Computational Thinking and Mental Models: From Kodu to Calypso

David S. Touretzky
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA
dst@cs.cmu.edu

***Abstract*—Reasoning about programs is an important component of computational thinking. Laws of computation give meaning to the formalisms in which programs are expressed, and can be used to predict or explain program behavior, or to uncover bugs. This paper presents Calypso, a language inspired by Microsoft's Kodu Game Lab but designed for programming actual mobile robots rather than characters in a virtual world. The initial implementation of Calypso uses the Cozmo robot by Anki.**

Like Kodu, the Calypso interpreter can be described by five key laws. An understanding of the laws and how to apply them constitutes a mental model of computation. Calypso provides a variety of affordances and scaffolding techniques to foster development of effective mental models and facilitate computational thinking.

***Index Terms*—Cozmo robot, mobile robots, robot programming**

I. Introduction

To help children learn to reason about computer programs, we need to expose them to computational systems whose laws are easy to grasp but rich enough to be interesting. Programs should be short for ease of analysis, but they must still be capable of non-trivial behavior. This means the language primitives must not be so primitive that it takes many instructions to accomplish anything significant. On the other hand, primitives that are too complex will be hard to understand. In previous work, colleagues and I have argued that Microsoft's Kodu Game Lab [1] occupies a good middle ground [2].

This paper describes Calypso [3], a variant of Kodu designed for programming actual mobile robots rather than characters in a virtual world. The initial implementation of Calypso uses the Cozmo robot by Anki. Like Kodu, Calypso can be described by five key laws of computation. But Calypso advances the language along several design principles: (1) visualization of the execution process, (2) visualization of program structure, (3) making the robot's sensations and internal representations transparent, (4) providing a tight connection between the robot's world map and objects in the physical world, and (5) improved affordances in the rule editor. Together these techniques facilitate development of effective mental models of computation, or what du Boulay called a "notional machine" [4].

This work was supported in part by a gift from Microsoft Research.

Fig. 1. The Cozmo robot, its three unique light cubes, and the game controller used with Calypso.

II. Hardware

A. The Cozmo Robot

Cozmo (Figure 1) is a vision-based mobile manipulator marketed by Anki as a children's toy, but sophisticated enough for use in robotics research. The robot weighs just 150 grams and includes motors, accelerometers, and a color camera. Cozmo has limited on-board processing; most of the computation is done by a smartphone or tablet running the Cozmo app and communicating with the robot via WiFi. Cozmo is accompanied by three light cubes that have built-in accelerometers of their own that can sense taps or motion, and controllable color LEDs. The cubes communicate with the robot via a proprietary radio protocol. They also bear special markers that allow Cozmo to visually detect them, and they can be lifted, carried around, rolled, or stacked.

Several games are built in to the Cozmo app, and a recent update added a small interpreter based on Scratch 3.0. Anki has also released an open source Python SDK, on which Calypso was built. To use the SDK, the phone or tablet running the Cozmo app must be connected to a laptop or workstation via a USB cable.

B. The Game Controller

One of the features that distinguishes Kodu from other children's programming languages is the use of the Xbox game controller as an input device. The developers thought a controller would offer a more comfortable interface for young children who have played computer games but not yet

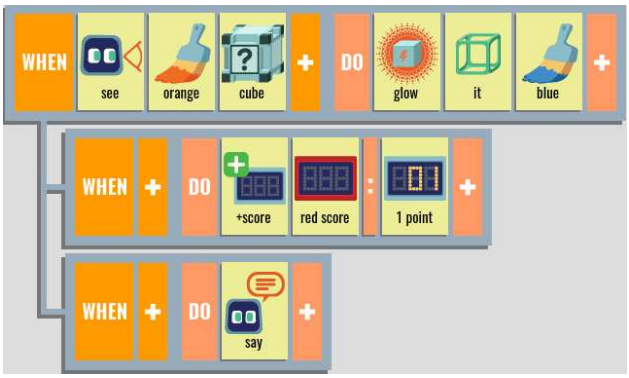

Fig. 2. Graphical display of indented rules in Calypso shows the dependency relationship between each rule and its parent.

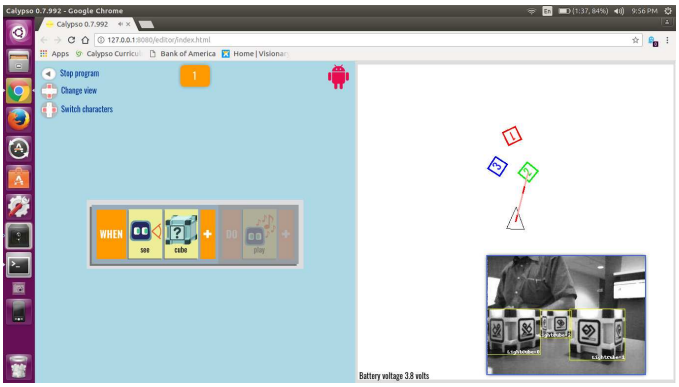

Fig. 3. The Calypso interface in execution mode. The current character (robot icon) is shown in the top right corner of the rules panel (left half of the screen). Button help is displayed along the left edge. The blue background of the rules panel shows that the character is on page 1. The right half of the screen shows the world map and camera viewer.

acquired much experience with the traditional keyboard/mouse combination. Mouse and touchscreen input are supported, but are more cumbersome to use.

Besides its use for menu selection, the game controller has two other roles. It offers a comfortable way to navigate in a 3D world: the left stick translates the cursor, the right stick orbits the camera around the cursor, and the shoulder buttons zoom in and out. With only a little practice it becomes natural to operate all three of these controls at once. In addition, the controller provides a way to interact with Kodu characters, e.g., they can be made user-driveable via one of the sticks, or have certain behaviors initiated by one of the triggers or ABXY buttons.

Calypso follows the same approach. With a single line of code, the robot can be driven using the left stick while the right stick tilts the camera and the shoulder buttons control the lift. This allows for a kind of "guided autonomy" where users can nudge the robot toward or away from particular objects, then allow control to revert to the rules they've written. In initial trials with several groups of children ages 8–15, they quickly become adept at teleoperating the robot this way.

III. The Kodu/Calypso Computational Framework

Kodu and Calypso are rule-based languages, not procedural languages. Other languages in the rule-based category include AgentSheets/AgentCubes [5], [6] and Ready (formerly Kandu) [7]. Rule-based languages also have a long history in artificial intelligence, where they have been used in cognitive modeling (e.g., SOAR [8]) and expert systems.

The left hand side of a Kodu or Calypso rule (the WHEN part) contains a pattern that is matched against the world state, possibly binding a variable called "it". The right hand side (the DO part) specifies an action that will be attempted if the left hand side is satisfied. See Figure 2. Rules are organized into pages. All the rules on the current page run repeatedly, typically about 70 times per second for Kodu. Unlike in procedural languages, the ordering of the rules on a page does not determine the rule cycles in which their actions will fire, but it does affect how rule conflicts are resolved. Figure 3 shows the Calypso interface in execution mode.

Beyond the elegance of the rule formalism, a second, equally important component of Kodu's success (and hopefully Calypso's) is the high level of the primitives provided. Kodu programs make no reference to screen coordinates, distances, or angles. Instead they refer to objects, and to natural operations on those objects such as grasping an object, giving an object to another character, or dropping or throwing the object. The user does not have to worry about the mechanics of how these operations are accomplished and rendered; they are primitives. This is quite different from graphics-oriented languages such as Scratch or Alice. Kodu is not a graphics language; it's a robot programming language that happens to use simulated robots. Calypso goes a step further by using actual robots.

A. The Laws of Kodu/Calypso

Laws of computation give meaning to the formalisms in which programs are expressed. They can be used to predict or explain program behavior, and to uncover bugs. The fundamentals of computation in Kodu/Calypso can be stated in five laws. These have been graphically illustrated and even turned into refrigerator magnets (see [9] for the Kodu version). Here we briefly review the laws of both languages.

The **First Law**, "Each rule picks the closest matching object", explains how variable binding conflicts are resolved. A rule that begins "WHEN see cube" could match any of several cubes. The conflict is resolved by picking the cube closest to the perceiver. When there are multiple rules looking for objects using different patterns, each rule chooses independently.

The **Second Law**, "Any rule that can run, will run", explains why the order in which rules are written does not determine the rule cycles in which their actions will fire. We say that a rule "can run" if its WHEN part is satisfied. If a rule can run, it *will* run, meaning it will fire (attempt) its action whether or not the preceding rules have run. However, its action may not be successful because actions can fail, or they can be blocked if they conflict with another action.

The **Third Law** explains how action conflicts are resolved: "When actions conflict, the earliest wins." Rules that appear earlier on the page have priority over rules that appear later, so if both rules run and fire their actions, the earliest rule's action takes precedence.

The **Fourth Law** explains rule dependency, which is shown by indentation. The Kodu version of this law states: "An indented rule can run only if its parent can." The parent of an indented rule is the first rule above it that has less indentation than the rule itself. An indented rule cannot run unless its parent's WHEN part is satisfied. Whether the parent's action succeeds, fails, or is blocked is not considered in Kodu.

In Calypso the situation is more precarious because actions can fail even when their preconditions are satisfied. This is discussed further in the section on Real World Actions vs. Simulation. To accomodate this, we have modified the Fourth Law to read: "An indented rule can run only if its parent's action succeeds."

The **Fifth Law** explains how the effects of actions are realized within a rule cycle: "Actions fire in order." Most rules on a page will have orthogonal actions unaffected by rule ordering. But there are some cases where order matters, such as multiple assignment statements referencing the same variable. Perhaps the most common case is the "switch to page" action, which short-circuits the execution of any later actions on the current page.

The Third and Fifth laws are complementary: the Third Law determines how action conflicts are resolved, while the Fifth determines how side effects are realized when there is no conflict. Variable assignment actions never conflict.

These five laws produce a rule language that can elegantly express the kinds of programs common in behavior-based robotics. Kodu's creators explicitly cite behavior-based robotics as one of their inspirations [1].

Although presented here all at once, the laws are not taught to children this way. We start with just the First Law and provide multiple activities where children can apply the law to explain or predict behavior. The next three laws are introduced one at a time, in order, with time provided to explore each law's meaning. The fifth law is new and has not yet been tested in the classroom.

There are additional minor laws beyond the five presented here. Two examples are "held objects aren't seen by the holder", and "timers cannot nest".

B. The Idiom Catalog

In previous work I defined a set of idioms for Kodu and designed a curriculum around them [10]. The curriculum is publicly available at [11]. Idioms are presented as flash cards for easy reference, inspired by the Scratch Cards of Rusk [12]. Each idiom consists of a name, a one-sentence description, an illustrative graphic, and sample code. Most of these idioms translate directly to Calypso.

The relationship of idioms to laws is explained once students have seen examples of both. Idioms are specific rules or sets of rules that occur frequently in programming because

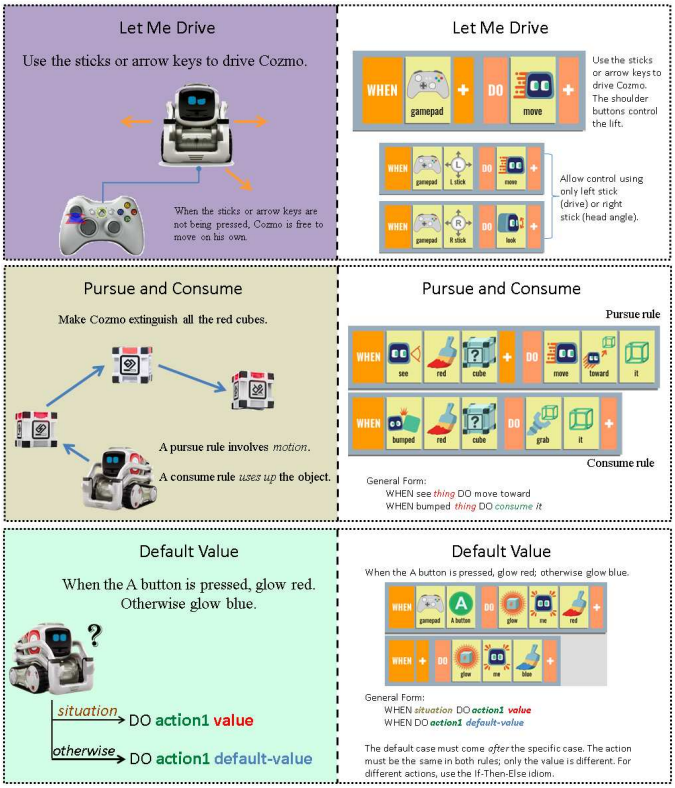

Fig. 4. Flash cards for the Let Me Drive, Pursue and Consume, and Default Value idioms.

they solve problems that come up often. There are many possible idioms. Each solves a specific problem, and a single program may only make use of a few of them. Laws, on the other hand, are universal. They apply to every program, and they are what make idioms work. For example, the Default Value idiom (Figure 4, bottom) depends on the Third Law to ensure that the specific cases take precedence over the default case, which must come last.

The first idiom students learn in Calypso is the teleoperation idiom Let Me Drive (Figure 4, top). In Kodu they start with Pursue and Consume, discussed below, and then add Let Me Drive. But with Calypso we introduce teleoperation first because we may need to guide the robot to find the objects it is to pursue.

Pursue and Consume (Figure 4, middle) is the richest idiom in the catalog, giving rise to an entire micro-domain of programs, bugs, symptoms of bugs, and strategies for analyzing programs to detect pursue and consume patterns. (This is reviewed in the section Reasoning About Calypso Programs, and discussed more extensively in [2] and [9].) Other idioms are more prosaic, such as Do Two Things (compound statements), Count Actions, and Default Value.

Our notion of idioms has much in common with what Ioannidou et al. colleagues have called "computational thinking patterns" (CTPs) in AgentSheets [13]. As with idioms, some CTPs are relatively simple, while others are more complex and can generate interesting micro-domains to reason about.

An example of a simple CTP is Absorption, which roughly corresponds to an "eat" or "vanish" action in Kodu, or the "consume" half of Pursue and Consume. Two more complex CTPs are Diffusion and Hill Climbing, which are used together to enable characters to perceive and move toward distant objects. Together they realize the "pursue" half of Pursue and Consume.

Like Scratch, AgentSheets is a graphics programming language, not a robot programming language. Scratch programmers work in the screen's coordinate system, and AgentSheets programmers work on a 3D grid. Characters in both frameworks can perceive items adjacent to them, but not farther away.[1] The Diffusion CTP implements a wavefront algorithm to overcome this perceptual limitation, and then Hill Climbing exploits the results to move in the direction of the gradient.

Kodu, by contrast, has no notion of grids or coordinates. Its primitives express relationships between objects. For example, traveling toward a distant, possibly moving object is handled by "move toward it". We should note, however, that the wavefront algorithm automatically finds paths around obstacles, while Kodu chooses a direct path which can leave characters stuck if there is a wall or other object in the way. But in Calypso "move toward it" uses the RRT-connect algorithm [14] to do probabilistic path planning, and thus can avoid obstacles and solve mazes as easily as the wavefront algorithm.

While AgentSheets is a graphics programming language and is domain-independent, Kodu is a robot programming language focused on a particular domain. Kodu's domain is virtual 3D worlds with built-in physics (gravity, momentum, collisions), certain kinds of pre-defined objects (apples, trees, roads, fish, etc.), and certain predefined ways of interacting with those objects. As a consequence, some operations that are programmed in AgentSheets using CTPs are simply primitives in Kodu. For example, the Absorption CTP relies on the user to specify the graphical effect and sound effect desired. In Kodu, built-in "eat", "boom", and "vanish" primitives generate appropriate sound and graphical effects automatically. But if one wants some other flavor of absorption, there is no way to create it.

In summary, both idioms and computational thinking patterns vary in complexity. The highest level instances generate interesting micro-domains with a rich set of phenomena that students can learn to reason about. But differences in the underlying computational model (grids for AgentSheets; 3D continuous worlds with physics for Kodu/Calypso) mean that the micro-domains will be different.

In the following sections I discuss ways in which Calypso differs from Kodu.

IV. Visualization of Program Execution

Rules in Kodu are only visible in the rule editor, not during execution. Calypso displays the rules on the current page while in execution mode, and uses several forms of annotation to make rule interpretation transparent:

[1]Scratch characters can turn toward a distant sprite only if it is referenced by name. To recognize an arbitrary object they must be touching it.

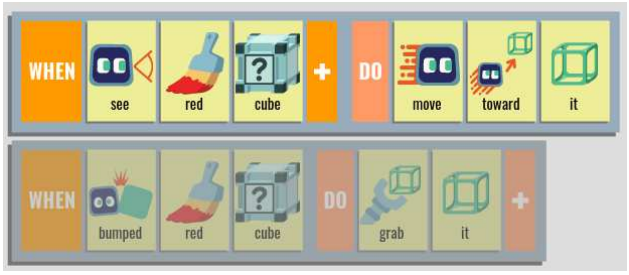

Fig. 5. Execution of the "Pursue and Consume" idiom. The robot can see a red cube and move toward it, but is not yet bumping it, so the second rule is dimmed.

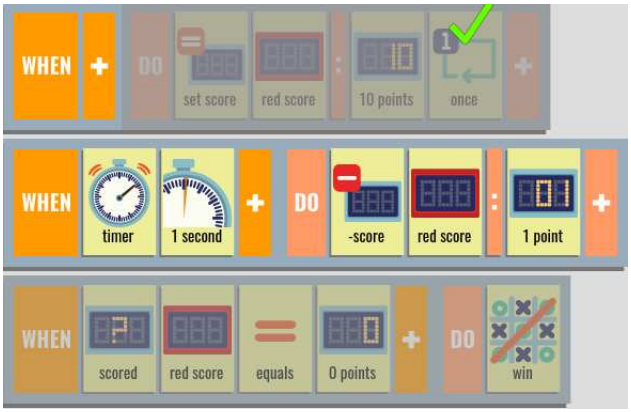

Fig. 6. In this example "once" is used to initialize a counter that then counts down to zero. When the "once" tile is satisfied, it receives a green checkmark and the DO part of the rule is dimmed, indicating that the action can no longer fire.

- If a WHEN part is not satisfied, the entire rule is dimmed (Figure 5, second rule).
- If a WHEN part is satisfied but the DO part is blocked (the Third Law), just the DO part is dimmed.
- If a rule contains a "once" tile and the once has been satisfied, the DO part is dimmed (because the action will not be repeated) and a green checkmark appears over the "once" tile (Figure 6, first rule). When the WHEN part is no longer satisfied and the "once" is reset, the checkmark is removed.

Repenning implemented a similar kind of display in AgentSheets, using color to indicate which rules (and which predicates within a rule) are satisfied. He calls this "conversational programming" [15] because the user can interact with the system by repeatedly changing the world state and observing the effect this has on the rules. This kind of experimentation is not currently possible in Calypso but could perhaps be offered when Calypso is run in simulator mode.

To make variable binding transparent, Calypso uses the same Line of Sight (LOS) indicators as Kodu to show which objects the rules have matched. This makes it easy for students to see the effect of the First Law. Figure 7 shows how the variable binding of rule 1 in the Pursue and Consume program of Figure 5 is displayed on the world map when the program is run. Rule 1 is choosing the closest red cube.

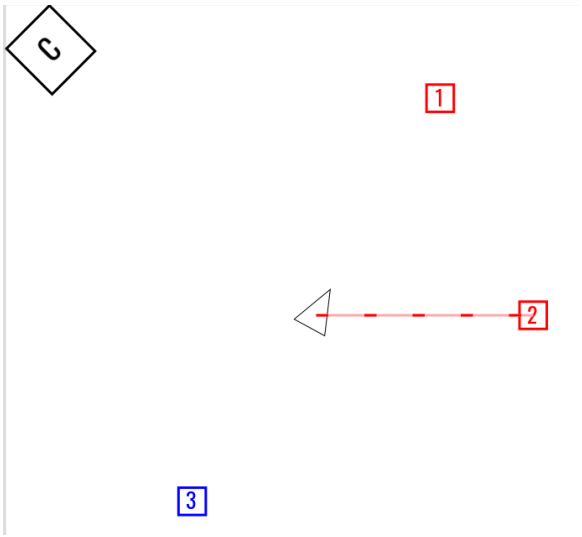

Fig. 7. The world map display when running the rules shown in Figure 5. The robot (black triangle) is moving toward the closest red cube, selected by rule 1. The Line of Sight indicator (dashed line) shows the selection.

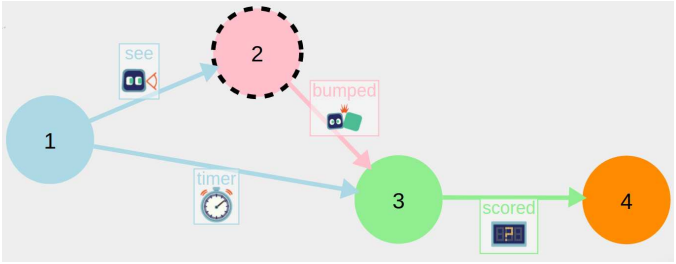

Fig. 8. Calypso automatically generates a state machine diagram from the user's code. The display shows the robot is currently on page 2.

V. Visualization of Program Structure

The rules of a Kodu or Calypso program are grouped into pages. Each page functions as a node in a state machine, and each "switch to page" tile can be seen as a state transition [16]. To help students see their programs as state machines, Calypso automatically generates a state machine diagram from user code (Figure 8). In editor mode, users can switch between viewing the rules on a page and viewing the state machine. During execution, if users switch to the state machine view they will see the currently active state marked, and as the program runs they will see activity flow through the state diagram.

Each character in Kodu has its own rules, and thus its own state machine. In Calypso the characters include the Cozmo robot, its three light cubes, and its charger. Light cubes can sense finger taps, display patterns using colored LEDs, and communicate with other characters via shared scores or Calypso's "say"/"hear" primitives, so their programming can be nontrivial. It is possible to switch characters during execution to observe any character's state machine or rules.

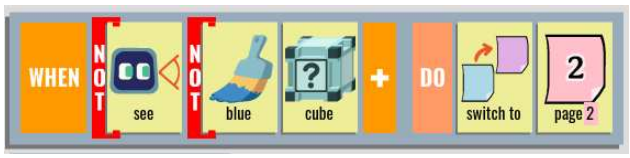

Fig. 9. A "not decorator" can be used to negate either a main predicate or one of its modifiers. This rule will run if the robot does not see any non-blue cubes.

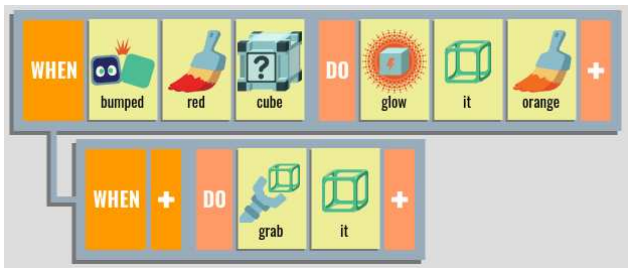

Fig. 10. In Calypso, an indented rule that does not bind "it" can reference the "it" binding of one of its ancestors.

VI. Semantic Enhancements

Kodu allows predicate arguments to appear in any order. So the "not" tile can appear anywhere after a predicate such as "see", but it applies only to the predicate itself. Thus, "*WHEN see apple not blue*", "*WHEN see blue not apple*", and "*WHEN see not blue apple*" all mean the same thing in Kodu; they are satisfied if the character does not see any blue apples.

Calypso expresses negation using a "not decorator" to negate specific tiles (Figure 9). Thus, "*WHEN not-see blue cube*" is satisfied if the character does not see any blue cubes, but "*WHEN see not-blue cube*" is satisfied if the character sees a cube that is not blue, and "*WHEN not-see not-blue cube*" is satisfied if all the cubes the character sees are blue, i.e., it does not see any non-blue cubes. The not decorator alone isn't powerful enough to express arbitrary boolean expressions since there is no mechanism for disjunctions, and negation only applies to individual terms, not conjunctions of terms. But it does handle many common cases.

Another area where Calypso extends Kodu semantics is the scope of the "it" tile. In Kodu, "it" can only be used on the DO side if it was bound on the WHEN side of that same rule. In Calypso, a rule whose WHEN part does not bind "it" can reference the "it" bound by one of its ancestors. This permits code such as that in Figure 10, where both rules reference "it".

VII. Transparency

Calypso characters exist in the physical world where perception is problematic due to limited camera field of view and resolution, and the possibility of occlusion. Calypso uses several techniques to make the robot's perception transparent so users will understand when the robot fails to see something.

A *camera viewer* window (Figure 11) shows what the robot is currently seeing, and any identified objects, such as light cubes or faces, are annotated in the camera image. This

Fig. 11. The camera viewer shows what the robot sees, with annotations indicating detected objects (cubes, faces, ArUco markers, etc.) Cozmo camera images are 320×240 grayscale; color images are also available, but at 160×240 resolution.

functionality is provided by the Cozmo SDK, but is extended in Calypso with additional object types, such as the ArUco markers [17] from OpenCV.

In execution mode, a *world map viewer* shows the robot's current map of the world, so it is immediately apparent what objects the robot is aware of. When an object is seen, it is placed on the world map and its position is continuously updated. If it goes out of view, it remains on the map in its last known position. Objects that are currently in view are highlighted on the map, so users can easily tell when an object is being tracked and when it has gone out of frame or been occluded. The "in-frame" tile used with "see" allows a rule to test whether an object is currently in view.

Tapping on a cube is a detectable event that can serve as input to a program. Cube taps are depicted on the world map by the cube briefly changing size. And when a cube changes color due to a "glow" action, that change also occurs on the world map.

An innovative feature of the Calypso world map is that it is not just a passive display. Users can mouse click on a cube to simulate a tap, which is confirmed by a tone unique to each cube. In addition, they can right-click on a cube and change its color on the map, which causes the physical cube to change color as well. We use the mouse rather than the game controller for world map interaction because the controller is dedicated to interaction with the robot.

VIII. Real World Actions vs. Simulation

Kodu actions never fail if their preconditions are met. The actions where precondition failure is possible are *eat*, *grab*, *launch*, *give*, and *squash*, all of which require as a precondition that the target object is within reach. In addition, *grab* and *launch* require the object to be movable, and *give* requires that it be capable of holding something. A few other actions can fail due to resource limits, e.g., missiles can only be fired at a certain rate, but these failures are of little consequence because the actions themselves are probabilistic (a missile isn't guaranteed to hit anything.)

In Kodu it is natural to write compound actions where a character increments a score, grabs an object, and switches to another page where it will transport the object somewhere. But real-world actions can fail, so Calypso programming style is slightly different. Although Calypso can detect failed actions and attempt to retry them, this may not always be successful. An example is a failed "grab" operation where the robot loses sight of the object and cannot reacquire it (perhaps it fell off the table). Even if the object is still in view, to prevent infinite loops an action will eventually be abandoned if it fails multiple retry attempts.

If an action fails outright but is still eligible to execute, it can be reattempted on the next rule cycle. But it would be a mistake to combine actions that can fail with other actions that will still execute when failure occurs. This is the reason we modified the Fourth Law to specify that if the parent rule's action fails, the child rules cannot run. Thus, in Calypso programs it is important to place unreliable actions in the parent rule, so that subsidiary actions and page switches will be blocked if the action fails.

IX. Reasoning about Calypso Programs

Pursue and Consume is a key idiom in both Kodu and Calypso. The pursue rule causes the robot to travel to an object and the consume rule operates on the object in some way such that it is no longer pursuable. In Kodu we pursue apples and "consume" them by eating them. In Calypso, the cubes can be programmed to glow with a certain color and turn off when the robot picks them up. Thus, we can use "grab" as a consume action provided that the pursue rule only selects cubes that are glowing. Once the cube is grabbed and no longer glowing, the pursue rule switches its attention to another cube and the robot heads off to pursue that. It will drop the first cube when it goes to pick up the next one, and it will never pick up a dropped cube again since it only looks for ones that are glowing.

Pursue and Consume is a rich micro-domain for exploring students' understanding of rule interpretation [9]. For example: given three red cubes, which one will Cozmo pursue first? Students have no trouble citing the First Law to justify their answer: it pursues the closest one. If we switch the order of the pursue and consume rules, what happens? Kodu students who hold the Sequential Procedure Fallacy[2] may be confused by this, but those who understand the Second Law realize that the change makes no difference. Will Calypso's visualization of rule interpretation help students escape the Sequential Procedure Fallacy and construct an accurate mental model of computation? Future work will investigate this.

Figure 12 shows a more complex problem in the Pursue and Consume micro-domain. Here we have two instances of the Pursue and Consume idiom, one for red cubes and one

[2]The Sequential Procedure Fallacy is the belief that the rules on a page must run one at a time, and in the order they are written [9].

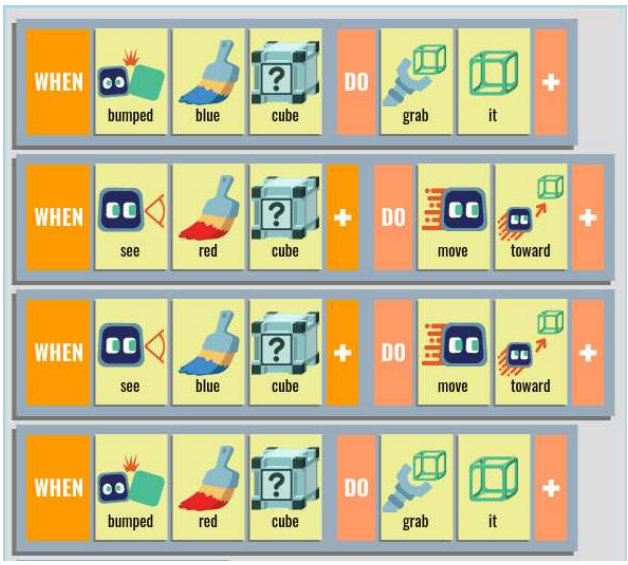

Fig. 12. A scrambled Pursue and Consume program, used to test students' ability to apply the Laws of Calypso.

for blue cubes, but the rule order is scrambled. If the robot is presented with a blue cube nearby and two red cubes further away, what will it pursue first? Answer: the closest red cube, because the first pursue rule (rule 2) takes precedence over the second one (rule 3) according to the Third Law. This pursue rule will pick the closest red cube (First Law). Students who think the two pursue rules jointly pick the closest cube are exhibiting the Collective Decision Fallacy [9].

When will the robot grab the blue cube? Answer: only when all the red cubes have been consumed. This is a consequence of the Third Law.

Additional problems in this micro-domain can be constructed by deleting one of the four rules. If a pursue rule has no matching consume rule, the robot will get stuck at the object it is pursuing. If a consume rule has no matching pursue rule, that consume rule will "starve" because the robot will never get close enough to an object to allow it to be consumed.

Previous studies using Kodu have shown that some children as young as 8 years old are capable of this kind of reasoning [9]. The additional scaffolding provided by Calypso may allow us to improve on these results.

X. Improved Affordances for Rule Editing

Experience with Kodu has led to some differences in the Calypso interface to optimize its affordances:

Novice Mode. Both Kodu and Calypso display button help along the left side of the screen, but beginners have not yet learned to refer to this. Until they become comfortable with the interface, it's easy to press a wrong button and end up editing code for the wrong character or adding rules to the wrong page. The problem is worse in Calypso because users can also accidentally switch from the editor view to a state machine view and might not know how to get back. To save users from confusing themselves, Calypso starts out in Novice Mode with the character switching, page switching, and view switching functions disabled, so they can only write code for the robot, and only on page 1.

No rule numbers. Kodu rules are numbered, making it easy to refer to specific rules in discussions. But the language itself does not use the numbers. In earlier work, Touretzky et al. showed that beginning Kodu programmers often exhibit the Sequential Procedure Fallacy [9]. Numbering the rules seems to reinforce this misunderstanding, so Calypso rules are unnumbered.

Current page made more salient. The Kodu rule editor displays the current page number at the top of the screen, and uses a different color number for each page. This can be a bit too subtle for beginners. Calypso changes the background color of the entire editor window to reflect the page number (blue for page 1, pink for page 2, etc.) This same convention is used in execution mode, where the rules are displayed on the left hand side of the screen and the world map on the right hand side (Figure 3). It is thus immediately apparent when a character changes pages.

Graphic display of rule dependency relationships. An indented rule in Kodu is dependent on its parent, but there is nothing to visually indicate that there is a relationship between the indented rule and a rule above it vs. a rule below it. Calypso graphically links each rule to its parent (Figure 2).

Visually distinguishing the first argument of assignment tiles. In Kodu, an action such as "+*score red-score 1-point blue-score*" means to increment the red score by 1 point plus the current value of the blue score. Kodu makes no visual distinction between the first argument to an assignment tile and the remaining arguments, although the first argument is treated quite differently (it's the target of the assignment). Calypso adds a marker between the first argument and the rest to reinforce this distinction (see Figure 2, second rule).

Sequential evaluation of rules. In Kodu, all the rules' WHEN parts are evaluated effectively in parallel at the start of each rule cycle. Then, for rules whose WHEN parts are true, the DO parts are executed sequentially, so the effects of one rule contribute to the effects of any following rules (the Fifth Law). However, the effects of a rule do not affect the evaluation of any rules' WHEN parts until the next rule cycle. This evaluation convention is not intuitive, and gives rise to certain bugs that are hard to explain or gracefully work around. For example, the rule "*WHEN DO set-score red-score 10 points once*" is intended to initialize a counter. If it is followed by the rule "*WHEN scored red-score = 0 points DO win*", the program will terminate at the end of the first rule cycle even though the red score's value at termination is 10, not 0, because the score *was* 0 when the two WHEN parts were evaluated.

In Calypso, evaluation of WHEN parts is interspersed with the performance of DO parts, so the text of the rules accurately describes the flow of effects of actions. In other words, evaluation respects the affordances provided by the rules, and the example above will work as intended. The cost of this change is that Calypso is not fully compatible with Kodu,

but in practice Kodu programmers rarely rely on its unusual evaluation convention.

XI. Discussion

Calypso facilitates a new kind of children's robot programming where the robots maintain a rich internal representation of the world that is transparent to and even manipulable by the user. This representation is primarily informed by computer vision. WHEN-DO rules match against the internal representation rather than being driven by raw sensor data. They fire high-level actions for navigating through the world, manipulating objects, and communicating with other characters.

Cozmo is a complex device with interesting behavior that children actually care about. Teaching them to carefully reason about Calypso programs introduces them to the notion of "lawfulness" (in the scientific sense), which is key to computational thinking [2].

Another key feature of both Kodu and Calypso is the use of state machines. State machines are ubiquitous in computer science, appearing in areas such as logic design, formal languages, network protocols, game programming, and behavior-based robotics. Much of the technology of everyday life can be described by state machines, from traffic lights to microwave ovens to smartphones. Yet computer languages usually provide no direct support for state machines; programmers must cobble together their own mechanisms using things like global variables and switch statements. Explicit instruction in state machines could allow students to construct and reason about more complex programs. Calypso goes a step further than Kodu by graphically displaying the state machine diagram, which should help children grasp the concept more quickly.

XII. Future Work

Many extensions are planned for Calypso. One of the first will be support for walls and places. The walls are constructed from Plasticor board with doorways cut in them at known locations. ArUco markers placed on the walls will allow the robot to identify them and build a map of the environment which it can then navigate through. "Places" can be defined relative to these landmarks, and a "visit" tile can be used to tell the robot to visit a particular place, using its path planner to calculate the route. The point of this extension is to provide the robot with richer internal representations which students can then learn to reason about.

Another line of extension is the pattern matching language used in the WHEN parts of rules. At present, patterns consist of a main unary predicate (e.g., "see") and some auxiliary unary predicates, such as a category test ("cube" or "charger") or color test (e.g., "red"). This language could potentially be extended to express more complex relationships between objects, which will be especially useful when multiple robots are considered.

Multi-robot support will allow students to explore much more complex behaviors as robots interact with each other in cooperative or competitive modes, as well as interacting with humans.

In addition to these technical extensions, future work will explore how children learn to reason about Calypso programs, and whether its improved affordances help them grasp concepts more quickly and avoid some common fallacies.

Acknowledgments

Thanks to Christina Gardner-McCune for assistance with testing Calypso and presenting it to novices, and to Christina, Stephen Coy, Ashish Aggarwal, and the anonymous referees for helpful comments on the manuscript.

References

[1] M. B. MacLaurin, "The design of Kodu: A tiny visual programming language for children on the Xbox 360," Proc. of POPL'11, pp. 241–246, 2011.

[2] D. S. Touretzky, C. Gardner-McCune, and A. Aggarwal, "Teaching 'lawfulness' with Kodu," Proc. of SIGCSE'16, March 02–5, 2016, Memphis, TN.

[3] Visionary Machines LLC, web site: http://calypso.software. Accessed August 29, 2017.

[4] B. Du Boulay, T. O'Shea, and J. Monk, "The bloack box inside the glass box: presenting computing concepts to novices," Intl. J. Man-Machine Studies, vol. 14, pp. 237–249, 1981.

[5] A. Repenning and T. Sumner, "Agentsheets: a medium for creating domain-oriented visual languages", IEEE Computer, vol. 28, no. 3, pp. 17–25, 1995.

[6] A. Repenning, "Moving beyond syntax: Lessons from 20 years of blocks programming in AgentSheets", Journal of Visual Languages and Sentient Systems, 2017.

[7] Anonymous, "About Ready", https://getready.io/about, accessed July 26, 2017.

[8] J.E. Laird, A. Newell, and P. S. Rosenbloom, "SOAR: an architecture for general intelligence", Artificial Intelligence 33(1):1–64, 1987.

[9] D. S. Touretzky, C. Gardner-McCune, and A. Aggarwal, "Semantic reasoning in young programmers," Proc. of SIGCSE'17, March 8–11, 2017, Seattle, WA.

[10] D. S. Touretzky, "Teaching Kodu with physical manipulatives," ACM Inroads, vol. 5, no. 4, pp. 44–51, 2014.

[11] D. S. Touretzky, "Kodu resources for teachers", https://www.cs.cmu.edu/~dst/Kodu, accessed July 26, 2017.

[12] N. Rusk, "Scratch cards", http://scratched.gse.harvard.edu/resources/scratch-cards, June 2009.

[13] A. Ioannidou, V. Bennet, A. Repenning, K. Koh, and A. Basawapatna, "Computational thinking patterns", Proc. 2011 Annual Meeting of the American Educational Research Association (AERA), New Orleans, April 8–12, 2011.

[14] J. Kuffner and S. M. LaValle, "RRT-Connect: An efficient approach to single-query path planning", Proc. 2000 IEEE Int'l. Conf. on Robotics and Automation (ICRA 2000), pp. 995–1001.

[15] A. Repenning, "Conversational programming: exploring interactive program analysis", Proc. of the 2013 ACM International Symposium in New Ideas, New Paradigms, and Reflections on Programming & Software (Indianapolis, October 2013).

[16] K. Stolee and T. Fristoe, "Expressing computer science concepts through Kodu Game Lab," Proc. of SIGCSE'11, pp. 99–104, 2011.

[17] S. Garrido-Jurado, R. Muñoz-Salinas, F.J. Madrid-Cuevas, and M.J. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion", Pattern Recognition, vol. 47, no. 6, pp. 2280–2292, 2014.