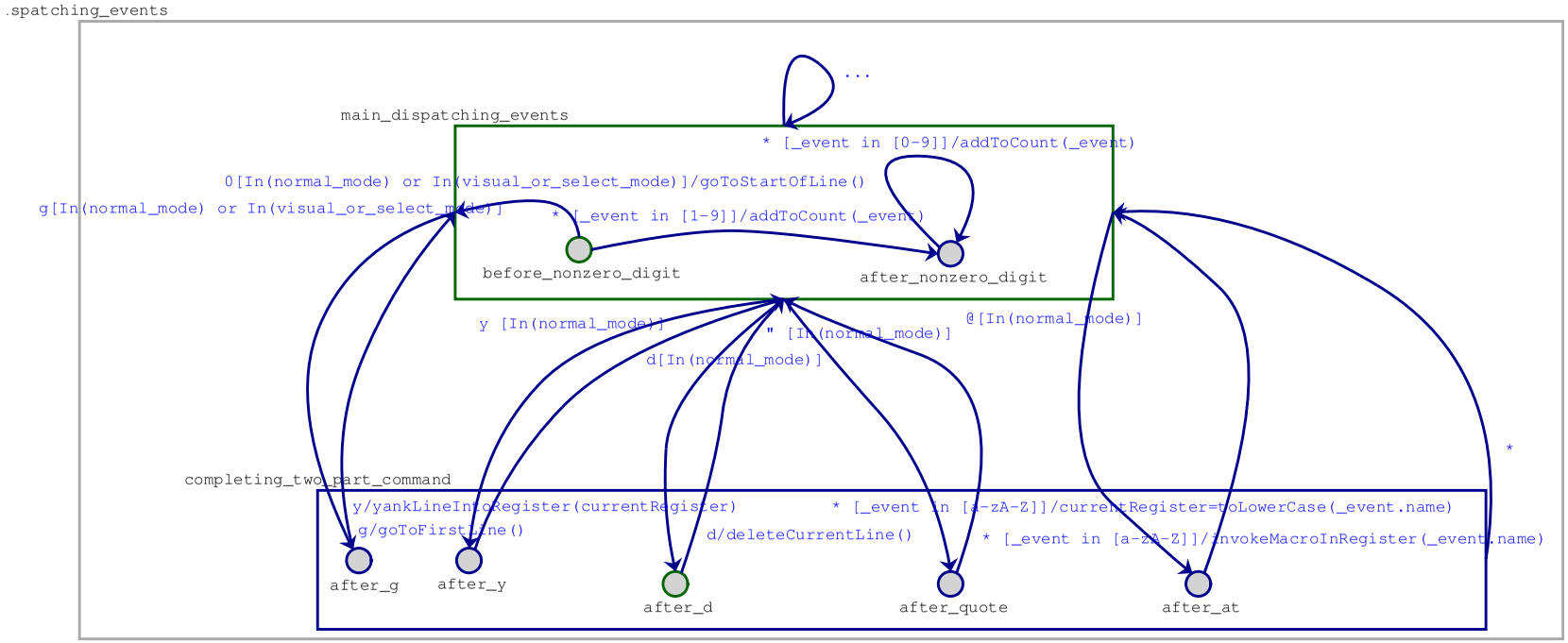

Figure 5.11: VimBehaviour Statechart `dispatching_events` Orthogonal Component

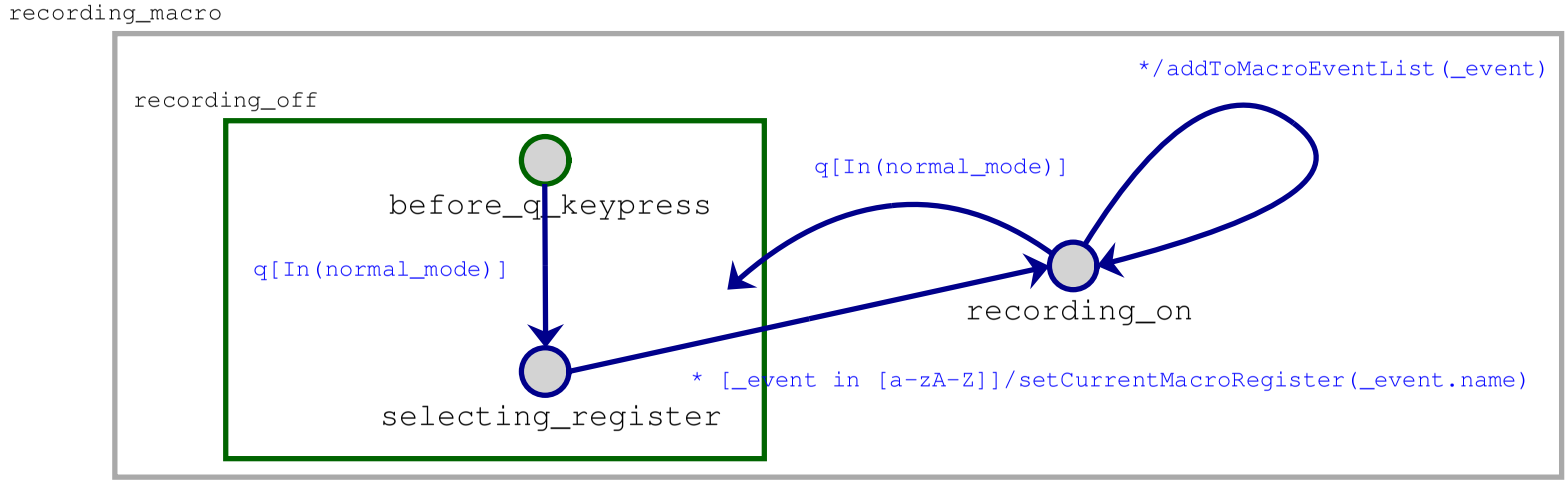

Figure 5.12: VimBehaviour Statechart `recording_macro` Orthogonal Component

5.3 Case Study 2: Vector Graphics Drawing Tool Based on Inkscape

5.3.1 High-Level Goals

The goal of this case study was to create a vector graphics drawing tool using SVG, ECMAScript, and Statecharts, whose functionality and user interface behaviour were derived from Inkscape, a free and open source vector graphics drawing tool. This case study will illustrate an application of the SCS design pattern.

The following subset of Inkscape's functionality was chosen to be implemented:

- Change between three tools on a toolbar: "transform", "draw rectangle", and "draw circle."
- Draw circles and rectangles on the canvas. These graphical objects will be referred to as "nodes" because of their close association with SVG DOM nodes.
- Select and deselect one or more graphical objects on the canvas.
- Graphically transform selected nodes on the canvas by rotating, translating and scaling them.

Furthermore, a meta-requirement was that the drawing tool must feel responsive at all times, particularly when the user was transforming a node by means of a dragging operation, which is to say, the user should not perceive a delay between the sending of a user interface event, and the corresponding visual change in the UI.

5.3.2 Natural-Language Specification of User Interface Requirements

The following is a natural-language specification of Inkscape's UI behaviour for the subset of functionality described in the previous section. In subsequent sections, it is shown how this natural-language specification of requirements was implemented by means of a Statecharts model.

There are five classes of objects with which the user could interact :

- A single Canvas.
- A single Rotation Handle, which the user manipulates to rotate selected nodes.
- A single Scale Handle, which the user manipulates to scale selected nodes.
- A single Toolbar with three buttons: the "Transform Button," "Draw Circle Button," and "Draw Rect Button." Only one button can be chosen at a time.
- A variable number of nodes, of which there are two sub-types: Rect and Circle.

In general, the user can perform the following operations on the above objects:

- "Create" a node by clicking and dragging on the Canvas.
- "Select" and "deselect" nodes by clicking on them, as well as in various other ways. To indicate which nodes are selected, selected nodes are surrounded by a transparent dashed rect, the "selection box", the dimensions of which are equal to the aggregate bounding box of of the selected nodes.
- Graphically transform selected nodes, by dragging them (to "translate"), by dragging the Rotation Handle (to "rotate"), or by dragging the Scale Handle (to "scale").
- "Choose" a tool on the toolbar. Note that, while "choose" and "select" are synonyms, in this context "choose" denotes an operation applied to a toolbar button, and "select" denotes an operation applied to a node. Likewise, a toolbar button will be referred to as "chosen", and a node as "selected".

The command syntax used to apply these operations is dependent on application state, primarily regarding which tool is currently chosen. The remainder of this section will provide a detailed natural-language description of this stateful command syntax.

Here, *mousedown*, *mouseup*, and *mousemove* refer to individual mouse events, while *mouseclick* refers to the sequence of events *(mousedown, mouseup)*, and *drag* refers to the sequence of events *(mousedown, mousemove+ mouseup)*, which is to say, one `mousedown` event, followed by one or more `mousemove` events, followed by a `mouseup` event.

Behaviour that Occurs Regardless of Which Tool is Chosen

A mouseclick on the canvas without holding the shift key results in all nodes being selected.

A mouseclick while holding the shift key on a node results in a node's selection state being toggled, such that a node which is not selected will be selected, and vice versa.

A mouseclick without holding the shift key on a non-selected node results in that node being selected, and all other nodes being deselected.

When the Transform Tool is Chosen

Rotation and Scale Handle If there are selected nodes, then either the rotate or scale handles will be visible, and will be positioned on the lower right corner (southeast position) of the aggregate bounding box of all selected nodes. Otherwise, if no nodes are selected, then neither the rotation nor the scale handle will be shown. A mouseclick (without shift) on a selected node results in rotation/scale handles being toggled, such that if the rotation handle is visible, it will be hidden and the scale handle shown, and vice versa. Dragging the rotation handle rotates all selected nodes about the center point of their aggregate bounding box. On mouseup, the position of the rotation handle is reset to the lower right corner of the aggregate bounding box of all selected nodes. Dragging the scale handle scales all selected nodes.

Additionally, when no nodes are selected, and a node is first selected, then the scale handles will be shown first, *unless* the last time nodes were selected, rotation handles were shown when they were deselected, *and* the nodes are selected via a drag, rather than a click.

Node Interaction Dragging a node without holding the shift key will deselect all currently selected nodes, drag the target node, and select the target node.

Dragging a node while holding the shift key has same effect as dragging on the canvas.

Dragging a selected node results in all selected nodes being dragged.

Dragging a non-selected node results in that node being selected, and all other nodes being deselected.

When the Rect or Ellipse Tools are Chosen

When the Rect or Ellipse Tools are chosen, rotate and scale handles will never be visible. Furthermore, mouseclick without holding the shift key on a selected node while in this mode has no effect. Finally, dragging anywhere (on the canvas or a node) results in a new element being draw, created and selected, and on mouseup, all other elements are deselected.

5.3.3 Application Architecture

The application's architecture is structurally similar to the application architecture described in the modal text editor case study in Section 5.2.4. Class diagrams containing an overview of the application architecture can be seen in Figures 5.13 and 5.14.

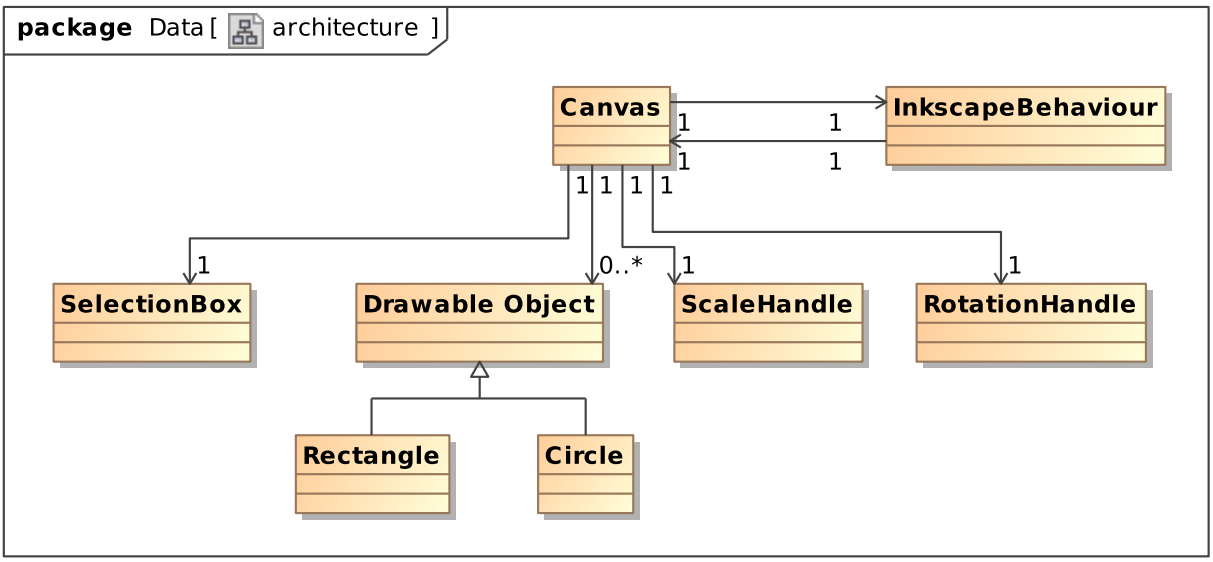

Figure 5.13: Class Diagram of Drawing Tool

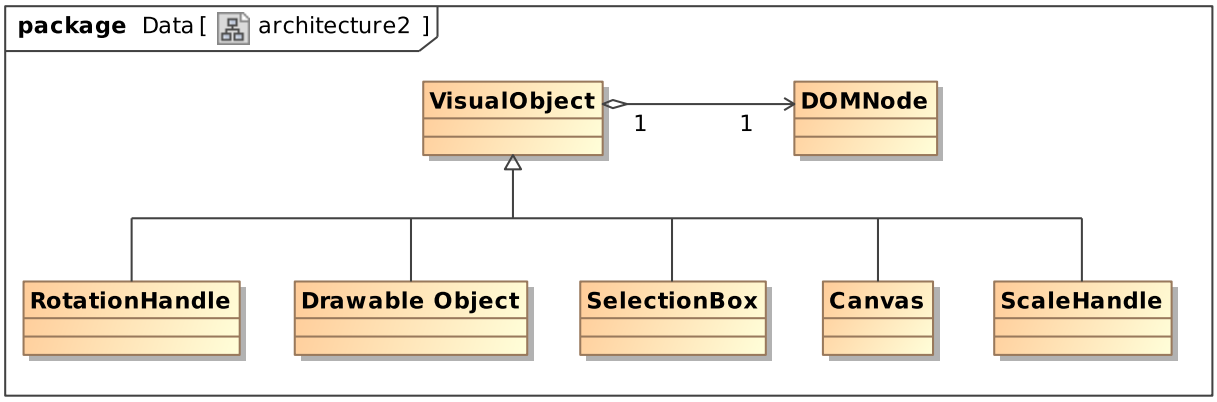

Figure 5.14: Classes extending VisualObject have a visual representation

Each class of graphical object described in Section 5.3.2 is represented as a class in Figure 5.13. There are one-to-one relationships between the Canvas, SelectionBox, ScaleHandle and RotationHandle classes, and a zero-to-many relationship between the Canvas and the DrawableObject class. Furthermore, the Canvas has a one-to-one association with an InkscapeBehaviour class which represents the InkscapeBehaviour statechart.

Each graphical object class is a subclass of VisualObject, which implies that it aggregates a DOM node. By changing the associated DOM node, its visual rendering will change on the page.

5.3.4 Mapping DOM User Interface Events to Statecharts Events

Mapping UI events to Statecharts events in this case study is not as complex as the mapping that was required for the text editor case study, described in Section 5.2.5. In this case, the only user interface events to which the application must react are three varieties of mouse events: `mousedown`, `mouseup` and `mousemove`. Furthermore, the Statecharts model should be able to detect whether the shift key was pressed, as well as the DOM node event target. These are both properties exposed by the DOM event, as specified in the Document Object Model (DOM) Level 2 Events Specification[DFF+10, Ch. 16]. Mapping the DOM event to a Statecharts event therefore entails mapping the DOM event name (e.g. `mousedown`) directly to the Statechart event's name (e.g. `mousedown`), and making the DOM event object the Statechart event's data. This will be clarified in the following sections.

5.3.5 InkscapeBehaviour Statechart Design

High-Level Overview

Figure 5.15 provides a high-level overview of the design of the InkscapeBehaviour statechart, which is similar to the VimBehaviour statechart described in the previous case study, in Section 5.2.5. Like VimBehaviour, there is an AND-state `main`, a final state, and a basic state

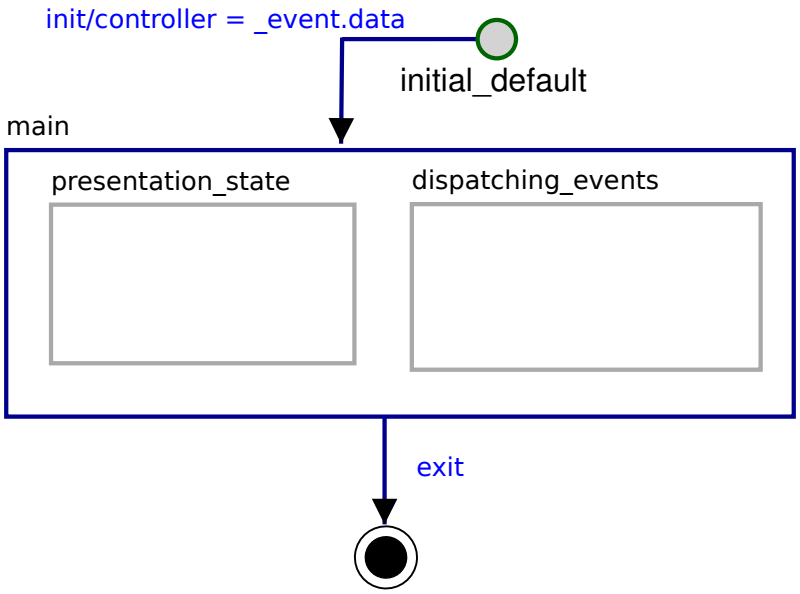

Figure 5.15: Top States

`initial_default`. `initial_default` is the top-level default state, and so the application begins in the `initial_default` state, and the statechart uses an `init` event to pass in a reference to a controller object, which provides an interface to the environment. Specifically, the InkscapeBehaviour controller object will contain references to the DOM nodes associated with the Canvas and Toolbar singletons, as well as a reference to an ECMAScript array that will contain references to all Rect and Circle DOM nodes on the canvas.

Upon receiving the `init` event, the statechart transitions to state `main`, an AND-state with two orthogonal components. `presentation_state` will model the application modes, and `dispatching_events` is responsible for encoding command syntax depending on mode.

Modelling Application Mode in Orthogonal Component `presentation_state`

Orthogonal component `presentation_state` captures high-level application state, or "mode". As described in the natural-language specification of the UI requirements, the high-level application state is most closely coupled with the state of the toolbar, which affects the presentation of the UI, for example changing which toolbar button is highlighted and whether the rotation and scale handles and selection box are shown, as well as changing the command syntax. To that end, the states in `presentation_state` have two main roles: to be inspected by the `dispatching_events` orthogonal component, thus encoding changes in command syntax, and to update the visual representation directly via action code in state entry and exit actions. This will be explained in detail below.

`presentation_state` has two top-level OR states: `transform_tool_selected` and `drawing_tool_selected`. `drawing_tool_selected` is furthermore broken down into two basic states, `rect_tool_selected` and `ellipse_tool_selected`. Together, these three states (`transform_tool_selected`, `rect_tool_selected` and `ellipse_tool_selected`) capture the

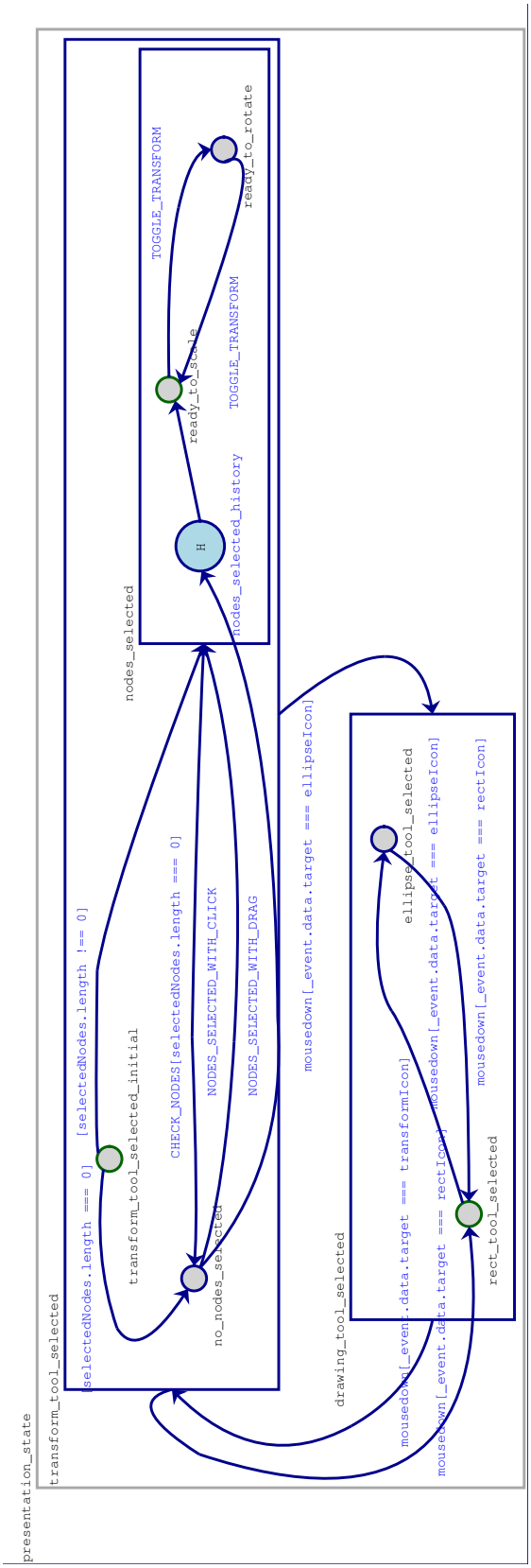

Figure 5.16: Orthogonal Component `presentation_state`

possible application states of the toolbar. Entering any of these states will highlight the associated button object, and exiting any of these states will remove the highlighting from the associated button object.

`transform_tool_selected`, `rect_tool_selected` and `ellipse_tool_selected` are all strongly connected by transitions, such that a `mousedown` event on a toolbar button will enable a transition to the button's associated state, regardless of the state the system is in. For example, a `mousedown` event on the ellipseIcon will cause a transition to `ellipse_tool_selected`, regardless of whether the system is in state `transform_tool_selected` or `rect_tool_selected`.

`transform_tool_selected` contains three substates: an initial state `transform_tool_selected_initial`, a basic state `no_nodes_selected`, and an OR state `nodes_selected`. Default transitions originating from `transform_tool_selected_initial` bring the system into states `no_nodes_selected` or `nodes_selected`, depending on whether or not the `selectedNodes` array is empty. A node is encoded as being selected by including it in the `selectedNodes` array. When the `selectedNodes` array is empty, this means that no nodes are selected.

In order to encode the logic that the dashed selection box and rotation handle or scale handle should only be shown when the transform tool is selected, the entry action on `nodes_selected` will show the selection box, and will hide it on exit. Likewise, the entry action of `ready_to_rotate` will show the rotation handle, and hide it on exit. `ready_to_scale` will do the same with the scale handle.

The system will transition between the substates of `transform_tool_selected` based on internal events which are sent from the other orthogonal components. For example, when a node is selected with a click, then the event `NODES_SELECTED_WITH_CLICK` will be sent, and if the system is in state `no_nodes_selected`, then it would transition to state `nodes_selected`, which makes sense, as semantically the `NODES_SELECTED_WITH_CLICK` event means that a node has been selected with a click, and thus the array `selectedNodes` would be nonempty. On the other hand, if a node is deselected, then the event `CHECK_NODES` is raised. If the system is in state `nodes_selected`, and `selectedNodes` is indeed empty, then the system would take the transition to `no_nodes_selected`, which again makes sense given the semantics of the raised event.

Finally, the history sub-state of `nodes_selected` is used to encode the special behavioural requirement mentioned in Section 5.3.2, which is that the same scale or rotation handle will be shown as the last time the user had any nodes selected, if the new node selection was initiated with a drag, as opposed to a click. As can be seen, if the system is in state `no_nodes_selected`, and a `NODES_SELECTED_WITH_DRAG` event is sent, then the system will transition to the history state, thus encoding the requirement that this command be initiated with a drag, rather than a click. The history state will then show either a scale handle, if a scale handle had been shown the last time any nodes were selected, or a rotation handle, if a rotation handle had been shown the last time any nodes were selected.

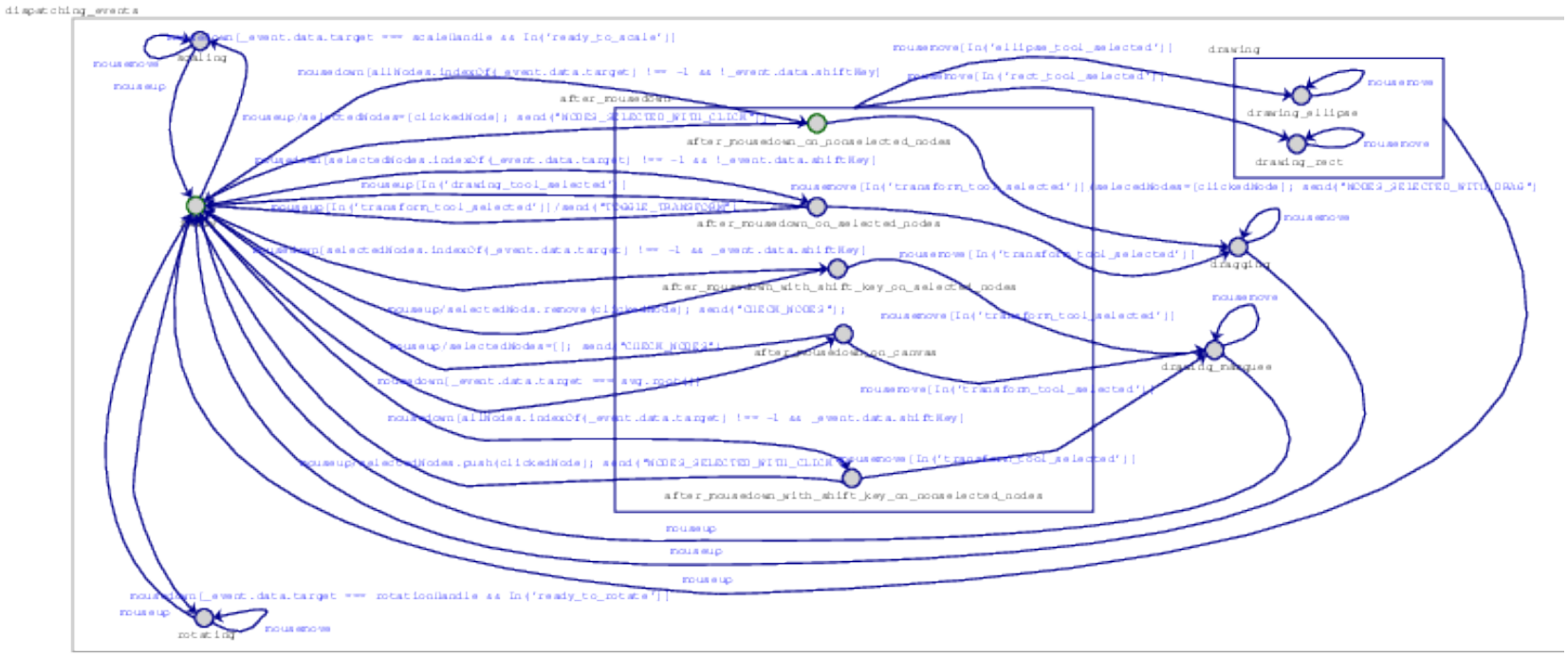

Figure 5.17: Orthogonal Component `dispatching_events`

Orthogonal Component `dispatching_events`

This orthogonal component implements the stateful command syntax specified in natural-language specification of the UI requirements. There are essentially two possible command *phrases*, which refer to two specific sequences of events. "mouseclick" refers to the sequence of events *(mousedown, mouseup)*, and "drag" refers to the sequence of events *(mousedown, mousemove+, mouseup)*, which is to say, one mousedown event, followed by one or more `mousemove` events, followed by a `mouseup` event. These "phrases" can have varying effects depending on the following factors: the event target, whether the shift key was pressed during the initial `mousedown` event, node selection state, and toolbar state. Node selection state and toolbar state are captured in the `presentation_state` orthogonal component and the `selectedNodes` array, while the event target and shift key state are captured in the triggered DOM event, which is passed to the system as event data. Transition conditions are used to inspect both the `presentation_state` orthogonal component, by using the SCXML `In()` predicate, and the DOM event, which is exposed as the `data` property on the SCXML object.

Generally, the system will begin in the `ready` state, and on `mousedown`, the system will transition to the first of possibly two intermediate states. If the system receives a `mouseup` as the second event, then the system will identify the phrase as a "click", generally transitioning back to the `ready` state, and performing some action on the way. However, if the system receives a second `mousemove` event, the phrase will be identified as a "drag", and the system will transition to the second intermediate state, possible performing some action on the way; subsequent `mousemove` events will cause the system to loop in the second intermediate state, performing some action based on the coordinates of the mouse event. `mouseup` will cause the system to return from the second intermediate state to the ready state.

A few examples will illustrate this process.

Consider the simple example where the user wishes to select an unselected node by clicking on it with the mouse. Assume that no other nodes are selected and the transform tool is chosen, and thus the system is in basic configuration {`no_nodes_selected,ready`}. To perform this command, the user would click a node. When the user clicks on a node, the system will receive a `mouseclick` where the node is the event target. The system will transition to `after_mousedown_on_nonselected_nodes`. The system will then receive a `mouseup` event, at which point it will transition back to `ready`. The transition action will set the `selectedNodes` array to contain only the original event target (`clickedNode`), and will send the `NODES_SELECTED_WITH_CLICK` event, causing the system to transition to state `ready_to_scale`, and reveal the selection box and scale handle due to entry actions on `nodes_selected` and `ready_to_scale`, respectively.

As another example, consider the case where the user wishes to draw a rectangle. Assume that the rect drawing tool is selected, and that the system is thus in configuration {`rect_tool_selected,ready`}. To perform this command, the user would drag anywhere, including on the canvas or on top of another node. Depending on where the user clicked, and whether the shift key was selected, the system will transition to a sub-state of `after_mousedown`. All transitions leaving from sub-states of `after_mousedown` have guard

condition `In('transform_tool_selected')`. As the system is in state `rect_tool_selected`, these transitions would evaluate to boolean `false`. There are two transitions with trigger `mousemove` leading from OR state `after_mousedown`: one with condition `In('ellipse_tool_selected')`, and one with condition `In('rect_tool_selected')`. The latter would be enabled, and would cause the system to transition to state `drawing_rect`. A new rect element would be created as the transition action. On subsequent `mousemove` events, the state would transition back to `draw_rect`, and on entry to `draw_rect` would update the newly-created rect based on the mousemove event's coordinates. On mouseup, the system would transition back to state `ready`.

`drawing_rect` is an example of a "second intermediate state" discussed earlier. Other examples of these states are `drawing_ellipse`, `dragging`, `rotating` and `scaling`, and all have similar behaviour, such that they loop back to themselves on `mousemove` event, and perform some action on entry, based on the `mousemove` event. For example, `dragging` will move the set of selected nodes by a delta computed from the mouse coordinates. `drawing_rect` will create a rect, and update its width and height based on the position of the mouse coordinates of the `mousemove` event. `drawing_marquee` works the same way, as does drawing_ellipse, but instead updates the `rx` and `ry` properties of the ellipse. `rotating` will rotate the selected objects based on a mouse delta, and `scaling` will scale the objects based on the mouse delta. The `mouseup` event from these states will return to `ready` without further action.

Finally, consider the case where the user wishes to translate a group of selected nodes. Assume the system is in basic configuration {`ready_to_scale, ready`}. To perform the command, the user would mousedown on a selected node and drag it. The initial `mousedown` event would bring the system to state `after_mousedown_on_selected_nodes`. The subsequent `mousemove` event would enable the transition targeting state `dragging`, as its condition `In('transform_tool_selected')` would evaluate to `true`. This would bring the system into state `dragging`, which would work in the same manner as described above.

5.4 Critique of SCS

The major critique against SCS is that its use of a single monolithic Statecharts model per application instance to describe all of a user interface's behaviour breaks encapsulation between individual UI components, and is thus unlikely to scale well with increased user interface complexity. There are two possible approaches that can be used to mitigate this concern.

The first is to compose the top-level orthogonal components, or sub-states of the orthogonal components, into individual "submachines," which is to say, to treat individual states of a statechart as architectural components in themselves. This could be facilitated by physical separation, as a single monolithic SCXML document can be divided into multiple XML documents which are then combined when the model is parsed. This can be accomplished with *XInclude*, a W3C standard which allows XML documents to be physically separated into multiple documents, such that they can be transformed into a single document when parsed. This strategy has has been suggested in previous iterations of the SCXML draft

specification[BAA+10].

This approach still relies on using a single Statecharts instance per application, which breaks encapsulation between individual UI components. A better solution may come in the form of creating multiple statechart instances, one for each graphical object created, such that each object aggregates its own statechart which encapsulates that object's state and behaviour. Objects would communicate with one-another via event-passing protocols.

This approach is likely to scale best with complex, object-oriented systems, but it does add complexity, and may lead to situations where state must be replicated between individual objects, thus violating the principal of "don't repeat yourself". For example, in the previous case study based on Inkscape, the toolbar state was a global concern that affected the behaviour of all other parts of the application. By using a monolithic statechart, that state could be inspected using the simple "In()" predicate from any transition, essentially making the toolbar state a global variable within the statechart. As in procedural programming languages, global variables are a very simple technique to share state across an application, but global variables are also well-known for breaking encapsulation, which is why many object-oriented languages, such as Java, do not directly support them. Moving the toolbar state out into its own object, such that it would need to be retrieved via a message-passing protocol, would have been possible, but it would have added complexity to the implementation.

The conditions under which SCS ceases to be a technique that scales well with complexity of a particular set of UI requirements is a problem which requires further investigation.

Chapter 6

Conclusion

The first three chapters of this thesis described the development of SCION, a Statecharts variant with a precise semantics built on SCXML and optimized for execution under ECMAScript and the Web browser environment. Chapters 2 and 3 described SCION's precise syntax and semantics, leveraging the work of the W3C SCXML draft specification for SCION's syntax, and "Big-Step Semantics", by Esmaeilsabzali, Day, et. al [EDAN09] for SCION's semantics.

Chapter 4 provided a rigorous empirical analysis of the performance of various optimization strategies of SCION semantics when run under different Web browser environments. The chapter described four optimization strategies that would be tested: the data structure used for transition selection; the Set data structure used throughout the algorithm; "model caching", where structural information about the model was precomputed and cached, rather than being computed at runtime; and finally, a flattening transformation to avoid traversing the state hierarchy. The chapter concluded that using a State Table data structure for transition selection, an ECMAScript Array as a Set implementation, and, finally, enabling model caching and the transition flattening transformation, provided the best overall performance and memory usage across a wide range of browsers and usage scenarios.

Chapter 5 described a design pattern that applied SCION to the development of a particular class of Web-based user interfaces, specifically those whose behaviour consists of a command syntax that varies depending on high-level application state. This design pattern described how to map user interface events to Statecharts events; how to encode high-level application state in the statechart through top-level orthogonal components; and finally, how to map the syntax of a user interface "command", often expressible as a regular expression, to a reduced state machine representation, annotated with embedded ECMAScript actions, that could then be embedded in the top-level orthogonal components. This design pattern was illustrated through detailed case studies of two highly interactive, but very different applications: a modal text editor based on vi, and a vector graphics drawing tool based on Inkscape. Because the SCS design pattern was sufficiently expressive to capture a non-trivial subset of the user interface behaviour of these two existing applications, it should be possible to develop other similarly complex applications using this technique.

Finally, SCION has been released as an open source project, and is currently being used in

production in several environments. For example, my employer, INFICON, a manufacturer of scientific instruments, has included it as a core component in its new software platform, where it is being used to control hardware. As another example, [24]7 inc., a consumer experience company, uses SCION to coordinate inputs across multiple modalities or browsers.

Future Work

The conclusions of Chapters 4 and 5 both describe opportunities for future work, regarding further optimizations of performance and memory usage of the SCION interpreter, and by refining the techniques by which SCION may be applied to Web user interface development.

Going forward, there are many problems to which SCION may be applied beyond the realm of Web-based user interface development. ECMAScript has been described as the "lingua franca" of scripting languages [Atw], and it is currently embedded in numerous development frameworks and environments outside of the Web browser. As SCION is implemented in ECMAScript, it can easily be embedded in these environments, and could thus be used to solve problems specific to the application domains targeted by these other environments.

The first way SCION may be applied outside of the Web browser environment is in desktop user interface development. Similar problems are faced in the development of desktop user interfaces as Web user interfaces. This is evident, as both case studies in Chapter 5 were based on applications that were originally desktop applications. It is possible to apply SCION in this way because there already exist high-quality ECMAScript bindings to many GUI toolkits used to develop desktop application. For example, SCION could potentially be used with the Qt GUI toolkit, which embeds ECMAScript via the QtScript bindings [qt]. As another example, SCION could be used in development of Java Swing applications via the Mozilla Rhino Java bindings.

SCION may also be applied to server-side Web development. Previous research has shown that Statecharts can be productively applied to the problem of modelling Web navigation [WP03]. SCION could be used to apply the same techniques described in existing literature to modern, ECMAScript-based web application frameworks, such as Node.js.

There is an interesting corollary to the previous application, which is that SCION could also potentially be used for embedded systems development including, possibly, robotics. Statecharts were originally invented for the purpose of controlling safety-critical embedded systems [Har87]. While ECMAScript cannot be run directly on embedded microcontroller hardware, there now exist several libraries and projects for the Arduino microcontroller, which facilitate the shift of embedded control logic from the microcontroller hardware to a Node.js server [M, Ped, Wal]. SCION could thus potentially be used with such a framework to control embedded applications.

A final domain to which SCION may be applied is game development. It has been shown that Statecharts may be productively applied to game development [MB], and ECMAScript is currently used as a scripting language in several existing game engines, including the popular Unity3D engine [uni]. SCION could thus potentially be used as an embedded Statecharts

controller in such game engines, or in the development of browser-based games based on technologies such as HTML5 Canvas.

Bibliography

[ABL08] C. Appert and M. Beaudouin Lafon. SwingStates: Adding state machines to Java and the Swing toolkit. *Software–Practice & Experience*, 38(11):1149–1182, 2008.

[arr] Array - MDN. https://developer.mozilla.org/en/JavaScript/ Reference/GlobalObjects/Array/.

[Atw] Jeff Atwood. Coding horror: JavaScript: The Lingua Franca of the Web. http://www.codinghorror.com/blog/2007/05/javascript-the-lingua-franca-of-the-web.html.

[BAA+10] Jim Barnett, Rahul Akolkar, RJ Auburn, Michael Bodell, Daniel Burnett, Jerry Carter, Scott McGlashan, Torbjörn Lager, Mark Helbing, Rafah Hosn, T.V. Raman, Klaus Reifenrath, and No'am Rosenthal. State Chart XML (SCXML): State Machine Notation for Control Abstraction. *W3C Working Draft*, 2010.

[Bea] Jacob Beard. [scxml] semantics of nested history in parallel state from jacob beard on 2011-02-28 (www-voice@w3.org from january to march 2011). http://lists.w3.org/Archives/Public/www-voice/2011JanMar/0033.html.

[bit] Bitwise operators - MDN. https://developer.mozilla.org/en/ JavaScript/Reference/Operators/Bitwise Operators.

[BL] Tim Berners-Lee. Pre-W3C Web and Internet Background. http://www.w3.org/2004/Talks/w3c10-HowItAllStarted/?n=15.

[BM08] Bram Moolenaar. Vim documentation: help, July 2008.

[BM09] Bram Moolenaar. Switching from mode to mode, April 2009.

[Bux86] W. Buxton. Chunking and phrasing and the design of human-computer dialogues. In *Proceedings of the IFIP World Computer Congress*, pages 475 – 480, 1986.

[C+99] J. Clark et al. XSL transformations (XSLT) version 1.0. *W3C recommendation*, 16(11), 1999.

[CD+99] J. Clark, S. DeRose, et al. *XML path language (XPath) version 1.0*. 1999.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[DFF+10] Erik Dahlström, Jon Ferraiolo, Jun Fujisawa, Anthony Grasso, Dean Jackson, Chris Lilley, Cameron McCormack, Doug Schepers, Jonathan Watt, and Patrick Dengler. Scalable Vector Graphics (SVG) 1.1 (Second Edition). *World Wide Web Consortium (W3C)*, 2010.

[DOTM01] M. C. F. De Oliveira, M. A. S. Turine, and P. C. Masiero. A statechart-based model for hypermedia applications. *ACM Transactions on Information Systems (TOIS)*, 19(1):2852, 2001.

[Dou00] B. P Douglass. Doing hard time: Developing Real-Time systems with UML, objects, frameworks and patterns. *Embedded Systems Programming*, page 199, 2000.

[EDAN09] Shahram Esmaeilsabzali, Nancy Day, Joanne M. Atlee, and Jianwei Niu. Big-Step semantics. *David R. Cheriton School of Computer Science, University of Waterloo, Technical Report*, February 2009.

[emb] Ember.js API docs. http://docs.emberjs.com/symbols/Ember.StateManager.html.

[Fen04] Huining Feng. *DCharts, a formalism for modeling and simulation based design of reactive software systems*. PhD thesis, McGill University, 2004.

[fla] flash.utils - details adobe ActionScript 3 (AS3) API reference. http://help.adobe.com/en_US/FlashPlatform/reference/actionscript /3/flash/utils/package.html#setTimeout.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, volume 206. Addison-wesley Reading, MA, 1995.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231274, 1987.

[has] HashSet (Java Platform SE 6). http://docs.oracle.com/javase/6/ docs/api/-java/util/HashSet.html.

[HH10] D. Hyatt and I. Hickson. HTML5. *World Wide Web Consortium WD WD-html5-20100304*, 2010.

[HK04] David Harel and Hillel Kugler. The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML). In Hartmut Ehrig, Werner Damm, Jrg Desel, Martin Groe Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert

Westkmper, editors, *Integration of Software Specification Techniques forApplications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 325–354. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-27863-4_19.

[Int12] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, sixth edition, December 2012.

[JB09] Jacob Beard. Modelling the Reactive Behaviour of SVG-based Scoped User Interfaces with Hierarchically-linked Statecharts, 2009.

[ki] Ki. https://github.com/FrozenCanuck/Ki.

[LHYT00] K. R. P. H. Leung, L. C. K. Hui, S. M. Yiu, and R. W. M. Tang. Modeling web navigation by statechart. In *Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International*, page 4147, 2000.

[Lun] Fredrik Lundh. Basic widgetmethods. http://effbot.org/tkinterbook/widget.htm.

[MB] Silvia Mur Blanch. *Statecharts modelling of a robots behavior*. Projecte fi de carrera, Escola Tcnica Superior DEnginyera.

[M] Sebastian Mller. Noduino - control arduino with node.js, WebSockets and HTML5. http://semu.github.com/noduino/.

[Net] Mozilla Developer Network. window.setTimeout - MDN. https://developer.mozilla.org/en/DOM/window.setTimeout.

[Net95] Netscape. Netscape and Sun announce JavaScript, the open, cross-platform object scripting language for enterprise networks and the Internet, 1995.

[Nia05] I. A Niaz. Automatic code generation from UML class and statechart diagrams. 2005.

[obj] Object - MDN. https://developer.mozilla.org/en/JavaScript /Reference/Global Objects/Object.

[Ped] Cam Pedersen. duino. https://github.com/ecto/duino.

[pyt] Built-in types python v2.7.3 documentation. http://docs.python.org/library/stdtypes.html#set.

[qt] Qt 4.3: QtScript module. http://doc.trolltech.com/4.3/qtscript.html.

[Rap10] Charles W. Rapp. SMC: The State Machine Compiler. http://smc.sourceforge.net/, March 2010.

[Sam02] Miro Samek. *Practical statecharts in C/C++*. Focal Press, 2002.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, 2006.

[spe] SPEC - standard performance evaluation corporation. http://www.spec.org/.

[sun] SunSpider JavaScript benchmark. http://www.webkit.org/perf/sunspider/sunspider.html/.

[SZ01] Emil Sekerinski and Rafik Zurob. istate: A statechart translator. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 376–390. Springer Berlin Heidelberg, 2001.

[uni] UNITY: game development tool. http://unity3d.com/.

[Wal] Rick Waldron. JavaScript arduino programming on nodejs - bocoup. http://weblog.bocoup.com/javascript-arduino-programming-with-nodejs/.

[Was03] Andrzej Wasowski. On efficient program synthesis from statecharts. *SIGPLAN Not.*, 38(7):163–170, 2003.

[Wel89] P. D. Wellner. Statemaster: A uims based on statecharts for prototyping and target implementation. *SIGCHI Bull.*, 20(SI):177–182, March 1989.

[WN68] William Newman. A system for interactive graphical programming. In *Proceedings of the April 30 – May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 47 – 54, New York, NY, USA, 1968. ACM. ACM ID: 1468083.

[WP03] M. Winckler and P. Palanque. StateWebCharts: a formal description technique dedicated to navigation modelling of web applications. *Interactive Systems. Design, Specification, and Verification*, page 279288, 2003.