Towards Automated Property Discovery within Hume

Gudmund Grov and Andrew Ireland

Heriot-Watt University (`G.Grov@hw.ac.uk`, `A.Ireland@hw.ac.uk`)

Abstract

Hume is a Turing-complete programming language, designed to guarantee space and time bounds whilst still working on a high-level. Formal properties of Hume programs, such as invariants and transformations, have previously been verified using the temporal logic of actions (TLA). TLA properties are verified in an inductive way, which often requires lemma discovery or generalisations. Rippling was developed for guiding inductive proofs, and supports lemmas and generalisation discovery through proof critics. In this paper we show how rippling and proof critics can be used in the verification of Hume invariants represented in TLA. Our approach is based on existing work on the problem of verifying and discovering loop invariants for an imperative program. We then extend this work to Hume program transformations.

1 Introduction

In [32], Storey identifies *complexity of definitions*, *expressive power*, *bounded space and time*, *logical soundness*, *security* and *verifiability* as the key features for a programming language targeting safety-critical systems. No language supports all of them, and several of them are conflicting: for example, many time and space properties are undecidable for Turing-complete languages, and low-level languages, where this is decidable, often have a high complexity.

Hume [19] is a novel Turing-complete programming language designed to reduce the complexity of definition, whilst guaranteeing bounds on space and time usage, both key features in Storey's list. This is achieved by a layered architecture: a low-level, concurrent, finite state automata language, termed the *coordination layer*, is built on top of a high-level (Turing-complete) strict functional language, termed the *expression layer*. Programming then involves balancing between the two layers, and due to failed costing, this often requires transformations from the expression layer into the coordination layer. Hence, resource costing and program transformations are at the heart of the Hume development methodology.

The time and space analysis is well developed for Hume, as described in e.g. [17]. Correctness verification, which is also found on Storey's list, has previously been applied to Hume programs in the temporal logic of actions (TLA)[28].This work appears in [13], and can handle both invariants of the coordination layer, and verification of program transformations, which can be reduced to an invariant proof [16]. The proof of an invariant often requires the discovery of auxiliary invariants[1]. In [13], several Isabelle/HOL [31] tactics are given for reasoning about Hume programs within TLA. While a high degree of proof automation was achieved, a key missing ingredient is invariant discovery. Here, we build upon *rippling* [8], a search control technique designed for reasoning about inductive conjectures. In particular, we focus on *proof critics* [22] for rippling, which, can be used to guide the discovery of inductive invariants. Previously, these ideas have been applied to the verification and discovery of loop invariants for imperative programs [25, 24]. In this paper we will show how this work can be applied to

[1]This is also the case for a generic TLA invariant, as discussed in [12].

Hume/TLA, and then extend it to program transformation verification. We also believe that this work is not limited to Hume, but applicable to generic Hume specification, which we will elaborate upon in §7.

The paper is structured as follows: we will first introduce the preliminaries in §2; this is followed by a discussion on the use of rippling to verify Hume invariants in §3; and proof critics to discover loop invariants in §4; this work is then extended to Hume program transformations in §5; before we discuss relevant work (§6); future work (§7); and conclude in §8.

2 Preliminaries

2.1 The temporal logic of actions (TLA)

The *temporal logic of actions* (TLA) [28] was developed to reason about concurrent systems, and combines temporal logic with actions. It is a uniform logic that can capture both safety and liveness requirements, however we will only discuss the safety aspect here. It is a three-tier logic where:

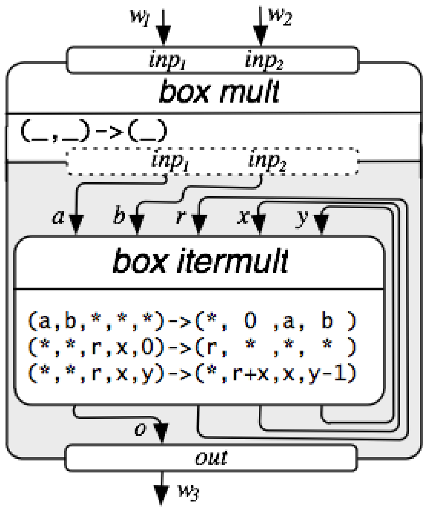
- in the *state level*, a *state function/predicate* is a function/predicate on one particular state, where a state is mapping from variables to values;
- in the *action level*, an *action* is a predicate on two states: a "before" and "result" state of the action;
- in the *temporal level*, a *formula* is a predicate on an infinite sequence of states.

All levels include a full predicate calculus. Additionally, the action level has a priming (') operator to separate variables in the "result" state (primed) from those of the "before" state. For example, $x' = x + 1$ is the computation that increments x by 1. At this level, "before" variable v and its "result" counterpart v', are distinct. The temporal level has two additional operators: $\Box P$, which denotes P holds iff it holds for all following states of the sequence; and the $\boldsymbol{\exists}$ operator, for (temporal) existential quantification. $\boldsymbol{\exists}$ is used to hide variables internal for a specification. To show that a property holds for a program, we must show that the program *implements* the property. In TLA, both programs and properties are specified in the same logic, hence this is formalised as logical implication.

TLA allows specifications to be written at different levels of abstraction. The key to proving such refinements between abstract and concrete representations, is to allow *stuttering steps*, i.e. steps that leave the state unchanged. These steps are seen as internal steps within a specification. To define a (monolithic[2]) program we must specify an initial state I, and an action N, representing the transitions. N is a predicate which compares a "result" and a "before state". The action must hold throughout execution, i.e. $\Box N$. However, this does not support stuttering steps. Let $\langle v, i\rangle$ be the tuple of all visible variables v, and internal variables i of the program. We refine $\Box N$ to $\Box(N \vee \langle v, i\rangle' = \langle v, i\rangle)$, which asserts that in all transitions either N holds, or the state is left unchanged. This supports stuttering, and is abbreviated by $\Box[N]_{\langle v,i\rangle}$. We will use monolithic specifications of our programs. Such specifications, with the internal variables hidden, are written:

$$\boldsymbol{\exists}\, i : I \wedge \Box[N]_{\langle v,i\rangle}. \tag{1}$$

[2]A detailed explanation of a *monolithic* TLA specification can be found in [29].


w1
w2
inp1
inp2
box mult
(_,_)->(_)
inp1
inp2
a
b
r
x
y
box itermult
(a,b,*,*,*)->(*, 0 ,a, b)
(*,*,r,x,0)->(r, * ,*, *)
(*,*,r,x,y)->(*,r+x,x,y-1)
o
out
w3

$$
\begin{array}{lrcl}
\text{(i)} & \Box(a * b & = & inp_1 * inp_2) \\
\text{(ii)} & \Box(\boldsymbol{r} + (\boldsymbol{x} * \boldsymbol{y}) & = & \boldsymbol{inp}_1 * \boldsymbol{inp}_2) \\
\text{(iii)} & \Box(o & = & inp_1 * inp_2) \\
\text{(iv)} & \Box(out & = & inp_1 * inp_2)
\end{array}
$$

Figure 1: Hume multiplication as iteration.

```
mult:  $inp_1 := w_1;\ inp_2 := w_2;$
       { True }
          $a := inp_1;\ b := inp_2;$
          $r := 0;\ x := a;\ y := b;$
          while ($y \neq 0$)
          begin
             $r := r + x;\quad y := y - 1;$
          end
          $o := r;$
          $out := o;$
       {$out = inp_1 * inp_2$ }
```

Figure 2: Imperative multiplication as iteration.

To ease reasoning, TLA requires that N must always specify the complete "result" state. Thus, unchanged variables must be explicitly stated. In a monolithic specification the subscript, i.e. $\langle v, i\rangle$, should therefore hold all the variables. Although $\boldsymbol{\exists}$ is semantically different from $\exists$, the proof rules are similar. For the work presented here for Hume, we can ignore $\boldsymbol{\exists}$ since our Hume invariants are independent of the $\boldsymbol{\exists}$ -bound variables. Moreover, in a transformation proof the witness for i will always be the same. Thus, to ease the reading, we will a assume TLA specifications of the form:

$$I \wedge \Box[N]_v. \tag{2}$$

2.2 Hierarchical Hume

The *Hume coordination* layer describes a system as concurrent *boxes* linked by *wires*. Boxes are scheduled in a cyclical way, where each runnable box is executed in each step, and this process never terminates. In this paper we will use the *Hierarchical Hume* [14, 16] extension to Hume, which allows nesting of boxes inside another box. Now, a box consists of a set of *matches* of the form

pattern `->` *expression*

where *pattern* is matched against the box's input wires. In a non-nesting box, a match will cause the *expression*, which belongs to the expression layer, to generate output to the output wires. In a nesting box, a match will copy the inputs into external input wires, and schedule the children boxes until the termination condition, defined by the *expression* is met. Then the internal output wires are copied to the output wires. If a pattern fails, the next match is attempted.

Hume boxes are scheduled in a two-phase lock step scheduling algorithm, where each step works as follows[3]:

- each box is executed and output is produced in a result buffer (e.g. `out` of the `mult` box of Figure 1) in the execute phase;
- this is followed by a uniform super-step where outputs are asserted to the wires.

For a nesting box, this scheduler is nested, i.e. the children of the nesting box are scheduled similarly, until termination. To ease the reading, and enable focus on the key issues, we will abstract over this scheduling for the boxes that are nested. Moreover, these boxes are assumed to not be nesting, i.e. we assume a box hierarchy of two levels. Here, box execution and output assertion in one uniform step. By way of illustration, we present in Figure 1 an iterative implementation of a multiplier in Hume. Note that an equivalent imperative program is shown in Figure 2. Figure 1 graphically illustrates the uniform scheduling step. Here, the nested `itermult` box does not contain an output buffer, whilst `out` is the output buffer of the first level (nesting) `mult` box. Note that for our proofs below, the nesting `mult` box requires a lower abstraction level, containing both an input and an output buffer, as well as the two phase scheduling.

The `mult` box of Figure 1 performs multiplication by iteration. This is accomplished by the nested `itermult` box, which multiplies the inputs by iterative addition and is achieved by the feedback loop wires `r`,`x` and `y`. Note that the result is produced in one first-level step, even though many internal `mult` steps may be required.

Figure 2 describes the same program in an arbitrary imperative language, with the correctness condition annotated by Hoare triples [20]: $\{P\}c\{Q\}$ denotes that if P holds and statement c terminates, then Q holds. The Hume program then works as follows. If the wires $\mathtt{w}_1$ and $\mathtt{w}_2$ contains a value, then these are copied to the input buffer $\mathtt{inp}_1$ and $\mathtt{inp}_2$, and to the internal wires `a` and `b`; the internal `itermult` box is then scheduled until the `o` wire has got a value, which on termination is copied to the output buffer `out` (and output wire $\mathtt{w}_3$). The first match of the `itermult` box then succeeds, which copies the `a` and `b` wires to the `x` and `y` wires, while `r` is set to `0`. This is the "entry step" of the loop. In Hume, `*` means 'ignore' in a pattern, and 'do not write' in an expression. The third match is the "loop step" of the imperative program, and will fire when the `x`, `y` and `r` wires contain values and $\mathtt{y} \neq \mathtt{0}$. It increments `r` by `y`, leaves `x` unchanged, and decrement `y` by `1`. The second match is the 'exit step' of the loop where the

[3]See for example [15] for details.

result of the tail-iteration `r` of the “loop steps” is copied to the `o` wire, and the termination condition of `mult` then holds, thus copying `o` to `out`.

2.3 Proof planning & rippling

Input sequent:

$$H \vdash G[f_1(\boxed{c_1(\underline{\ldots})}^{\uparrow}, f_2(\lfloor\ldots\rfloor), f_3(\boxed{c_2(\underline{\ldots})}^{\uparrow}))]$$

Method preconditions:

1. there exists a subterm T of G which contains wave-front(s), *e.g.*

$$f_1(\boxed{c_1(\underline{\ldots})}^{\uparrow}, f_2(\lfloor\ldots\rfloor), f_3(\boxed{c_2(\underline{\ldots})}^{\uparrow}))$$

2. there exists a wave-rule which matches T, *e.g.*

$$C \rightarrow f_1(\boxed{c_1(\underline{X})}^{\uparrow}, Y, Z) \Rightarrow \boxed{c_5(\underline{f_1(X, \boxed{c_3(\underline{Y})}^{\downarrow}, \boxed{c_4(\underline{Z})}^{\downarrow})})}^{\uparrow}$$

3. the wave-rule condition follows from the context, *e.g.*

$$H \vdash C$$

4. resulting inward directed wave-fronts are potentially removable, *e.g. sinkable or cancellable, i.e.*

$$\ldots \boxed{c_3(\underline{f_2(\lfloor\ldots\rfloor)})}^{\downarrow} \ldots$$

or

$$\ldots \boxed{c_4(\underline{f_3(\boxed{c_2(\underline{\ldots})}^{\uparrow})})}^{\downarrow} \ldots$$

Output sequent:

$$H \vdash G[\boxed{c_5(\underline{f_1(\ldots, \boxed{c_3(\underline{f_2(\lfloor\ldots\rfloor)})}^{\downarrow}, \boxed{c_4(\underline{f_3(\boxed{c_2(\underline{\ldots})}^{\uparrow})})}^{\downarrow})})}^{\uparrow}]$$

Figure 3: The rippling method.

Proof planning is a technique for automating proof search. Central to the technique is the notion of a *proof plan* [6], a high-level proof outline which encodes a common pattern of reasoning. We will focus here on a proof plan called *rippling*. Rippling is a rewriting technique based upon a difference reduction strategy. To illustrate, consider a conjecture where you are given a hypothesis of the form $(\forall b'.\ f(a, b'))$ while the goal takes the form $f(c_1(a), b)$. Note that the $c_1(\ldots)$ embedded within the goal prevents a match with the given hypothesis. In rippling, such embedded structures are called *wave-fronts*. The goal of rippling is to identify and reduce the number of wave-fronts such that a hypothesis can be applied. Wave-fronts can be represented using explicit annotations that are added to the goal. For example, using shading to denote wave-fronts, the goal given above becomes:

$$f(\boxed{c_1(a)}^{\uparrow}, \lfloor b \rfloor)$$

In addition to the shading, note that a wave-front is annotated with an arrow. The arrow indicates which direction the wave-front can be moved, *i.e.* upward or downward through the goal structure. There are two reasons for moving a wave-front downward. Firstly, if a wave-front can be moved to a position corresponding to a universally quantified variable within the given hypothesis, then the wave-front can be eliminated via the specialisation of the universal hypothesis. This is known as *sinking* a wave-front, and the $\lfloor \ldots \rfloor$ annotation within the goal is used to indicate sink positions. Secondly, multiple wave-fronts can sometimes cancel each other out, a kind of destructive interference. So moving wave-fronts closer together can also make sense. The manipulation of wave-fronts is achieved via so called *wave-rules*. A wave-rule is a rewrite rule that has been annotated by wave-fronts. A key property of wave-rules is that they preserve the unannotated structure of the goal, the so called *skeleton*. Preserving skeleton maximises the chances of eventually applying the given hypothesis. In the schematic example given above, the following represents an applicable wave-rule:

$$f(\boxed{c_1(X)}^{\uparrow}, Y) \Rightarrow \boxed{c_2(f(X, \boxed{c_3(Y)}^{\downarrow}))}^{\uparrow}$$

Note that $\Rightarrow$ represents a rewrite, while $\rightarrow$ is used for logical implication. In general a proof plan contains *methods* and *critics* [21]. Rippling is represented by a single method as given in Figure 3. While methods represent common patterns of reasoning, critics are used to define patchable exceptions. When a method fails, its associated critics analyse the proof-failure and initiate a proof patch [21, 22, 23]. Typically the proof patching process makes use of meta-variables as place-holders for missing structure with the expectation that the constraints of the proof will provide instantiations during the planning of the remainder of the proof. This style of patching a proof is known as *middle-out reasoning* [7]. An example of a proof patch which exploits rippling and middle-out reasoning will be given in §4. For a complete description of rippling see [8].

3 Invariant verification

In [24], rippling was used to verify Hoare-triple properties as illustrated in Figure 2. To verify a Hoare-triple, it is converted into a *verification condition* (VC), a purely logical statement, by a *verification condition generator* (VCG). The VC is then verified by a theorem prover. However, before this can be done each statement must be turned into a Hoare-triple. This is mostly an automatic process, however finding and verifying an invariant which holds for the ***while*** loop, known as the *loop invariant*, is the hardest part. Thus, we will only focus on the loop invariant here. In the case of Figure 2, we use an invariant of the form $r + (x * y) = inp_1 * inp_2$. In the proof, the invariant is assumed beforehand, and this assumption is called the *invariant hypothesis* (IH)

$$\mathsf{IH}:\ r + (x * y) = inp_1 * inp_2. \tag{3}$$

Using the assumption, it is shown to hold after the loop has executed. By using wave-annotation, this goal is expressed as $\boxed{(\underline{r}+x)}^{\uparrow} + (x * \boxed{(\underline{y}-1)}^{\uparrow}) = inp_1 * inp_2$. The proof requires the following wave-rules:

$$\boxed{(\underline{X}+Y)}^{\uparrow} + Z \Rightarrow X + \boxed{(Y + \underline{Z})}^{\downarrow} \tag{4}$$

$$X * \boxed{(\underline{Y}-1)}^{\uparrow} \Rightarrow \boxed{(\underline{X*Y}) - X}^{\uparrow} \tag{5}$$

$$\boxed{(X + \underline{\boxed{\underline{Y}-X}^{\uparrow}})}^{\downarrow} \Rightarrow Y, \tag{6}$$

and is derived as follows:

$$\begin{array}{rl}
\boxed{(\underline{r}+x)}^{\uparrow} + (x * \boxed{(\underline{y}-1)}^{\uparrow}) = inp_1 * inp_2 & \text{[apply (4)]} \\
r + \boxed{(x + \underline{(x * \boxed{(\underline{y}-1)}^{\uparrow})})}^{\downarrow} = inp_1 * inp_2 & \text{[apply (5)]} \\
r + \boxed{(x + \underline{(\boxed{(\underline{x*y}) - x}^{\uparrow})})}^{\downarrow} = inp_1 * inp_2 & \text{[apply (6)]} \\
r + (x*y) = inp_1 * inp_2 & \text{[apply IH]}
\end{array}$$

Hoare logic was developed for sequential programs, and cannot be be applied directly to Hume programs due to issues of concurrency. Moreover, note that although Hume has a finite state machine architecture, *model checking* [10] is not in general suitable for program verification, due to a strong dependency with the data-centric expression layer, as described in [14]. However, it is applicable for small subsets of Hume, as described in [18], using the TLC model checker for the TLA$^+$ [29].

In TLA, programs and properties are represented in the same uniform logic: a property P holds for a program S, if S implements P, and implementation is represented as logic implication:

$$\vdash S \to P.$$

TLA always attempts to reduce temporal properties to the action level. This is illustrated by the derived induction rule for proving invariants of specification as shown in (2):

$$\frac{\vdash I \to P \qquad \vdash P \wedge V' = V \to P' \qquad \vdash P \wedge N \to P'}{\vdash I \wedge \Box[N]_V \to \Box P}. \tag{7}$$

The first sub-goal (assumption) is the base case, which states that P holds in the initial state I. The second case captures that the sub-script of the action V, at least contains the free variables of P, which is required for stuttering invariance. The last case, $\vdash P \wedge N \to P'$ states that P is preserved by action N. Here, P' means that P is a predicate over the result state of N. We will only discuss the action level here since the temporal level proofs are trivial for our examples. Moreover, the first two sub-goals are normally trivial, thus we will only discuss the main sub-goal, i.e. $\vdash P \wedge N \to P'$.

The partial correctness property of the `mult` box, is given by (iv) in Figure 1, which corresponds to the Hoare triples for the imperative program of Figure 2. As in the imperative case of Figure 2, the main part of the overall proof is the loop invariant. This corresponds to (ii) of Figure 1. The proof of the loop invariant depends on (i), the "loop entry" step. The invariant follows directly from the Hume semantics, while the partial correctness property (iv) follows directly from (iii), where (iii) is the "loop exist" step. (iii) can also be proven directly, using the loop invariant (ii). As in the imperative case, we will only discuss the loop invariant (ii) henceforth.

The last match of `itermult` corresponds to the ***while*** loop in the imperative program. Since the match expression is the result of executing a Hume box, it refers to the primed result state of an action. Let $[\![C]\!]$ be the semantics of a Hume construct C represented in TLA[4]. Due to the wiring, graphically illustrated in Figure 1, the annotated result of $[\![\mathsf{(*,r\text{-}x,x,y\text{-}1)}]\!]$ is represented as:

$$\begin{array}{c} r' = \boxed{(\boxed{r} + x)}^{\uparrow} \quad x' = x \quad y' = \boxed{(\boxed{y} - 1)}^{\uparrow} \\ inp_1' = inp_1 \qquad inp_2' = inp_2. \end{array} \tag{8}$$

The "loop invariant" in TLA is the same as in the imperative program, and the IH for the invariant proof is also the same (IH: $r + (x * y) = inp_1 * inp_2$). The ripple proof derivation starts of as

$$\begin{array}{rl} r' + (x' * y') = inp_1' * inp_2' & \text{[apply (8)]} \\ \boxed{(\boxed{r} + x)}^{\uparrow} + (x * \boxed{(\boxed{y} - 1)}^{\uparrow}) = inp_1 * inp_2 & [\cdots] \end{array}$$

and the remaining proof is identical to the imperative program. Note, that the annotation step shown here, is handled by the verification condition generator (VCG) in the imperative version.

4 Loop invariant discovery

The hardest part of Hoare-triple proofs, is the undecdable tasks of finding a strong enough loop invariant, like (3). [24] contains novel work, where proof critics [22] are used to explore partial ripple successes to discover a strong enough loop invariant. Firstly, both a ***while*** loop and a Hume feedback loop, as in Figure 1, require a tail-invariant, and the proof critic thus provides a tail-invariant patch. We will now apply the work described in [24] to discover the Hume "feedback loop invariant", required for the proof in the previous section. The post-condition for the property is $out = inp_1 * inp_2$, which is updated as follows: $out' = o$ and $o' = r$. Thus, an obvious first approximation of the loop invariant becomes:

$$\mathsf{IH}: \; r = inp_1 * inp_2.$$

By the TLA "induction rule" (7), and (8), the proof of the "loop action" blocks when attempting to ripple

$$\underbrace{\boxed{\boxed{r} + x}^{\uparrow}}_{\textbf{blocked}} = inp_1 * inp_2.$$

A proof is blocked when there are no applicable wave rules, hence rippling is not possible. However, the precondition of the tail-invariant proof critic succeeds, as illustrated in Figure 4. That is, a partial match between the blocked wave front and the available wave rules suggests a schematic invariant of the form

$$F_1(r, x, y) = inp_1 * inp_2, \tag{9}$$

[4]The Hume to TLA translation is not the topic here, and has thus been omitted. However, note that his is an operational representation of the Hume semantics, and in the translated TLA, we will use *italic font* instead of `sans serif`. Please see [13, 15, 18] for details of the TLA representation of Hume programs.

Blockage:

$$\boxed{\underline{r} + x}^{\uparrow} = inp_1 * inp_2$$

Critic precondition:

- Precondition 1 of rippling succeeds, *i.e.*
  1. there is a subterm T of G which contains a wave-front(s), e.g.

$$\boxed{\underline{r} + x}^{\uparrow}$$

- Precondition 2 of rippling partially succeeds, *i.e.*
  2. there exists a wave-rule which partially matches, e.g.

$$\boxed{\underline{r} + x}^{\uparrow} \quad \text{with} \quad \boxed{(\underline{X} + Y)}^{\uparrow} + Z \Rightarrow \ldots$$

Proof patch:
Speculate additional term structure within the conjecture such that preconditions 2, 3 and 4 will also potentially succeed, *i.e.*

$$F_1(\boxed{\underline{r} + x}^{\uparrow}, x, \boxed{\underline{y} - 1}^{\uparrow}) = inp_1 * inp_2,$$

where F_1 is a higher-order meta-variable.

Figure 4: A tail-invariant proof critic instantiation.

where F_1 is a second-order meta-variable. The expectation is that F_1 will be instantiated during the course of a rippling proof. The primed variables in $F_1(r', x', y') = inp_1' * inp_2'$ are first rewritten using (8)

$$F_1(\boxed{\underline{r} + x}^{\uparrow}, x, \boxed{\underline{y} - 1}^{\uparrow}) = inp_1 * inp_2.$$

By wave-rule (4), a new meta-variable F_2 is introduced by instantiating F_1 to $\lambda X.\lambda Y.\lambda Z.X + F_2(X, Y, Z)$. Application of (4) then derives

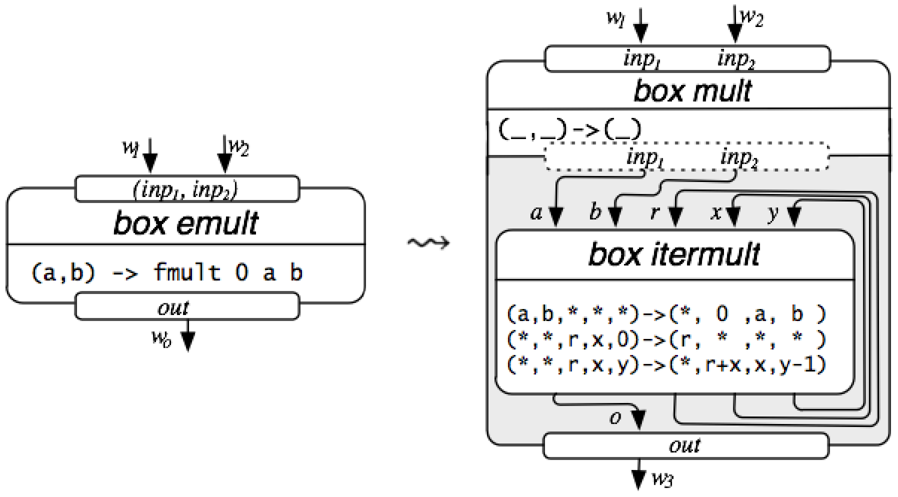
$$r + \boxed{x + \underline{F_2(\boxed{\underline{r} + x}^{\uparrow}, x, \boxed{\underline{y} - 1}^{\uparrow})}}^{\downarrow} = inp_1 * inp_2.$$

Here, wave rule (5) suggests an instantiation for F_2, i.e. $\lambda X.\lambda Y.\lambda Z.F_3(X, Y, Z) * (Z - 1)$, which gives

$$r + \boxed{x + \underline{\boxed{\underline{F_3(\boxed{\underline{r} + x}^{\uparrow}, x, \boxed{\underline{y} - 1}^{\uparrow}) * y} - F_3(\boxed{\underline{r} + x}^{\uparrow}, x, \boxed{\underline{y} - 1}^{\uparrow})}^{\uparrow}}}^{\downarrow} .$$

Finally, wave rule (6) suggest that F_3 is instantiated to $\lambda X.\lambda Y.\lambda Z.Y$, resulting in the following invariant:

$$r + (x * y) = inp_1 * inp_2.$$



```
(a,b) -> fmult 0 a b
```

```
(_,_)->(_)
```

```
(a,b,*,*,*)->(*, 0 ,a, b )
(*,*,r,x,0)->(r, * ,*, * )
(*,*,r,x,y)->(*,r+x,x,y-1)
```

Figure 5: Box multiplication as recursion to iteration transformation

Note, the invariant is identical to the invariant in §3, and the proof structure is the same.

5 Hume program transformations

To formalise the relationship between expressiveness/high-levelness and resource bounds, Hume introduces a set of *levels*, where each downward-level restricts expressiveness and thus increases the set of decidable resource properties. Now, the Hume methodology is based around decidability analysis, which explores the different Hume levels: a high-level and expressive program is first created and resource costing is attempted; if it succeeds we are done; if it fails, the problem is identified and resource bounds are either altered, or the violating program constructs are transformed to a lower level and costing is reapplied. This process is iterative until costing succeeds. In the previous two sections, we showed how rippling and proof critics, used to verify imperative programs, can be applied to verify and discover Hume invariants using TLA. We will now show how this work can be extended to verify Hume transformations, a key concept of the Hume methodology.

Figure 5 shows the translation from a high-level Hume box, `emult`, which performs multiplication by primitive recursive addition, into the `mult` box previously discussed. In `emult` the addition is achieved by the `fmult` function:

```
fmult r _ 0 = r;
fmult r x y = fmult (r+x) x (y-1);
```

This `emult` box is in the *PR-Hume* level, where resource properties are undecidable. The `mult` box from Figure 1 on the other hand, is in the *FSM-Hume* level, which guarantees strong resource bounds. A typical Hume transformation, transforms a "recursive box", such as `emult`, into a more costable "iterative box", such as `mult`. Such a transformation is graphically illustrated in Figure 6, for an arbitrary program $prog_1$ containing `emult`. Note that

$$\texttt{fmult}\ 0\ x\ y = x * y$$

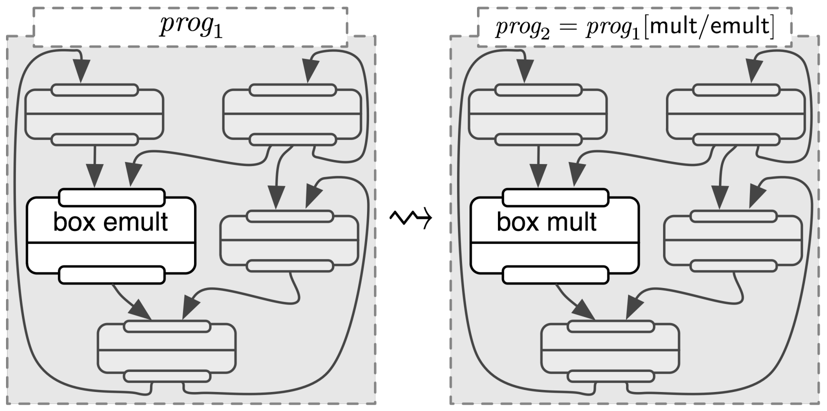


Figure 6: Program transformation example.

This is verified by first generalising it to $\forall r, x. \texttt{fmult}\ r\ x\ y = r + (x * y)$, which is then verified by induction on y. This proof has been mechanised in Isabelle/HOL, and is given in [13].

5.1 Transformation verification

In TLA, a transformation is represented as a *refinement*: a Hume program P_1 transforms into a Hume program P_2, which we write $P_1 \rightsquigarrow P_2$, iff P_2 *implements* P_1. Programs and properties are represented in the same logic, hence this is represented as a logical implication:

$$P_1 \rightsquigarrow P_2 \equiv [\![P_2]\!] \rightarrow [\![P_1]\!] \tag{10}$$

Both $[\![P_2]\!]$ and $[\![P_1]\!]$ are assumed to be of form shown in (2). $P_1 \rightsquigarrow P_2$ is then verified by the following derived TLA proof rule:

$$\frac{\vdash J \rightarrow I \qquad \vdash W' = W \rightarrow V' = V \qquad \vdash M \rightarrow [N]_V}{\vdash J \wedge \Box[M]_W \rightarrow I \wedge \Box[N]_V}. \tag{11}$$

If the two first givens are indeed theorems, then they are normally trivial to verify. The main part is the proof of $\vdash M \rightarrow [N]_V$ which asserts that a step in the transformed program is either a step of the original source program, or all variables are left unchanged. Note, that as in (7), the rule (11) reduces the temporal level to the action level.

Let *correct nesting* of a box-to-box transformation denote that the inputs and outputs are the same, and any computation on wires is nested inside a box, hence both boxes compute results in one scheduling step. For example, the `emult` ($prog_1$) to `mult` ($prog_2$) transformation depicted in Figure 6 has correct nesting. Although informally, [16] shows that correctly nested box-to-box transformations, reduces to a proof of function correctness, meaning the following derivation holds:

$$\begin{array}{cl} prog_1 \rightsquigarrow prog_2 & \text{[apply (10)]} \\ [\![prog_2]\!] \rightarrow [\![prog_1]\!] & \text{[apply definition of } prog_2\text{]} \\ [\![prog_1\text{[mult/emult]}]\!] \rightarrow [\![prog_1]\!] & \text{[apply result from [16]]} \\ [\![\text{mult}]\!] \rightarrow [\![\text{emult}]\!] & (\Pi) \end{array}$$

Moreover, [16] also shows that the proof of (Π) reduces to a pre-post condition proof, similar to the ones in the previous sections[5]. Here, the output buffer in the transformed box (e.g. `mult`) must produce the same output as the source box expression (e.g. `fmult 0 a b` of `emult` where `a` equals inp_1 and `b` equals inp_2). Hence, (Π) reduces the transformation proof to the following invariant:

$$out = \mathsf{fmult}\ 0\ inp_1\ inp_2 \tag{12}$$

Note that since the `mult` box is the assumption in the implication in (Π), *out*, inp_1 and inp_2 are "assumed updated" by the mult box. Moreover, with respect to the overall $\vdash M \rightarrow [N]_V$ conjecture, all but the termination step implies $\vdash V' = V$, while in the termination step $\vdash M \rightarrow N$. Since, N implies the application of `fmult`, this conjecture follows from this pre-post condition.

As in the previous sections, the key to the proof of (12), is the loop invariant of the nested feedback loop of mult. From the definition of `fmult` the following conditional wave-rule is derived:

$$Y \neq 0 \rightarrow \mathsf{fmult}\ \boxed{(\boxed{R} + X)}^{\uparrow}\ X\ \boxed{(\boxed{Y} - 1)}^{\uparrow} \Rightarrow \mathsf{fmult}\ R\ X\ Y \tag{13}$$

This is the only rule required in the proof. The loop invariant here is $\mathsf{fmult}\ r\ x\ y = \mathsf{fmult}\ 0\ inp_1\ inp_2$, and the invariant hypothesis is obviously the same:

$$\text{IH: } \mathsf{fmult}\ r\ x\ y = \mathsf{fmult}\ 0\ inp_1\ inp_2$$

The "loop step" of the `mult` box then induces the following derivation:

$$\begin{array}{cl} \mathsf{fmult}\ r'\ x'\ y' = \mathsf{fmult}\ 0\ inp_1'\ inp_2' & \text{[apply (8)]} \\ \mathsf{fmult}\ \boxed{(\boxed{r} + x)}^{\uparrow}\ x\ \boxed{(\boxed{y} - 1)}^{\uparrow} = \mathsf{fmult}\ 0\ inp_1\ inp_2 & \text{[apply (13)]} \\ \mathsf{fmult}\ r\ x\ y = \mathsf{fmult}\ 0\ inp_1\ inp_2 & \text{[apply IH]} \end{array}$$

Note that in the application of (13), the pre-condition of the rule holds, by the definition of the corresponding pattern. If $y = 0$, the second match would have succeeded.

5.2 Invariant discovery

The tail-invariant critic, where one particular instantiation is illustrated in Figure 4, can be reused to discover the loop invariant in this example, although the particular rules will deviate. Similar to §4, our first approximation of the invariant is $r = \mathsf{fmult}\ 0\ inp_1\ inp_2$, gives rise to a blocked ripple, i.e:

$$\underbrace{\boxed{\boxed{r} + x}^{\uparrow}}_{\textbf{blocked}} = \mathsf{fmult}\ 0\ inp_1\ inp_2$$

However, the precondition of the tail-invariant patch succeeds, which results in the introduction of a meta-variable F_1:

$$F_1(r, x, y) = \mathsf{fmult}\ 0\ inp_1\ inp_2$$

The definition of the priming operators (8) results in:

$$F_1(\boxed{\boxed{r} + 1}^{\uparrow}, x, \boxed{\boxed{y} - 1}^{\uparrow}) = \mathsf{fmult}\ 0\ inp_1\ inp_2$$

[5]Mechanised case-studies in Isabelle/HOL, which appear in [13], have provided empirical evidence for this approach as well.

Wave-rule (13) then suggests that F_1 is instantiated in terms of fmult, and the same loop invariant as in the previous section is obtained:

$$\mathsf{fmult}\ r\ x\ y = \mathsf{fmult}\ 0\ inp_1\ inp_2$$

6 Relevant work

Our Hume/TLA work build directly on [25, 24], which uses rippling [8] and proof critics [22] to verify imperative programs. Further, we have extended these ideas to transformation proofs [16]. In [16], it is shown how a Hume transformation proof can be reduced to an invariant proof. The work with rippling and critics is novel for Hume/ TLA and transformations, and the generic work with Hume and TLA is novel with respect to using TLA at the programming language level. We believe TLA is more suitable for Hume verification compared to process algebras [4], like CCS, CSP or the Π-calculus.

The work presented here comes out from the first authors PhD thesis [13]. Parts of this work involves a mechanisation of TLA in Isabelle/HOL [31], and a representation of the semantics of Hume programs on top of this. Several proof tactics were developed to automate these proofs. The invariant discovery ideas presented in this paper would obviously increased the degree of automation of these tactics.

Previously, transformation verification has been described [16]. [14] outlines a box calculus for Hume. There, a set of behaviour preserving rules and strategies were defined to transform a Hume program into another. TLA has also been used to model check programs at the lowest, least expressive, HW-Hume level [18], and to reason about different Hume scheduling strategies [15].

Finally, the work presented here is at the action level of TLA, which is similar to Action Systems [3] and Event-B [2]. Thus, we believe this work is also applicable in these formalisations.

7 Future work

This paper has applied rippling and critics to one small example. Extending this to programs of multiple concurrent boxes will not have any impact on the approach, since the invariant will remain invariant due to the strict wire communication. We have not implemented the ideas presented in this paper, and this will be our next step. For imperative programs, the loop invariant generation techniques have already been implemented and tested [25, 24], In the Hume case, Hume/TLA proofs have been mechanised in the Isabelle/HOL theorem prover [13], so via IsaPlanner [11], invariant discovery should not be hard to implement.

A longer term goal is to be able to synthesise transformations. Building directly upon [26], we will now outline how we believe the work described here can be extended to the synthesis of transformations.

In [26], the problem is stated as: given a Hoare triple $\{P\}C\{Q\}$, find an instantiation of C in a small generic imperative program (containing assignment, conditionals and ***while*** loops) such that the triple is valid. The approach combines proof planning with conventional partial order planning. Here, proof planning is used for the local perspective, i.e. finding correct statements, while partial order planning is used to achieve a correct order of statements. This is implemented in a tool called Bertha, which is automatic with the exception that the user has to provide loop invariants.

In Hume, this work would be adapted to automate the synthesis of transformations which are a result of a failed costing. However, an additional requirement here is that the transformations are level-reducing (i.e. it becomes more decidable with respect to time and space properties). To illustrate, consider again the recursion to iteration transformations shown in Figures 5 and 6. The program starts with $prog_1$, and the coster fails on the `emult` box. Next, failure analysis is applied and the problem is that the recursive `fmult` function cannot be costed, which causes a "`recursion_to_iteration` proof method (plan)" to be applied. Now, this uses the expression of the `emult` box to create the specification, i.e.

$$out = \mathsf{fmult}\ 0\ inp_1\ inp_2$$

Moreover, the program knows that a nested box with feedback loop must be created, since this is the only way to represent iteration in Hume. Moreover, we know the program must contain one or more entry matches, loop matches and exit matches, while the loop invariant can be directly found using the techniques described in Section 5.

Partial order planning could then be used to find the correct order of the matches, for example when pattern matching is used, the loop exit (*,*,r,x,0) -> (r,*,*,*) match must precede the loop body (*,*,r,x,y) -> (*,r+x,x,y-1) match. If not, the box will never terminate. Moreover, the loop entry step has different input wires (a,b,*,*,*) than the body and exit steps, while the exit step has different output wires (r,*,*,*). Note that with respect to the box calculus [14] described in the previous section, this approach is more flexible since it is not constrained by an existing set of rules.

TLA, and the full TLA$^+$ specification language [29], which combines TLA with a variant of ZF set theory, has been used both in industry and academia [5, 12, 27]. In [12], Gafni and Lamport illustrate the building of a sufficiently strong invariant by verifying the Disc Paxos algorithm. The algorithm is verified in a bottom-up fashion, where smaller invariants are gradually built up until they are strong enough to verify the main theorem. In a top-down approach the main theorem is first attempted to be verified, and from a partial success, a required invariant is found and verified. This process is iterated until a sufficiently small invariant is found which can be proved directly. Gafni and Lamport argues that both a top-down and bottom-up approach can be used. The top-down approach is a similar approach to the one discussed here. The work described in this paper is exploring Hume programs represented as TLA formulas. Hence, we are only manipulating the TLA terms, and not the Hume code. Thus, we believe that an approach based on rippling and proof planning is applicable to automate the verification of generic TLA invariants, also outside the Hume/TLA context, like in [12].

Another possible role of proof planning is properties involving the $\boldsymbol{\exists}$ operator, which is used to hide internal details of a specification. In this paper, we have ignored the $\boldsymbol{\exists}$ operator, since it was not that relevant for the Hume context. This follows from the fact that $\boldsymbol{\exists}$ is handled in the same way for all programs[6]. In general, to prove a refinement with bound variables in TLA, that is, of the form:

$$(\boldsymbol{\exists}\, B.\ J \wedge \Box[M]_W) \rightarrow (\boldsymbol{\exists}\, A.\ I \wedge \Box[N]_V)$$

one must first remove/instantiate the $\boldsymbol{\exists}$ bound variables A and B. Then, (11) can be applied. This is achieved with proof rules similar to those for standard predicate existential quantification $\exists$. A key step in such a proof, is to find the correct witness for A, known as the *refinement mapping* [1]. Proof planning has previously been used to find complex witnesses for $\exists$ bound variables [9, 30]. We believe that due to the similarity between the rules for $\exists$ and $\boldsymbol{\exists}$, a proof planning approach can also be used to find refinement mappings.

[6]The details will appear in [13]

8 Conclusion

Hume is a Turing-complete programming language, designed to guarantee space and time bounds, whilst working on a high-level. Correctness properties, such as invariants and transformations, have previously been verified using the temporal logic of actions (TLA). Such verification efforts require mathematical induction. Rippling was developed for guiding inductive proofs, and rippling based proof critics have been developed to discovery and generalisations of invariants.

Here, we have shown the use of rippling to both verify and discover invariants, based on existing work on verifying and discovering loop invariants for imperative programs. This work has then been extended to transformations. We believe that the work is also applicable in a generic TLA setting, which we have elaborated upon.

Acknowledgements

This work has been supported by EU FP6 EmBounded.

References

[1] Martín Abadi and Leslie Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, 31 May 1991.

[2] Jean-Raymond Abrial. *Modelling in Event-B: System and Software Engineering.* Cambridge University Press, 2009. To be published.

[3] Ralph-Johan Back. Refinement calculus, part ii: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop*, pages 67–93, London, UK, 1990. Springer-Verlag.

[4] J. C. M. Baeten. A Brief History of Process Algebra. *Theoretical Computer Scisence*, 335(2-3):131–146, 2005.

[5] Brannon Batson and Leslie Lamport. High-Level Specifications: Lessons from Industry. In *Formal Methods for Components and Objects*, number 2852 in Lecture Notes in Computer Science, pages 242–261. Springer, March 17 2003.

[6] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *Proc of CADE'88*, pages 111–120. Springer-Verlag.

[7] A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L. H. Clarke, editor, *Proc. of UK IT 90*, pages 221–6. IEE, 1990.

[8] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling – Meta-level Guidance for Mathematical Reasoning.* Cambridge University Press, 2005.

[9] Alan Bundy, Alan Smaill, and Jane Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L. H. Clarke, editor, *Proc. of UK IT 90*, pages 221–6. IEE, 1990.

[10] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts, 1999.

[11] Lucas Dixon and Jaques Fleuriot. IsaPlanner: A Prototype Proof Planner in Isabelle. In *Proceedings of CADE'03*, volume 2741 of *LNCS*, pages 279–283, 2003.

[12] Eli Gafni and Leslie Lamport. Disk Paxos. *Distributed Computing*, 16(1):1–20, 2003.

[13] Gudmund Grov. *Reasoning about Correctness Properties of a Coordination Programming Language.* PhD thesis, Heriot-Watt University, 2009. January 2009 submission. Subject to oral examination.

[14] Gudmund Grov and Greg Michaelson. Towards a Box Calculus for Hierarchical Hume. In Marco T. Morazan, editor, *Trends in Functional Programming*, volume 8, chapter 5, pages 71 – 88. Intellect, 2007.

[15] Gudmund Grov, Greg Michaelson, and Andrew Ireland. Formal Verification of Concurrent Scheduling Strategies using TLA. In *3rd IEEE International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems*, number CFP07036-USB in IEEE Catalog Number. IEEE, 2007.

[16] Gudmund Grov, Robert Pointon, Greg Michaelson, and Andrew Ireland. Preserving Coordination Properties when Transforming Concurrent System Components. In *Coordination Models, Languages and Applications Track of the 23rd Annual ACM Symposium on Applied Computing*, volume 1 of 3, pages 126 – 127, 1515 Broadway, New York, March 2008. The Association for Computing Machinery, Inc.

[17] Kevin Hammond, Christian Ferdinand, Reinhold Heckmann, Roy Dyckhoff, Martin Hofman, Steffen Jost, Hans-Wolfgang Loidl, Greg Michaelson, Robert Pointon, Norman Scaife, Jocelyn Sérot, and Andy Wallace. Towards Formally Verifiable WCET Analysis for a Functional Programming Language. In *Proceedings of 6th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[18] Kevin Hammond, Gudmund Grov, Greg Michaelson, and Andrew Ireland. Low-level programming in Hume: an exploration of the HW-Hume level. In Zoltán Horváth, Viktória Zsók, and Andrew Butterfield, editors, *In Proceedings of Implementation of Functional Languages (IFL 2006)*, volume 4449 of *Lecture Notes in Computer Science*, pages 91 – 107. Springer, 2006.

[19] Kevin Hammond and Greg Michaelson. Hume: A domain-specific language for real-time embedded systems. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*, volume 2830 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2003.

[20] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[21] A. Ireland. The use of planning critics in mechanizing inductive proofs. In A. Voronkov, editor, *In Proc of LPAR'92*, LNCS 624, pages 178–189. Springer-Verlag, 1992.

[22] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.

[23] A. Ireland, B. J. Ellis, A. Cook, R. Chapman, and J. Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning: Special Issue on Empirically Successful Automated Reasoning*, 36(4):379–410, 2006.

[24] Andrew Ireland and Jamie Stark. On the Automatic Discovery of Loop Invariants. In *The Fourth NASA Langley Formal Methods Workshop*, number 3356. NASA Conference Publication, 1997. Also available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/1.

[25] Andrew Ireland and Jamie Stark. Proof Planning for Strategy Development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):65–97, February 2001.

[26] Andrew Ireland and Jamie Stark. Combining Proof Plans with Partial Order Planning for Imperative Program Synthesis. *Automated Software Engineering Journal*, 13(1):65–105, 2006.

[27] Peter B. Ladkin, Leslie Lamport, Bryan Olivier, and Denis Roegel. Lazy Caching in TLA. *Distributed Computing*, 12(2-3):151–174, 1999.

[28] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[29] Leslie Lamport. *Specifying Systems — The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, Reading, Massachusetts, 2002.

[30] Erica Melis and Jörg Siekmann. Knowledge-based proof planning. 115(1):65–105, 1999.

[31] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[32] N. Storey. *Safety Critical Computer Systems*. Addison-Wesley, 1996.