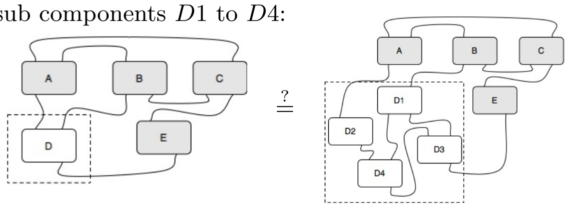
Preserving Coordination Properties when Transforming Concurrent System Components[*]

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

ABSTRACT

Complexity in concurrent or distributed systems can be managed by dividing component into smaller components. For example, suppose component D is replaced by an assembly of sub components $D1$ to $D4$:



While the new assembly may have the same *functional correctness* properties as the original component, the *coordination* properties of the whole system may have changed radically, as the additional processes must now be scheduled with attendant impact on the scheduling of the original processes. If the original system is non-deterministic or time dependent, then the system's functional properties may also change. A well known solution for managing large systems is to structure the components into sub-parts, an approach that was first taken by Harel [2] for finite state automata (FSA). By illustrating with the Hume programming language, we will argue for a similar approach for program transformation, where the overall structure is preserved by nesting new components inside a super-component.

Hume[1] explicitly separates *coordination* and *computation* concerns. It is based on autonomous *boxes* linked by *wires*, which are defined in the finite state *coordination language*. Transitions within a box is defined in the *expression language* by a list of *matches*, each of the form

$$pattern \rightarrow expression$$

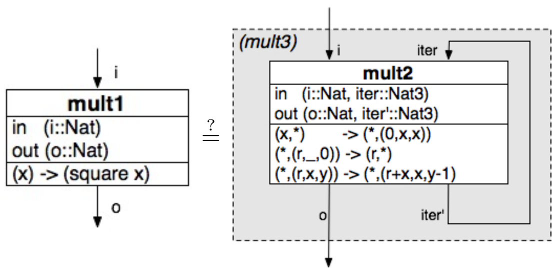
where each *pattern* is matched against the box input and the associated *expression* generates output by associated recursive actions. Hume targets safety-critical resource bounded systems – thus Hume programs are deterministic. This is achieved by an execution model which is based on a cyclical two phase execution: in the first phase each box is run once and attempts to consume input and generate output; and in the second phase the output changes are resolved in a unitary super-step.

Wires are single-buffered and partial matching is allowed, thus a box may fail on matching inputs and block when attempting to write to a full wire. In the first case the box enters a **Matchfail** state, while the latter induces a **Blocked** state. If not, the box is **Runnable**. Hume uses *lock-step* scheduling which proceeds as follows:

```
for ever
    for each Runnable or Matchfail box
        execute (box)
    super_step (each Runnable or Blocked box)
```

At the heart of Hume is the ability to statically cost time and space usage [1]. However, not all properties are costable, and thus Hume program development involves the key concept of *program transformation*, to reduce constructs into a more decidable representation. This often involves transforming from the expression to the coordination layer. For example, the well-known linear recursion to iteration transformation [4], is in Hume represented as recursion expression (`mult1`) to box iteration (`mult2`):

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



where the symbol '`*`' denotes ignore input or no output, '`_`' requires that input is present. Functional correctness is preserved in `mult2` so that with the same input (`i`) the same result is produced at the output (`o`). However, coordination properties are altered: `mult1` produces the result in 1 scheduling step, while `mult2` requires $N+1$ steps for a recursive call of depth N. A consequence of the '`*`' pattern is that Hume programs are time-dependent. Thus, replacing

[*]This work has been supported by EU FP6 EmBounded and a James Watt Scholarship. A full version of the paper is available at `http://www.macs.hw.ac.uk/techreps/`.

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil
Copyright 2008 ACM 978-1-59593-753-7/08/0003 ...$5.00.

`mult1` by `mult2` in an arbitrary program, may induce different matches in connected boxes, and so change the overall (functional and coordinational) behaviour of the program.

A solution is to structure boxes by allowing boxes to nest inside a box. We call this extension *Hierarchical Hume*. This allows us to split and merge boxes both vertically and horizontally, without changing the overall program structure. A box remains an input-output relation, without any state (side effects). Hence, when a *nesting box* (parent) terminates, all dangling state will be reset before the next cycle. Further, we need to know when a nesting box should start executing and when it should stop, and this must be specified by the programmer. For example, we can create a hierarchical box, by nesting the shaded area encapsulating `mult2` and the feedback wire (`iter'`$\rightarrow$`iter`). We call this box `mult3`. This requires the `i` and `o` wires to split into the ones (externally) connecting `mult3` and those (internally) connecting `mult3` and `mult2`. Finally, we want `mult3` to start executing, i.e. schedule it's children, when there is an input present (on external `i`) and terminate when there is a value on the internal `o` wire. This is achieved by a `(_) -> (_) match` in match. To achieve this scheduling, a **Terminated** state is separated from **Runnable**. This means that the box (in the execute phase) has terminated. Moreover, the scheduler is now nested and scheduling can be defined inductively. However, this requires two additional potential box states: **Execute** denotes that the children boxes (of the box) are in the execute phase; and **Super** denoting that the children are in the super-step phase. The inductive scheduler is then defined as:

```
schedule (boxes ,condition) ≜
    until condition
        for each Runnable or Matchfail box in boxes
            if box is nesting
                then schedule(children(box),termination_condition(box))
                else execute(box)
        super_step (each Terminated or Blocked box in boxes)
```

where *schedule(all top-level boxes,False)* is the top-level scheduler. Here, the termination condition is *False* since a Hume program never terminates. We have proven that this extension is conservative (see full paper) and updated the Hume interpreter with a prototype implementation of this scheduling. Experiments there has provided empirical evidence for the discussion above, and that hierarchies overcome these problems. It has also shown some efficiency gain due to reduced unnecessary scheduling.

We can also formally prove that the behaviour is preserved in a transformation, and we have found TLA [3] suitable for such proofs. Here, both program and properties are given in the same logic. `mult3` preserves the behaviour of `mult1` if it implements it, which is represented as logical implication. We will now outline this proof (It has also been verified with the TLC model checker). Let X be the input value of the component, i and o the internal input (`i`) and output (`o`) wires of `mult3`, and f the feedback loop (`iter'`$\rightarrow$`iter`) of `mult2`. Firstly, the same wires must always be consumed, which is trivial since there are only one input, and both boxes match iff input is available. Secondly, on termination the internal output wire (o) of `mult3` must hold the same value, as produced by `mult1`. A box terminates when o has a value, i.e. is not empty, which is written $o \neq \perp$. By unfolding `square` this is written as:

$$I \triangleq \Box\big(o \neq \perp \Rightarrow o = \texttt{mult}\ 0\ X\ X\big)$$

The proof of I requires the following invariants:

$$I_1 \triangleq \Box\big(o = \perp \Rightarrow i \neq \perp \Rightarrow\ i = X\big)$$
$$I_2 \triangleq \Box\big(o = \perp \Rightarrow f \neq \perp \Rightarrow \texttt{mult}\ 0\ X\ X = \texttt{mult}\ f[1]\ f[2]\ f[3] \wedge f[2] = X\big)$$

$f[n]$ accesses the n^{th} element of tuple f, and I_2 assumes I_1. Invariant proofs follow an induction principle where the invariant must initially hold (base case), and must be preserved in a transition (step case). Initially, I,I_1 and I_2 holds since $i = o = f = \perp$. The majority of the step cases will not change the values of i, o or f. Hence, we will only discuss the non-trivial steps that alter the values. i, and thus I_1, is by definition, only updated when `mult3` matches the input, and is then set to the input value X. Hence, I_1 holds.

I_2 is the "loop invariant" of the "iteration", and depends on f. There are two cases where `mult2` updates f: firstly, when $i \neq \perp$, then wire f is set to $(0, i, i)$. This is the "entry step" of the "loop". Since $i = X$ by I_1, we have $f = (0, X, X)$ thus I_2 holds since

$$(\texttt{mult}\ 0\ X\ X = \texttt{mult}\ 0\ X\ X\ \wedge X = X)$$

Secondly, in the "iteration step" of the "loop" we assume (IH): $\texttt{mult}\ 0\ X\ X = \texttt{mult}\ f[1]\ f[2]\ f[3]\ \wedge\ f[2] = X$. The values are then updated such that we must prove

$$\texttt{mult}\ 0\ X\ X = \texttt{mult}\ (f[1] + f[2])\ f[2]\ (f[3] - 1)\ \wedge\ f[2] = X$$

By rewriting the definition of `mult` from right to left we have

$$\texttt{mult}\ 0\ X\ X = \texttt{mult}\ f[1]\ f[2]\ f[3]\ \wedge\ f[2] = X.$$

which can be unified with the assumption (IH). Hence I_2 holds.

I is verified using I_2. o is only written to at the "exit step" of the loop. Pattern matching in `mult2` ensures that $f[3] = 0$ in this case. o is then given the value of $f[1]$. Thus, by I_2, we have:

$$\texttt{square}\ X = \texttt{mult}\ 0\ X\ X = \underbrace{\texttt{mult}\ f[1]\ f[2]\ 0}_{f[1]\ by\ def.\ of\ \texttt{mult}} = f[1] = o$$

which shows that I holds. This value is copied to the output wire in the super-step phase, thus the behaviour is also preserved in this phase, which completes the proofs. Note that this is a general approach, which enables proof automation.

The main difference between this work and *statecharts* [2], is that we are introducing *hierarchies of transitions* rather than *hierarchies of transitions*. However, due the FSM model the work is comparable, creating an AND super-step.

1. REFERENCES

[1] 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.

[2] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[3] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[4] Z. Manna. *Mathematical Theory of Computing.* McGraw-Hill, 1974.