Formal Verification of Concurrent Scheduling Strategies using TLA

Gudmund Grov, Greg Michaelson and Andrew Ireland
School of Mathematical and Computer Sciences
Heriot-Watt University, Riccarton, Scotland, EH14 4AS
{gudmund,greg,air}@macs.hw.ac.uk

Abstract

There is a high demand for correctness for safety critical systems, often requiring the use of formal verification. Simple, well-understood scheduling strategies ease verification but are often very inefficient. In contrast, efficient concurrent schedulers are often complex and hard to reason about.

This paper will show how the TLA logic can be used to verify schedulers of concurrent components. TLA allows us to prove that one program preserves the behaviour of another program, in particular that an efficient scheduling strategy preserves the behaviour of a simpler one. For an arbitrary program we can use the simpler scheduler for correctness verification, knowing that the properties also hold in the more efficient one, which we then implement. This approach is illustrated with the Hume programming language, which is based on concurrent rich automata. We show an efficient extension to the Hume scheduler, and prove that this extension preserves the behaviour of the standard Hume scheduler.

1. Introduction

There is a high demand for correctness for safety critical systems. Thus, many standards for developing such systems, like the UK MoD 00-55 [16], require the application of formal methods. Due to the concurrent, rather than sequential, nature of distributed systems, composition of pre-post conditions of components, as found in e.g. Hoare logic [10], is not valid for such verification. More details are required for the coordination of the components and, *temporal logic* has successfully been applied to verify correctness of these types of systems. However, it is notoriously hard to both specify and reason about properties in temporal logic. Thus, to reason about concurrent systems inside a temporal logic, a simple and well-understood formal model of the system is highly desirable. However, simple scheduling strategies are often very inefficient. In this paper we will show how TLA [12] can be applied to the formal verification of schedulers. We will show how we can formally verify that an efficient scheduling strategy preserves the behaviour of a simple model. Consequently, correctness properties of the simpler strategy also hold for the more efficient one, thus allowing us to reason inside the simple strategy, knowing that these properties hold in the more efficient one. The efficient scheduler can therefore be implemented, while the simple one is used for verification.

We will illustrate our approach with the Hume [15] programming language. In Hume, a program is a set of concurrent finite state automata, making it well suited to implement concurrent systems. We motivate and define a more efficient scheduling strategy for Hume. We then formalise both the current and the more efficient scheduling in TLA, and prove that the efficient extension preserves the behaviour of the current scheduler.

2. TLA

Temporal Logic of Actions (TLA) [12] combines temporal logic with actions. 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.

TLA can be "lifted" on top of any predicate logic, hence all three layers have a full propositional calculus. Additionally, the action level has a priming (') operator to separate variables in the "result" state (primed) from those of the "before" state. At this level, "beforeW" variable v and its "result" counterpart v', are distinct. The temporal level has two additional operators: $\Box P$ which denotes P is True if it holds in all states of the sequence, and an operator $\exists$ which is used to hide internal details.

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

$$\mathit{Program} \Rightarrow \mathit{Property}. \tag{1}$$

Programs are compared in a similar fashion. For example, to prove that a more efficient program preserves the behaviour of the original *Program* we must show that

$$\mathit{Efficient\ Program} \Rightarrow \mathit{Program}. \tag{2}$$

Due to transitivity, (1) ensures that *Property* also holds for *Efficient Program*. Thus, if (2) holds, it is sufficient to prove the required properties for *Program*. The key in proving properties such as (2) is allowing *stuttering steps*, i.e. steps that leave the state unchanged. This allows the use of internal action, which is important when comparing many scheduling algorithms. However, this is not required here, and will not be discussed further (see [1, 12] for details).

To define a program we must specify an initial state I, and an action $\mathcal{N}$, representing the transitions. $\mathcal{N}$ is a predicate which compares a "result" and a "before state". The action must hold throughout execution, i.e. $\Box\mathcal{N}$. However, this does not support stuttering steps. Let $\langle v, i\rangle$ be the tuple of all visible v and internal i variables of the program. We refine $\Box\mathcal{N}$ to $\Box(\mathcal{N} \vee \langle v, i\rangle' = \langle v, i\rangle)$, which asserts that in all transitions either $\mathcal{N}$ holds, or the state is left unchanged. This supports stuttering, and is abbreviated by $\Box[\mathcal{N}]_{\langle v,i\rangle}$. We will use monolithic specifications of our programs. Such specifications, with the internal variable hidden, are written:

$$\exists\, i : I \wedge \Box[\mathcal{N}]_{\langle v,i\rangle}. \tag{3}$$

To ease reasoning, TLA requires that $\mathcal{N}$ 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 for all the variables.

3. Hume

Hume [15] is a novel programming language which explicitly separates the *coordination* and the *computation* concerns. It is based on autonomous *boxes*, linked by *wires*, and controlled by generalised transitions, termed *match*es, of the form:

$$pattern \rightarrow expression$$

A *pattern* is matched with the inputs and, if it succeeds, causes the *expression* to generate output. If it fails, the following match is tried. If none of the matches succeed, then no transition is made. Boxes and wires are defined in the finite state *coordination language*. Transitions within boxes are defined in the purely functional *expression language* through pattern matching and associated recursive actions. This discussion will focus on the scheduling of boxes, and we will therefore not go into the full expression layer and pattern matching details.

3.1. Hume Scheduling

The Hume coordination layer is static. Thus, each wire has a source and destination which will not change. For simplicity, we will assume that all wires are between boxes, i.e. there are no connected external devices. A wire is a single value buffer that connects two boxes. An important feature is the ability of a pattern to ignore inputs on given wires, and an expression to not produce outputs on all output wires. This is identified by '$*$' in the source code. Since a wire is single buffered, a box will block with pending output until all of its required output wires become empty.

Since Hume targets safety critical systems, there is a strong focus on formal analysis, which requires deterministic programs, and a simple execution model. In Hume, this is achieved by cyclical execution, where each cycle has two phases: in the first phase each box is run once and attempts to consume inputs and generate outputs; and in the second phase the output changes are resolved in a unitary super-step. At the end of each phase a box will be in one of the following states:

- **Rn** (Runnable) The box has successfully consumed inputs and asserted outputs.
- **Bl** (Blocked) The box has successfully consumed inputs but failed to assert outputs. It will attempt to assert outputs on subsequent cycles.
- **Mf** (Matchfail) The box has failed to match inputs.

Hume then uses a *lock-step* scheduling, thus achieving deterministic programs. The scheduling works as follows:

for ever
  for each ***Rn*** *and* ***Mf*** *box*
    execute (box)
  super step

3.2. Hume Scheduling Formalised in TLA

We will formalise a high-level definition of Hume in TLA. A fuller formalisation, at a lower grain of atomicity, is given in [4] which also gives a detailed exploration of §5.1 and §6.

The layering of TLA fits well into the Hume design. Properties of a one-shot execution of a box are action predicates, while properties of the coordination layer are temporal formulas. In the formalisation we assume the following:

there is a set *BS* of box identifiers, henceforth *boxes*; all components below, except the scheduling variable s, are tuples of distinct variables. The i^{th} element is accessed by a sub-script, e.g. st_i. For boxes, we assume that the tuple is ordered with respect to the box identifier, that is, st_i is the state for box $i \in BS$; there is a tuple *ws* of wires; each box i has a tuple of input iws_i and output ows_i wires; feedback loops are allowed, thus a wire in iws_i may also occur in ows_i; if iws_i or ows_i for all $i \in BS$ are composed this it will equal *ws*; a wire in *ws* will occur in only one iws_i and one ows_i; each box i has a state variable st_i and a result buffer res_i; for all $i \in BS$ $st_i \in \{\mathbf{Rn}, \mathbf{Bl}, \mathbf{Mf}\}$; res_i has the same number of elements as ows_i.

$$
\begin{array}{rcl}
\mathcal{S} & \triangleq & (s = E \Rightarrow s' = S) \wedge (s = S \Rightarrow s' = E) \\
\mathcal{B}_i^e & \triangleq & st_i \neq Bl \Rightarrow \langle iws_i, res_i, st_i \rangle' = e_i(iws_i) \\
& \wedge & st_i = Bl \Rightarrow \langle iws_i, res_i, st_i \rangle' = \langle iws_i, res_i, st_i \rangle \\
\mathcal{B}_i^s & \triangleq & Q_i \Rightarrow \langle ows_i, res_i, st_i \rangle' = \langle w(res_i, ows_i), \bot, Rn \rangle \\
& \wedge & \neg Q_i \Rightarrow \langle ows_i, res_i, st_i \rangle' = \langle ows_i, res_i, Bl \rangle \\
\mathcal{B}_i & \triangleq & (s = E \Rightarrow \mathcal{B}_i^e) \wedge (s = S \Rightarrow \mathcal{B}_i^s) \\
H_l & \triangleq & I \wedge \Box[\mathcal{S} \wedge \bigwedge_{i \in BS} \mathcal{B}_i]_{\langle s, ws, res, st \rangle} \\
H & \triangleq & \exists\, s, res, st : H_l
\end{array}
$$

Figure 1. Lock-step scheduling in TLA

Figure 1 shows the TLA actions. $\mathcal{S}$ is defined to alternate between the execute E and super-step S phase. $\mathcal{B}_i^e$ is the box action for an arbitrary box i in the E phase. Note, the check that a box is **Rn** or **Mf** is moved into the box action, and not before it is called, as explained in §3.1. This is a consequence of TLA's requirement for explicitly defining unchanged variables, and eases the specification of the action. If the check succeeds then $e_i(iws_i)$ will match and consume the required inputs, produce the result output, and set the new state – i.e. **Rn** if a match succeeds and **Mf** if not. If the check fails, then iws_i and res_i are left unchanged.

Similarly, $\mathcal{B}_i^s$, represents the super-step phase of a box. The Q_i predicate succeeds if, for all corresponding elements k of res_i and ows_i, one of them is '$*$':

$$Q_i \triangleq \forall k \in len(ows_i) : \big((res_i)_k = *\big) \vee \big((out_i)_k = *\big)$$

where $len(ows_i)$ is the number of element in tuple ows_i. If Q_i succeeds then the new computed values of res_i are written to the output wires, achieved by $w(res_i, ows_i)$. The output buffer is set to empty, represented by $\bot$, and the state is **Rn**. If it fails, the output buffer's value is left dangling, and the box will be **Bl** at least until the next super-step, where Q_i is re-checked.

$\mathcal{B}_i$ uses the "scheduling phase value" s to determine if $\mathcal{B}_i^e$ or $\mathcal{B}_i^s$ should be called for an arbitrary box i. H_l is the program specification. We leave the initial state I undefined. The complete next-state action is the conjunction of the scheduler $\mathcal{S}$ and the next actions for all boxes $i \in BS$. Finally, in Hume we are only interested in the values on the wires, thus H hides all variables, except *ws*, using $\exists$.

4. An Efficiency Problem

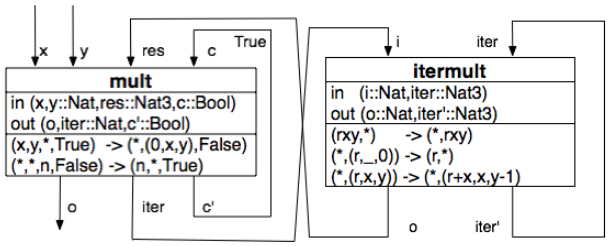

Figure 2. Multiplication as Iteration

The motivation behind the layered design of Hume was to add the ability for static costing of time and space usage [8, 14]. However, many such properties are undecidable in the expression layer. Thus, the balance between expressiveness and decidability is a key factor in Hume development, and often requires us to find more decidable representation of expressive constructs. One such example is to represent repetition as iteration in the coordination layer, instead of recursion in the expression layer – where the latter is hard to cost. Iteration is achieved by the use of a feedback wire, in the same way as in hardware systems.

Figure 2 illustrates *coordination iteration* by implementing multiplication using two boxes. `mult` is the "interface" to the rest of the program. When it receives inputs on input wires `x` and `y`, the iteration begins if the previous result has produced output on `o`. This is achieved by the `c'` $\rightarrow$ `c` wire. A match on these inputs causes the `(0,x,y)` triple to be sent to `itermult` box. In each iteration `itermult` will add `x` to the previous result, represented by `r`, initially set to 0 – and decrement `y` by 1. The iteration "terminates" when `y` is 0. The result is sent back to `mult2`. Note that '_' in a pattern denotes that a value must be present, and this value is consumed. Further, note that `(`$\cdots$`)` is used for both pattern/expression and tuples. We have left the rest of the program unspecified for now.

The Hume scheduler will always attempt to run boxes which do not have a **Bl** state. In an iteration of depth N (the initial `y` value), the program requires $N+4$ steps to compute the output. For $N+2$ of these steps, the `mult` box will be in a **Mf** state. This will be the case for all boxes that depend, directly or indirectly, on such an iteration. Hence, it must be possible to reduce this unnecessary scheduling overhead.

5. Towards Staged Scheduling

In the usual execution of a box, an **Rn** box may have either asserted outputs to other boxes and itself, or just to other boxes, or just to itself. If it has asserted outputs just to itself then it can have no impact on the ability of any other box to consume inputs. We say that such boxes have the *self-output* property.

Thus, in principle, such boxes may execute repeatedly until they assert an output for another box, without affecting the overall outcome of program execution, provided there are no strong timing dependencies elsewhere in the program. To achieve this, **Rn** is divided up into two sub-states:

- $\mathbf{Rn}^s$ (Runnable) The box has successfully consumed inputs and asserted outputs, and *is not only* writing to internal wires.
- **Sf** (Selfout) The box has successfully consumed inputs and asserted outputs, and *is only* writing to internal wires.

Then the staged scheduling strategy is:

for ever
  if no box is $\boldsymbol{Rn}^s$
    then for each ***Sf*** *box*
      execute (box)
    else for each $\boldsymbol{Rn}^s$*,****Sf*** *and* ***Mf*** *box*
      execute (box)
  super step

Both the interpreter and the compiler have been updated with this staged scheduling, resulting in the expected efficiency gain. For example, for the interpreter, in the repetition as iteration example in §4, by connecting `x` and `y` to a generator producing the first 1411 integers, and connecting ○ to standard output, the staged scheduling had an increased timing saving of 14% (57s vs. 70.9s) compared to lock-step scheduling.

5.1. Staged Scheduling Formalised in TLA

Figure 3 shows the TLA actions for staged scheduling. $\mathcal{S}$ and I are unchanged from Figure 1. *SO* is used in the execute phase – which is now defined by $s\mathcal{B}_i^e$ for an arbitrary box i. It behaves as specified in §5, by executing only **Sf** boxes when *SO* holds, and as for lock-step scheduling when it does not hold. $s\mathcal{B}_i^s$ defines the box action in a super-step. Here, we assume a sub-tuple $nows_i$ of ows_i for all $i \in BS$. This tuple holds all the wires that are *not* wired back to box i. Moreover, $emp(nows_i')$ holds if $nows_i'$ only holds '$*$'s, i.e. all output values are on the wires wired back to i. If the box only writes to internal wires, i.e. $emp(nows_i')$, then it is in an **Sf** state.

$$
\begin{array}{rcl}
SO & \triangleq & \bigwedge_{i \in BS} st_i \neq \mathbf{Ru}^s \\
s\mathcal{B}_i^e & \triangleq & \textbf{if } (SO \Rightarrow st_i = \mathbf{Sf}) \wedge (\neg SO \Rightarrow st_i \neq \mathbf{Bl}) \\
 & & \quad \textbf{then } \langle iws_i, res_i, st_i \rangle' = e_i(iws_i) \\
 & & \quad \textbf{else } \langle iws_i, res_i, st_i \rangle' = \langle iws_i, res_i, st_i \rangle \\
s\mathcal{B}_i^s & \triangleq & Q_i \Rightarrow \langle ows_i, res_i \rangle' = \langle w(iws_i), \bot \rangle \\
 & \wedge & st_i' = \textbf{ if } emp(nows_i') \textbf{ then Sf else } \mathbf{Rn}^s \\
 & \wedge & \neg Q_i \Rightarrow \langle ows_i, res_i, st_i \rangle' = \langle ows_i, res_i, Bl \rangle \\
\mathcal{N} & \triangleq & \bigwedge_{i \in BS} (s = E \Rightarrow s\mathcal{B}_i^e) \wedge (s = S \Rightarrow s\mathcal{B}_i^s) \\
sH_l & \triangleq & I \wedge \Box[\mathcal{S} \wedge \mathcal{N}]_{\langle st,s,ws,res \rangle} \\
sH & \triangleq & \exists\, st,s,res : sH_l
\end{array}
$$

Figure 3. Staged scheduling in TLA

6. A Proof of Correctness

We will now outline the proof that staged scheduling preserves the behaviour of lock-step scheduling, that is, proving (2) from §2.

$$\text{R1.}\ \frac{I_2 \Rightarrow I_1 \quad (\mathcal{B}_2 \Rightarrow \mathcal{B}_1) \quad (x' = x) \Rightarrow (y' = y)}{I_2 \wedge \Box[\mathcal{B}_2]_x \Rightarrow I_1 \wedge \Box[\mathcal{B}_1]_y}$$

$$\text{E1.}\ \vdash F\{f/x\} \Rightarrow \exists\, x : F \qquad \text{E2.}\ \frac{F \Rightarrow G}{(\exists\, x : F) \Rightarrow G} \ \textit{x does not occur free in G}$$

Figure 4. TLA Proof Rules

We will rely on the proof rules shown in Figure 4. Formulas above the line are assumptions and the formula below the line is the conclusion of a rule. Rule R1 is a specialisation of standard TLA rules for this particular example. It states that a monolithic program implements another if the initial state implements the initial state, the next-action implements the next-action and, the new state space is strictly larger than the old. E1 and E2 are direct copies from [12]. E1 is the introduction rule for $\exists$. If F holds with variable x replaced by a state function f, then F holds with x hidden. E2 is the elimination rule for $\exists$. It states that if a variable is free in the conclusion of an implication, then it can be bound in the assumption/given.

To prove our theorem we rely on the following fact, which follows directly from the definition of $\mathbf{Rn}^s$ and **Sf**:

Theorem 1 $st_i = \mathbf{Rn}$ *iff* $st_i \in \{\mathbf{Rn^s}, \mathbf{Sf}\}$

In the proof we must provide witnesses for the $\exists$ bound variables in H, using the state space of *sH*. In TLA the sum of these witnesses are called the *refinement mapping* [1]. If we ignore the subtle type differences between variables

and state functions then s and *res* are just replaced by themselves. All the st_i are replaced by the following function, based on Theorem 1:

$$\overline{st_i} \triangleq \textbf{if } st_i \in \{\mathbf{Rn^s}, \mathbf{Sf}\} \textbf{ then } \mathbf{Rn} \textbf{ else } st_i$$

For any action, formula, etc. $\cdots$ we use $\overline{\cdots}$ to denote $\cdots\{\overline{st_i}/st_i\}$ (for now we ignore the trivial s and *res* substitution). The substitution $\overline{\cdots}$ distributes over all operators.

Before the proof of the main Theorem 3, we prove Theorem 2. Here we remove all the $\exists$ and prove that sH_l implements $H_l\{\overline{st_i}/st_i\}$. Note that we unfold $H/H_l/sH/sH_l$ in the theorems. We will use Lamport's structured way of writing mathematical proofs [13]: The proof should be read hierarchically. The j^{th} step of the current level i proof is labelled $\langle i\rangle j$. Further, $\langle i\rangle j$ QED denotes the proof of the current level $i-1$ goal. $A \Rightarrow B$ is written ASSUME: A PROVE: B, and in case-splits, ASSUME is replaced by CASE while the goal is the same as the $i-1$ goal.

Theorem 2

$\langle 1\rangle 1$. ASSUME: $I \wedge \Box[\mathcal{S} \wedge \mathcal{N}]_{\langle s,ws,res\rangle}$
PROVE: $\overline{I} \wedge \Box[\overline{\mathcal{S}} \wedge \bigwedge_{i\in BS} \overline{\mathcal{B}_i}]_{\overline{\langle s,ws,res,st\rangle}}$
PROOF: Rule R1 reduces the proof to
$\langle 2\rangle 1$. ASSUME: I
PROVE: $\overline{I}$
PROOF: I and $\overline{I}$ are identical. □
$\langle 2\rangle 2$. ASSUME: $\mathcal{S} \wedge \mathcal{N}$
PROVE: $\overline{\mathcal{S}} \wedge \bigwedge_{i\in BS} \overline{\mathcal{B}_i}$
PROOF: Standards propositional reasoning and unfolding of $\mathcal{N}$ reduces the proof to:
$\langle 3\rangle 1$. ASSUME: $\mathcal{S}$
PROVE: $\overline{\mathcal{S}}$
$\langle 3\rangle 2$. ASSUME: $\bigwedge_{i\in BS} \big(s = E \Rightarrow s\mathcal{B}_i^e) \wedge (s = S \Rightarrow s\mathcal{B}_i^s\big)$
PROVE: $\bigwedge_{i\in BS} \overline{\mathcal{B}_i}$
PROOF: It is sufficient to show the goal for an arbitrary box $i \in BS$. The proof follows by a case-split on s and some simplifications:
$\langle 4\rangle 1$. ASSUME: $s = E \wedge s\mathcal{B}_i^e$
PROVE: $\overline{\mathcal{B}_i^e}$
PROOF: The proof is by a case analysis on both the *SO* property and on st_i. We will only address the key cases (see [4] for full details):
$\langle 5\rangle 1$. CASE: $SO \wedge st_i = \mathbf{Sf}$
PROOF: By assumptions $\langle 5\rangle 1$ and $\langle 4\rangle 1$ and the definition of $s\mathcal{B}_i^e$ we have
$$\langle iws_i, res_i, st_i\rangle' = e_i(iws_i) \qquad (4)$$
Further, assumptions $\langle 5\rangle 1$ implies by definition that $\overline{st_i} = \mathbf{Rn}$. This and (4) implies $\overline{\mathcal{B}_i^e}$ by definition. □
$\langle 5\rangle 2$. CASE: $SO \wedge st_i = \mathbf{Rn}^s$
PROOF: By contradition. Assumption $\langle 5\rangle 1$ implies both *SO* and $st_i = \mathbf{Rn}^s$. However, by definition *SO* implies $st_i \neq \mathbf{Rn}^s$ and we obtain a contradiction. □
$\langle 5\rangle 3$. Q.E.D.
PROOF: By $\langle 5\rangle 1$, $\langle 5\rangle 2$ and all the other remaining cases. □
$\langle 4\rangle 2$. ASSUME: $s = S \wedge s\mathcal{B}_i^s$
PROVE: $\overline{\mathcal{B}_i^s}$
PROOF: By the refinement mapping, both $emp(nows_i')$ and $\neg emp(nows_i')$ will give the state **Rn**, which is the case in $\overline{\mathcal{B}_i^s}$. The remaining parts of the proof follow directly. □
$\langle 4\rangle 3$. Q.E.D.
PROOF: By $\langle 4\rangle 1$ and $\langle 4\rangle 2$. □
$\langle 3\rangle 3$. Q.E.D.
PROOF: By $\langle 3\rangle 1$ and $\langle 3\rangle 2$. □
$\langle 2\rangle 3$. ASSUME: $\langle s,ws,res\rangle' = \langle s,ws,res\rangle$
PROVE: $\overline{\langle s,ws,res,st\rangle}' = \overline{\langle s,ws,res,st\rangle}$
PROOF: This is trivial by the definition of $\overline{\cdots}$. □
$\langle 1\rangle 2$. Q.E.D.

Using this theorem, we can now prove the main theorem that staged scheduling preserves the behaviour of lock-step scheduling.

Theorem 3 *Staged scheduling behaves like lock-step scheduling.*

$\langle 1\rangle 1$. ASSUME: $\exists\, s,res,st : I \wedge \Box[\mathcal{S} \wedge \mathcal{N}]_{\langle s,ws,res\rangle}$
PROVE: $\exists\, s,res,st : I \wedge \Box[\mathcal{S} \wedge \bigwedge_{i\in BS} \mathcal{B}_i]_{\langle s,ws,res,st\rangle}$
$\langle 2\rangle 1$. ASSUME: $(I \wedge \Box[\mathcal{S} \wedge \bigwedge_{i\in BS} \mathcal{B}_i]_{\langle s,ws,res,st\rangle})\{\overline{st}/st\}$
PROVE: $\exists\, s,res,st : I \wedge \Box[\mathcal{S} \wedge \bigwedge_{i\in BS} \mathcal{B}_i]_{\langle s,ws,res,st\rangle}$
PROOF: By applying E1 with $\overline{st}$ for f and st for x to assumption $\langle 2\rangle 1$ followed by E1 with the dummy substitutions $\{res/res\}$ and $\{s/s\}$ (and ignoring the subtle type differences between variables and state functions) we obtain the goal. □
$\langle 2\rangle 2$. ASSUME: $I \wedge \Box[\mathcal{S} \wedge \mathcal{N}]_{\langle s,ws,res\rangle} \Rightarrow H$
PROVE: $\exists\, s,res,st : I \wedge \Box[\mathcal{S} \wedge \mathcal{N}]_{\langle s,ws,res\rangle} \Rightarrow H$
PROOF: By the definition of H, *s,res* and *st* are bound by $\exists$. Rule E2 can therefore be applied three times to assumption $\langle 2\rangle 2$ to obtain the goal. □
$\langle 2\rangle 3$. Q.E.D.
By applying Theorem 2 sequentially to $\langle 2\rangle 1$ followed by $\langle 2\rangle 2$. □
$\langle 1\rangle 2$. Q.E.D.
By $\langle 1\rangle 1$. □

7. Relevant Work

Both algorithmic and deductive verification techniques have been applied to verify schedulers. For example, Cadena [9] uses *model checking*. Model checkers have the advantage of being fully automating, but require a finite model and thus, cannot be applied to verify scheduling

strategies. Moreover, the model must be sufficiently small. Narasimhan et al [17] applies theorem proving to verify the correctness of the FDLS scheduling strategy. Our approach is distinct from these by following the "refinement method" where we reason with a simpler model and verify that a more efficient strategy refines it. This eases reasoning since we are only required to prove the refinement once (for the strategy) – and can use the simpler scheduling to reason about all programs. The B tool [3] is probably the most well-known approach based on refinement. However, B cannot deal with concurrency, thus making TLA, which targets such systems, more suitable. TLA has previously been applied to other problems of a concurrent nature, like [2, 11]. However, with the exception of previous Hume work [7], we are not familiar with the use of TLA at the programming language level. Consequently, it has not been used for schedulers at this level either.

In [6] we explore a different scheduling strategy for Hume, using hierarchies of boxes. However, this work is not purely a scheduling problem, and is mainly motivated by transformations. Further, we do not discuss the use of TLA. [5] is based on the hierarchical structure in [6], and as in [6], the TLA aspect is not considered.

8. Conclusion

We have shown how to formally verify properties of concurrent programs with complex schedulers, by reasoning with a simpler scheduler and showing that the complex strategy preserves the behaviour of the simpler. This has been illustrated by showing that an efficient extension to the standard scheduler in Hume is behaviour preserving. The extension is now standard in both the Hume interpreter and compiler. As a result of the theorem proven here, we can still use the easier lock-step scheduling in our proofs, although the staged approach is used in actual programs. We have formalised both strategies in TLA and given a formal proof.

We are interested in the verification of properties of Hume programs and have found TLA very useful for properties of the coordination layer. We are exploring both model checking [7] and a theorem proving approach using Isabelle. We intend to mechanise the proof of this theorem in Isabelle.

Acknowledgements

Thanks to Robert Pointon. This work is supported by EU FP6 EmBounded project and a James Watt Scholarship.

References

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

[2] M. Abadi, L. Lamport, and S. Merz. A TLA solution to the RPC-memory specification problem. volume 1169 of *Lecture Notes in Computer Science*, pages 21–66. Springer, 1994.

[3] J.-R. Abrial. *The B-Book - Assigning Programs to Meanings*. Cambridge University Press, Aug. 1996.

[4] G. Grov. A Formal Account of Hume Scheduling. Technical Report HW-MACS-TR-0052, Heriot-Watt University, 2007. `http://www.macs.hw.ac.uk/techreps/`.

[5] G. Grov and G. Michaelson. Towards a Box Calculus for Hierarchical Hume. Under review for TFP, 2007.

[6] G. Grov, R. Pointon, G. Michaelson, and A. Ireland. Preserving Coordination Properties when Transforming Concurrent System Components. Submitted to APLAS, 2007.

[7] K. Hammond, G. Grov, G. Michaelson, and A. Ireland. Low-Level Programming in Hume: an Exploration of the HW-Hume Level. In *IFL*, 2006. Accepted for publication in LNCS 4449.

[8] K. Hammond and G. Michaelson. Hume: A Domain Specific Language for Real-Time Embedded Systems. In *Proceedings of GPCE'03: Generative Programming and Component Engineering, Erfurt, Germany*. Springer, LNCS, September 2003.

[9] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In *International Conference on Software Engineering*, volume 1169, 2003.

[10] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–585, October 1969.

[11] P. B. Ladkin, L. Lamport, B. Olivier, and D. Roegel. Lazy caching in TLA. *Distributed Computing*, 12(2-3):151–174, 1999.

[12] L. Lamport. The Temporal Logic of Actions. *ACM Toplas*, 16(3):872–923, May 1994.

[13] L. Lamport. How to Write a Proof. *American Mathematical Monthly*, 102(7):600–608, Aug./Sept. 1995.

[14] G. Michaelson, K. Hammond, and J.Serot. FSM-Hume: Programming Resource-Limited Systems using Bounded Automata. In *SAC*, pages 1455–1461. ACM Press, March 2004.

[15] G. Michaelson and K.Hammond. The Hume Language Definition and Report, Version 0.2. Technical report, Heriot-Watt University and University of St Andrews, Jan. 2002.

[16] Ministry of Defence. The procurement of safety critical software in defence equipment (part 1: Requirements, part 2: Guidance). Defence Standard 00-55, Issue 1, Ministry of Defence, Directorate of Standardization, 1991.

[17] N. Narasimhan, E. Teica, R. Radhakrishnan, S. Govindarajan, and R. Vemuri. Theorem proving guided development of formal assertions in a resource-constrained scheduler for high-level synthesis. *Formal Methods in System Design*, 19(3):237–273, 2001.