IST-510255
EmBounded
Automatic Prediction of Resource Bounds for Embedded Systems

Specific Targeted Research Project (STReP)
FET Open

D26 (WP7d): Compiler support tools

Due date of deliverable: July 2007
Actual submission date: November 2007

Start date of project: 1st March 2005 Duration: 48 months

Lead contractor: Heriot-Watt University Revision: 1.8

Purpose: To describe `humemc`, a model checker for Hume programs.

Results: We have implemented a program that accepts a subset of FSM-Hume programs annotated with specifications of temporal properties. The program verifies that the properties holds for the given program. It also verifies that the program does not contain any deadlocks. The off-the-shelp SPIN model checker is used for the verification. We describe the Hume to SPIN model checker, show how to use the tool, discuss verification results, and provides data on execution time and workload performance for the underlying model checker.

Conclusion: We have been able to implement `humemc` using already existing Hume constructs. The tool has been applied to several examples, which has given us a better understanding of potential errors in the Hume coordination layer.

Project co-funded by the European Commission within the 6^{th} Framework Programme (2002-06)			
Dissemination Level			
PU	Public		※
PP	Restricted to other programme participants	(including the Commission Services)	
RE	Restricted to a group specified by the consortium	(including the Commission Services)	
CO	Confidential only for members of the consortium	(including the Commission Services)	

Compiler support tools

Gudmund Grov <gudmund@macs.hw.ac.uk>
School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, Scotland

Abstract

We describe the `humemc` tool for model checking of Hume programs. The tool accepts a subset of annotated FSM-Hume programs, where the annotations are defined using current Hume constructs, and translates these to Promela. Program and specification are then verified using the SPIN model checker.

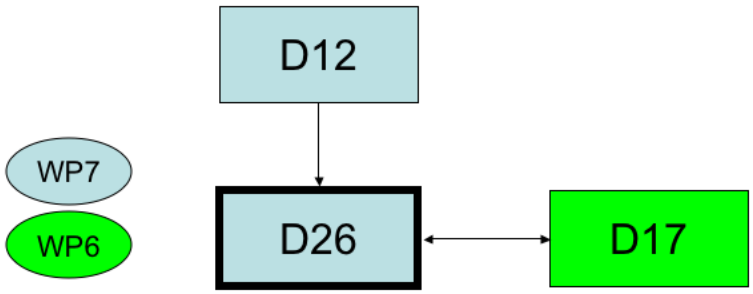
Here, we give an introduction to SPIN, define the source and specification language for our tool, and show how this is embedded into SPIN. Finally, we show how the tool works, describe some results, and provide some empirical evidence for the correctness of our approach using a small set of examples.

The positioning of this deliverable is discussed in Section 1.

Major Revisions		
Revision	Date	Changes
1.8	14 Aug. 2009	table-of-contents (addressing Review Report Year 4)
1.6	30 July 2008	relationship to D17 in Sec 1 (addressing Review Report Year 3)
1.5	30 Nov. 2007	initial version

Contents

1 Positioning of Deliverable D26



D12	WP7a	Hume Formal Semantics
D26	WP7d	Compiler support tools
D17	WP6a	Assertion language

This deliverable describes how Hume source programs are translated into Promela so that properties may be checked using model-checking. It builds on deliverable D12(WP7a) [Jos06], which defines the formal Hume semantics.

Furthermore, there is a mutual dependency with D17(WP6a) [LG07] which defines an assertion language for Hume. To achieve a better integration between the expression and coordination layer, the Temporal Logic of Actions [Lam94] is used express assertions of the coordination layer in D17, instead of the LTL logic used here. However, the assertion language is based on the experience from the work described here, and the specification language in D17 and here are fairly similar, thus showing the mutual dependency.

2 Introduction

We have implemented `humemc`, a model checker for Hume programs. The tool accepts a subset of FSM-Hume as a source language, together with a set of annotations which specifies temporal properties that must hold for the given program. The annotations are defined using the existing Hume `data` and `expression` constructs. The Hume program and annotations are translated into PROMELA and LTL, which are the modelling and specification language/logic for the SPIN model checker. SPIN is then used to verify that the program cannot induce a deadlock, and that the specified properties holds for the given program.

The report is organised as follows: First, we define the source language for the tool, followed by a short introduction to SPIN and PROMELA. We then describe how the Hume program is translated into PROMELA. This is followed by a description of the specification language, and how the annotations are translated into PROMELA/LTL. We then show how to use `humemc` and applies it to a set of examples. There, we give some results on the time and space consumption of our tool, and typical examples of properties that can be verified.

2.1 Model Checking

Model checking [CGP99] is a fully automatic verification technique which can verify temporal properties of a system. The system, or model, M is represented as a Kripke structure over a quadruple $M = (S, S_0, R, L)$:

1. S is a finite set of states;
2. $S_o \subseteq S$ is the set of initial states;
3. $R \subseteq S \times S$ is a total transition for the states;

4. $L : S \rightarrow 2^{AP}$ is a labelling function that labels each state $s \in S$ with teh set of atomic propositions that holds in that states.

If liveness properties are used then the Kripke structure must be extended with a fairness constraint $F \subseteq 2^S$. The model checking problem for a (temporal) property π of the model is defined as:

$$M \models \pi$$

The model checker will then search (the entire state space of) M for a violation of π. The advantage of model checking is that it is fully automatic, so no user interaction is required. Most model checkers also provides counter-examples when a violation of π is found, thus helping the debugging process. However, all model checkers suffers from the *state space explosion problem*. This occurs when the model/system is too big, and the model checker runs out of memory.

2.2 Main Contributuions of this Deliverable

The main contribution of this deliverable is to add model checking support to the Hume tool set. It also provides a specification language for the coordination layer of Hume, which is novel by specifying events rather than reason with program variables.

The model checker targets a subset of FSM-Hume, which focuses more on the coordination aspects, rather than the expression aspects, of Hume. However, it is still a model checker for a functional programming language. Except for work with Erlang [NFG01], we are not familiar with other model checkers for functional languages.

Finally, the deliverable contains a set of examples where the tool was used to verify properties of some programs and, more importantly, find errors in other programs. Thus, we provide empirical evidence for the tool, and show the importance of formal verification in general.

3 The Source Language

The source language for the model checker is a subset of FSM-Hume, essentially HW-Hume extended with integers and words. Consequently, no data type or functions are allowed, thus the program is mainly just boxes, and the accepted box syntax is shown in Figure 1.

4 SPIN and PROMELA

SPIN [Hol03] is designed to simulate and verify asynchronous process systems, with a focus on the interaction between processes. It is typically used to check for *deadlocks*, *unspecified receptions* and *inexcusable code*, and supports both a simulation and a full blown verification mode. SPIN models are written in the PROMELA modelling language. It is an imperative language, with a syntax that resembles C.

A SPIN model consists of *processes*, *message channels* and *variables*, where processes communicates over global variables or channels. Variables and channels can also be used locally by a process.

A *process* is a finite state automata (FSA), and is declared using the `proctype` keyword. It can either be instantiated by prefixing it with the `active` keyword, or explicitly with the `run` statement. A SPIN model may also have a declared `init` process, which works the same way as `main` in C. SPIN works by always attempting to execute the next executable statement of an arbitrary process, hence each process has it's own program counter. If a process reaches a statement that cannot be executed and hence is blocked, the process also blocks. To control the flow inside a process, repetitions, goto and case/if statements are supported.

A *message channel* has type `chan`. It has a defined buffer length and type. For example,

```
chan q = [ 5 ] of { byte , int }
```

⟨boxdecl⟩ ::= **box** ⟨boxid⟩ ⟨prelude⟩ ⟨body⟩

⟨prelude⟩ ::= **in** (⟨inout⟩ , ⟨inoutlist⟩)
out (⟨inout⟩ , ⟨inoutlist⟩)

⟨inoutlist⟩ ::= ⟨inout$_0$⟩ **,** ... **,** ⟨inout$_n$⟩ n ⩾ 0

⟨inout⟩ ::= ⟨varid⟩ , ⟨varidlist⟩ **::** ⟨type⟩

⟨varidlist⟩ ::= ⟨varid$_0$⟩ , ... , ⟨varid$_n$⟩ n ⩾ 0

⟨type⟩ ::= ⟨basetype⟩
| **(** ⟨types⟩ **)**

⟨types⟩ ::= ⟨type$_0$⟩ **,** ... **,** ⟨type$_n$⟩ n ⩾ 1

⟨basetype⟩ ::= **word** ⟨precision⟩
| **int** ⟨precision⟩
| **char**

⟨precision⟩ ::= **1** | ... | **64**

⟨body⟩ ::= (**match** | **fair**) ⟨boxmatches⟩

⟨boxmatches⟩ ::= ⟨match⟩ | ⟨matches⟩

⟨matches⟩ ::= ⟨match$_0$⟩ | ... | ⟨match$_n$⟩ n ⩾ 0

⟨match⟩ ::= ⟨patt⟩ → ⟨expr⟩

⟨patt⟩ ::= ⟨constant⟩
| ⟨varid⟩
| **(** ⟨patts⟩ **)**

⟨patts⟩ ::= ⟨patt$_0$⟩ **,** ... **,** ⟨patt$_n$⟩ n ⩾ 1

⟨expr⟩ ::= ⟨constant⟩
| ⟨varid⟩
| **(** ⟨exprs⟩ **)**

⟨exprs⟩ ::= ⟨expr$_0$⟩ **,** ... **,** ⟨expr$_n$⟩ n ⩾ 1

Figure 1: Abstract Syntax for a Box of the Source Language

declares a channel called 'q' which can hold 5 messages at a given time. Each message consists of a `byte` and an `int`. Messages are sent using the '`!`' operator and received using the '`?`' operator. Rendezvous communication is achieved by using a buffer length of `0`.

PROMELA support standard data types like `bit`,`bool` and `int`. It can also declare sets of symbolic constants using `mtype`, which eases specifications. *Constants* are declared using a C style macro `# define`.

There are several ways of specifying the desired and error behaviour of the model, however, we will only use it's linear temporal logic (LTL). We can then specify properties such as P is always true ($\Box P$), Q is eventually true ($\Diamond Q$), and every time P is true, then at some later point Q holds ($\Box(P \Rightarrow \Diamond Q)$)

5 Translating Hume into PROMELA

A Hume box is represented as a PROMELA process, and this is the most substantial part of the translation. The translation is formalised by a set of translation rules, and some selected rules for a box translation is shown in Figure 2. Rules for declaring variables, for implementing boxes that use unfair matching, and for the creation of individual matches are omitted here (all the rules are listed in [Gro04])

Families of translation rules have been collected into *rule classes*. Three sets of rule classes are used by all the other classes: *data types*, *data values* and *names*. The first of these is translated by the *Types* class and returns a corresponding PROMELA representation of the data types allowed in Hume. Translating an instance of a type, e.g. the value of a variable, is achieved in the *Values* class. Finally, the *Names* class handles the translation of the names used, like variables and box identifiers. This for example avoids conflicts with PROMELA keywords.

The channels used for *interprocess communication* are created in the *Chan* class. One channel is created for each `wire` in Hume.

19: $\mathcal{N}ames[\![$ ⟨terminal⟩ $]\!] \mapsto$ ⟨terminal⟩1
20: $\mathcal{N}ames[\![$ ⟨string⟩'⟨string⟩$]\!] \mapsto$ ⟨string⟩q⟨string⟩
35: $\mathcal{P}roc[\![$ ⟨boxdecl⟩ $]\!] \mapsto \mathcal{P}relude[\![$ ⟨boxdecl⟩ $]\!]$ { $\mathcal{V}ars[\![$ ⟨boxdecl⟩ $]\!]$ $\mathcal{B}ody[\![$ ⟨boxdecl⟩ $]\!]$ }
36: $\mathcal{P}relude[\![$ ⟨boxdecl⟩ $]\!] \mapsto \mathcal{P}relude[\![$ name ⟨boxdecl⟩ $]\!]$ $\mathcal{P}relude[\![$ param ⟨boxdecl⟩ $]\!]$
37: $\mathcal{P}relude[\![$name box ⟨boxid⟩ ⟨prelude⟩ ⟨body⟩ $]\!] \mapsto$ **proctype** $\mathcal{N}ames[\![$ ⟨boxid⟩ $]\!]$
38: $\mathcal{P}relude[\![$ param box ⟨boxid⟩ ⟨prelude⟩ ⟨body⟩$]\!] \mapsto \mathcal{P}relude[\![$ ⟨prelude⟩ $]\!]$
39: $\mathcal{P}relude[\![$ in (⟨inoutlist$_1$⟩) out (⟨inoutlist$_2$⟩) $]\!] \mapsto$ **(chan** $\mathcal{P}relude[\![$ in ⟨inoutlist$_1$⟩ $]\!]$ **,** $\mathcal{P}relude[\![$ out ⟨inoutlist$_2$⟩ $]\!]$ **, control)**
40: $\mathcal{P}relude[\![$in ⟨inout⟩ , ⟨inoutlist⟩ $]\!] \mapsto \mathcal{P}relude[\![$ in ⟨inout⟩ $]\!]$, $\mathcal{P}relude[\![$ in ⟨inoutlist⟩ $]\!]$
41: $\mathcal{P}relude[\![$in ⟨varid⟩ , ⟨varidlist⟩ :: ⟨exprtype⟩ $]\!] \mapsto \mathcal{P}relude[\![$ in ⟨varid⟩ $]\!]$, $\mathcal{P}relude[\![$ in ⟨varidlist⟩ :: ⟨exprtype⟩ $]\!]$
42: $\mathcal{P}relude[\![$ in ⟨varid$_{\langle i \rangle}$⟩ $]\!] \mapsto$ **in**⟨i⟩
43: $\mathcal{P}relude[\![$out ⟨inout⟩ , ⟨inoutlist⟩ $]\!] \mapsto \mathcal{P}relude[\![$ out ⟨inout⟩ $]\!]$, $\mathcal{P}relude[\![$ out ⟨inoutlist⟩ $]\!]$
44: $\mathcal{P}relude[\![$out ⟨varid⟩ , ⟨varidlist⟩ :: ⟨exprtype⟩ $]\!] \mapsto \mathcal{P}relude[\![$out ⟨varid⟩ $]\!]$, $\mathcal{P}relude[\![$ out ⟨varidlist⟩ :: ⟨exprtype⟩$]\!]$
45: $\mathcal{P}relude[\![$out ⟨varid$_{\langle i \rangle}$⟩ $]\!] \mapsto$ **out**⟨i⟩
55: $\mathcal{B}ody[\![$ box ⟨boxid⟩ ⟨prelude⟩ ⟨body⟩ $]\!] \mapsto$ **start: control?BEGIN;** $\mathcal{B}ody[\![$ ⟨prelude⟩ ⟨body⟩ $]\!]$ **finish: control!END; goto start;**
56: $\mathcal{B}ody[\![$ ⟨prelude⟩ fair ⟨boxmatches⟩ $]\!] \mapsto \mathcal{F}air[\![$ ⟨prelude⟩ fair ⟨boxmatches⟩ $]\!]$
58: $\mathcal{F}air[\![$⟨prelude⟩ fair ⟨boxmatches⟩ $]\!] \mapsto$ **if** $\mathcal{F}air[\![$⟨prelude⟩ ⟨boxmatches⟩ $]\!]$ **::else goto finish fi;**
59: $\mathcal{F}air[\![$⟨prelude⟩ ⟨match⟩ | ⟨matches⟩ $]\!] \mapsto \mathcal{F}air[\![$⟨prelude⟩ ⟨match⟩ $]\!]$ $\mathcal{F}air[\![$⟨prelude⟩ ⟨matches⟩ $]\!]$
60: $\mathcal{F}air[\![$⟨prelude⟩ ⟨match⟩ $]\!] \mapsto$ **::** $\mathcal{M}atch[\![$ ⟨prelude⟩ ⟨match⟩ $]\!]$
64: $\mathcal{M}atch[\![$ in (⟨inoutlist$_1$⟩) out (⟨inoutlist$_2$⟩ ⟨patt⟩ → ⟨expr⟩ $]\!]$
$\mapsto$ **(** $\mathcal{P}att[\![$ ⟨inoutlist$_1$⟩ ⟨patt⟩$]\!]$ **)** → $\mathcal{E}xp[\![$ ⟨inoutlist$_1$⟩ ⟨patt⟩ $]\!]$ $\mathcal{E}xp[\![$⟨inoutlist$_2$⟩ ⟨expr⟩ $]\!]$

We show all the translation rules needed to translate a Hume box into its PROMELA representation, apart from those for matches. The rule numbering used here is the same as for the full set of rules (see [Gro04]).

Figure 2: Translation Rules

PROMELA processes are created in the $\mathcal{P}roc$ class. The translation is quite cumbersome, and thus has been divided into three different parts. The $\mathcal{P}relude$ sub-class is used to create the prelude of the process, i.e. the process signature. In Hume variables can be created/bound in a match. In PROMELA, on the other hand, all variables must be declared before they can be used. The $\mathcal{V}ars$ sub-class declares these variables. The main body, i.e. the matches, are translated in the $\mathcal{B}ody$ sub-class. Fair matching is dealt with in the $\mathcal{F}air$ sub-class and unfair matching in the $\mathcal{U}nfair$ sub-class.

In order to ensure determinism, Hume specifies the order boxes are executed within a single scheduling superstep. This must be mapped into the unrestricted scheduling order used by PROMELA, using a designated PROMELA process which is created in the $\mathcal{C}ontrol$ class, and which uses the channels created in the $\mathcal{C}ontrolChan$ class to communicate with the processes. These are rendezvous channels between the control box and each other box.

The translation currently considers only stream-based input and output, with Hume *input* and *output* streams being wired to boxes as required. Since input streams allow the model to be manipulated by the execution environment, and we require a *closed environment* in order to verify properties in the model checker, the operation of each input stream is simulated by a special PROMELA process. These are created in the $\mathcal{I}n\mathcal{S}tream$ class. Output streams do not have this problem, and are therefore modelled as output-producing PROMELA processes, created in the $\mathcal{O}ut\mathcal{S}tream$ class.

One process is created initially by the *Init* class, the `init` process. The `init` process is used to instantiate all processes with the correct channels, and to initialise channels if an initial value is specified in the Hume program

We will use a half adder implemented in Hume to illustrate the translation. The Hume source code is shown in Figure 3, while Figure 4 shows the resulting PROMELA model of the translation. The full PROMELA model is 146 lines of code – thus the figure is restricted to key elements.

```
type Bit = word 1;
stream output to "std_out";

box gen
in (t::int 64)
out (t'::int 64,x,y::Bit)
match
 0 -> (1,0,0) |
 1 -> (2,0,1) |
 2 -> (3,1,0) |
 3 -> (0,1,1);

box xor
in (x,y::Bit)
out (z::Bit)
match
 (0,0) -> 0 |
 (0,1) -> 1 |
 (1,0) -> 1 |
 (1,1) -> 0;

box fanout
in (x,y::Bit)
out (x1,y1,x2,y2::Bit)
match
 (x,y) -> (x,y,x,y);

box and
in (x,y::Bit)
out (z::Bit)
match
 (0,0) -> 0 |
 (0,1) -> 0 |
 (1,0) -> 0 |
 (1,1) -> 1;

box show
in (z,c::Bit)
out (zc::(Bit,Bit,char))
match
 (z,c) -> (z,c,'\n');

wire gen(gen.t' initially 0)(gen.t,fanout.x,fanout.y);
wire fanout(gen.x,gen.y)(xor.x,xor.y,and.x,and.y);
wire xor(fanout.x1,fanout.y1)(show.z);
wire and(fanout.x2,fanout.y2)(show.c);
wire show(xor.z,and.z)(output);
```

Figure 3: Hume code for Half Adder

The large difference in the source code size of the two representations is due to Hume's *deterministic* execution model – while Spin is *non-deterministic*. Moreover, Hume boxes are atomic in the execution model, meaning a box finishes a cycle before the next box starts. These properties are encoded by introducing semaphores in the `Control` process.

Lines 1 and 2 of Figure 4 show the skeleton of the channels. A wire is semantically equivalent to a one-buffer message channel, and is shown on line 3. The only difference is that a channel is not restricted to be point-to-point. Rather, this is ensured by the translation.

The `xor` and `show1`[1] PROMELA processes on lines 6 and 26 are full representations of the corresponding Hume boxes. Since Hume programs do not terminate, the body of the processes are embedded in an infinite loop.

Each process receives as arguments its input and output channels together with a control channel. The main body of the process (e.g. line 10 to 22) consists of non-deterministic `if-else` clauses – where fair matching is achieved through nesting. Each Hume match is an option in the statement, prefixed by `::`, and is divided into three parts:

$$(\ \langle \textit{check inputs} \rangle\ )\ \rightarrow\ \langle \textit{read input} \rangle;\ \langle \textit{write output} \rangle$$

where inputs are checked by wrapping the arguments of the `?` operator into square brackets, and where each statement is separated by &&. If the check/guard succeeds, the right hand side of $\rightarrow$ is evaluated. Here the values are read and removed from the channels, and the output channels are written to using the `!` operator. If it fails the next option is checked. In Hume, *variables* are implicitly declared in the pattern of a match.

[1] We use `show1` rather than `show`, since the latter is a reserved word in PROMELA.

```
 1: mtype = { BEGIN , END };
 2: chan <procname>_control = [0] of { mtype };
   ...
 3: chan <procname>_<varname> = [1] of { <type> };

 4: proctype gen(chan in1, ... ,  control ){ ... }
 5: proctype fanout(chan in1,...,control){ ... }

 6: proctype xor(chan in1,in2,out1,control){
 7:
 8:  start:
 9:   control?BEGIN;
10:   if
11:   :: (in1?[0] && in2?[0]) -> in1?0;in2?0;out1!0;
12:   :: else if
13:       ::(in1?[0] && in2?[1]) -> in1?0;in2?1;out1!1;
14:       :: else if
15:          ::(in1?[1] && in2?[0]) -> in1?1;in2?0;out1!1;
16:          :: else if
17:             ::(in1?[1] && in2?[1]) -> in1?1;in2?1;out1!0;
18:             :: else goto finish
19:              fi;
20:          fi;
21:       fi;
22:   fi;
23:  finish:
24:   control!END; goto start;}

25: proctype and(chan in1,...,control){ ...}

26: proctype show1(chan in1,in2,out1,control){
27:   bit z; bit c;
28:  start:
29:   control?BEGIN;
30:   if
31:   :: (in1?[z] && in2?[c]) -> in1?z;in2?c;out1!z,c,-1;
32:   :: else goto finish
33:   fi;
34: finish:
35:   control!END; goto start;}


36: active proctype Control(){
37:   do
38:   :: <procname>_control!BEGIN; <procname>_control?END; ...
39:   od;}

40: active proctype output(){ ... }
41: init{ ... }
```

Figure 4: Half Adder in Promela

6 Specifying Temporal Properties in Hume

The specification language used to specify Hume properties is based on SPINs LTL logic. In imperative language with assertions, such as Spark Ada or JML (an extended version of Java), properties are specified by referring to values of the variables. Similarly, properties in SPIN refer to values of propositional variables. In Hume, in contrast, all state is held on wires. However, there is no way to refer directly to the wires: wires can only be reached via the boxes that they connect. Thus, there are (at least) two ways of specifying and reasoning about Hume properties in the coordination layer:

1. By using the wires, and reasoning about properties by reasoning about the values on the wires. For example, a wire can be accessed using a <boxid>.<varid> notation.

2. By using the matches, and reasoning about how they are triggered.

The first approach is more abstract since we ignore the inside part of a box, while the latter is closer to Hume and more directed towards the functional programmer. Both approaches have their pros and cons, and can even be combined. However, since we believe that the latter is easier to understand for the Hume programmer, we have thus followed this approach. Furthermore, to keep the specification language as simple as possible, we have chosen not to combine it with a ”wire based” language.

Programs are normally annotated by either special comments or with a separate specification file. We have instead exploited the existing Hume language constructs. The specification language is defined using the following three inductive data types:

```
data Spec = TEMP Expr;
data Expr = IMPLIES Expr Expr
          | AND Expr Expr
          | OR Expr Expr
          | ALWAYS Expr
          | EVENTUALLY Expr
          | NOT Expr
          | UNTIL Expr Expr
          | BOX string Pattern;
data Pattern = CONST string
             | IGNORE | CONSUME
             | TUPLE [Pattern];
```

This definition must be included in the source code of all programs that are verified by our tool. A property is expressed using Hume's `expression` construct, where `expression e` evaluates `e` without changing the behaviour of the program. A property is expressed by

```
expression TEMP <property> ;
```

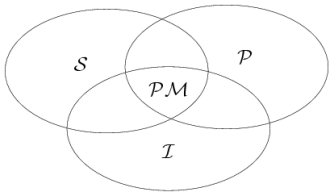
For example, we want to express the property that if the generator (`gen`) is `0` then the the sum will be `0`, with no carry bit – i.e. both inputs in the `show` box are `0`. Using standard LTL notation, this is expressed as:

$$\Box(gen\,0 \rightarrow \Diamond show(0,0)) \tag{1}$$

and is expressed in Hume as

```
expression (TEMP (ALWAYS(IMPLIES (BOX ‘‘gen’’ (TUPLE [CONST ‘‘0’’]))
                 (EVENTUALLY (BOX ‘‘show’’ (TUPLE [CONST ‘‘0’’, CONST ‘‘0’’]))))));
```

The translation of these events is not as direct as the translation of the Hume programs. Informally, we can look at a match as a *rule* – and the body of a box as a set of rules in a case expression. A box has a set of input wires, which may or may not have available input. We let $\mathcal{I}$ denote this set of available input. The premise (pattern) p of a rule specifies the required inputs, and if $p \in \mathcal{I}$ for a rule being evaluated, the rule fires. An event in the specification language is identified as a ⟨BoxId⟩ ⟨Pattern⟩ couple. We let s represent ⟨Pattern⟩ and assume s and p belong to the same box. The relationship between the set of all s ($\mathcal{S}$), the set of all p ($\mathcal{P}$), and the available inputs $\mathcal{I}$ for a box can be illustrated by a Venn diagram:



where $\mathcal{PM}$ ($\mathcal{S} \cap \mathcal{P} \cap \mathcal{I}$) illustrates a potential event/match. A premise, s or p, is a tuple of elements. An element v is not necessarily a constant – which complicates the intersection. If it is a variable or a consume '_' element the corresponding input $\mathcal{I}_v$ must exist. If $v \in Const$ then $\mathcal{I}_v$ must in addition equal v. The ignore element '$*$' is the only elements that succeeds if $\mathcal{I}_v$ does not hold any value. This comparison with the input $\mathcal{I}$ is represented by the function *check*:

$$check(v, \mathcal{I}) \triangleq \begin{cases} v \in \mathcal{I} & \begin{cases} v \in Const \ \wedge \ v \neq \mathcal{I}_v & \Rightarrow \ False \\ otherwise & \Rightarrow \ True \end{cases} \\ v \notin \mathcal{I} & \begin{cases} v \in Ignore & \Rightarrow \ True \\ otherwise & \Rightarrow \ False \end{cases} \end{cases}$$

where the existence of $\mathcal{I}_v$ is first investigated – followed by investigating the type of v. A set $\mathcal{P}$ of premises p intersects with the input $\mathcal{I}$ for the premises p when all its variables v passes $check(v, \mathcal{I})$:

$$\mathcal{P} \cap \mathcal{I} \triangleq \bigcup_{p \in \mathcal{P}} \forall v.\ v \in p \rightarrow check(v, \mathcal{I})$$

$\mathcal{PM}$ denotes the set of potential events for a box. We are interested in the occurrence of an event, expressed as $\mathcal{M}$. This happens if a rule with premise p fires and p is a potential event $\mathcal{PM}$:

$$\mathcal{M} \triangleq \lambda p.\ fires(p) \ \wedge \ p \in \mathcal{PM}$$

where $fires$ depends on two conditions. Firstly, all the required inputs must be available, $p \in \mathcal{I}$. Secondly, no other rules can have matches earlier in that run. This depends on the patterns of the other rules, together with the type of matching, i.e. fair or unfair. To ease reading, the syntax $\mathcal{M}^b$ ($\mathcal{PM}^b/p^b/\ s^b$) is used to express that the event/premise is part of box b.

We will illustrate the translation with the example shown in (1). The example has two properties ($\mathcal{M}$). We use a boolean flag to represent each of them – initially set to `false`[2].

```
bool prop1 = false;
bool prop2 = false;
```

For a property $\mathcal{M}$, all rules with a premise p must be identified and, depending on their position in the formula's term tree, the property is set to `true` when the rule fires.

$$\mathcal{M} \rightarrow (\texttt{prop}_{\mathcal{M}} \texttt{ = true})$$

[2]As a consequence $\Box\mathcal{M}$ will always fail. If we want to support this, further analysis is required.

In the example p is `0` in the `gen` box and `(s,z)` in the `show` box. In the `gen` box one match satisfies this condition – and the corresponding option in the `gen` `proctype` is updated (**bold font** is used for the added annotation in the PROMELA model):

```
(in1?[0]) -> prop1=true; in1?0;out1!1;out2!0;out3!0;
```

The case for `prop2` is more complicated. The property refers to the `show` box which holds one match, where p yields `(s,z)`. The corresponding specification s yields `(0,0)` – making p more general than s. Predicate *check* asserts that if an element v of a premise is a constant, the corresponding input $\mathcal{I}_v$ must be equal to v. If it is a variable it is only required that $\mathcal{I}_v$ exists, i.e. that there is some input on the corresponding channel. Since p holds, we must check that `s` and `z` are `0`, before we trigger the event (by setting `prop2` to `true`), creating the following righthand side of the match:

```
if
  ::  (in1?[0] &&  in1?[0]) -> prop2=true;
  ::  else skip;
fi; in1?z; in2?c; out1!z,c,-1;
```

The if-clause that we have introduced ensures that event $\mathcal{M}$ only occurs if $\mathcal{I}$, $\mathcal{S}$ and $\mathcal{P}$ intersect. Since we are reasoning about causal properties the conclusion of an implication is only interesting if the premise already holds, i.e. $\mathcal{M}_{premise}$ has to occur before $\mathcal{M}_{conclusion}$ is considered . This means that only if `prop1` has already occurred, i.e. it is set to `true`, will `prop2` occur. The righthand-side of the match clause for `show` is then as follows:

```
if
  ::  (in1?[0] &&  in1?[0])
             -> if
                  ::  (prop1) -> prop2=true;
                  ::  else skip;
                fi;
  ::  else skip;
fi; in1?z; in2?c; out1!z,c,-1;
```

In order to enforce a correct semantics for these formulae we need to know when to reset a property to `false`. This depends on the position of the expression in the formula's term tree. In an implication, which is the case for the example, we require that `prop1` must be `true` before `prop2` can occur. Consequently, we will not set `prop1` back to `false` when it is triggered. When `prop2` occurs we have proven that the formula holds (at some point in the run), and is reset to verify the formula for the remaining parts of the run. Both `prop1` and `prop2` are set back to `false` in this case. We can now show the final version of the right hand side of the match:

```
if
  ::  (in1?[0]&& in1?[0])
             -> if
                  ::(prop1) -> prop2=true;
                               prop1=false;prop2=false;
                  ::  else skip;
                fi;
  ::  else skip;
fi; in1?z; in2?c; out1!z,c,-1;
```

In order to reason about the properties, we must define a set of *propositional variables* for each event of the defined properties – which is $True$ if corresponding property is `true`. In our example two propositional variables are created – related to if `prop1==true` and `prop2==true`:

```
#define p1 (prop1 == true)
#define p2 (prop2 == true)
```

We are then left at proving the property(See (1)):

$$\Box(p1 \rightarrow \Diamond p2)$$

This LTL-formula is translated into a Büchi automata in SPIN – which is then used in the verification process. Note that if a program contains several temporal formulas, the conjunction of them are verified.

The rules describing the general case for when an event is triggered and when properties are set back to `false` are formalised in the functions of Figure 5 and 6. If the function returns $True$ it denotes the triggering of a match or resetting of a formula respectively.

$$\begin{array}{lcl} match_e[\![\ \mathcal{E}\ ]\!] & \triangleq & \textbf{if } \mathcal{E} \in \{\mathcal{E}_1 \rightarrow \mathcal{E}_2, \mathcal{E}_1\ \mathrm{U}\ \mathcal{E}_2\} \textbf{ then} \\ & & \quad \textbf{if } e \in \mathcal{E}_2 \textbf{ then } sem_e[\![\ \mathcal{E}_1\ ]\!] \\ & & \quad \textbf{else } True \\ & & \textbf{else } True \end{array}$$

Figure 5: Triggering of a Match

Figure 5 shows how an event is triggered. The function assumes that $\mathcal{M}$ is already satisfied. If the event is the conclusion of an implication or the "until" operator, it is required that the premise already holds, i.e. the $\mathcal{M}_{premise}$ has already occurred. Function *sem* from Figure 6 is used to check if the premise holds.

$$\begin{array}{lllcl} (1) & sem_e[\![\ \mathcal{E}_1 \rightarrow \mathcal{E}_2\ ]\!] & \triangleq & \textbf{if } e \in \mathcal{E}_1 \textbf{ then } False \\ & & & \textbf{else } sem_e[\![\ \mathcal{E}_1\ ]\!] \wedge sem_e[\![\ \mathcal{E}_2\ ]\!] \\ (2) & sem_e[\![\ \mathcal{E}_1 \wedge \mathcal{E}_2\ ]\!] & \triangleq & sem_e[\![\ \mathcal{E}_1\ ]\!] \wedge sem_e[\![\ \mathcal{E}_2\ ]\!] \\ (3) & sem_e[\![\ \mathcal{E}_1 \vee \mathcal{E}_2\ ]\!] & \triangleq & sem_e[\![\ \mathcal{E}_1\ ]\!] \vee sem_e[\![\ \mathcal{E}_2\ ]\!] \\ (4) & sem_e[\![\ \mathcal{E}_1\ \mathrm{U}\ \mathcal{E}_2\ ]\!] & \triangleq & \textbf{if } e \in \mathcal{E}_1 \textbf{ then } False \\ & & & \textbf{else } sem_e[\![\ \mathcal{E}_1\ ]\!] \wedge sem_e[\![\ \mathcal{E}_2\ ]\!] \\ (5) & sem_e[\![\ \Box\mathcal{E}\ ]\!] & \triangleq & sem_e[\![\ \mathcal{E}\ ]\!] \\ (6) & sem_e[\![\ \Diamond\mathcal{E}\ ]\!] & \triangleq & sem_e[\![\ \mathcal{E}\ ]\!] \\ (7) & sem_e[\![\ \neg\mathcal{E}\ ]\!] & \triangleq & sem_e[\![\ \mathcal{E}\ ]\!] \\ (8) & sem_e[\![\ \mathcal{M}\ ]\!] & \triangleq & prop(\mathcal{M}) \end{array}$$

Figure 6: Semantics for Reset Properties

Function *sem* recursively defines the semantics that determines whether or not properties should be reset to `false` when an event $\mathcal{M}$ is triggered. This is vital in order to achieve a sound embedding of the semantics of the logic. If *sem* returns $True$ all properties belonging to a formula is set back to `false`. The formula uses the *root* of the term tree, as well as the position of the event, during evaluation. By calling the reseting function $reset$, this process is illustrated as follows:

$$\mathcal{M} \rightarrow sem_{\mathcal{M}}[\![\ \mathcal{M}_{root}\ ]\!] \rightarrow reset(\mathcal{M}_{root})$$

The *reset* function accepts the *root* element of the term tree since all properties are set to `false`.

If the property is in an implication (1) or "until" (4) when applied to *sem*, it must be part of the conclusion and both the premise and conclusion evalues to $True$ to induce a rest. Conjunctions (2) are reset if both elements are $True$, while disjunctions (3) only require one of them to be. Temporal properties (5 and 6) and negations (7) are handled by Spin, and therefore ignored. A property $\mathcal{M}$ (8) evaluates to $True$ if the value it currently holds is `true`. The evaluation of $\mathcal{M}$ is achieved by a function $prop$.

7 Usage of the tool

The translation rules were implemented in Java. In addition to a pure source-to-source translation, the program includes options to simulate and verify Hume program directly. The *specification* option verifies the properties specified using the `expression` statement. By selecting the *verification* option a standard verification search for violation of safety properties, i.e. deadlocks, is performed. If a verification fails, a *guided* simulation can be run, following the a trail leading to a counter-example. The trail is generated by a failed verification. There is also an option for *random* simulation.

The tool is command-line based. It has been tested under RedHat Linux, and requires

- the Hume parser (`hparse`) to parse the source code;
- an installation of the SPIN model checker version 4.1.3 (from 24 April 2004);
- Java Virtual machine version 1.4, with the `java.io` and `java.util` libraries;

The tools must be accessible using the `PATH` environment variable. The general syntax for running the program is:

```
$ java humemc [ -r | -g | -v | -s ] <filename>
```

The options has the following meaning:

Translation: `java humemc <filename>` Following execution, the tool indicates the success of the operations, together with the translation time and name of the generated PROMELA file. For example, successful translation of a program `half` would be shown as:

```
$ java humemc half.hume
-------------------------------------
Translation Successful
-------------------------------------
File Generated:   half.pml
Translation Time: 236 ms
-------------------------------------
```

Random Simulation: `java humemc -r <filename>` The simulation is terminated by terminating the `humemc` program, achieved by pressing 'Ctrl-C' from a Unix shell. The program uses SPIN to run the simulation. The following example shows the beginning of the output when performing a random simulation of the *half adder* in 'humemc':

```
$ java humemc -r half.hume
           0 0 -1
           1 0 -1
           1 0 -1
```

```
0 1 -1
0 0 -1
1 0 -1
1 0 -1
0 1 -1
0 0 -1
1 0 -1
. . .
```

Standard Verification: `java humemc -v <filename>` Verification is also terminated by pressing 'Ctrl-C' from a Unix shell. In the verification, the PROMELA code is first translated to ANSI C. It is then compiled using the standard C compiler, 'cc', and finally the generated C program is executed. Issuing the following commands is equivalent to running `humemc` with the '-v' argument:

```
$ java humemc model.hume
$ spin -a model.pml
$ cc -o pan pan.c
$ ./pan
```

Assertion Verification: `java humemc -s <filename>` This option is similar to the option above, but also verifies assertions specified inside the `expression` statement, in the correct format.

Guided Simulation: `java humemc -g <filename>` This option is to run a *guided simulation* on a previously failed verification, which is used mainly for debugging/error detection. A trail created by the failed verification is used. The simulation is also terminated by pressing 'Ctrl-C' from the Unix shell. When using this option no code is translated. The result is therefore equivalent to running the guided simulation in SPIN directly:

```
$ spin -t -p model.pml
```

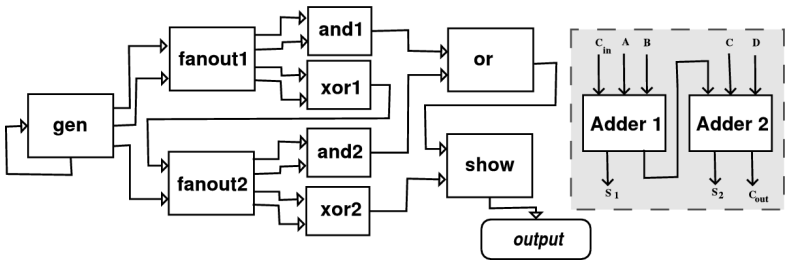
8 Examples and Results

We will now show a set of examples that has been verified with the tools. However, first we will discuss some translation characteristics for our tool.

8.1 Half-Adder and Adder

The analysis is performed with half-adder used above. Further, it is extended into a full adder, by allowing an addition carry input. N-bit adders can then be constructed by wiring the carrier bits of the adders. This is illustrated in Figure 7.

Firstly, we explored how the *time consumption*, *amount of work* and the *number of code lines* scaled with program size. We used un-annotated one-, two-, three-, four- and five-bit adders for these analysis, and the results are listed in Table 1. We noted a large difference in the *number of code lines* required for the Hume and PROMELA representations. On average each line of Hume code required *6.7* lines of PROMELA code, increasing *monotonically* towards a constant value. The amount of work required by the verifier is identified by the number of transitions between the different states. We found that when using SPIN wiith the default state vector of 1028 bytes, it was unable to cope with examples larger than a five-bit adder, respresenting 45 processes and 124 channels. For comparison, the four bit adder required 37 processes and 100 channels, giving a total of 2431 states. Based on this metric, the amount of work for the SPIN checker increases *linearly* with the number of processes generated. The time needed to translate each example was consistency around 0.5 seconds,



The adder extends the half-adder from Figure 3 by also allowing a carry bit as input. An additional type of box, `or`*, is introduced – representing logical or(A ∨ B). The area with a shaded background shows how a two-bit adder is created by wiring together carry bit. N-bit adders are created by wiring N adders in the same way.*

Figure 7: Full Adder in HW-Hume

	1 Bit	2 Bit	3 Bit	4 Bit	5 Bit	Average
State Vector Size (Bytes)	256	456	652	848	1028	**648**
Number of Transitions	694	1273	1852	2431	N/A	**1563**
Verification Time (ms)	5735	8297	11458	15942	N/A	**10358**
Lines of Code Factor	4.27	5.84	7.02	7.87	8.51	**6.70**
Number of Boxes	9	17	25	33	41	**25**
Number of Processes	13	21	29	37	45	**29**
Number of Channels	28	52	76	100	124	**76**

Table 1: Translation Characteristics for Binary Adders

reflecting the similar source size of the examples. Verification time, however, increased *exponentially* in line with the number of processes and channels. Additional experimentation revealed that increased inter-processor communication had a greater impact on the total verification time than an increased number of processes.

Secondly, we proved that in no runs, can we induce `1` as a result in addition to a carrier bit – which is the case when both inputs of `show` are `1`:

$$\Box\neg(show(1,1))$$

We can also verify that if one of the inputs of $fanout$(the binary numbers being added) is `1`, then the sum (first input of `show`) is `1`:

$$\Box((fanout(1,*) \vee fanout(*,1) \rightarrow \Diamond show(1,*))$$

Since the adder accepts a carry bit as input, the result can be `1` plus a carry output. This means that the formula

$$\Box\neg(show(1,1))$$

will *fail*. We can prove that by stimulating the two-bit adder with some input, the correct output is created (black box test) – and by doing so with all inputs we can show the correctness. For example, we can prove that adding three bits will induce two `1`s at the first `show` box and one at the second:

$$\Box(gen(1,1,1) \rightarrow \Diamond(show1(1,1) \ \wedge \ show2(1,0)))$$

8.2 Counter

The standard features of SPIN help us identify *deadlocks*, which means the program cannot make any progress. This problem was experienced in a three-bit counter. Three boxes, each representing one of the the bits, are connected together to create the counter. One of their outputs is wired to a box `counter` – which converts the binary input into decimal format and displays the result. The program hangs when reaching `4`. By using a standard verification and a *guided* simulation following the counter-example, we determined an error in a pattern. The problem was the binary representation of `5` was `011` and not `101`, which is the correct one. Like most errors, this is both trivial and easy to make. An automated tool, such as a model checker, can quickly and easily find such errors and provide counter-examples. We also used the specification language to prove that the counter always resets to `0` when reaching `7`:

$$\Box(counter(1,1,1) \rightarrow \Diamond counter(0,0,0))$$

This formula is expressed in Hume as follows:

```
expression (TEMP (ALWAYS (IMPLIES
     (BOX ‘‘counter’’ (TUPLE [CONST ‘‘1’’,CONST ‘‘1’’,CONST ‘‘1’’]))
     (EVENTUALLY(BOX ‘‘counter’’ (TUPLE [CONST ‘‘0’’,CONST ‘‘0’’,CONST ‘‘0’’])))))));
```

To decide the triggering of an event during the embedding into Promela, we use the function *match* – which will obtain one corresponding match for each property. We will illustrate the use of *sem* to decide whether or not each event induces a reseting. For this illustration we abbreviate $counter(1,1,1)$ and $counter(0,0,0)$ to `prop1` and `prop2` respectively. This is the same name as they are given in Promela:

$$\begin{array}{ll}
sem_{prop1}[\![\Box(prop1 \rightarrow \Diamond prop2)]\!] & \\
\quad \equiv sem_{prop1}[\![prop1 \rightarrow \Diamond prop2]\!] & \textit{by (5)} \\
\quad \equiv \textbf{if } prop1 \in \{prop1\} \textbf{ then } False & \\
\quad\quad \textbf{else } sem_{prop2}[\![\, prop1 \,]\!] \wedge sem_{prop2}[\![\, \Diamond prop2 \,]\!] & \textit{by (1)} \\
\quad \equiv False &
\end{array}$$

$$\begin{array}{ll}
sem_{prop2}[\![\Box(prop1 \rightarrow \Diamond prop2)]\!] & \\
\quad \equiv sem_{prop2}[\![prop1 \rightarrow \Diamond prop2]\!] & \textit{by (5)} \\
\quad \equiv \textbf{if } prop2 \in \{prop1\} \textbf{ then } False & \\
\quad\quad \textbf{else } sem_{prop2}[\![\, prop1 \,]\!] \wedge sem_{prop2}[\![\, \Diamond prop2 \,]\!] & \textit{by (1)} \\
\quad \equiv sem_{prop2}[\![\, prop1 \,]\!] \wedge sem_{prop2}[\![\, prop2 \,]\!] & \textit{by (6)} \\
\quad \equiv \textbf{if } prop1 \wedge sem_{prop2}[\![\, prop2 \,]\!] & \textit{by (8)} \\
\quad \equiv \textbf{if } prop1 \wedge \textbf{if } prop2 & \textit{by (8)} \\
\quad \equiv \textbf{if } prop1 \wedge prop2 &
\end{array}$$

`prop1` is the premise of an implication and will therefore not induce a reseting. `prop2` is the conclusion, and will induce a reset if both the premise and the conclusion holds. Since it is the only part of the conclusion, reseting only depends on `prop1`. The following code fragment shows the Promela code of the two matches:

```
::  (in1?[0] && in2?[0] && in3?[0])
     -> if
          ::  (prop1)-> prop2=true; prop1=false; prop2=false;
          ::  else skip; fi; in1?0; in2?0; in3?0; out1!0,-1;
...
::  (in1?[1] && in2?[1] && in3?[1])
       -> prop1=true; in1?1; in2?1; in3?1; out1!7,-1;
```

8.3 Roscoe's Railway

In his book on concurrency [Ros98], Roscoe illustrates deadlocks with a small railway network, shown on the left side of Figure 8. In a deadlock no components can make progress The railway network is implemented in CSP (Communicating Sequential Processes), and model checked using FDR (Failures/Divergences Refinement), the main proof and analytic tool for CSP. The railway comprises of a closed ring and a bypass, each made up by a set of track sections. Each track section is represented as a process. It can be either empty of full, where a full section has one train on it and an empty none. A track section cannot be occupied by more than one train each time and a train can only enter a track section which is empty.

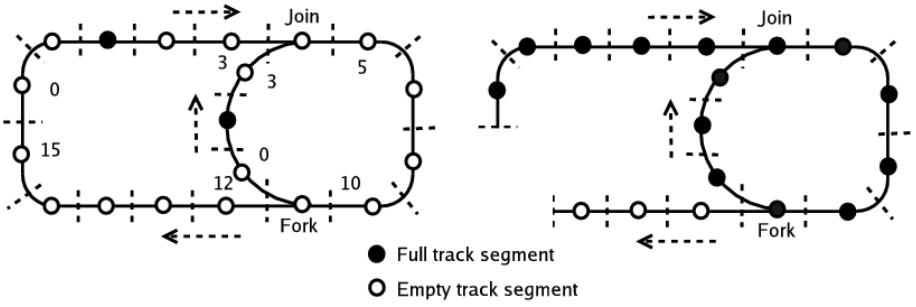


Figure 8: Roscoe's Railway Network

Since the network shown on the right hand side of Figure 8 is not closed, it is not *hereditary* deadlock-free (where the network and all sub-networks are deadlock-free): a sub-network may fill with trains and induce deadlock. If there are exactly two trains in the network, however, deadlock cannot occur, as the left side shows. Our Hume version of the railway extends the original CSP example in several ways, notably in the use of an asynchronous handshaking protocol, which increases the concurrency of the system, models actual railway practice more closely, and prevents deadlock except where there are no empty track sections, the use of high-level structuring and meta-programming constructs, which increases the flexibility of the example, and the use of strong types to allow the example to be reused. In the rest of this section we will define and prove a set of temporal properties of the railway.

PROPERTY 1: *If a track section is full, a train cannot enter it*

A track section becomes full when a train arrives, and remains full until it departures. By specifying that if a train arrives, another train cannot arrive until the current train departures, we achieve this. This can be expressed quite elegantly with the until operator:

$$\Box(\mathcal{M}^{track_n}_{arrive} \rightarrow \ \neg\mathcal{M}^{track_n}_{arrive} \ \mathrm{U} \ \mathcal{M}^{track_n}_{departure})$$

Figure 9 shows part of Hume implementation. We must prove this property for each *instantiation* of a `Track` segment, in addition to `Fork` and `Join`. For each `Track` segment, if `state` is **6**, a train arrives, and `inp` holds the train identifier. When a train departures `state` will have the train identifier – forcing us to specify the property for each possible train – and `outctl` equals **0**. For `Ring1` this is expressed as:

$$\Box(\mathit{Ring}1(6,\_,*) \rightarrow \neg\,\mathit{Ring}1(6,\_,*)\ \mathrm{U}\ (\mathit{Ring}1(1,*,0)\ \vee\ \mathit{Ring}1(2,*,0)))$$

The rest of the `Track` is expressed the same way – only with the box identifier changed. The `Join` segment, being the most complex with two potential inputs, is defined as follows:

```
box Join
in (value ::State,in1,in2::Channel,outctl::Ctl)
out(value'::State,in1ctl,in2ctl::Ctl,outp::Channel)
match
  (5,0,*,*) -> (6,0,*,*)
| (5,*,0,*) -> (6,*,0,*)
   ...

box Fork
in (value :: State,inp::Channel,out1ctl,out2ctl::Ctl)
out(value':: State,inctl::Ctl,out1,out2::Channel)
match
  (5,0,*,*) -> (6,0,*,*)
| (5,0,*,*) -> (6,0,*,*)
  ...

template Track
in (value ::State,inp::Channel,outctl::Ctl)
out(value'::State,inctl::Ctl,outp::Channel)
match
   (5,0,*) -> (6,0,*)
|  (6,1,*) -> (1,1,0)
|  (6,2,*) -> (2,1,0)
|  (6,3,*) -> (3,1,0)
  ...

for i =  0 to RingSize-1 except (ForkPos, JoinPos)
   instantiate Track as Ring{i};
for i = 0 to BypassSize-1
   instantiate Track as Bypass{i};
...
```

The figure illustrates the implementation of the railway. `template` *gives structure to the track segments, and each segment is instantiated as* $\texttt{Ring}_n$ *or* $\texttt{Bypass}_n$ *according to their location. The* `Fork` *and* `Join` *segments have a different structure. The figure only shows the relevant part of the code, and only a few of the matches.*

Figure 9: Hume Code of Roscoe's Railway

$$\Box(Join(6,\_,*,*) \ \vee Join(6,*,\_,*) \rightarrow \neg(Join(6,\_,*,*) \ \vee \ Join(6,*,\_,*)) \ \mathrm{U} \ (Join(1,*,*,0) \ \vee \ Join(2,*,*,0)))$$

PROPERTY 2: *If a train leaves a track section, it will eventually leave the next section as well*

We need to use the coordination layer to identify adjacent tracks. In most cases this is obvious, but Figure 8 shows the track numbers that may lead to confusion. We show this property for two adjacent segments; `Ring6` and `Ring7`:

$$\Box(Ring6(1,*,0) \ \vee \ Ring1(2,*,0) \rightarrow \ \Diamond(Ring7(1,*,0) \ \vee \ Ring7(2,*,0)))$$

The most complicated segments is when a train departures `Ring10` – it will then enter `Fork`, and consequently either continue on the ring or enter the bypass:

$$\Box(Ring10(1,*,0) \ \vee \ Ring10(2,*,0) \rightarrow \ \Diamond(Fork(1,*,*,0) \ \vee \ Fork(2,*,*,0) \vee Fork(1,*,0,*) \ \vee \ Fork(2,*,0,*)))$$

Note this property can also be divided into two sub-formulas – one expressing if a train leaves one segment it will enter the next and the other expressing if a train enter a segment it will eventually leave it:

$$\Box(\mathcal{M}^{track_n}_{departure} \rightarrow \Diamond\mathcal{M}^{track_{n+1}}_{arrive}) \wedge \Box(\mathcal{M}^{track_{n+1}}_{arrive} \rightarrow \Diamond\mathcal{M}^{track_{n+1}}_{departure})$$

PROPERTY 3: *If a train leaves the* `Fork` *section, it will eventually either leave the first bypass track or the next track in the outer ring*

From Figure 8 we can see that `Ring12` and `Bypass0` are the segments following `Fork`. We can then specify the property as:

$$\Box(\mathcal{M}^{Fork}_{departure} \rightarrow \mathcal{M}^{Ring12}_{arrive} \vee \mathcal{M}^{Bypass0}_{arrive})$$

When identifying the patterns p in $\mathcal{M}$, we can express the property:

$$\begin{aligned}&\Box(Fork(1,*,*,0) \vee Fork(2,*,*,0) \vee Fork(1,*,0,*) \vee Fork(2,*,0,*)\\ &\quad\rightarrow \Diamond\, Ring12(1,*,0) \vee Ring12(2,*,0) \vee Bypass0(1,*,0) \vee Bypass0(2,*,0))\end{aligned}$$

PROPERTY 4: *The Fork section includes fairness*

The meaning of *fairness* in this context is that in any element of a run, a train will eventually arrive in the bypass and a train will eventually arrive in the ring. It is sufficient to show that it arrives in the first elements – `Ring12` and `Bypass0`:

$$\Box\Diamond\mathcal{M}^{Ring12}_{arrive} \wedge \Box\Diamond\mathcal{M}^{Bypass0}_{arrive}$$

However, this property *fails*. This is due to the implementation of the railway and the deterministic nature of Hume. Unfair matching means the outer network is always checked first when a train is set for departure. If this is always empty, which it might be, the train will never enter the bypass. After identifying the correct patterns p, the formula is expressed as:

$$\Box\Diamond Ring12(6,\_,*) \wedge \Box\Diamond Bypass0(6,\_,*)$$

In defining the properties of this example, we encountered a version of the railway system which was believed to deadlock. In fact, although we were able to prove that the example was deadlock-free, using our technique we were also able to determine that it contained a *livelock*. Our experience is that livelock is, in fact, much more common than deadlock in Hume, often being introduced through the use of asynchronous communication, and that this may be difficult to detect through testing alone. We can thus provide useful feedback on the liveness of Hume programs.

9 Conclusion

We have implemented `humemc`, a model checker for Hume programs annotated with temporal properties. It uses the off-the-shelf SPIN model checker for the actual verification.

`humemc` allows the user to specify correct order of events, by annotating the source code using the specification language provided. The tool will then attempt to verify these annotations. If a violation is found, a counter-example will be provided, which will help the user to find and correct the error. The program can also verify absence of deadlock

We have shown the translation from a Hume program into PROMELA, the input language for the SPIN. We have also shown how the translation of the annotations into a combination of PROMELA and the LTL logic used by SPIN.

The approach is illustrated by a set of small to medium sized Hume programs. In experiments with adders it could handle 33 boxes (4 bit adder) but failed when the program had 41 boxes (5 bit adder). Since all these boxes are small, and he type is only bits, we do not expect the program to handle any programs consisting of more than around 15 to 20 decent sized boxes. However, scalability is a subject of further research.

In this report, we discuss the deployment of direct translation of HW-Hume into Promela for model-checking with Spin. In contrast, in D17 (Assertion Language) we consider the verification of formal properties of Hume expressed through a novel specification notation, using a TLA embedding in Isabelle. Thus it might appear that an opportunity has been lost to develop a unified approach. However, there are important differences in requirement between this work and D17.

First of all, this report is concerned with the highly impoverished HW-Hume level, with minimal expression layer content. Thus, here we are almost entirely focused on the coordination layer for which an approach intended for verifying inter-process communication and coordination properties is more appropriate. In contrast, D17 is concerned with verification of the Hume expression layer, a Turing Complete language requiring full strength semantic description and theorem proving. Thus the use of a lighter-weight approach is not appropriate.

References

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999. 2.1

[Gro04] G. Grov. Model Checking HW-Hume. Master's thesis, Heriot-Watt University, 2004. 5

[Hol03] G. J. Holzmann. *The Spin Model Checker — Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003. 4

[Jos06] S. Jost. Formal Hume Semantics. EmBounded Project Deliverable, March 2006. Deliverable D12. 1

[Lam94] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994. 1

[LG07] H-W. Loidl and G. Grov. Assertion Language. EmBounded Project Deliverable, October 2007. Deliverable D17. 1

[NFG01] Thomas Noll, Lars Ake Fredlund, and Dilian Gurov. The Erlang Verification Tool. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 582–585. Springer–Verlag, 2001. 2.2

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998. 8.3