A Reasoning Infrastructure for the Embedded Systems Language Hume

Hans-Wolfgang Loidl[1] and Gudmund Grov[2]

[1] Ludwig-Maximilians-Universität München,
Institut für Informatik, D 80538 München, Germany, hwloidl@tcs.ifi.lmu.de
[2] School of Mathematics and Computer Science
Heriot-Watt University, EH14 4AS Edinburgh, U.K, G.Grov@hw.ac.uk

Abstract. In this paper we present a formalisation of the embedded systems language Hume in the Isabelle/HOL theorem prover. This formalisation integrates two logics, a VDM-style program logic for the functional sub-language and a TLA-style logic for the coordination sub-language of Hume. We present a soundness proof of the program logic, and demonstrate the usability of the these logics on two example proofs.

1 Introduction

Typically resources on embedded systems, such as memory, are very scarce. Thus, an important property of an embedded program is to execute within these limited resources. The Hume embedded systems language achieves a high degree of predictability of resource consumption by defining two language layers: a strict, higher-order functional language at the expression layer, and a restricted language of interacting boxes at the coordination layer. With this design the full computational power of a modern programming language can be used at the expression layer. Where verification and analysis of an expression become intractable, the code can be decomposed into a network of boxes. This language is far more restricted, but easier to analyse.

The different layers require different formalisms for proving (resource) properties. At the expression layer we apply standard techniques from program verification and develop a VDM-style [15] program logic to reason about programs. At the coordination layer, where we find a network of interacting boxes, we choose the Temporal Logic of Actions (TLA) [17] as a suitable mechanism for reasoning. We use a shallow embedding of the coordination language and of TLA into the Isabelle/HOL theorem prover and combine it with a deep embedding of the expression language and a shallow embedding of its assertion language.

While these two kinds of logics, and their formalisations, have been studied in isolation, this paper makes three contributions in combining them into one *integrated reasoning infrastructure* for a concrete language, in proving soundness for the VDM-style program logic, and in demonstrating the usability of the integrated infrastructure by proving functional correctness and bounded heap consumption for two simple Hume programs.

2 The Hume Language

Hume [10, 12] is designed as a layered language where the *coordination layer* is used to construct reactive systems using a finite-state-automata based notation, representing a static system of interconnecting *boxes*; while the *expression layer* describes the behaviour of a box using a strict, higher-order, purely functional rule-based notation. A central design goal of Hume is predictability of resource consumption, so that each expression-layer program can execute within bounded time and space constraints. Thus, we are mainly interested in proving resource bounds for Hume programs.

The expression layer of Hume corresponds to a strict, higher-order language with general algebraic datatypes. We formalise the expression layer of Hume in the form that is used as an intermediate language in the compiler. This language makes some structural restrictions on the source code, most notably it is in let-normal-form, and uses general algebraic data-types:

$$\begin{aligned}
\text{Patt} \ni\ & p ::= x \mid v \mid c\ x_1 \ldots x_n \mid _ \\
\text{Expr} \ni\ & e ::= x \mid v \mid c\ x_1 \ldots x_n \mid f\ x_1 \ldots x_n \mid x\ x_1 \ldots x_n \mid \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \\
& \quad x_1 \oplus x_2 \mid \texttt{let}\ x\ = e_1\ \texttt{in}\ e_2 \mid \texttt{case}\ x\ \texttt{of}\ p_1 \rightarrow e_1\ \texttt{otherwise}\ e_2
\end{aligned}$$

A pattern p is either a variable $x \in \text{Var}$, a value $v \in \text{Val}$, a constructor application $c\ x_1 \ldots x_n$ ($c \in \text{Constr}$) or a wildcard $\_$. An expression e is either a variable x, a value v, a constructor application $c\ x_1 \ldots x_n$, a first-order function call of f or a higher-order function call of x with arguments $x_1 \ldots x_n$, a conditional, a binary primitive operation $\oplus$, a let-expression, or a case expression. The latter is a one-step matching expression, with a default branch to be used if the match was unsuccessful. The entire program is represented by a table funTab, which maps a function name (f) to its formal arguments ($args_f$) and to its body ($body_f$).

On coordination layer, Hume programs comprise a set of concurrent, asynchronous boxes, scheduled in an alternating sequence of execute and super-step phases. In the execute phase any box with sufficient available inputs will be executed. In the following super-step phase, the results of the boxes will be copied to single-buffer wires, provided they are free. Boxes are defined as a series of pattern-matching rules, of the form $p \rightarrow e$, using the syntax of the expression language above. The semantics of these rules is similar to the one of expression-layer pattern matching, but additionally accounts for possibly missing input values.

As an example, the Hume code to the right shows a box that takes a 16-bit integer as input value, and produces as output value the input value increased by one.

```
box inc
in  ( i  :: int 16 )
out ( i' :: int 16 )
match  i -> i+1;
```

In general, the body of a box is comprised of a sequence of rules, with the left hand side being matched against the input data, and first successful match enabling execution of its right hand side. Networks of boxes on the Hume coordination layer are best presented in figures, like the examples in Figure 5.

$$
\begin{array}{lll}
l, a, r \in \text{Locn} & = \mathbb{N} \uplus \{\text{nil}\} & E, E' \in \text{Env} = \text{Var} \Rightarrow \text{Val} \\
h, h' \in \text{Heap} & = \text{Locn} \rightharpoonup_f \text{Val} & G \in \text{Ctxt} = \mathcal{P}\,\{(\text{Expr}, \text{Assn})\} \\
v \in \text{Val} & = \{\bot\} \uplus \mathbb{Z} \uplus \mathbb{B} \uplus \text{Locn} \uplus (\text{Constr}, \text{Locn}^*) \uplus (\text{Funs}, \mathbb{N}, \text{Locn}^*) & \\
P, Q, A \in \text{Assn} & = \text{Env} \Rightarrow \text{Heap} \Rightarrow \text{Heap} \Rightarrow \text{Val} \Rightarrow \text{Resources} \Rightarrow \mathbb{B} & \\
p \in \text{Resources} & = (\mathit{clock} :: \mathbb{Z},\ \mathit{callc} :: \mathbb{Z}, \mathit{maxstack} :: \mathbb{Z}) &
\end{array}
$$

Fig. 1. Basic Domains

3 A VDM-style Program Logic for the Expression Layer

3.1 Operational Semantics

Figure 1 summarises the domains used in the formalisation. The sets of identifiers for variables ($x \in \text{Var}$), functions ($f \in \text{Funs}$), and data constructors ($c \in \text{Constr}$) are disjoint. We represent heaps ($h \in \text{Heap}$) by finite mappings from locations to values (written $\rightharpoonup_f$), and environments ($E \in \text{Env}$) by total functions of variables to values (written $\Rightarrow$). A location $l \in \text{Locn}$ is either a natural number (representing an address in the heap), or the constant nil. We write Locn^* for the domain of sequences of elements in Locn, $E \star \vec{xs}$ for mapping E over the sequence $\vec{xs}$, prefix $\#$ for the length of a sequence and $+\!\!+$ for append of sequences. We model heap usage of h as the size of its domain, ie. $|\,\text{dom}\ h\,|$. Other resources, such as time, are modelled in the resource vector $p \in \text{Resources}$ and $\smile$ combines these. We obtain costs for basic operations from a parameterised table $\mathcal{R}$. The operation freshloc has the property: freshloc $s \notin s$, for all s.

A judgement of the big-step operational semantics, shown in Figure 2, has the form $E, h \vdash e \Downarrow_m (v, h', p)$ and is read as follows: given a variable environment, E, and a heap, h, e evaluates in m steps to the value v, yielding the modified heap, h', and consumes resources p. A value v is itself the result (Value). For a variable x a lookup is performed in E (Var). For a constructor application $c\,\vec{xs}$ the heap is extended with a fresh location, mapped to c and its argument values (Constr). For primitive operations a shallow embedding is used so that the operator $\oplus$ can be directly applied to the values of its variables (PrimBin). A conditional returns the value of the then branch e_1 or the else branch e_2, depending on the value of x. A let binds the result of e_1 to the variable x and returns the value of the body e_2 (Let). The judgement of the pattern-matching semantics, $\texttt{MATCH}\ E, h \vdash\ p\ \texttt{at}\ l \Downarrow (E', l')$, matches the value at location l against pattern p, returning a modified environment and l on success, or the same environment and $\bot$ on failure. A case is a one-step matching expression, that matches a variable x against a pattern p; if successful its value is that of e_1 (CaseTrue), if unsuccessful its value is that of e_2 (CaseFalse). In first-order function calls, we bind the values in the arguments to the formal parameters (args_f), and use the modified environment to evaluate the body (body_f) of the function f (CallFun). For higher-order function calls, with function closures represented as triples $(f, i, \vec{rs})$ of function name, number of arguments and their values, we have to distinguish two cases. If we have fewer than the required number of arguments, the result will be a

$$\frac{}{E, h \vdash v \Downarrow_1 (v, h, \mathcal{R}^{\texttt{konst}})}\ \textsc{Value} \qquad \frac{E\ x = v}{E, h \vdash x \Downarrow_1 (v, h, \mathcal{R}^{\texttt{var}})}\ \textsc{Var}$$

$$\frac{\begin{array}{c} l = \text{freshloc (dom } h) \\ \vec{rs} = E \star \vec{xs} \qquad h' = h(l \mapsto (c,\ \vec{rs})) \end{array}}{E, h \vdash c\ \vec{xs} \Downarrow_1 (l, h', \mathcal{R}^{\texttt{constr}\ \#\vec{xs}})}\ \textsc{Constr} \qquad \frac{\begin{array}{c} v_1 = E\ x_1 \qquad v_2 = E\ x_2 \\ v = v_1 \oplus v_2 \end{array}}{E, h \vdash x_1 \oplus x_2 \Downarrow_1 (v, h, \mathcal{R}^{\oplus})}\ \textsc{PrimBin}$$

$$\frac{E\ x = \text{true} \qquad E, h \vdash e_1 \Downarrow_n (v, h', p')}{E, h \vdash \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow_{n+1} (v, h', p' \frown \mathcal{R}^{\texttt{if true}})}\ \textsc{IfTrue}$$

$$\frac{E\ x = \text{false} \qquad E, h \vdash e_2 \Downarrow_n (v, h', p')}{E, h \vdash \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow_{n+1} (v, h', p' \frown \mathcal{R}^{\texttt{if false}})}\ \textsc{IfFalse}$$

$$\frac{E, h \vdash e_1 \Downarrow_m (v, h', p') \qquad E(x := v), h' \vdash e_2 \Downarrow_n (v'', h'', p'')}{E, h \vdash \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 \Downarrow_{m+n} (v'', h'', p' \frown p'' \frown \mathcal{R}^{\texttt{let}})}\ (\textsc{Let})$$

$$\frac{l = E\ x \qquad \texttt{MATCH}\ E, h \vdash p\ \texttt{at}\ l \Downarrow (E', v') \qquad v' \neq \bot \qquad E', h \vdash e_1 \Downarrow_n (v, h', p')}{E, h \vdash \texttt{case}\ x\ \texttt{of}\ p \to e_1\ \texttt{otherwise}\ e_2 \Downarrow_{n+1} (v, h', p' \frown \mathcal{R}^{\texttt{case true}})}\ (\textsc{CaseTrue})$$

$$\frac{l = E\ x \qquad \texttt{MATCH}\ E, h \vdash p\ \texttt{at}\ l \Downarrow (E', v') \qquad v' = \bot \qquad E, h \vdash e_2 \Downarrow_n (v, h', p')}{E, h \vdash \texttt{case}\ x\ \texttt{of}\ p \to e_1\ \texttt{otherwise}\ e_2 \Downarrow_{n+1} (v, h', p' \frown \mathcal{R}^{\texttt{case false}})}\ (\textsc{CaseFalse})$$

$$\frac{E' = E(args_f := E \star \vec{xs}) \qquad E', h \vdash body_f \Downarrow_n (v, h', p)}{E, h \vdash f\ \vec{xs} \Downarrow_{n+1} (v, h', p \frown \mathcal{R}^{f\ \#\vec{xs}})}\ (\textsc{CallFun})$$

$$\frac{\begin{array}{c} l = E\ x \qquad \text{Some } (f,\ i,\ \vec{rs}) = h\ l \qquad i + \#\vec{xs} < \#args_f \\ l' = \text{freshloc (dom } h) \qquad h' = h(l' \mapsto (f,\ (i + \#\vec{xs}),\ (\vec{rs} +\!\!+ (E \star \vec{xs})))) \end{array}}{E, h \vdash x\ \vec{xs} \Downarrow_1 (l', h', \mathcal{R}^{\texttt{ap false}\ \#\vec{xs}})}$$
$$(\textsc{CallVarUnderApp})$$

$$\frac{\begin{array}{c} l = E\ x \qquad \text{Some } (f,\ i,\ \vec{rs}) = h\ l \qquad i + \#\vec{xs} = \#args_f \\ E' = E(args_f := \vec{rs} +\!\!+ (E \star \vec{xs})) \qquad E', h \vdash body_f \Downarrow_n (v, h', p) \end{array}}{E, h \vdash x\ \vec{xs} \Downarrow_{n+1} (v, h', p \frown \mathcal{R}^{\texttt{ap true}\ \#\vec{xs}})}$$
$$(\textsc{CallVarExact})$$

Fig. 2. Operational Semantics of Expression-layer Hume

new closure consisting of f, the number of arguments supplied so far and their values. If we have precisely the number of arguments that the function requires, the values in the closure and the those of the arguments ($\vec{xs}$) are bound to the formal parameters ($args_f$) of the function, before the function body is evaluated using these new bindings (CallVarExact).

3.2 Program Logic

A judgement has the form $G \rhd e : P$, meaning that expression e fulfills the assertion P in a context G. As shown in Figure 1, an *assertion* is a predicate over the components of the operational semantics, and a context is a set of pairs of program expression $e \in$ Expr and assertion $P \in$ Assn.

The rules in Figure 3 define the program logic for expression-layer Hume. A value w is the result value, and the heap is unchanged (vdmValue). A variable x requires a lookup in the environment, and the heap is unchanged (vdmVar). In a constructor application, the newly allocated location is existentially quantified, and the heap is updated with a binding to this location (vdmConstr). The two possible control flows in the conditional are encoded in the logic as implications, based on the boolean contents of variable x (vdmIf). In the let rule (vdmLet), the intermediate value, heap and resources are existentially quantified, and the environment for executing e_2 is updated accordingly. The control flows in the case construct are encoded as implications, based on the result from the pattern match (vdmCase). In a first order call (vdmCallFun), the assertion to be proven for f is added with the call into the context, and the function body has to fulfill the assertion, after modifying environment and resources. The higher-order function call (vdmCallVar) directly encodes the preconditions of the two cases of under application (Φ) and of exact application (Ψ) from the operational semantics. Thus, the definition of Φ reads as follows: for the pre-state E, h we find in variable x a closure with i arguments, and the total number of arguments is smaller than the function's arity ($i + \#\vec{xs} < \#args_f$). In this case, the result state v, h', p is constructed such that the result value is a new closure, containing all arguments, and the result heap is updated accordingly. The definition of Ψ reads as follows: For the pre-state E', h we find in variable x a closure with i arguments, and the total number of arguments matches the function's arity ($i + \#\vec{xs} = \#args_f$). In this case, the second premise in rule vdmCallVar demands that the body of f ($body_f$) fulfils the assertion P, with the environment adjusted for parameter passing and a modified resource vector to account for the costs of the function call. Note that Ψ has to be parameterised over the function f and the argument values $\vec{rs}$ captured in its closure, so that the vdmCallVar rule can quantify over f and $\vec{rs}$. In particular, the quantification over f scopes over the entire second judgement in the premise of vdmCallVar, because it is needed to retrieve the function's body via $body_f$. Finally, the quantification over $\vec{rs}$ scopes over the entire pre-condition inside the assertion. The environment E is constructed out of E' by binding the formal parameters to the values in the closure ($\vec{rs}$) and those retrieved from the arguments $\vec{xs}$. The final two rules,

$$\frac{}{G \ \triangleright \ w \ : \ \lambda E\ h\ h'\ v\ p.\ v = w \ \wedge \ h = h' \ \wedge \ p = \mathcal{R}^{\texttt{konst}}} \quad (\textsc{vdmValue})$$

$$\frac{}{G \ \triangleright \ x \ : \ \lambda E\ h\ h'\ v\ p.\ v = E\ x \ \wedge \ h = h' \ \wedge \ p = \mathcal{R}^{\texttt{var}}} \quad (\textsc{vdmVar})$$

$$\frac{}{\begin{array}{l} G \ \triangleright \ c\ \vec{xs} \ : \ \lambda E\ h\ h'\ v\ p.\ \exists l\ \vec{rs}. \\ \qquad l = \text{freshloc}(\text{dom}\ h) \ \wedge \ \vec{rs} = E \star \vec{xs} \ \wedge \ v = l \ \wedge \ h' = h(l \mapsto (c,\ \vec{rs})) \ \wedge p = \mathcal{R}^{\texttt{constr}\ \#\vec{xs}} \end{array}} \quad (\textsc{vdmConstr})$$

$$\frac{}{G \ \triangleright \ x_1 \oplus x_2 \ : \lambda E\ h\ h'\ v\ p.\ \exists v_1\ v_2.\ E\ x_1 = v_1 \ \wedge \ E\ x_2 = v_2 \ \wedge \ v = v_1 \oplus v_2 \ \wedge \ h = h' \ \wedge \ p = \mathcal{R}^{\oplus}} \quad (\textsc{vdmPrimBin})$$

$$\frac{G \ \triangleright \ e_1 \ : P_1 \qquad G \ \triangleright \ e_2 \ : \ P_2}{\begin{array}{c} G \ \triangleright \ \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \ : \ \lambda E\ h\ h'\ v\ p. \\ ((E\ x = \text{true} \ \longrightarrow \ \exists p'.\ P_1\ E\ h\ h'\ v\ p' \ \wedge \ p = p' \frown \mathcal{R}^{\texttt{if true}}) \wedge \\ (E\ x = \text{false} \ \longrightarrow \ \exists p'.\ P_2\ E\ h\ h'\ v\ p' \ \wedge \ p = p' \frown \mathcal{R}^{\texttt{if false}})) \end{array}} \quad (\textsc{vdmIf})$$

$$\frac{G \ \triangleright \ e_1 \ : P_1 \qquad G \ \triangleright \ e_2 \ : \ P_2}{\begin{array}{c} G \ \triangleright \texttt{let}\ x \ = \ e_1\ \texttt{in}\ e_2 \ : \ \lambda E\ h\ h'\ v\ p.\ \exists v'\ h''\ p'\ p''. \\ (P_1\ E\ h\ h''\ v'\ p' \ \wedge \ P_2\ E(x := v')\ h''\ h'\ v\ p'' \ \wedge \ p = p' \frown p'' \frown \mathcal{R}^{\texttt{let}}) \end{array}} \quad (\textsc{vdmLet})$$

$$\frac{G \ \triangleright \ e_1 \ : \ P_1 \qquad G \ \triangleright \ e_2 \ : \ P_2}{\begin{array}{c} G \ \triangleright \ \texttt{case}\ x\ \texttt{of}\ p_1 \rightarrow e_1\ \texttt{otherwise}\ e_2 \ : \ \lambda E\ h\ h'\ v\ p.\ \forall l.\ l = E\ x \ \longrightarrow \ \forall E'\ v'. \\ \texttt{MATCH}\ E,\ h \vdash \ p_1\ \texttt{at}\ l \ \Downarrow \ (E', v') \ \longrightarrow \\ ((v' = \bot \ \longrightarrow \ \exists p'.\ P_2\ E\ h\ h'\ v\ p' \ \wedge \ p = p' \frown \mathcal{R}^{\texttt{case false}}) \ \wedge \\ (v' \neq \bot \ \longrightarrow \ \exists p'.\ P_1\ E'\ h\ h'\ v\ p' \ \wedge \ p = p' \frown \mathcal{R}^{\texttt{case true}})) \end{array}} \quad (\textsc{vdmCase})$$

$$\frac{\{(f\ \vec{xs}, P)\} \cup G \ \triangleright \ \mathit{body}_f \ : \ \lambda E\ h\ h'\ v\ p.\ \forall E'.\ E = E'(\mathit{args}_f := E' \star \vec{xs}) \longrightarrow P\ E'\ h\ h'\ v\ (p \frown \mathcal{R}^{f\ \#\vec{xs}})}{G \ \triangleright \ f\ \vec{xs} \ : \ P} \quad (\textsc{vdmCallFun})$$

$$\frac{\begin{array}{c} \forall E\ h\ h'\ v.\ \Phi_{x,\vec{xs}}\ E\ h\ h'\ v \longrightarrow \ P\ E\ h\ h'\ v\ \mathcal{R}^{\texttt{ap false}\ \#\vec{xs}} \\ (\forall f.\ \{(x\ \vec{xs}, P)\} \cup G \ \triangleright \ \mathit{body}_f \ : \ \lambda E\ h\ h'\ v\ p. \\ \forall E'.\ (\exists\ \vec{rs}.\ \Psi_{x,\vec{xs}}\ E'\ h\ h'\ v\ f\ \vec{rs} \ \wedge E = E'(\mathit{args}_f := \vec{rs} +\!\!+ (E' \star \vec{xs})) \ \longrightarrow \ P\ E'\ h\ h'\ v\ (p \frown \mathcal{R}^{\texttt{ap true}\ \#\vec{xs}})) \end{array}}{G \ \triangleright \ x\ \vec{xs} \ : \ P} \quad (\textsc{vdmCallVar})$$

$$\frac{(e, P) \in G}{G \ \triangleright \ e \ : \ P}\ \textsc{vdmAx} \qquad \frac{\forall E\ h\ h'\ v\ p.\ P\ E\ h\ h'\ v\ p \ \longrightarrow \ Q\ E\ h\ h'\ v\ p \qquad G \ \triangleright \ e \ : \ P}{G \ \triangleright \ e \ : \ Q}\ \textsc{vdmConseq}$$

$$\begin{array}{l} \Phi_{x,\vec{xs}} \equiv \lambda E\ h\ h'\ v.\ \exists l\ l'\ f\ i\ \vec{rs}.\ E\ x = l \ \wedge \ h\ l = \text{Some}(f,\ i,\ \vec{rs}) \ \wedge \ i + \#\vec{xs} < \#\mathit{args}_f \ \wedge \\ \qquad\qquad l' = \text{freshloc}\ (\text{dom}\ h) \ \wedge \ v = l' \ \wedge \ h' = h(l' \mapsto (f,\ (i + \#\vec{xs}),\ (\vec{rs} +\!\!+ (E \star \vec{xs})))) \\ \Psi_{x,\vec{xs}} \equiv \lambda E'\ h\ h'\ v\ f\ \vec{rs}.\ \exists l\ i\ .\ E'\ x = l \ \wedge \ h\ l = \text{Some}\ (f,\ i,\ \vec{rs}) \ \wedge \ i + \#\vec{xs} = \#\mathit{args}_f \end{array}$$

Fig. 3. Program Logic for Expression-layer Hume

VDMAx and VDMConseq, are the standard rules for using an axiom, present in the context, and for logical consequence in the meta-language.

3.3 Soundness

We define (relativised) semantic validity of an assertion for an expression in a context as follows.

Definition 1 (validity). *Assertion P is valid for expression e ($\models_n e : P$) iff*

$$\forall m \leq n.\ \forall E\ h\ h'\ v\ p.\ (E,\ h\ \vdash\ e\ \Downarrow_m\ (v, h', p))\ \longrightarrow\ P\ E\ h\ h'\ v\ p$$

Assertion P is valid for expression e in context G ($G \models e : P$) iff

$$\forall n.\ ((\forall (e', P')\ \in\ G.\ \models_n\ e'\ :\ P')\ \longrightarrow \models_n\ e\ :\ P)$$

Based on these simple definitions of validity, exploiting the shallow nature of our embedding of assertions, the soundness theorem can be stated as follows.

Theorem 1 (soundness). *For all contexts G, expressions e, assertions P*

$$G \rhd e : P \Longrightarrow G \models e : P$$

Proof structure: By induction over the rules of the program logic. In the case of function calls, induction over the index n in the semantics relation. □

Our approach to proving *mutually recursive functions*, building on [1], is to define a predicate *goodContext*. In the first-order case, it requires context elements to be function calls associated to entries in the function specification table $\mathcal{F}$, and, informally, the context must be powerful enough to prove all of its assertions. This means that it has to encode information about all functions called in the body of the function under consideration. Note the universal quantification over the arguments $\vec{ys}$ of the function. This allows us to adapt the function arguments to the concrete values provided at the call site.

Definition 2. *A context G is called a good context w.r.t. specification table $\mathcal{F}$, written goodContext $\mathcal{F}$ G, iff*

$$\begin{array}{l}(\forall e\ P\ .\ (e, P) \in G\ \longrightarrow \\ \quad (\exists f\ \vec{xs}.\ e = f\ \vec{xs}\ \wedge\ P = \mathcal{F}\ f\ \vec{xs}\ \wedge \\ \quad\quad (\forall \vec{ys}.\ G \rhd body_f\ : \\ \quad\quad\quad \lambda E\ h\ h'\ v\ p.\ \forall E'.\ E = E'(args_f := E' \star \vec{ys})\ \longrightarrow \\ \quad\quad\quad\quad (\mathcal{F}\ f\ \vec{ys})\ E'\ h\ h'\ v\ (p \smile \mathcal{R}^{f\ \#\vec{ys}})))\end{array}$$

Figure 4 shows a set of *admissible rules* that are useful in proving concrete program properties. The rules CTXTWeak and CUT are proven by induction over the derivation of $G \rhd\ e : P$. Rule MUTREC is proven by induction over the size of G. Finally, VDMAdapt, which is used when proving a property on a function call, follows directly from MUTREC. Note that VDMAdapt reduces the proof to one of the above good context predicate that has to be constructed for the function under consideration. Thus, proving *goodContext* $\mathcal{F}$ G, for a specially constructed context, becomes the main step in proving a property for (mutually) recursive functions. Section 5.2 gives an example.

$$\frac{G \ \rhd \ e : P}{G \cup G' \ \rhd \ e : P} \ \textsc{ctxtWeak} \qquad \frac{G \ \rhd \ e' : P' \qquad (\{(e', P')\} \cup G) \ \rhd \ e : P}{G \rhd \ e : P} \ \textsc{cut}$$

$$\frac{\text{finite } G \qquad |\ G \mid = n \qquad \mathit{goodContext}\ \mathcal{F}\ G \qquad (e, P) \in G}{\emptyset \ \rhd \ e : P} \quad (\textsc{mutrec})$$

$$\frac{\mathit{goodContext}\ \mathcal{F}\ G \qquad \text{finite } G \qquad (f\ \vec{xs}, \mathcal{F}\ f\ \vec{xs}) \ \in G}{\emptyset \ \rhd \ f\ \vec{ys} \ : \ \mathcal{F}\ f\ \vec{ys}} \quad (\textsc{vdmAdapt})$$

Fig. 4. Admissible Rules

4 Incorporating the Coordination Layer using TLA

The Temporal Logic of Actions (TLA) [17] was developed to reason about concurrent systems, and can capture both safety and liveness properties in the same uniform logic. We have previously formalised the Hume coordination layer, embedded in TLA, in Isabelle/HOL [8], and found it suitable for proving properties [11, 9]. In this work we focus on the VDM-TLA integration, thus following the "state-behaviour integration" approach (see Section 6). TLA combines temporal logic with actions, thus creating three tiers: a *state level*, where a state is a mapping from variables to values; an *action level* over two states; and a *temporal level* over an infinite sequence of states. All levels include a full predicate calculus. The action level has an additional priming (') operator, which separates variables in the "result" state (primed) from those of the "before" state. At this level, "before" variable x and its "result" counterpart x', are distinct.

On the temporal level, a TLA embedding of a Hume program has the form $I \wedge \Box[\mathcal{N}]_v$, where I is a state predicate defining the initial state, $\mathcal{N}$ is an action which defines the computation, and v is a tuple containing all program and meta variables. An invariant P, which is the focus here, is prefixed by the temporal always operator: $\Box P$. To show that an invariant 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. Moreover, temporal formulas are always attempted to be reduced to the simpler action level. For example, Inv reduces a temporal invariant proof to the action level, where P' is the invariant over the "result" state:

$$\frac{I \longrightarrow P \qquad P \wedge \mathcal{N} \longrightarrow P' \qquad P \wedge v' = v \longrightarrow P'}{I \wedge \Box[\mathcal{N}]_v \longrightarrow \Box P} \quad (\textsc{Inv})$$

TLA is mechanised in Isabelle/HOL using possible world semantics [8]. This embedding is shallow, which enables direct use of existing Isabelle/HOL tools and theorems. Moreover, the Hume embedding is also shallow, which simplifies the expression layer integration.

Hume boxes are executed in a two-step lock step algorithm [9]: in the first execute phase all runnable boxes are executed sequentially; this is followed by a super-step, which copies box outputs to wires. This is embedded by two (enumeration) meta-variables: a scheduler s and a program counter pc.

The scheduling is formalised by an action S, which updates the scheduling variable s. The scheduler depends on whether pc' equals the (defined) "last box" in the enumeration type.

Definition 3. *By giving the value of this "last box" predicate as argument, the scheduler is defined as follows:*

$$S\ last\_box \equiv s' = (\texttt{if}\ s = Execute \wedge last\_box\ \texttt{then}\ Super\ \texttt{else}\ \texttt{Execute})$$

Each Hume wire is represented by a variable, and to achieve the lock-step scheduling, each box has one result buffer variable for each output. This is illustrated in Figure 5, which shows the examples discussed in Section 5. In addition, each box has a state/mode variable (e.g. *even_st*) of the enumeration type consisting of *Runnable*, *Blocked*, and *Matchfail*: a *Runnable* box can be executed; *Matchfail* denotes that the box has failed in matching the inputs and cannot be executed; a *Blocked* box has failed asserting the output buffer with the output wires, and cannot be executed. All the result buffers and wires are of type Val. In addition each program has one heap $h \in$ Heap and a $p \in$ Resources which specifies resource properties. Every box has one action for each phase.

In the super-step, box outputs are checked, and if succeeded the result buffer is copied to the wires. For heap structures, this is a shallow copy of the references on the stack, thus the heap is left unchanged. In the execute phase, only the current box (identified by pc) is executed in a given step. Moreover, it is only executed when it is not *Blocked*. In a box execution, there is first a check that there is a match that will succeed (a box may not be total). If this check succeeds, $E,\ h \vdash e \Downarrow (v, h', p)$ is used to represent the execution of the expression layer e, under the environment E, with a correct variable (Var) to value (Val) mapping and the current heap h.

To enable usage of the inductively defined judgement $E,\ h \vdash e \Downarrow (v, h', p)$ within the TLA-embedded expression layer, it must be seen as a function, accepting the old heap h, environment E and expression e as input, and returning the new heap h', the return value v and resources used p.

Definition 4. *We define the* exe *function as follows:*

$$((v, h', p) = exe\ E\ h\ e) \equiv (E,\ h \vdash e \Downarrow (v, h', p))$$

The *exe* function is then used to update the result buffer, heap h and resource variable p in the TLA representation. The proof of TLA invariants of the coordination layer relies on pre-post conditions of the *exe* function. To achieve this, the *exe* function must be linked with the the VDM logic for proofs of properties of the form $E,\ h \vdash e \Downarrow (v, h', p)$. This is achieved by the following theorem, which is the key for the integration between the TLA and VDM logic:

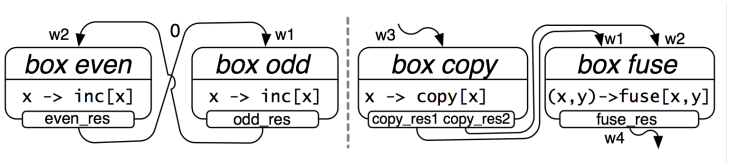

Fig. 5. Box Diagrams of EvenOdd and ListCopy Examples

Theorem 2 (vdmexe). $[\![((v, h', p) = exe\ E\ h\ e); \emptyset \rhd e : A]\!] \Longrightarrow A\ E\ h\ h'\ v\ p$

Proof. The assumption applied to the *soundness* theorem gives $\emptyset \vDash e : A$. By the definition of *validity in context*, this gives $|\vDash \emptyset \longrightarrow (\vDash e : A)$ which gives (G1) $\vDash e : A$. Moreover, the definition of *exe* and the assumption of *vdmexe*, implies $E, h \vdash e \Downarrow (v, h', p)$. The goal then follows from unfolding (G1), by using the definition of *validity*, followed by an application of it. □

This integration works directly due to the shallow TLA embedding and the shallow assertion language for a VDM property. VDM properties are then explored within TLA, since the *vdmexe* theorem turns the VDM property into a HOL predicate, enabling the VDM proof to be independent of TLA. We will now illustrate how the VDM and TLA logics are combined in the proof of a Hume invariant $\Box P$. This approach will be known as the "*standard Hume/TLA invariant structure*." Firstly, the INV rule above is applied. The base (initial state) $I \Rightarrow P$ and "unchanged" $P \wedge v' = v \Rightarrow P'$ cases are handled by the Isabelle/HOL simplifier, and not discussed further.

In the main $P \wedge \mathcal{N} \Rightarrow P'$ case, $\mathcal{N}$ is a conjunction of the scheduler S and all the box actions. It starts by case-analysis on s, creating two cases. The first case, $s = Super$ is achieved purely within the TLA logic. The second case, $s = Execute$ is followed by a case analysis on *pc*. If P' depends on the result of execution *pc*, then it is followed by a case-split on $pc\_st = Blocked$ (the box state/mode variable). Let e be the expression layer of this box. If the check for a match succeeds, then the proof of P' depends on a sufficiently strong A such that $\emptyset \rhd e : A$ can be proved by the VDM-logic. The *vdmexe* theorem is applied to enable the use of A in the main TLA proof of P'.

5 Examples

5.1 Example: Even-Odd

Our first example is the even-odd program depicted in Figure 5 (left). It is a closed network with two boxes, each incrementing its argument by one. Our goal is to show that one wire can only contain an even number, while the other wire can only contain an odd number. The variables have the same names as in Figure 5. The execution order of the boxes is defined to be *peven* then *podd*,

followed by the super-step phase *Super*. In the execute phase, *pc* is updated by the "active box." The box action *even_exe* for the execute phase of *even* is defined as follows:

$$
\begin{array}{l}
\texttt{if } pc \neq peven \texttt{ then } Unchanged(w2, even\_res, even\_st) \\
\texttt{else } pc' = podd \wedge \\
\quad \texttt{if } even\_st = Blocked \texttt{ then } Unchanged(w2, even\_res, even\_st, h, r) \\
\quad \texttt{else if } w2 = \bot \texttt{ then } Unchanged(w2, h, r) \wedge \\
\qquad\qquad even\_res' = \bot \wedge even\_st' = Matchfail \\
\qquad \texttt{else } w2' = \bot \ \wedge \ even\_st' = Runnable \\
\qquad\quad (even\_res', h', r') = exe\ \emptyset(x := w2)\ h\ (inc\ [x])
\end{array}
$$

where $Unchanged\ x \equiv x' = x$ and *inc* is the expression layer program:

$$\text{inc } [x] \equiv \texttt{case } x \texttt{ of } y \rightarrow\ (\lambda x\ y.x + 1)\ y\ y \quad \texttt{otherwise } x$$

The box action *odd_exe* for *odd* is defined similarly. To specify the exchange of data between boxes, the super-step action *even_sup* is defined as follows

$$
\begin{array}{l}
\texttt{if } ao\ even\_res\ w1 \\
\texttt{then } even\_st' = Runnable \ \wedge \ w1' = nw\ even\_res\ w1 \ \wedge \ even\_res' = \bot \\
\texttt{else } even\_st' = Blocked \ \wedge \ Unchanged(w1, even\_res)
\end{array}
$$

where $ao\ A\ B \ \equiv \ A = \bot \vee B = \bot$ and $nw\ A\ B \ \equiv \texttt{if } A = \bot \texttt{ then } B \texttt{ else } A$ ($\bot$ denotes unavailability of data). The super-step action *odd_sup* has a similar definition. The "next" action $\mathcal{N}$ is defined as:

$$S \wedge (s = Execute \longrightarrow even\_exe \wedge odd\_exe) \wedge (s = Super \longrightarrow even\_sup \wedge odd\_sup)$$

In the initial state I, $w1 = I\ 0$, while all other values are $\bot$, and all boxes are *Runnable*, $s = Execute$ and $pc = peven$.

To verify the property we first define even and odd numbers as mutually inductive sets *Even* and *Odd*.

$$\frac{}{0 \in Even}(\mathsf{base}) \qquad \frac{n \in Odd}{n+1 \in Even}(\mathsf{step1}) \qquad \frac{n \in Even}{n+1 \in Odd}(\mathsf{step2})$$

From the expression layer, we require the following VDM-property.

Lemma 1 (even-odd1).

$$\emptyset \ \rhd \ inc\ [x] \ : (\lambda E\ h\ h'\ v\ p.\ (\forall i.\ (E\ x = (I\ i)) \ \longrightarrow \ v = (I\ (i+1))))$$

Proof structure. The proof is structure directed, and standard simplification is sufficient to solve the remaining verification conditions. □

The first invariant to verify is a typing invariant stating that all wires and buffers are integers when they are not empty.

Lemma 2 (even-odd2).

$$\Box \begin{pmatrix} ((\exists\ i.\ w1 = (I\ i)) \vee w1 = \bot) \wedge ((\exists\ i.\ even\_res = (I\ i)) \vee even\_res = \bot) \\ \wedge\ ((\exists.\ i\ w2 = (I\ i)) \vee w2 = \bot) \wedge ((\exists\ i.\ odd\_res = (I\ i)) \ \vee\ odd\_res = \bot) \end{pmatrix}$$

Proof structure. The proof follows the "standard Hume/TLA invariant structure." It follows from INV, the *even-odd1* lemma, and variable instantiations. □

Due to the closed wiring between the two boxes, there is a circular dependency between the wires and result buffers. These must be proven at the same time.

Lemma 3 (even-odd3).

$$\Box\left(\begin{array}{l}(\forall\ i.\ w1 = (I\ i) \longrightarrow\ w1 \in \mathit{Even}) \wedge (\forall\ i.\ \mathit{even\_res} = (I\ i) \longrightarrow\ \mathit{even\_res} \in \mathit{Even}) \\ \wedge\ (\forall.\ i\ w2 = (I\ i) \longrightarrow\ w2 \in \mathit{Odd}) \wedge (\forall\ i.\ \mathit{odd\_res} = (I\ i) \longrightarrow\ \mathit{odd\_res} \in \mathit{Odd})\end{array}\right)$$

Proof. Using the "standard Hume/TLA invariant structure," it follows from INV, *even-odd1* and *even-odd2*, and the inductive definitions of *Even* and *Odd*. □

Theorem 3 (even-odd).

$$\Box\big((\forall\ i.\ w_1 = (I\ i) \longrightarrow i \in \mathit{Even})\ \wedge\ (\forall\ i.\ w_2 = (I\ i) \longrightarrow i \in \mathit{Odd})\big)$$

Proof. The proof follows directly from *even-odd3* using standard TLA reasoning.

5.2 Example: List-Copy

We are now proving a resource property over the heap. The program is shown in Figure 5 (right). The *copy* box performs a list-copy over its input list, producing two identical lists; the *fuse* box takes these two lists and combines them into one list, using only the smaller of the two elements from the input lists.

$$\begin{array}{l}\mathit{copy}\ [x]\ \equiv \texttt{case}\ x\ \texttt{of} \\ \quad (\text{CONS}\ [h,t]) \rightarrow\ \texttt{let}\ y\ =\ \mathit{copy}\ [t]\ \texttt{in}\ \text{CONS}\ [h,y]\quad \texttt{otherwise}\quad \text{NIL}\ [] \\ \mathit{fuse}\ [x_0,x_1] \equiv \texttt{case}\ x_0\ \texttt{of}\ (\text{CONS}\ [h_0,t_0]) \rightarrow\ \texttt{case}\ x_1\ \texttt{of}\ \ (\text{CONS}\ [h_1,t_1]) \rightarrow \\ \quad \texttt{let}\ y\ = \mathit{fuse}\ [t_0,t_1]\ \texttt{in let}\ b\ = (h_0 < h_1)\ \texttt{in} \\ \quad\ \ \texttt{if}\ b\ \texttt{then}\ \text{CONS}\ [h_0,y]\ \ \texttt{else}\ \text{CONS}\ [h_1,y] \\ \quad \texttt{otherwise}\quad \text{NIL}\ []\quad \texttt{otherwise}\quad \text{NIL}\ []\end{array}$$

The TLA embedding of this program follows the same structure as the Even-Odd example. In this case, the *fuse* box accepts two inputs (x, y), binding them in the environment, and the *copy* box returns two values. This is captured by the following interface: $(\mathit{fuse\_res}', h', r') = \mathit{exe}\ \emptyset(x := w1, y := w2)\ h\ (\mathit{fuse}\ [x,y])$. For *copy* we model the fact that the input from $w3$ is forwarded directly to its first output as follows: $(\mathit{copy\_res1}', (\mathit{copy\_res2}', h', r')) = (w3, \mathit{exe}\ \emptyset(x := w3)\ h\ (\mathit{copy}\ [x]))$. The super-step is similar to the one already described, although the copy box checks and updates two wires.

In this example the box network is open, with input wire $w3$ and output wire $w4$. Therefore we have to extend $\mathcal{N}$ with an additional conjunct *env*, which models the environment. Both $w3$ and $w4$ are assumed to be updated by other boxes (or streams). In the coordination-layer proof below, we model a single, non-overlapping evaluation where wire $w3$ initially contains a list structure (*mList*) and is never updated, while wire $w4$ consumes the result.

We specify the layout of a list structure by the following inductive definition *mList*. Informally, $(n, a, U, h) \in mList$ means that at address a in heap h we find a list structure with n CONS and 1 NIL cells, covering the addresses in U.

$$\frac{h\ a = \text{Some}\ (\text{NIL}, [])}{(0, a, \{a\}, h) \in mList}\ \textsc{mListNIL}\qquad \frac{\begin{array}{c} h\ a = \text{Some}\ (\text{CONS}, [r, r']) \\ a \notin U \qquad (n, r', U, h) \in mList \end{array}}{(n+1, a, U \cup \{a\}, h) \in mList}\ \textsc{mListCONS}$$

The following definition is useful to express that address a contains a list of length n: $list\ h\ a\ n \equiv \exists\ U.\ (n, a, U, h)\ \in\ mList$

We are interested in the heap consumption, specified as a relation between the pre-heap h and the post-heap h'. We define these resource properties as entries in the *function specification table* $\mathcal{F}$. The specification for *copy* states, that provided the argument x points to a list structure of length n in the pre-heap h, the size of the post-heap h' is at most the size of the pre-heap plus $n+1$. Thus, the heap consumption of the *copy* function is $n + 1$. Furthermore, the result of the function (in v) is a list of length n at location r' in the post-heap h'. The remaining clauses assure that the input data structure is not modified. The clause $h \sim_{(\text{dom}\ h') - U'} h'$ states that the heaps h and h' contain the same values at addresses $(\text{dom}\ h') - U'$, i.e. only the values in U' are modified. The clause $U' \cap (\text{dom}\ h) = \emptyset$ states that the copy of the list does not overlap with the input list, and the $(n, r, U, h') \in mList$ clause states that the structure of the input list is indeed unchanged. Analogously, the specification for *fuse* states, that provided the arguments x_0 and x_1 point to list structures, of the same length n_0 and n_1, at locations r_0 and r_1 in the pre-heap h, the size of the post-heap h' is at most the size of the pre-heap plus $n_0 + 1$. The result of the function is a list of length n_0 at location r' in the post-heap.

$$\begin{array}{l}
\mathcal{F}\ copy\ [x] \equiv \lambda E\ h\ h'\ v\ p.\ \forall n\ r\ U. \\
\quad E\ x\ =\ r\ \wedge\ (n, r, U, h) \in mList \longrightarrow \\
\qquad |\ \text{dom}\ h'\ | \leq |\ \text{dom}\ h\ | + n + 1\ \wedge (\exists r'\ U'.\ v = r'\ \wedge\ (n, r', U', h') \in mList\ \wedge \\
\qquad h \sim_{(\text{dom}\ h') - U'} h'\ \wedge\ U' \cap (\text{dom}\ h) = \emptyset\ \wedge\ (n, r, U, h') \in mList) \\
\mathcal{F}\ fuse\ [x_0, x_1] \equiv \lambda E\ h\ h'\ v\ p.\ \forall n_0\ n_1. \\
\quad (\exists r_0\ U_0.\ E\ x_0\ =\ r_0\ \wedge\ (n_0, r_0, U_0, h)\ \in mList) \wedge \\
\quad (\exists\ r_1\ U_1.\ E\ x_1\ =\ r_1\ \wedge\ (n_1, r_1, U_1, h)\ \in\ mList)\ \wedge\ n_0 = n_1) \longrightarrow \\
\qquad |\ \text{dom}\ h'\ | \leq |\ \text{dom}\ h\ | + n_0 + 1\ \wedge\ (\exists n'\ r'\ U'.\ v = r'\ \wedge\ (n', r', U', h') \in mList\ \wedge\ n' = n_0)
\end{array}$$

In order to prove these two assertions, we first construct for each function a context, with specifications for all function calls in its body. We then prove the good context predicate, for these contexts. In both functions we find one direct recursive call, and therefore construct a one-element context for each function.

Lemma 4. $goodContext\ \mathcal{F}\ \{(copy\ \vec{xs},\ \mathcal{F}\ copy\ \vec{xs})\} \wedge goodContext\ \mathcal{F}\ \{(fuse\ \vec{xs}, \mathcal{F}\ fuse\ \vec{xs})\}$

Proof structure. The same structure as in the proof for Theorem 4, but using VDMAx when encountering the recursive call. □

Now the resource consumption can be proven in an empty context:

Theorem 4. $\emptyset \rhd \mathit{copy}\ \vec{xs} : \mathcal{F}\ \mathit{copy}\ \vec{xs} \wedge \emptyset \rhd \mathit{fuse}\ \vec{xs} : \mathcal{F}\ \mathit{fuse}\ \vec{xs}$

Proof structure. In both clauses, by first applying the syntax-directed rules, and by applying CTXTWEAK, VDMADAPT and Lemma 4 when encountering the recursive call. The proof of the remaining subgoal proceeds by case distinction over the structure of the input, and over the cases in the conditional. □

On coordination layer the main theorem below states that at each point in the execution any wire and any expression-layer result is either empty ($\bot$), or contains a reference to a list structure with N elements ($\mathit{list}\ldots$).

Theorem 5. $\Box \left(\begin{array}{l} (w1 = \bot \vee\ \mathit{list}\ h\ w1\ N) \wedge (w2 = \bot \vee\ \mathit{list}\ h\ w2\ N) \\ \wedge\ (w3 = \bot \vee\ \mathit{list}\ h\ w3\ N) \wedge (w4 = \bot \vee\ \mathit{list}\ h\ w4\ N) \\ \wedge\ (\mathit{copy\_res1} = \bot \vee\ \mathit{list}\ h\ \mathit{copy\_res1}\ N) \\ \wedge\ (\mathit{copy\_res2} = \bot \vee\ \mathit{list}\ h\ \mathit{copy\_res2}\ N) \\ \wedge\ (\mathit{fuse\_res} = \bot \vee\ \mathit{list}\ h\ \mathit{fuse\_res}\ N) \end{array} \right)$

Proof structure. Standard TLA reasoning reduces the proof of each conjunct as a separate invariant, which is outlined below. □

We will focus on the proof of one conjunct, shown as Lemmas 8. The other conjuncts are verified similarly. We start with the proof for an auxiliary theorem, relating the expression-layer result to the coordination-layer structure.

Lemma 5. $\mathit{list}\ H\ V\ M \wedge (W, H', S) = \mathit{exe}\,\emptyset(x := V)\ H\ (\mathit{copy}\,[x]) \longrightarrow \mathit{list}\ H'\ V\ M$

Proof structure. This is proven from the expression-layer specification of *copy* using the Integration Theorem (vdmexe). □

Note that this property asserts that after box-execution the input V is still a *list*. Capital letters represent free variables. The invariant in Lemma 6 ensures that no other boxes (incl. the environment) can execute when $\mathit{copy\_res1} \neq \bot$.

Lemma 6. $\Box(\mathit{copy\_res1} \neq \bot \longrightarrow w1 = \bot \wedge\ w2 = \bot \wedge w3 = \bot \wedge\ \mathit{fuse\_res} = \bot)$

Proof. The proof follows the "standard Hume/TLA invariant structure" and follows from $\Box(w3 \neq \bot \longrightarrow w1 = \bot \wedge\ w2 = \bot \wedge \mathit{copy\_res1} = \bot \wedge \mathit{copy\_res2} = \bot \wedge\ \mathit{fuse\_res} = \bot)$, which can be proven directly using the same structure. □

The following invariant asserts that wires and expression-layer results are well-formed, and that box executions do not interfere.

Lemma 7. $\Box(w3 = \bot \vee\ \mathit{list}\ h\ w3\ N)$

Proof structure. This is proved using the "standard Hume/TLA invariant structure" with the same strengthening as in the proof of Lemma 6. □

Lemma 8. $\Box(\mathit{copy\_res1} = \bot \vee\ \mathit{list}\ h\ \mathit{copy\_res1}\ N)$

Proof. The proof follows the "standard Hume/TLA invariant structure", strengthened by Lemmas 6 and 7, instantiating the free variables of Lemma 5. □

The proofs of the other conjuncts of Theorem 5 have the same structure, using lemmas over the expression-layer result of *copy* and *fuse* similar to Lemma 5. Since the entire expression-layer heap is discarded after execution, the size of the live heap at coordination-layer is the sum of all data structures reachable from the wires and results. From the 7 *list* predicates in Theorem 5 we can immediately conclude that $7N + 7$ is such a bound. Moreover, Lemma 6 and similar lemmas for the other expression-layer results restrict the possible combinations of non-$\perp$ values in the wires and results: after executing *copy* only $w1, w2$, *copy_res1* and *copy_res2* are non-$\perp$; after executing *fuse* only $w4$ and *fuse_res* are non-$\perp$. Therefore, we can refine the upper bound on the size of the live heap to $4N + 4$.

6 Related Work

Our VDM-style, resource-aware program logic builds on our logic in [1], especially in handling mutual recursion and parameter adaptation, where the source language was an abstraction of JVM bytecode, but without algebraic data-types or higher-order functions as in Hume. In our formalisation we use techniques studied in Nipkow's Hoare-style logic for a simple while language in [21] and the Hoare-style logic for Java-light by von Oheimb [26], both formalised in Isabelle/HOL. Our choice of a VDM-style logic was partially driven by the study of variants of Hoare-style and VDM-style logics explored in Kleymann's thesis [16]. Isabelle/HOL formalisations of a program logic for a call-by-value, functional language are studied in [18]. DeBoer et al. [5, 22] present a sound and complete Hoare-style logic for a sequential object-oriented language with inheritance and subtyping, with tool support for generating VCGs from annotated flowcharts, which are solved in HOL [3]. Other related reasoning infrastructures, mostly for object-oriented languages, are: the Jive system [20], building on a Hoare-style logic for a Java subset [23]; the LOOP project's encoding of a Hoare-style logic, with extensions for reasoning about abrupt termination and side-effects, in PVS/Isabelle [14]; the *Why* system [6], with *Krakatoa* [19] as front-end, using Coq for formalisation and interactive proofs of JML-annotated Java programs.

Our choice of TLA for the coordination layer comes from the stateful nature of both systems, whereas approaches such as process algebras are stateless. For the "independent" mechanisation of the coordination layer, we have developed several proof tactics, which automate the verification of "standard Hume/TLA invariant structure" [8], albeit using Isabelle/HOL directly to encode the expression layer. Our integration of the two formalisations follows the "state-behaviour integration" approach, exemplified by csp2b [4], CSP||B [25] and CSP-OZ [7]. However, in this integration computation is state based (B or Z), and the communication uses a stateless *process algebra* [2]. In Hume on the other hand, computation is within a (stateless) purely functional language, and communication is within a (stateful) finite state machine language. All these tools are

motivated by using suitable tools for the different aspects of a system. csp2b [4] works by translating the CSP (process algebra) [13] into a B machine, while CSP||B [25] and CSP-OZ [7] give a CSP semantics to the computational aspect (B and Object-Z). In our case, both the TLA and VDM representations are built on top of Isabelle/HOL, and integration is naturally achieved by "unlifting" into the Isabelle/HOL level, and no (resource preserving) translation is required.

7 Conclusion

We have presented an integrated Isabelle/HOL formalisation of the two language layers of Hume. Our formalisation combines a high level of abstraction through the VDM-style program logic for the expression layer, ensured through the meta-theoretic soundness result, with a direct, shallow embedding of a TLA logic for the coordination layer. By formalising the most suitable style of logics for these two layers in a theorem prover, we obtain a powerful reasoning infrastructure. We have applied this infrastructure to two Hume programs, verifying both functional and resource properties of the code, refining the latter through proofs of the availability of wire-values. These examples show that while the proofs can become lengthy, the proof strategies that can be applied follow naturally from the respective logics.

Notably, the TLA proofs follow a common structure, and we have explored their automation, through proof tactics, in the stand-alone coordination layer embedding [8]. In our integrated system, the proofs are (manually) driven by this structure at the coordination layer. Where needed, we make use of the VDM logic to prove the required properties for expressions. For resource properties only simple inequalities, eg. over heap sizes, have to be proven. The main complication comes from asserting disjointness of data structures, and here techniques from separation logic could help [24]. To raise the level of abstraction further, we are currently working on a specialised resource logic for the expression layer and on tactics for combining the two layers.

References

1. D. Aspinall, L. Beringer, M. Hofmann, H-W. Loidl, and A. Momigliano. A Program Logic for Resources. *Theoretical Computer Science*, 389(3):411–445, 2007.
2. J. C. M. Baeten. A Brief History of Process Algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.
3. F.S. de Boer and C Pierik. Computer-Aided Specification and Verification of Annotated Object-Oriented Programs. In *FMOODS 2002*, volume 209 of *IFIP Conference Proceedings*, pages 163–177. Kluwer, 2002.
4. M.J. Butler. csp2B: A Practical Approach to Combining CSP and B. *Formal Aspect of Computing*, 12(3):182–198, 2000.
5. F.S. de Boer. A WP-calculus for OO. In *Foundations of Software Science and Computation Structures (FoSSaCS'99)*, LNCS 1578, pages 135–149. Springer, 1999.
6. J.-C. Filliâtre. Why: a Multi-language Multi-prover Verification Tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

7. C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, pages 423–438, 1997.
8. G. Grov. *Reasoning about Correctness Properties of a Coordination Programming Language*. PhD thesis, Heriot-Watt University, 2009. To appear.
9. G. Grov, G. Michaelson, and A. Ireland. Formal Verification of Concurrent Scheduling Strategies using TLA. In *International Conference on Parallel and Distributed Systems (ICPADS'07)*, pages 1–6, Hsinchu, Taiwan, December 2007. IEEE.
10. K. Hammond. Hume: a Bounded Time Concurrent Language. In *Conf. on Electronics and Control Systems (ICECS '2K)*, pages 407–411, Kaslik, Lebanon, 2000.
11. K. Hammond, G. Grov, G. Michaelson, and A. Ireland. Low-Level Programming in Hume: an Exploration of the HW-Hume Level. In *Implementation of Functional Languages (IFL'06)*, LNCS 4449, pages 91–107. Springer, 2006.
12. K. Hammond and G.J. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Conf. Generative Programming and Component Engineering (GPCE '03)*, LNCS 2830, pages 37–56. Springer-Verlag, September 2003.
13. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
14. M. Huisman and B. Jacobs. Java Program Verfication via a Hoare Logic with Abrupt Termination. In *Fundamental Approaches to Software Engineering (FASE'00)*, LNCS 1783, pages 284–303, 2000.
15. C. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990.
16. T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, University of Edinburgh, 1999.
17. L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
18. J. Longley and R. Pollack. Reasoning About CBV Functional Programs in Isabelle/HOL. In *Theorem Proving in Higher Order Logics (TPHOLs'04)*, LNCS 3223, pages 201–216. Springer, 2004.
19. C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA Tool for Certification of JAVA/JAVACARD Programs Annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.
20. P. Müller, J. Meyer, and A. Poetzsch-Heffter. Programming and Interface Specification Language of JIVE - Specification and Design Rationale. Technical report, Fachbereich Informatik and Fernuniversitat Hagen, 2000.
21. T. Nipkow. Hoare Logics for Recursive Procedures and Unbounded Nondeterminism. In *Computer Science Logic (CSL'02)*, LNCS 2471, pages 103–119, 2002.
22. C. Pierik and F.S. de Boer. A Syntax-Directed Hoare Logic for Object-Oriented Programming Concepts. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS'03)*, LNCS 2884, pages 64–78. Springer, November 2003.
23. A. Poetzsch-Heffter and P. Müller. A Programming Logic for Sequential Java. In *European Symposium on Programming (ESOP'99)*, LNCS 1576, pages 162–176, 1999.
24. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Symposium on Logic in Computer Science (LICS'02)*, pages 55–74, Copenhagen, July 2002. IEEE Computer Society.
25. S. Schneider and H. Treharne. CSP Theorems for Communicating B Machines. *Formal Aspects of Computing: Applicable Formal Methods*, 17(4):390–422, 2005.
26. D. von Oheimb. Hoare Logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.