2014 IEEE Security and Privacy Workshops

LEGO® Bricks for Reactive Programming

Dennis Volpano
Computer Science Department
Naval Postgraduate School
Monterey, California 93943
Email: volpano@nps.edu

***Abstract*—A fundamental unit of computation is introduced for reactive programming called the LEGO® brick. It is targeted for domains in which JavaScript runs in an attempt to allow a user to build a trustworthy reactive program on demand rather than try to analyze JavaScript. A formal definition is given for snapping bricks together based on the standard product construction for deterministic finite automata.**

I. Introduction

As the trend for software as a service continues, more functionality is expected of the code running within web browsers, specifically, HTML5, Flash and JavaScript. Complex JavaScript programs are behind many applications, for instance, Google's gmail, maps and drive. It is well known that such script has tracked information about users. Much attention continues to be given to analyzing source code or executables for certain properties. With the expressiveness of JavaScript, automating this analysis is no easier than automating it for a general-purpose programming language. At best, we can semi-decide *unsafe* programs, not safe ones. In other words, it is impossible to merely confirm that a given safe program is safe. With respect to information flow, the situation is just as bad. We can semi-decide programs that leak sensitive data but not those that don't.

Anyone who wants us to use their software at our own risk should bear the burden of convincing us the code does what we expect and no more. Shipping only JavaScript, or worse, binaries, is certainly not in this spirit as it places the burden squarely on the user. An early attempt to shift this burden from the user to the programmer is Necula's work in proof-carrying code [1]. It requires programmers to develop safety proofs for their code. But why accept code in the first place? It's typically too low level and inscrutable. Is there a way to supply instead easily-understood building blocks and a blueprint that describes how they can be assembled to build the desired application? If the blueprint were easier to inspect than code, we would likely have more confidence in the application than if it were delivered to us in binary form.

This paper sketches such an approach for a class of programs called reactive programs. These are programs structured around responding to external signals. Microcontrollers and user-interface code are just two examples. What sets the approach apart from work in declarative reactive programming [2], [3] is that the building blocks, called LEGO bricks, are not new program combining forms but rather fundamental computing elements that are primitive enough to distinguish between states and signals. States are used to remember history while signals forget the past. The cost of remembering the past is a function of input encoding and is reflected directly in brick size, just as it is in deterministic finite automata. The idea is to make the added complexity from having to remember the past reflected directly in brick size so that unwanted or extraneous behavior might become more obvious.

A definition of LEGO bricks and their systems is given in the next section. An operation for "snapping" a brick to a LEGO system is defined. It parallels the product construction for two deterministic finite automata. Examples of LEGO systems for a simple input transducer and ripple-carry adders are given. Finally a LEGO system for managing a drop-down menu is presented. Results presented here are preliminary as the work is in progress.

II. The LEGO brick

Bricks have two purposes. The first is to prescribe an admissible ordering of some set of external signals (events) that can occur over consecutive clock cycles of execution. Thus bricks prescribe transitions between signals over cycles. The second purpose is to count signals, up to a limit, in the context of knowing that other events have already occurred. States are used for this purpose. Thus bricks also prescribe transitions between states. Signals and states are defined as follows:

$$\begin{array}{lcl} \textit{signals} & C & ::= \quad B \mid C \,\&\, C \\ \textit{basic signals} & B & ::= \quad c_1, c_2, \ldots \\ \textit{states} & Q & ::= \quad p, q, r, s, \ldots \end{array}$$

where '&' denotes conjunction. A basic signal is a two-valued propositional variable having truth values T or I where I is the *indeterminate* truth value. In one clock cycle, a basic signal has one of these two values. Execution cycles are modeled as transitions between truth assignments to a basic signal, much like signal transition systems used to model circuits [5], [6] but with one key difference. Non-occurrence is not represented by assignment of truth value F to the signal but rather by assignment of truth value I. The reason is because of the need to distinguish negation from indeterminacy. Some basic signals are mutually exclusive. For example, a mouse is up or down but not both. The occurrence of one implies the negation of the other. However, the absence of a signal in a cycle does not imply its negation. It only means the signal did not occur, or in other words, was indeterminate. A key on a keyboard, for instance, may be pushed during a cycle. If not pushed, then absent another signal in the cycle that prevents it from being pushed, the most one can say is that the key push signal was

LEGO is a trademark of the LEGO Group of companies which does not sponsor, authorize or endorse this research.

DOI 10.1109/SPW.2014.30

Fig. 1. Binary keyboard

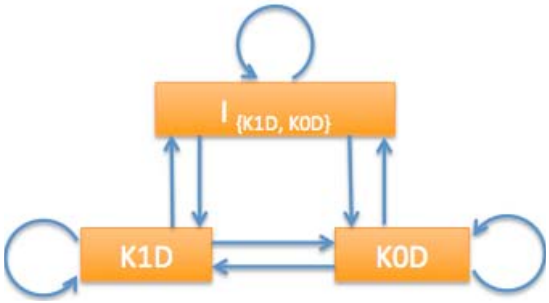

Fig. 2. Binary keyboard with delay

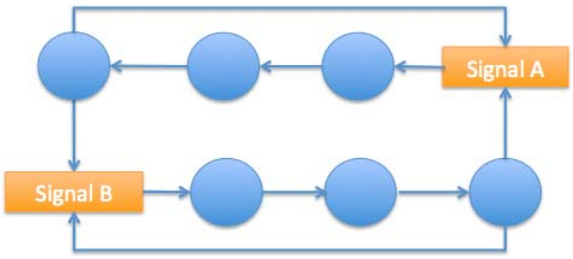

Fig. 3. Memory brick to count up to 3 signals after A or B

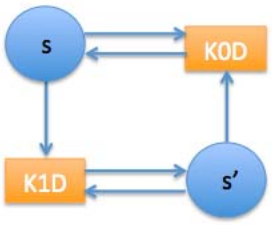

Fig. 4. 1-bit memory brick

indeterminate, not that the signal was false because a mutually-exclusive signal occurred. Indeterminacy also turns out to be useful for handling basic signals like a mouse over a button. If the physical button doesn't exist in a cycle (e.g. in a dynamic web page) then the basic signal's value is indeterminate.

Conjunction is interpreted in Kleene's weak three-valued logic which has truth values T, F and I [4]. In this logic, $F \,\&\, I = T \,\&\, I = I \,\&\, I = I$. Therefore, $(c = T) \,\&\, (c = I)$ becomes $c = I$ for every basic signal $c \in B$. We will use the notation $I_{\{c_1,\ldots,c_n\}}$ for $c_1 = I \,\&\, \cdots \,\&\, c_n = I$ if $n > 0$. Then $c \,\&\, I_B = I_B$ if $c \in B$. Finally, a conjunction of basic signals is a signal unless it comprises two or more mutually-exclusive basic signals in which case it is *inconsistent*.

A LEGO brick is a pair (H, δ), where $H \subset C \cup Q$ is finite and $\delta : H \to 2^H$ is a transition function.[1] H contains signals and states. A brick transitions from a signal or state to another signal or state. For example, Fig. 1 shows a brick for a binary keyboard with signals K0D (key 0 down) and K1D (key 1 down). Each edge corresponds to one clock cycle. At the end of each cycle, exactly one of the two keys was pushed during that cycle. The one pushed determines to which signal the brick transitions. While there may be more than one signal in a brick, each occurs exactly once in the brick (signals are shown as rectangles in brick diagrams). The brick in Fig. 1 merely makes transitions between two signals without remembering any previous ones. If we do not expect one of the two keyboard signals to occur on every cycle then we can introduce delay (called stutter in [7]) using indeterminate signal $I_{\{\mathsf{K1D},\mathsf{K0D}\}}$. The result is the brick shown in Fig. 2.

Note that both of the preceding examples of bricks have no state. Neither remembers any signal values prior to the most recent one. So in a sense they operate like Markov machines. Using states, a brick can remember a finite portion of the past while counting other signals. An example of this is given in the LEGO memory brick shown in Fig. 3. The states, which are drawn as circles, count occurrences (or non-occurrences) of some other signal while remembering that signal A has occurred (top three states) or B has occurred (bottom three states). The signal being counted is identified when the brick is snapped into place. A concrete example of a memory brick is the 1-bit memory brick shown in Fig. 4. It has two states, named s and s', that recall the previous input bit. We shall see in Section IV how this brick can be used, and how a 2-bit version of it is handy for building a ripple carry adder.

[1]There are no designated starting signals as it is assumed that any signal in H can be initial.

III. LEGO SYSTEMS

Intuitively, a LEGO system represents the accumulation of repeatedly snapping bricks together. It is a pair (K, Δ) where $K \subset C \times 2^A$ is finite, $A \subset Q$ is finite and $\Delta : K \to 2^K$ is a transition function. Notice that unlike the domain of δ, the domain of Δ always includes a signal because every element $(c, S) \in K$ comprises a signal c. Each element is equipped with a signal c and a set of states S, which are used to decide what to output. We shall say that (c, S) is a signal in a LEGO system if S is empty and a state otherwise. If the set of states of every member of K is empty then every member is just a signal and the LEGO system degenerates to a brick. Likewise, any brick comprised only of signals, as in Fig. 2, is also a LEGO system. The brick in Fig. 3, however, is not. It transitions from states but no state belongs to $C \times 2^A$.

A LEGO output system is a triple (K, Δ, O) where (K, Δ) is a LEGO system and $O : K \to E$ an output function where E is some set of output elements. The signals of K can be defined with respect to members of E. For instance, E might contain elements of a document object model (DOM) tree in a system running within a web browser. In this case, a signal such as mouse over a button might cause the button to be rewritten in the DOM tree into another object such as a textbox explaining the button. There may be signals defined with respect to the textbox and those signals are indeterminate (have value I) until

the textbox exists. We shall see examples of this in Section V with mouse-over-menu signals. An output system executes by producing a sequence of output elements as determined by the transition function Δ and output function O.

A. Signals vs state

In general, a signal does not remember the past, however, it may in some special cases. The unique occurrence of a signal c in a total ordering of signals indicates the presence of those signals that preceded it. Since it's unique, an output function can map *the* occurrence of it to some output appropriate for that past. However, if the signals that can precede it are partially ordered then there may be more than one signal path to c's occurrence and the output function will be unable to distinguish them there. So signals in general have amnesia. Unlike states within bricks, they do not have names that can uniquely identify different historical paths and enable an output function to discriminate them. States allow you to uniquely identify such paths if you want.

IV. The snap operation

The snap operation is similar to the product construction for two deterministic finite automata [8]. Given a LEGO brick $L = (C_1 \cup A_1, \delta)$ and a LEGO system $S = (C_2 \times 2^{A_2}, \Delta)$, where $A_1, A_2 \subset Q$, S *snapon* L is the LEGO system $(C_3 \times 2^{A_1 \cup A_2}, \Delta')$ where $C_3 = \{c_1 \,\&\, c_2 \mid c_1 \in C_1 \text{ and } c_2 \in C_2\}$ and Δ' is the transition function, defined in Table I. In the definition of Δ', S is a set of states that may be empty. Like the product construction for finite automata, snapping can produce unreachable signals but for a different reason; no inconsistent signal is reachable. Since consistency can be determined for two signals, Δ' is only defined for consistent ones.

The fourth rule of Δ' needs explanation. It talks about a transition from $(c, S \cup \{q'\})$. On the LEGO system side, we have $(c_1, S_1) \in \Delta(c, S)$ and on the LEGO brick side, we have a brick in state q'. The product LEGO system allows the LEGO system to proceed as it would in one cycle, however, the brick remains "stuck" in q'. That's because the LEGO system proceeds under a signal, namely c_1, under which there is no way for the brick to proceed from q' without a conflict with c_1. Two mutually-exclusive signals would occur in the same cycle or a signal would occur and also be indeterminate.

For example, a two-bit decoder LEGO system with delay is produced by snapping the 1-bit memory brick of Fig. 4 to the binary keyboard LEGO system of Fig. 2. In the result are two signals $\mathsf{K1D} \,\&\, \mathsf{I}_{\{\mathsf{K1D},\mathsf{K0D}\}}$ and $\mathsf{K0D} \,\&\, \mathsf{I}_{\{\mathsf{K1D},\mathsf{K0D}\}}$ which are equivalent in weak three-valued logic. These two signals can be merged leaving just $\mathsf{I}_{\{\mathsf{K1D},\mathsf{K0D}\}}$. Next we specify an output function to yield the desired decoder. To decode two input bits as ASCII, when the most significant bit is input first, we introduce the ASCII output function. It is defined here by annotating states of the resulting LEGO system. Upon doing this, we get the resulting LEGO output system in Fig. 5. The occurrence of I within the states stands for $\mathsf{I}_{\{\mathsf{K1D},\mathsf{K0D}\}}$.

V. Examples of LEGO systems

Two examples of LEGO systems are given. The first example is a ripple carry adder in two forms. The first form is incremental, meaning two operands are added by supplying one bit from each operand at a time, least significant bit first. This is the most efficient of the two forms as it only requires remembering the last bit input and whether there was carry from the previous two bits added. The second form is not incremental as it allows one operand to be input entirely before the other. Consequently, the whole operand must be remembered before any addition can be done. As inputs are encoded in binary, the size of the LEGO brick is exponential in operand length just as the size of a deterministic finite automaton under the same requirement would be. This growth should be embraced as it reflects in brick size the added complexity from having to remember history that is exponential in input size. The second example is the start of a full menu management LEGO system. Some rudimentary bricks and LEGO systems formed from them are presented. They illustrate the use of signals defined with respect to elements in the range of a LEGO system's output function.

$(c_1 \,\&\, c_2, S_1) \in \Delta'(c \,\&\, c', S)$
if $(c_1, S_1) \in \Delta(c, S)$, $c_2 \in \delta\ c'$ and $c \,\&\, c'$ and $c_1 \,\&\, c_2$ are each consistent

$(c_1, S_1 \cup \{q'\}) \in \Delta'(c \,\&\, c', S)$
if $(c_1, S_1) \in \Delta(c, S)$, $q' \in \delta\ c'$ and $c \,\&\, c'$ is consistent

$(c_1 \,\&\, c_2, S_1) \in \Delta'(c, S \cup \{q'\})$
if $(c_1, S_1) \in \Delta(c, S)$, $c_2 \in \delta\ q'$ and $c_1 \,\&\, c_2$ is consistent

$(c_1, S_1 \cup \{q'\}) \in \Delta'(c, S \cup \{q'\})$
if $(c_1, S_1) \in \Delta(c, S)$ and $c_1 \,\&\, c'$ is indeterminate or inconsistent for all $c' \in \delta\ q'$

$(c_1, S_1 \cup \{q_2\}) \in \Delta'(c, S \cup \{q'\})$
if $(c_1, S_1) \in \Delta(c, S)$ and $q_2 \in \delta\ q'$

TABLE I. Definition of transition function Δ'

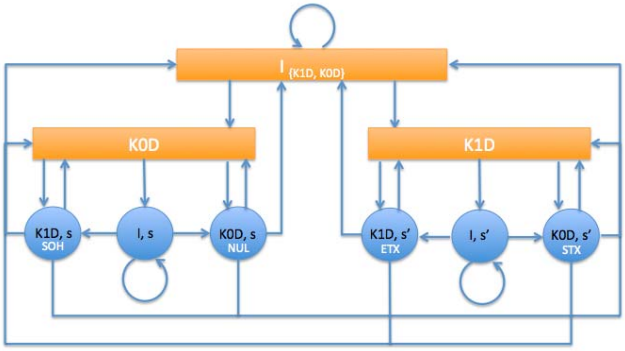

Fig. 5. 2-bit ASCII decoder

A. Ripple carry adders

A LEGO output system for an incremental ripple carry adder is shown in Fig. 6. It is an adder for two binary operands of arbitrary length such that the least-significant two bits of each operand are given first as input. There are no signals in this LEGO output system, only states (a formal definition of it would give unique state names to all states). The columns are split into two groups depending on whether there is carry in for the two bits being added. The third column of states stores only

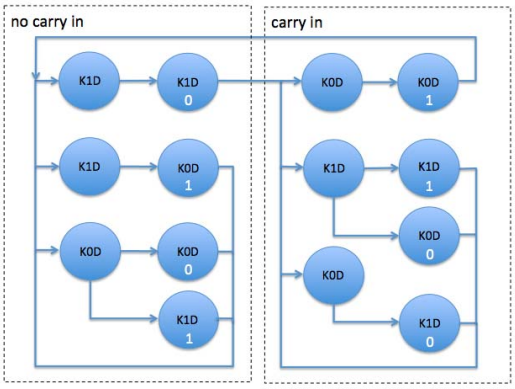

Fig. 6. Incremental ripple-carry-adder

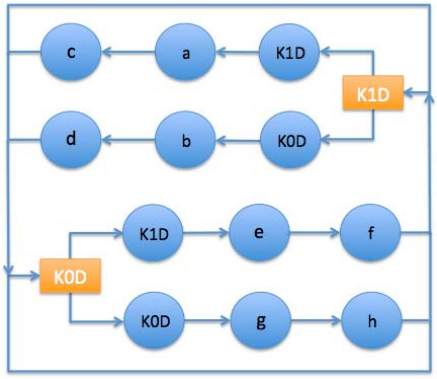

Fig. 7. 2-bit memory brick

carry-in true whereas the fourth column stores both carry-in true and the previous keyboard signal. The fourth and second column states have enough information for the output function to produce a sum bit. Those states are annotated with output 0 or 1. Snapping the brick for a binary keyboard with delay in Fig. 2 to the incremental ripple carry adder would produce an adder LEGO output system with delay.

A nonincremental version of the ripple carry adder can be built from the 2-bit memory brick in Fig. 7. This brick expects one 2-bit operand to be input completely and then remembers it while the second operand is input. Snapping this brick to the binary keyboard with delay LEGO system gives a LEGO system for which an output function can be defined for adding two 2-bit inputs where one summand is input completely before the other. The output function will depend on input convention. Whether a sum bit can be output in a state depends on whether the input is msb or lsb first. For instance, in the state $(\mathsf{K0D}, \{\mathsf{a}\})$ of the resulting LEGO system, no sum bit can be generated if K0D is the msb of the second summand. But if K0D is the lsb then 1 can be output in this state ($0 \oplus 1$ where 1 comes from the previous signal K1D). And in the state $(\mathsf{K1D}, \{\mathsf{c}\})$, 10 can be output ($1 \oplus 1$ preceded by the carry 1) giving final sum 101 on inputs 11 and 10, in that order, lsb first. As one operand, given in binary, must be input entirely before addition happens, the size of this LEGO system is exponential in the size of the summands.

B. *Two-option drop-down menu*

We describe the development of a LEGO output system for a drop-down menu of two elements that mirrors the behavior of an HTML select in the Safari web browser. A menu in general has several facets. We shall focus on just two, opening it and selecting an option. When our menu is open, it shows two options (0 and 1) where the most recent option selected is checked. An option is highlighted if the mouse is positioned over it. When the menu is closed, it shows the most recent option selected. It may also, when closed, have a highlighted perimeter which indicates it's in an active state, meaning the arrow keys can open it. Signals are mouse events, specifically, mouse up, down, over and not over. The latter two are defined with respect to objects of the document object model (DOM). The objects of interest for our purpose are as follows:

1) DDC-0, DDC-1 – Drop-down menu closed showing last option selected (0 or 1).
2) ADDC-0, ADDC-1 – Active drop-down menu closed showing last option selected (0 or 1).
3) 0, 1 – Menu options 0 and 1 of open menu.

With these objects, we introduce signals MO-DDC-0 and MO-NDDC-0. The former occurs when the DDC-0 object is in the DOM tree and the mouse is over it while the latter occurs if the object is in the tree but the mouse is not over it. If the object is not in the tree then both signals are indeterminate. The object may not be in the tree because the menu may be open. There are signals MO-DDC-1 and MO-NDDC-1 as well. Lastly there are signals MD and MU, for mouse down and mouse up, and MO-0, MO-1 and MO-N0/1 for mouse over 0, mouse over 1, and mouse over neither 0 or 1 when both exist in the DOM tree. Signal MU is considered instantaneous in that it happens in just once cycle, when the mouse transitions from down to up, whereas MD can happen over contiguous cycles by keeping the mouse button down.

Our goal is to develop and snap together some bricks into a system for which an output function can be defined that updates the DOM tree to produce the same effect as the HTML select. We begin with a LEGO system for opening the menu. The question is under what conditions a DDC object in the tree should be replaced by the open-menu objects 0 and 1. A first cut might say open the menu when the mouse is down and positioned over the DDC object. But this wouldn't preserve HTML's select semantics since one can drag the mouse to the closed menu object yet this shouldn't cause it to open. The menu should open only if a MD event occurred and there hasn't been one since the last MU. In other words, the MD event must be informally what is called a mouse "click". So history of a previous MD must be retained. To this end, a LEGO system for mouse drag is created; it is shown in Fig. 8. The system has one state to remember whether there has been a previous MD event without an intervening MU. That way when another brick is snapped to it, the resulting LEGO system will be able to distinguish between a click and a drag. Thus DDC can be replaced by open menu objects upon a click only.

For instance, the brick in Fig. 9 describes mouse position relative to the closed menu object DDC-0; there's a similar brick for DDC-1. Note the transition from MO-NDDC-0 to the indeterminate signal. It is there because in one cycle, the menu object DDC-0 may exist in the DOM tree and in the next

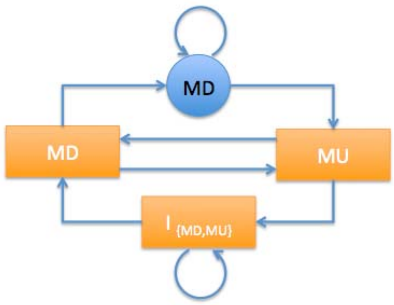

Fig. 8. Mouse drag LEGO system

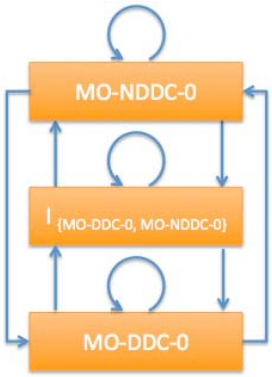

Fig. 9. Mouse position relative to closed menu DDC-0

cycle it doesn't because the mouse was re-positioned and the menu opened. A portion of the LEGO system resulting from snapping this brick to the mouse drag LEGO system is shown in Fig. 10. The complete system has nine signals and three states. Two of those states are shown in the figure with state set {r} (r is the name chosen for the only state in the mouse drag LEGO system). The output function for this system would map signal MD & MO-DDC-0 to the pair of objects 0 and 1 which effectively opens the menu. It would be undefined at state (MD & MO-DDC-0, {r}) because the occurrence of r tells us there was a prior occurrence of MD without an intervening MU. In other words, the mouse is being dragged. A similar LEGO system is constructed for DDC-1.

Next we build a LEGO output system for managing the selection of menu options. An option is selected by the occurrence of two events in a cycle, a MU event and either MO-0 or MO-1. The mouse is dragged over either option and released to make a selection. During dragging, the previous selection (or default) is indicated with a check mark and the option over which the mouse is currently positioned is highlighted. Any LEGO system then must provide states or signals that can be mapped by the output function to update the DOM tree in a way that is consistent with this behavior. This can be accomplished with two snap operations.

To manage mouse movement over an open menu, we introduce a brick for mouse position as shown in Fig. 11. Note the transition from MO-N0/1 to the indeterminate signal. It's there since in Safari, an open menu can disappear with a click away from it. The mouse-position brick is snapped

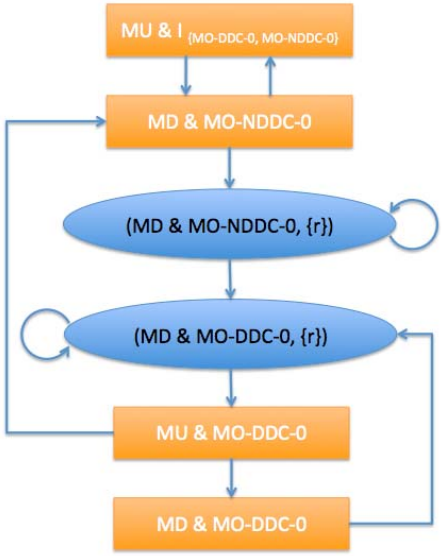

Fig. 10. A portion of the LEGO system for opening a two-option menu

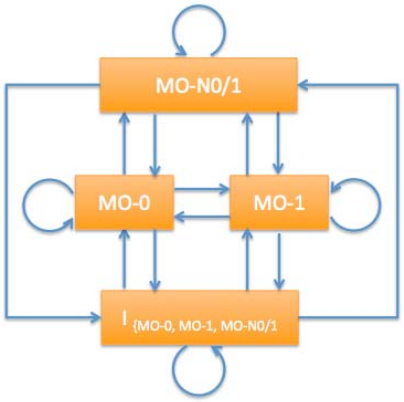

Fig. 11. Mouse position relative to open menu

to the mouse drag LEGO system. To that resulting LEGO system, we snap the 1-bit memory brick shown in Fig. 12. The memory brick is used to remember the previous selection. The resulting LEGO system has 28 states and two signals. A portion of it is shown in Fig. 13. Notice how states r (from the mouse drag LEGO system) and s (from the 1-bit memory brick) appear in the result. In each of the four states shown, s appears which implies 0 was most recently selected. The output function makes use of this fact in the three states where MD occurs by mapping each to the appropriate object highlighted or checked; (MD & MO-0, {s}) and (MD & MO-0, {r, s}) each get mapped to an open menu with 0 highlighted and 0 checked, while (MD & MO-1, {r, s}) is mapped to an open menu with 1 highlighted and 0 checked. The portion of the LEGO system not shown has symmetric behavior for state s' where 1 is checked instead for those states that contain s'. Signals MU & MO-0 and MU & MO-1 are mapped to objects ADDC-0 and ADDC-1 respectively. The presence of state r in {r, s} tells us the mouse is being dragged, however, the output function does not exploit that information here.

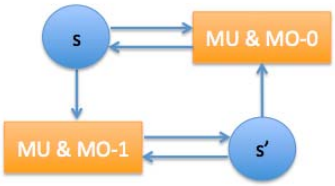

Fig. 12. 1-bit memory brick to remember last selection

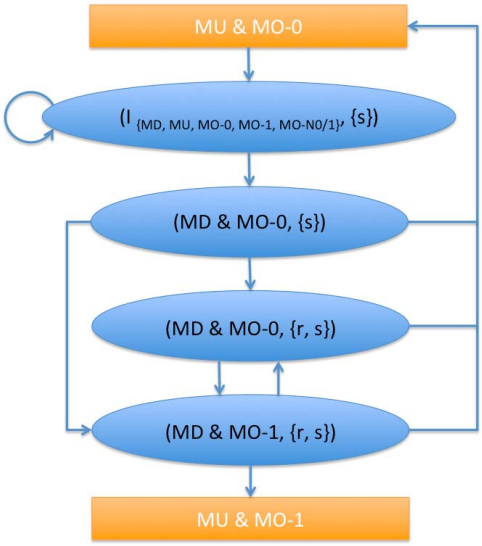

Fig. 13. A portion of the LEGO system for selecting a menu option

C. LEGO runtime system within a browser

A collection of LEGO output systems are run in parallel by a system that resembles a commercial logic analyzer. A set of signals is sampled at the end of each clock cycle. Based on them and the current configuration (c, S) of each, a transition is made in each of the systems deterministically and the target configuration becomes the current one. Their output functions are evaluated for any targets in their domain. Development of a runtime system for executing LEGO systems that manipulate a DOM tree is under way. It is written in JavaScript and is intended to run within any web browser. Whether we can harness signals in small enough "clock cycles" and update the DOM tree while preserving the fluid behavior a user expects is an open question. It will likely depend on the browser.

VI. Conclusion

The LEGO project grew out of an attempt to replace JavaScript within Adobe Reader, the exploits of which are well known. The premise was that Reader didn't need such power and that more basic functionality could be provided with fewer sharp edges. The scope then broadened to that of web browsers and applications running within them. A long-term goal is to avoid running downloaded JavaScript within a browser. Instead, a user would download a blueprint for building a LEGO system. The blueprint would be intelligible and draw upon local LEGO bricks as well as downloaded ones if necessary. More work is needed to develop other LEGO pieces that have high potential for reuse such as the memory brick described in this paper. Eventually there should be a manageable set of such pieces whose level of reuse rivals the level achieved in the electronic games industry.

VII. Acknowledgements

Thanks to Avner Biblarz for dissecting menu behavior in Safari for the purpose of identifying LEGO bricks and systems, and to Joe Lukefahr for providing the initial runtime system for executing LEGO output systems. Also thanks to the reviewers, especially Anna Shubina, whose comments helped bring some much needed transparency to the paper. This research was supported by the Office of Naval Research.

References

[1] G. Necula, "Proof-Carrying Code. Design and Implementation," in *Proof and System Reliability*. NATO Science Series, 2002, vol. 62, pp. 261–288.

[2] C. Elliott, "Declarative Event-oriented Programming," in *Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 2000, pp. 56–67.

[3] L. Meyerovich, A. Guha, J. Baskin, G. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi, "Flapjax: a Programming Language for Ajax Applications," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Systems, Languages and Applications*, 2009, pp. 1–20.

[4] Y. Nakayama, "DRT and Many-valued Logics," in *Logic, Language and Computation*, S. Akama, Ed. Kluwer Academic Publishers, 1997, pp. 131–142.

[5] F. García-Vallés and J.-M. Colom, "Structural Analysis of Signal Transition Graphs," in *Petri Nets in System Engineering*, 1997.

[6] A. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli, "A Unified Transition Graph Model for Asynchronous Control Circuit Synthesis," in *Proceedings of the 1992 IEEE/ACM International Conference on Computer-aided Design*, 1992, pp. 104–111.

[7] L. Lamport, "The Temporal Logic of Actions," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 872–923, 1994.

[8] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, 3rd ed. Addison Wesley, 2006.