Automatic Verification for Interactive Graphical Programs

Carl Eastlund
Northeastern University
Boston, MA, USA
cce@ccs.neu.edu

Matthias Felleisen
Northeastern University
Boston, MA, USA
matthias@ccs.neu.edu

ABSTRACT

Modern software applications come with interactive graphical displays. In the past, verification efforts for such programs have usually ignored the I/O aspects of programs and focused instead on their core functionality. This approach leaves open the question of how errors in the interactive part of the program can affect its overall functionality.

In this paper we present an extension of Dracula (the ACL2 development environment for DrScheme) with a simple graphical framework. With Dracula we can automatically prove theorems about interactive graphical programs, guaranteeing their complete behavior. We have successfully verified theorems about a number of interactive programs with Dracula; we have also successfully used Dracula as a motivational tool to introduce students to the world of automated theorem proving.

Categories and Subject Descriptors

D [**2**]: 4—*Software/Program Verification*; F [**3**]: 1—*Specifying and Verifying and Reasoning about Programs*; I [**3**]: 2—*Graphics Systems*

General Terms

Verification

Keywords

ACL2, Dracula, Interactive graphical programs

1. THEOREM PROVERS IN THE AGE OF GRAPHICAL APPLICATIONS

Modern applications interact with their users via rich graphical interfaces. The complexity of concurrent user interactions makes these applications error-prone. Simultaneously, the integration of I/O with the fundamental program logic (often called the model) means even shallow interface bugs can cause deep, subtle, and persistent errors. Developers need tools to find and rule out this kind of problem.

Researchers have developed abstract models of interactive systems [7] and applied automated testing [19] and model checking [10] techniques to help developers debug their user interfaces. These methods give developers an assurance of correctness about their programs. Large test suites check properties of a finite subset of the program's inputs; model checkers verify a model based on the program.

Formal verification has been applied to other forms of I/O; notably, the Sparkle theorem prover can verify a model of concurrent filesystem operations [9]. We go even further and automatically prove correctness theorems about interactive graphical programs. Our approach relies on an event loop with functional callbacks, exploits images as first-class values, and introduces liveness properties. We use ACL2 [16] because it is the most successful industrial-strength theorem prover.[1] ACL2 has verified large projects including commercial floating point processors, a complete micro-processor, and a Java virtual machine.

In this paper we present Dracula, the first tool providing automatic theorem proving for interactive, graphical programs. The rest of the paper proceeds as follows. We describe the language and user interface of Dracula in section 2 and its graphical toolkit in section 3. Section 4 explains how we prove theorems about Dracula programs, and section 5 details the projects we have verified. We conclude with section 6 on related work and section 7 to summarize our contribution.

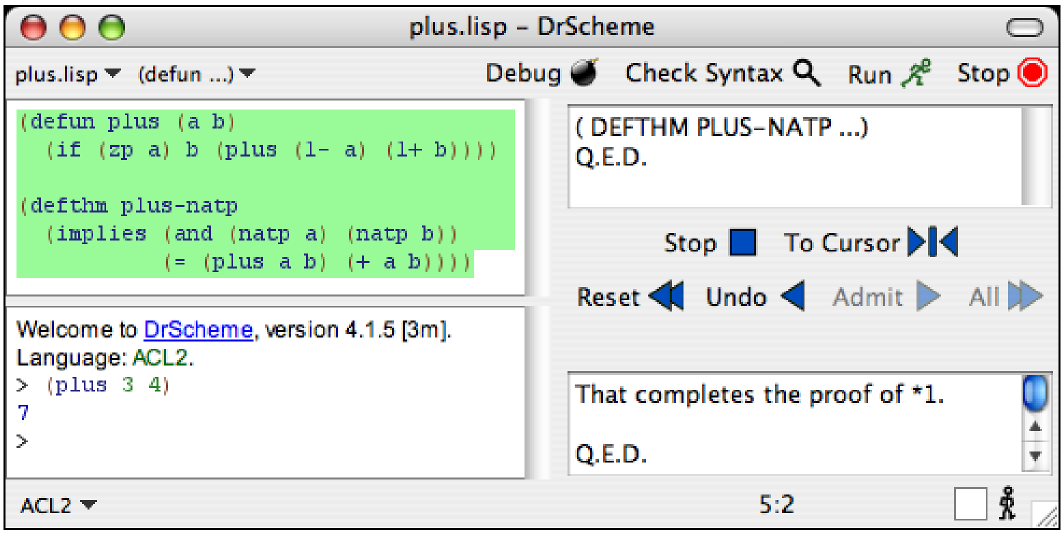
2. DRACULA: AN IDE FOR ACL2

Dracula [23] is an extension of the ACL2 theorem prover, implemented as a language level in the DrScheme integrated development environment (IDE) [13]. Dracula uses the DrScheme runtime system to simulate ACL2 programs, and the ACL2 theorem prover to reason about them. Every feature of Dracula comes with both an executable behavior and a logical meaning for ACL2.

ACL2 stands for "Applicative Common Lisp: a Computational Logic". It consists of a first-order functional language ("Applicative Common Lisp") and a first-order equational logic ("Computational Logic"). The theorem prover infers induction strategies to prove a programmer's conjectures.

Figure 1 shows a small program in ACL2 containing a function *plus* of two arguments and a conjecture *plus-natp*

[1] `campus.acm.org/public/pressroom/press_releases/3_2006/software.cfm`



```
(defun plus (a b)
  (if (zp a) b (plus (1- a) (1+ b))))

(defthm plus-natp
  (implies (and (natp a) (natp b))
           (= (plus a b) (+ a b))))
```

```
Welcome to DrScheme, version 4.1.5 [3m].
Language: ACL2.
> (plus 3 4)
7
>
```

```
( DEFTHM PLUS-NATP ...)
Q.E.D.
```

```
That completes the proof of *1.

Q.E.D.
```

Figure 1: A verified ACL2 program in the Dracula IDE.

stating that *plus* performs addition on the natural numbers. When entered into ACL2, the theorem prover first ensures that *plus* terminates on all inputs. Once ACL2 admits *plus*, it attempts to verify the statement of *plus-natp* for all assignments to a and b. ACL2 proves this claim by induction on the structure of the natural number a.

DrScheme is an IDE for a collection of programming languages, ranging from Scheme to Java to Algol 60. Its implementation and underlying language provide facilities for easily creating and deploying new languages [5, 14].

DrScheme also provides a wealth of tools for programmers using these languages, including online documentation, stack traces, a syntax checker, and a unit testing framework. Of special interest are DrScheme's *teachpacks*: libraries of interactive functionality designed for students. These include the World teachpack, which provides graphical I/O.

Dracula. Figure 1 shows a screenshot of Dracula in action. The two sides of the Dracula interface reflect the executable and logical components of the language. The left-hand side provides two windows for formulating and executing programs: the definitions window, where users edit their programs, and the interactions window, where users try out their functions.

The right-hand side of the display is Dracula's interface to the ACL2 theorem prover. It provides buttons to invoke ACL2 and to send each term from the definitions window to the theorem prover. Dracula paints functions green, once shown to terminate on all inputs, and also paints theorems green once verified; red highlights failures otherwise. Green terms are locked from further editing to faithfully represent ACL2's logical state; users may edit red terms or undo the admission of green terms to edit them. Below the control buttons, Dracula shows the theorem prover's output; above them, it shows a proof tree naming key checkpoints for quick diagnosis of a failed proof attempt.

3. INTERACTIVE GRAPHICAL PROGRAMS FOR ACL2

Dracula incorporates the World teachpack: an interactive, functional graphics framework originally developed as part of an undergraduate curriculum to teach program design principles [12]. The teachpack provides a simple functional framework for I/O, images as values, and an event loop.

Figure 2 shows a simple interactive graphical program called Darts based on this framework. The player has three tries to throw a dart at a target by clicking on it. The game window shows the target (a big red circle) and the remaining darts (little blue triangles). A single hit wins the game; three consecutive misses loses. On the right of figure 2, we see the player miss twice and finally hit the target.

We represent the current game "world" as the number of remaining darts, or the symbol 'win. The function *win-or-lose* determines the end of the game (0 or 'win); any other *World* is an *ActiveWorld* (a positive integer).

During the game, when the user clicks the mouse the *throw-dart* function compares the mouse position to the target using *dart-hits* (elided for space). The input signature of *throw-dart* includes *ActiveWorld*; *throw-dart* decrements the value of the world under the assumption that it is positive. It assumes the player has a dart left to throw, and relies on *win-or-lose* to end the game otherwise.

The *show* function turns the game state into an image (a first-class value), including an informative message when the game ends. Otherwise it shows the target and remaining darts using *show-target* and *show-darts* (both elided).

At the end of the program, we start the interactive world with **big-bang**, providing an initial value of 3. We instruct Dracula to end the game should the player *win-or-lose*, to *throw-dart* every time the player uses the mouse, and to *show* the game on a 120×120 canvas.

The World machine. In general, the World teachpack operates as an "abstract machine" with five instructions. As in the Darts game, the programmer supplies a data representation for the state of the world and functions defining the behavior of each user interaction. The **big-bang** command assembles the World machine and sets it in motion. It accepts five optional function arguments as shown in figure 3. The machine uses *World_0* as the initial world value; each subsequent clause defines the behavior of one of the machine's instructions.

```
;; win-or-lose : World → Boolean
(defun win-or-lose (w)
  (or (equal w 'win) (equal w 0)))

;; throw-dart : ActiveWorld Int Int Action → World
(defun throw-dart (w x y a)
  (if (equal a 'button-down)
      (if (dart-hits x y) 'win (1- w))
      w))

;; show : World → Image
(defun show (w)
  (cond ((equal w 'win) (text "You win!" 24 'blue))
        ((equal w 0) (text "You lose." 24 'blue))
        (t (show-darts w (show-target)))))

(big-bang 3
 (stop-when win-or-lose)
 (on-mouse-event throw-dart)
 (on-redraw show 120 120))
```

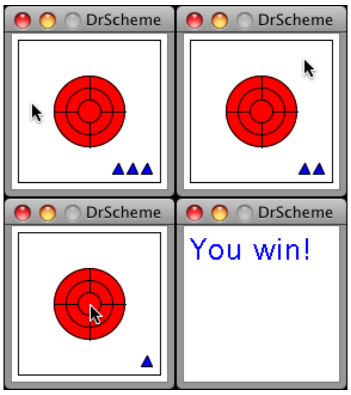
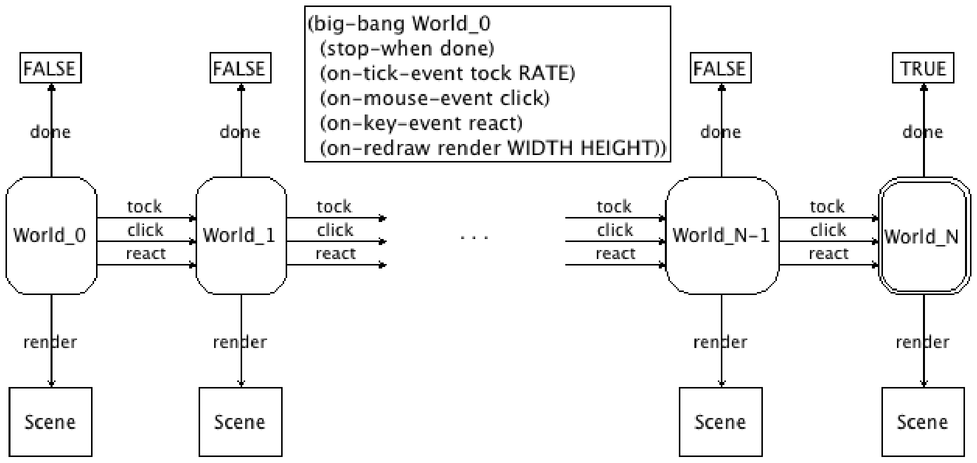


Figure 2: Source code and frames from the Darts game.



```
(big-bang World_0
  (stop-when done)
  (on-tick-event tock RATE)
  (on-mouse-event click)
  (on-key-event react)
  (on-redraw render WIDTH HEIGHT))
```

Figure 3: Structure of the World event loop.

```
(stop-when done)
;; done : World → Boolean
```

After each event, the World machine checks for a *done* state. When the current world satisfies *done*, no further events are processed. The event handlers rely on this behavior; their first input is an *ActiveWorld*, meaning a *World* for which *done* does not hold.

```
(on-tick-event tock RATE)
;; tock : ActiveWorld → World
;; RATE : Rational
```

Every *RATE* seconds, the clock ticks and the World machine transforms the world with the *tock* function.

```
(on-key-event react)
;; react : ActiveWorld Key → World
;; A Key is a Character or Symbol.
```

Each time the user presses a key, the World machine *react*s accordingly, producing a new world based on the current one and the keystroke. Regular keys (e.g., A, 0, or +) are represented as characters, while special keys (e.g., F1 or Escape) are represented as symbols.

```
(on-mouse-event click)
;; click : ActiveWorld Int Int Action → World
;; An Action is 'button-up, 'button-down, 'drag,
;; 'move, 'enter, or 'leave.
```

Whenever the user moves the mouse or presses (or releases) a button, the World machine updates the world with *click*. It produces a new world based on the current one, the mouse's x and y coordinates, and the type of action performed.

```
(on-redraw render WIDTH HEIGHT)
;; render : World → Image
;; WIDTH, HEIGHT : Nat
```

The World machine opens a canvas of dimensions *WIDTH* by *HEIGHT* for graphical output. It *render*s the image of the world to the canvas after each event.

The World machine processes events, updates the canvas, and keeps track of the current world value until *done*. Figure 3 gives a visual depiction of this process. From a semantic perspective, this form of I/O control is closely related to Clean's model of input and output [1]. It is also somewhat reminiscent of the Yale school of functional reactive programming [4, 11].

For convenience, the function arguments to **big-bang** are optional. The World machine ignores events for which no handler is provided, allowing the programmer to focus on events of interest (as seen in Darts). If there is no *render* function, the world comes without a visual presentation. If the *done* predicate is omitted, the World machine processes events indefinitely.

Images. Most of the datatypes used by the World machine already exist in ACL2: numbers, symbols, and characters. Graphical output introduces a new datatype: images. The World teachpack includes operations defining a functional image representation; figure 4 describes some of the most important operations.

The representation of images in Dracula's runtime simulation is complex to support efficient rendering, but to the user they are essentially two-dimensional vectors of pixels. They are distinct from other datatypes (not merely cons-based lists, for instance). Images are compared for equality pixel-by-pixel, so two images constructed differently may still be treated as the same. Dracula renders images graphically (rather than textually) when produced as a result in the interactions window, so their representation is never exposed.

4. REASONING ABOUT INTERACTIVE GRAPHICAL PROGRAMS

Dracula inherits the executable implementation of the World teachpack from DrScheme. To preserve the system's duality of execution and logical reasoning, we must assign to the World machine a logical meaning in ACL2. Otherwise, it is impossible to state and prove theorems about interactive graphical programs. In addition, we establish common patterns for verifying safety and liveness properties.

As implemented in DrScheme, the World machine event loop is a potentially non-terminating program. Input events may never occur, or the world may not reach a final state. Standard techniques exist for reasoning about non-terminating state machines in ACL2. For one instance, a program may model finite prefixes of a computation [3]. For another, a program may represent a partial, tail-recursive function with a total function extending its behavior [18]. We choose the former approach.

We model the World machine's event loop as a function operating on a fixed list of input events representing a prefix of the user's interactions. This function is constructed automatically by **big-bang** using the macro facilities of ACL2 and DrScheme. An event loop with all handlers provided has this form:

```
;; event-handler : ActiveWorld Event → World
(defun event-handler (w e)
  (cond
   ((tickp e) (tock w))
   ((keyp e) (react w e))
   ((mousep e)
    (click w (mouse-x e) (mouse-y e) (mouse-action e)))
   (t w)))


;; event-loop : ActiveWorld EventList → World
(defun event-loop (w es)
  (declare (xargs :measure (len es)))
  (cond
   ((endp es) w)
   ((done w) w)
   (t (event-loop (event-handler w (car es)) (cdr es)))))
```

The event-loop function dispatches individual events to event-handler until the computation is *done*, or it runs out of events.[2] The *done* clause is omitted if **big-bang** has no **stop-when** clause. We elide the full representation of events, but the three disjoint types are recognized by the predicates tickp, keyp, and mousep. Mouse events support the accessors mouse-x, mouse-y and mouse-action.

The event-handler auxiliary function processes individual events. It has one clause per input handler supplied to **big-bang**. The event handler updates the world via *tock* for timer events; *react* for keyboard events, passing along the keystroke; and *click* for each mouse event, passing in x, y,

[2]The **declare** clause in event-loop helps ACL2 make a termination argument.

Basic shape constructors They consume dimensions, a drawing mode ('solid or 'outline), and a color (e.g., 'red).	circle *: Nat Mode Color → Image* rectangle *: Nat Nat Mode Color → Image* triangle *: Nat Mode Color → Image* star *: Nat Nat Nat Mode Color → Image*
Various image constructors Includes text rendering, lines, blank scenes, and image overlay with offset.	text *: String Nat Color → Image* line *: Int Int Color → Image* empty-scene *: Nat Nat → Image* place-image *: Image Int Int Image → Image*
Predicate and accessors Support for recognizing images and computing their dimensions.	imagep *: Any → Boolean* image-width *: Image → Nat* image-height *: Image → Nat*

Figure 4: A few image functions from the World teachpack.

and *action* data. Any events not explicitly handled leave the world unchanged.

The complete **big-bang** computation for a sequence of events *es* is equal to (event-loop *World_0 es*). If an **on-redraw** clause is given, the image on the canvas after the same sequence of events is (*render* (event-loop *World_0 es*)).

Returning to Darts, **big-bang** produces the following event loop:

```
;; event-handler : ActiveWorld Event → World
(defun event-handler (w e)
  (cond
   ((mousep e) (throw-dart w
                           (mouse-x e)
                           (mouse-y e)
                           (mouse-action e)))
   (t w)))

;; event-loop : World EventList → World
(defun event-loop (w es)
  (declare (xargs :measure (len es)))
  (cond
   ((endp es) w)
   ((win-or-lose w) w)
   (t (event-loop (event-handler w (car es)) (cdr es)))))
```

The event loop handles each event until the player manages to *win-or-lose*. The handler reacts to mouse events by calling *throw-dart*, and ignores other events.

Safety. With an explicit representation of the World machine, we can begin to verify properties of interactive graphical programs. Let's look at a simple safety property: the event loop preserves some invariant of the world. To begin, we state the property:

```
;; invariant : Any → Boolean
(defun invariant (w) ...)

(defthm event-loop-invariant
  (implies (invariant w)
           (invariant (event-loop w es))))

(defthm big-bang-invariant
  (invariant (event-loop World_0 es)))
```

The *invariant* predicate formalizes our invariant; *event-loop-invariant* states that the event loop preserves it. The *big-bang-invariant* corollary states that the **big-bang** computation starting at *World_0* satisfies the invariant. These three pieces constitute a general pattern for safety proofs about the World machine.

For the Darts game, we use the representation of *World* as our invariant. We instantiate the pattern above to state our conjecture, adding sufficient annotation for the theorem prover to verify it:

```
;; dart-gamep : Any → Boolean
(defun dart-gamep (w)
  (or (natp w) (equal w 'win)))

(defthm event-loop-dart-gamep
  (implies (dart-gamep w)
           (dart-gamep (event-loop w es))))

(defthm big-bang-dart-gamep
  (dart-gamep (event-loop 3 es))
  :hints
  (("Goal"
    :in-theory (disable event-loop-dart-gamep)
    :use (:instance event-loop-dart-gamep (w 3)))))
```

The *dart-gamep* predicate recognizes the Darts world: either a number of remaining darts or 'win. The *event-loop-dart-gamep* conjecture states the preservation of *dart-gamep* throughout the event loop; its corollary *big-bang-dart-gamep* states that whole World computations satisfy *dart-gamep*. ACL2 automatically proves *event-loop-dart-gamep*. It requires only two hints to verify *big-bang-dart-gamep*, specifying how to apply *event-loop-dart-gamep* to the initial world.[3]

The Dracula World teachpack supports this pattern of safety proofs with a macro, (*world-preserves invariant*). It expands into conjectures of the form described above and provides appropriate default hints to the theorem prover.

Liveness. Once we can verify properties of the world representation, we can turn our attention to externally observable program behavior. We illustrate the point with a termination property: after some number of events, the computation must yield a final world with respect to *done*. We formulate the conjecture based on the kind of events that contribute to termination:

```
;; interesting-eventp : Event → Boolean
(defun interesting-eventp (e) ...)
```

[3]It is common for ACL2 to infer inductive proofs and rely on hints for corollaries.

```
;; count-interesting-events : EventList → Nat
(defun count-interesting-events (es)
  (cond
   ((endp es) 0)
   ((interesting-eventp (car es))
    (1+ (count-interesting-events (cdr es))))
   (t (count-interesting-events (cdr es)))))

(defthm big-bang-measure
  (implies (≥ (count-interesting-events es) MAXIMUM)
           (done (event-loop World_0 es))))
```

The conjecture *big-bang-measure* requires the world to reach a final state after *MAXIMUM* events satisfying *interesting-eventp*, counted by *count-interesting-events*. Other events are ignored; for instance, keystrokes may advance the world computation, while moving the mouse may not.

Our statement suggests a more general lemma. At any point in the computation, there is a measure of "interesting" events remaining before a final world:

```
;; measure : World → Nat
(defun measure (w) ...)

(defthm event-loop-measure
  (implies (and (invariant w)
                (≥ (count-interesting-events es)
                   (measure w)))
           (done (event-loop w es))))
```

The *event-loop-measure* lemma assumes that *invariant* is indeed true for all reachable states of the world, and states that *measure* computes the maximum number of interesting events before reaching a final world. The *big-bang-measure* conjecture is a corollary, where *MAXIMUM* is the measure of *World_0*. This pattern captures termination for many World-based programs; proving other liveness properties is also possible with variations on this strategy.

By applying this formula to the Darts game, we can verify that cheating is impossible: the player cannot get more than three attempts to hit the target. After at most three mouse clicks, the game must end one way or the other; see figure 5 for the text necessary to verify this property. The *darts-remaining* function computes the number of attempts the player has left; the *clickp* predicate recognizes mouse clicks and *count-clicks* counts them. The *event-loop-darts-remaining* conjecture states that a Darts game w ends after no more than (*darts-remaining* w) mouse clicks; *big-bang-darts-remaining* says the whole game allows 3 clicks. ACL2 verifies *event-loop-darts-remaining* automatically, and again needs only two hints to verify its corollary.

Dracula supports this pattern of termination proofs with another macro:

```
(world-ends-after measure interesting-eventp invariant)
```

This form expands into a count function and conjectures following the pattern shown above with appropriate hints.

Images. So far we have reasoned about the inputs and control behavior of our game. It is also important to reason about our outputs. The Dracula World teachpack includes an axiomatization of the portable properties of images, without translating the full details of their implementation to ACL2. Observations such as the imagep predicate and the dimensions of basic shapes are guaranteed in the logic. Others depend on system attributes, such as the font of text images, and are left abstract. Properties such as the commutativity and associativity of non-overlapping image overlay are partially axiomatized based on conservative bounding box approximations.

Dracula programs can verify output properties such as the size of images and whether all parts of a rendering fit on the screen at once; proofs are fully portable and hold on machines with different graphics drivers and hardware.

```
;; clickp : Event → Boolean
(defun clickp (e)
  (and (mousep e)
       (equal (mouse-action e) 'button-down)))

;; count-clicks : EventList → Nat
(defun count-clicks (es)
  (cond
   ((endp es) 0)
   ((clickp (car es)) (1+ (count-clicks (cdr es))))
   (t (count-clicks (cdr es)))))

;; darts-remaining : World → Nat
(defun darts-remaining (w)
  (if (natp w) w 0))

(defthm event-loop-darts-remaining
  (implies (and (dart-gamep w)
                (≥ (count-clicks es)
                   (darts-remaining w)))
           (win-or-lose (event-loop w es))))

(defthm big-bang-darts-remaining
  (implies (≥ (count-clicks es) 3)
           (win-or-lose (event-loop 3 es)))
  :hints
  (("Goal"
    :in-theory (disable event-loop-darts-remaining)
    :use (:instance event-loop-darts-remaining (w 3)))))
```

Figure 5: Text from which ACL2 can prove that Darts must end after 3 throws.

5. VERIFIED INTERACTIVE GRAPHICAL PROGRAMS

To validate the applicability of Dracula and the above recipes for safety and liveness, we conducted six experiments proving properties of interactive programs. The first five are small video games representative of student verification projects; the last is a simple text editor.

UFO. Our UFO game is a one-enemy version of Space Invaders. The UFO falls from the top of the screen, weaving left and right, dropping bombs. Meanwhile, the player controls a tank at the bottom of the screen, moving left and right and firing missiles upward. The player wins if a missile hits the UFO; the player loses if a bomb hits the tank or the UFO lands.

We can verify that the objects in the game (the UFO, tank, missiles, and bombs) stay within the visible portion of the screen (a safety property), and that the game ends in a fixed amount of time (a liveness property) because the UFO must eventually land. Figure 6 shows these properties.

```
;; objects-in-bounds : Any → Boolean
(defun objects-in-bounds (w)
  (and (gamep w)
       (in-bounds (game-tank w))
       (in-bounds (game-ufo w))
       (all-in-bounds (game-missiles w))
       (all-in-bounds (game-bombs w))))

(defthm objects-always-in-bounds
  (implies (objects-in-bounds w)
           (objects-in-bounds (event-loop w es))))

;; ufo-distance : World → Nat
(defun ufo-distance (w)
  (+ (ufo-lateral-distance w)
     (* (ufo-vertical-distance w) (1+ *WIDTH*))))

;; count-ticks : EventList → Nat
(defun count-ticks (es)
  (cond
    ((endp es) 0)
    ((tickp (car es)) (1+ (count-ticks (cdr es))))
    (t (count-ticks (cdr es)))))

(defthm ufo-eventually-lands
  (implies (and (objects-in-bounds w)
                (≥ (count-ticks es) (ufo-distance w)))
           (game-over (event-loop w es))))
```

Figure 6: Excerpts from verified properties about the UFO program.

Hangman. The player guesses letters in a word; correct guesses expose more of the word, while incorrect guesses reveal a condemned stick figure. The game ends when either the full word or full stick figure becomes visible.

Our Hangman game is proven to terminate after a fixed number of keystrokes; each advances the player one step closer to winning or losing.

Worm. In the Worm game, a.k.a. Snake or Nibbles, the player controls the direction of a worm on a grid. The grid has walls and, somewhere, a piece of food. If the worm eats the food, it grows and a new piece of food appears. If the worm runs into a wall or its own tail, the game ends.

For this game, we can prove that the worm's tail never crosses itself; the head can cross the tail, but the game must end before it gets any further. We can also prove that (for similar reasons) the worm's tail is always inside the grid, and that the segments of the worm are always adjacent to each other.

Blocks. This is a one-block version of Tetris: single blocks, rather than formations of four, drop down the screen; the player must guide them into position. Completed rows vanish and the blocks above drop down.

We can verify two properties of the correct stacking of blocks. They never fall off the bottom of the screen, and never overlap one another.

Bikes. Inspired by the movie TRON, the Bikes game enables each player to control a bike that leaves a permanent colored trail behind it. Players are eliminated as their bikes run into trails (others' or their own); the last player standing wins. Input from multiple players is distinguished by assigning players different keys. The trails left by bikes are represented as lists of coordinates of their corners.

Project	**Lines**	**Conjectures**	**Time (s)**
Hangman	365	11	1.48
Blocks	450	16	0.86
UFO	696	23	13.97
Worm	824	34	4.90
Editor	1,117	59	5.04
Bikes	1,354	84	202.11

Figure 7: Statistics about our experiments.

In this experiment, we are able to prove that segments of the bikes' trails always run in cardinal directions, never diagonally.

Editor. Our text editor, inspired by a previous case study on algebraic specifications [21], allows the user to manipulate a single line of text. The controls include typing, deleting, selecting text, navigating left and right, and an unbounded undo history. The editor's display shows a prefix of the current text, dropping any characters that do not fit entirely within the window.

We are able to verify key properties of the controls, including: typing enters the appropriate text, deleting does not remove text beyond the selection, undo restores the text before the last typing operation, and navigation doesn't affect the undo history.

This example also allows us to verify two properties of the editor's rendered image, namely the displayed text prefix always fits on the screen, and is always maximal. In other words, no longer prefix of the typed text fits.

Results. Figure 7 shows statistics on the size of each project, the theorem proving effort (by counting conjectures, including both lemmas and main theorems), and how long the theorem prover takes to verify them. The Bikes game required the most conjectures, and by far the longest time for ACL2 to verify, because we used a compact and indirect representation (leaving out points of the trail between the corners). An implementation storing each individual point in the trails might have been easier to verify. However, our choice of representation demonstrates Dracula's ability to scale to nontrivial problems.

For each project, we were able to verify one or more meaningful properties, including (over all the projects) safety and liveness conditions, as well as reasoning about graphical outputs.

There was a class of liveness conditions we were not able to express, however. Informally, they represent conditions that "can" happen, rather than "must". At any point in the UFO game before the UFO lands, the tank can fire a missle that destroys the UFO. In the Worm game, the worm can always make its way to the food, unless it has backed itself into a corner. With appropriate placement, the Blocks can always clear a row. These are liveness conditions—they represent eventualities that must be possible until the end of the game becomes inevitable—but no single type of event makes them inevitable. Instead, a sequence of event choices must be made to bring about these conditions; the game is not "fair" if these choices are impossible.

Our formulation of liveness is too näive for these conditions. Merely choosing the final condition *worm-eats-foodp* (in Worm) is insufficient; no fixed choice of *interesting-eventp* expresses the worm's path to the food. We need new patterns of proofs to reason about this kind of property.

6. RELATED WORK

Dowse et al. have used the Sparkle theorem prover [20] to verify a monadic filesystem API augmented with deterministic concurrency [8, 9]. Their system permits non-termination by including a bottom element in all types, which cannot later be ruled out. Davis reported on proofs about file I/O in ACL2 [6], which uses an explicit state-passing style and disallows non-termination. Dracula allows users to write interactive graphical programs without monads or state passing, and can express both non-terminating interactions and provable liveness properties, e.g., termination.

Other work has focused on verifying fully featured interactive systems such as interactive web programs or hierarchical, event-driven GUIs. Godefroid et al. [2] and Memon [19] have developed automated testing tools for these domains. While practical, this method of verification does not provide the level of assurance of formal reasoning, as no test suite for a complex program is ever exhaustive.

Krishnamurthi et al. [17] and Dwyer et al. [10] have applied model checking to interactive (web and GUI) programs. In this approach, verification systems extract a model from a source program; this is usually a graph with nodes representing program states and edges representing control flow. Each node is annotated with atomic propositions: simple properties known to hold in the associated program state. A model checker can then verify properties of the program expressed as temporal logic propositions about the model. For instance, verification of a banking web site might require reaching a state where "PIN submitted" holds true before proceeding to a "withdrawal approved" state. Much like our approach, there are methods for model checking tailored to safety and liveness properties. The set of properties that can be verified by a model checker depends on the method of constructing a model, the set of atomic propositions, the temporal logic used (e.g. computation tree logic vs. linear temporal logic), and the choice of model checking algorithm.

There is ongoing work in the verification of operating systems, such as the Verisoft [15] and L4.verified [22] projects, that includes automatic verification of interactive I/O at the hardware and device driver level. This is an important branch of verification, but we consider interactions between machine components to play a significantly different role in programs and correctness criteria from concurrent interactions with a human user.

7. CONCLUSIONS

We present Dracula, a tool based on the ACL2 theorem prover, providing automatically verified, executable, interactive graphical programs. We have demonstrated Dracula's ability to reason about the internal representation of such programs as well as their inputs and outputs.

Dracula represents another step toward automated formal verification of complex, real-world interactive programs. We are already working on an extension of the World teachpack with traditional GUI widgets, e.g., buttons, sliders, drop-down menus, etc. Based on a preliminary design, we conjecture that our technique extends naturally to GUIs that include these widgets as well as GUIs that use a hierarchical organization.

In collaboration with Rex Page (Oklahoma U.), we intend to ensure that Dracula's support of interactive programming is accessible to senior students in software engineering capstone courses. Conversely, we consider the verification of interactive graphical programs as an ideal tool to prepare senior and master-level students for positions and courses where they verify state machines and hardware processors.

Acknowledgements. Many thanks to Pete Manolios for ideas on verifying state machines, to Dale Vaillaincourt and Sky O'Mara for their contributions to the implementation, to Shriram Krishnamurthi for comments on an early draft, and to the NSF for a research grant supporting this work.

8. REFERENCES

[1] Achten, P. and M. J. Plasmeijer. The ins and outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.

[2] Benedikt, M., J. Freire and P. Godefroid. VeriWeb: Automatically testing dynamic web sites. In *Proceedings of the 11th International World Wide Web Conference*, 2002.

[3] Boyer, R. S. and J. S. Moore. Mechanized formal reasoning about programs and computing machines. In Veroff, R., editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos.* MIT Press, 1996.

[4] Cooper, G. H. and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th European Symposium on Programming*, volume 3924 of *Lecture Notes in Computer Science*, p. 294–308. Springer, 2006.

[5] Culpepper, R., S. Tobin-Hochstadt and M. Flatt. Advanced macrology and the implementation of Typed Scheme. In *Proceedings of the 8th Workshop on Scheme and Functional Programming*, p. 1–14. ACM, 2007.

[6] Davis, J. Reasoning about ACL2 file input. In *Proceedings of the 6th International Workshop on the ACL2 Theorem Prover and its Applications*, p. 117–126. ACM, 2006.

[7] Dix, A. J. and C. Runciman. Abstract models of interactive systems. In *People and Computers: Designing the Interface*, p. 13–22. Cambridge University Press, 1985.

[8] Dowse, M., A. Butterfield, M. van Eekelen, M. de Mol and R. Plasmeijer. Towards machine-verified proofs for I/O. In *Proceedings of the 16th International Workshop on the Implementation and Application of Functional Languages*, Technical Report 0408, Institute of Computer Science and Applied Mathematics of the University of Kiel, p. 469–480. ACM, 2004.

[9] Dowse, M., A. Butterfield and M. C. J. D. van Eekelen. Reasoning about deterministic concurrent functional I/O. In *Proceedings of the 16th International Workshop on the Implementation and Application of Functional Languages*, p. 177–194, 2004.

[10] Dwyer, M. B., Robby, O. Tkachuk and W. Visser. Analyzing interaction orderings with model checking.

In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, p. 154–163. IEEE, 2004.

[11] Elliot, C. and P. Hudak. Functional reactive animation. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*, p. 196–203. ACM, 1997.

[12] Felleisen, M., R. B. Findler, M. Flatt and S. Krishnamurthi. *How to Design Programs.* MIT Press, 2001.

[13] Findler, R. B., C. Flanagan, M. Flatt, S. Krishnamurthi and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Programming Languages: Implementations, Logics, and Programs*, volume 1292 of *Lecture Notes in Computer Science*, p. 369–388. Springer, 1997.

[14] Flatt, M. Composable and compilable macros: You want it *when?* In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, p. 72–83. ACM, 2002.

[15] Hillebrand, M. A. and W. J. Paul. On the architecture of system verification environments. In *Hardware and Software: Verification and Testing*, volume 4899 of *Lecture Notes in Computer Science*, p. 153–168. Springer, 2008.

[16] Kaufmann, M., P. Manolios and J. S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.

[17] Licata, D. R. and S. Krishnamurthi. Verifying interactive web programs. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, p. 164–173. IEEE, 2004.

[18] Manolios, P. and J. S. Moore. Partial functions in ACL2. *Journal of Automated Reasoning*, 31:107–127, 2003.

[19] Memon, A. M. An event-flow model of GUI-based applications for testing. *Software Testing, Verification, and Reliability*, 17(3):137–157, 2007.

[20] Mol, M. D., M. V. Eekelen and R. Plasmeijer. Theorem proving for functional programmers—Sparkle: a functional theorem prover. In *The 13th International Workshop on Implementation of Functional Languages*, volume 2312 of *Lecture Notes in Computer Science*, p. 55–72. Springer, 2001.

[21] Partsch, H. On the use of algebraic methods for formal requirements definitions. In *Requirements Engineering, Arbeitstagung der GI*, p. 138–158. Springer, 1983.

[22] Tuch, H., G. Klein and G. Heiser. OS verification—now! In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, p. 7–12. USENIX, 2005.

[23] Vaillancourt, D., R. Page and M. Felleisen. ACL2 in DrScheme. In *Proceedings of the 6th International Workshop on the ACL2 Theorem Prover and its Applications*, p. 107–116. ACM, 2006.