USING SIMPLE PUSHLOGIC

David Greaves
University of Cambridge, Computer Laboratory
Cambridge, UK

Daniel Gordon
University of Cambridge, Computer Laboratory
Cambridge, UK

Keywords: Compensation, Model Checking, Code Reflection, Ubiquitous Computing, Control Plane, Application Scripting.

Abstract: Pushlogic is a scripting language for a dynamic population of devices (sensors/processors/actuators) and dynamic number of concurrent applications in a reliable or safety-critical system. System stability is assured by idempotency constraints and intrinsic error recovery capabilities arise from the reversible nature of Pushlogic. It is a constrained language amenable to automated reasoning. It defines 're-hydration' for dynamic binding of rules to new device instances and a load-time model checker that runs before a new bundle of rules may join a domain of participation. In a typical application, complex embedded devices are partitioned into passive components known as 'pebbles'. API reflection is then used to expose the interfaces offered by the pebbles. All proactive and interactive behaviour between pebbles or over the network must then be implemented with Pushlogic and 'code reflection', as we call it, exposes this behaviour for automated reasoning.

1 INTRODUCTION

In this paper, we introduce a new scripting language, called Pushlogic, that generates declarative byte code. The code can be canned to ROM for embedded applications, or run on server platforms (e.g. a PDA) for reliable interaction with a dynamic population of devices and other application scripts.

In software terms, a 'script' is a collection of commands to be performed in a particular order under various conditions. Imperative programming languages, such as assembly language, Java and the unix shell language are frequently used for scripting. These languages are used to control a collection of devices or to otherwise automate a process. They are unrestricted in expressibility and hence reasoning about their behaviour or their interaction with other such scripts is hard. When a script phrased in a decidable language controls and reacts to objects containing undecideable code (or exhibting unpredicatable behaviour), the system becomes undecidable as a whole. Nonetheless, it is our belief that there are significant benefits from using decidable code at the highest levels - the level of application scripting.

Network systems suffer from errors as the result of intentional and unintentional arrival and departure of new entities (devices or interacting applications that control the devices) and from network errors and disconnects. The BPEL4WS (Schlingloff et al., 2005), and StAC (Chessell et al., 2002) languages were designed to provide reliable completion of business transactions in this environment. They both provide *Compensation* mechanisms that allow the programmer to structure additional code to be executed should components of a partial transaction need to be rolled back. Programs in these languages have been subjected to automated formal analysis, for instance using Petri Nets, but they are not any more restricted than C or Java in their expressiveness, and so automated solutions encounter the usual problems (decidability etc.). Pushlogic does not require the programmer to provide his own compensation code because only reversible programs are allowed.

Pushlogic also provides means for a device to publish its proactive behaviour, that is, to announce what it will do when introduced to an environment. There are a number of technologies that enable devices to publish their APIs and to receive commands over the network, such as WSDL, XMLRPC and UPnP (Microsoft, 2000). We embrace them, but go further: to enable automated reasoning about the behaviour of a device we force the embedded *code* inside devices to

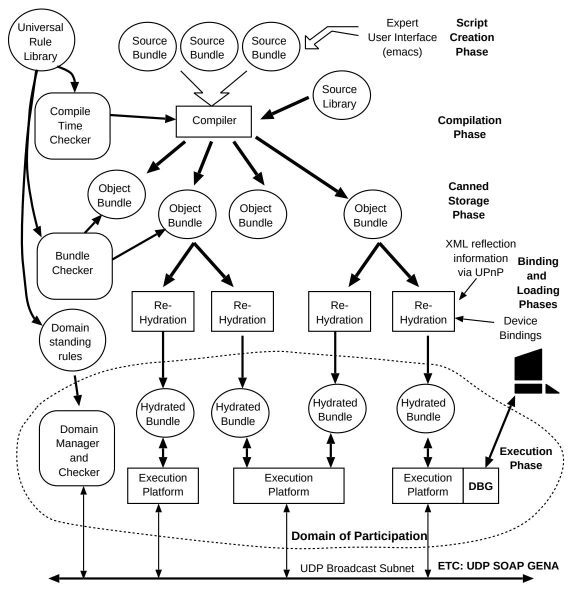


Figure 1: The write/compile/re-hydrate/execute toolchain for Pushlogic.

be implemented via a declarative bytecode that is exposed alongside the/any command APIs. For execution, this code may be interpreted or further compiled to native code. For checking, it must be inspectable over the network - a facility we call 'code reflection'.

Pushlogic object level is a declarative byte code, designed as an intermediate code for automated reasoning using model checkers. Pushlogic object is held in bundle files containing rules. Rules are either temporal logic assertions or else executable rules that define a finite state machine or '*mechanism*.' Bundles run inside a domain of participation (DoP). Dynamic storage allocation only occurs when new bundles of rules are loaded into a running DoP. Bundles arrive either when a new pebble that requires control arrives, or when a new application is started, expressed in Pushlogic. Before a bundle can join, the union of the rules in the new bundle is formed against those already in the domain. If any of the rules are inconsistent or any of the temporal logic rules (existing or new) will not hold under the combined mechanism, the bundle cannot be loaded.

We use the term '*mechanism*' for our combination of FSMs because it models not only the effect of inputs on outputs and internal state, but because a mechanical system of levers and cogs can sometimes be operated in reverse, with pressure applied to an output causing an 'input' to change.

A user-level device, such as a DVD player, is considered to consist of some number of physical or logical devices, called '*pebbles*'. Pebbles only interact with each other through an application program coded as one or more bundles of Pushlogic. Pebbles provide input and output to various sensors, actuators and other interfaces. Pebbles are like device drivers, except they are first-class entities on the network that can register their command API and capabilities, so that they are a resource to be used by any number of applications.

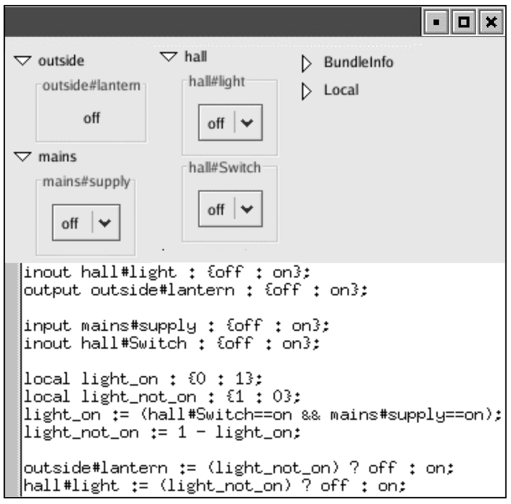
Pushlogic has been developed for a year or so, and its first compiler and run time system are becoming stable. We are now implementing the DoP manager for our Ethernet-based implementation. The manager provides real-time checking of bundles joining the DoP and DoP merging. Pushlogic therefore provides a scripting language for for a dynamic population of sensors, actuators and applications suitable for safety-critical systems. We are also implementing a Controller Area Network (CAN) (Kaiser and Mock, 1999) version, where, for current applications, device API reflection will not be needed and all checking is done before system assembly.

Figure 1 shows the Pushlogic toolchain. Source bundles are compiled with libraries to generate dry object bundles that do not refer to specific pebbles by name. A subsequent re-hydration stage implements such bindings, and a given bundle may join the DoP more than once, as illustrated, but using different bindings for each instance. Several bundles may run on a single execution platform, but the behaviour of the system is, as far as possible, the same as though they were distributed over the network. For a self-contained device using ROM'd code, such as the Heating Controller presented later, part of the rehydration can be performed before canning the code to ROM, so that the code is bound to the local pebbles, and part of it can be done later, for instance to bind to other devices encountered in the domain at run time.

Our first implementation holds all run-time variables as fields in the tuples of a distributed tuple space. Fields range over constant values, local tuples or remote tuple pointers.

Communication between pebbles and bundles is through shared fields. The tuple space is navigated using URIs and heirarchic names with hash symbols as separators. Network traffic uses our own temporary protocol, called ETC (evolving tuple core protocol), that is essentially UDP versions of SOAP RPC and GENA eventing, found in UPnP (Microsoft, 2000).

A GTK GUI may be connected to any running DoP to allow variables to be viewed and edited over the network. Alternatively, it can be run standalone, on a workstation, with a number of bundles loaded from the command line. Figure 2 shows a bundle called Lanterns under the GUI, The output 'outside#lantern' is a label and cannot be changed directly with the GUI. It is updated when the value of this variable changes. The input 'mains#supply' has a menu from which the user can select 'on' or 'off'. The inout vari-



```
inout hall#light : {off : on};
output outside#lantern : {off : on};

input mains#supply : {off : on};
inout hall#Switch : {off : on};

local light_on : {0 : 1};
local light_not_on : {1 : 0};
light_on := (hall#Switch==on && mains#supply==on);
light_not_on := 1 - light_on;

outside#lantern := (light_not_on) ? off : on;
hall#light := (light_not_on) ? off : on;
```

Figure 2: Lanterns - An Example of Pushlogic under GTK GUI.

ables 'hall#light' and 'hall#Switch' can be changed by the user as well as by a Pushlogic program. Program counters and other local variables are stored in tuples held under the 'Local' tab, in a unique subtuple for each bundle instantiated on the platform. We also have a locally-written universal UPnP control point, that can perform roughly the same function for a subnet of UPnP devices. We will shortly merge the functionality of these two GUIs.

In other work, formal validation of Ladder Logic and timed transition systems has received significant attention (Glässer, 1995). Commercial products for formal validation of reactive control systems against safety standards are becoming available. e.g. (ReactiveSystems, 2003). Cypress has just released a synthesiser for embedded controller software to eliminate hand-crafted device drivers and network stacks (Pearson, 2005). Formal specification of web resources and services using ontologies and assume/guarantee reasoning is advancing (Monika Solanki, 2003), as are proof-carrying imperative code tool chains. However, our practice of converting imperative code to declarative intermediate form seems novel.

2 EXECUTION SEMANTICS

Each executable rule in Pushlogic object is an assignment of the form

$$f := exp : pbind$$

where f is a variable in a global, heirarchic name space (the tuple space paradigm) and exp is a Pushlogic object expression and $pbind$ is information that assists in reversing the operation of the rule. Fields may be local to a bundle or shared between the current bundle and other bundles and pebbles. Where shared, they are declared as input, output or inout. Input fields are only changed by external bundles and pebbles. Output fields are changed only by the current bundle. Inout fields are changed by the current bundle and by other bundles and pebbles and also by timeouts in network protocols.

Each field ranges over a set of constant values, certain of which may be declared as the *safe values* of that field. The values are integers or strings, or the reserved constants 'true' and 'false'. Where a bundle alters the value of a field held on a remote resource, the run time system generates network remote write using the ETC protocol. Where a bundle is sensitive to changes on remote resources, it uses a periodic soft-state registration protocol within ETC that causes it to receive notification of changes for a period (e.g. one hour). In any domain, an inout field may be set to one of its non-safe values by at most one bundle or pebble. Multimedia applications use the notion of third-party setup, where a field in a source pebble is set to the same circuit identifier as a field in a sink pebble.

The reference execution model for an executable Pushlogic rule is that all sub-expressions occurring in the expression are re-evaluated whenever there are changes to any of their support. Likewise, changes to the result of the top expression become scheduled as updates to the assigned field. Updates are **gated**, by which we mean that all updates to fields held on a common execution platform resulting from a single event are batched and made at once (atomically). Further changes arising from a batch of gated updates are collected and deferred to the next batch. Our interpreter for the Pebbles project is a direct implementation of the reference execution model, but it uses too much RAM for use on low-cost microcontrollers, where native code should be run from ROM.

A push logic expression may generate a special value, backstop ($\perp$). When backstop is assigned to a field, the field's value is unchanged. When multiple rules assign to the same field, static analysis must show that they either generate a common value or else backstop.

The gated nature of updates to fields held on a common platform enables certain rule combinations to operate deterministically when they would not otherwise. Consider the following pair of rules where d, $d1$, and $d2$ are held on the same execution platform:

$$d1 := d; \quad d2 := (d\&\&!d1)?1 : \perp$$

This pair will reliably set d2 to one whenever d starts to hold. Without the gated-update constraint, the second rule might always be executed after the first rule and hence the guard would never hold.

The reference execution model implies that the union of executable rules for a DoP may be thought

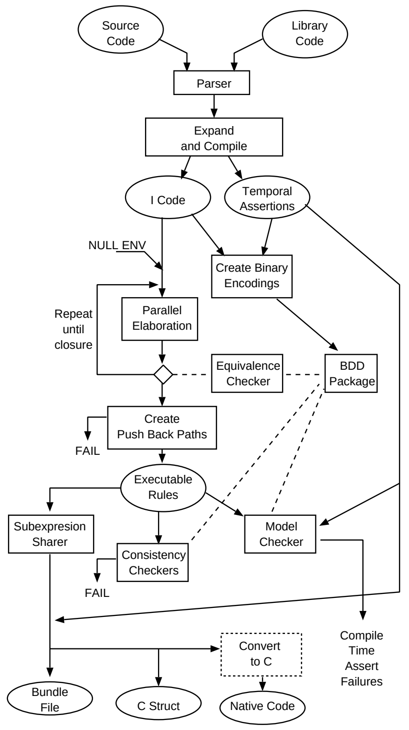


Figure 3: Structure of the Pushlogic Compiler.

of as an assertion over the system state. The assertion holds at all times that the system is passive, and when any event occurs that breaks the assertion, the mechanism implements corrective action so that the assertion once again holds. Inout fields are essentially outputs that can unilaterally change back to one of their safe values. The interpreter contains a second set of rules to evaluate every operator or function application in reverse, so that a change of an inout can be 'pushed back' to another inout or local field, so that once again, a consistent state prevails. This is the Pushlogic implementation of *compensation*. The $pbind$ provides information to specify the compensation behaviour where there is more than one possible way to interpret a rule in reverse. For example, with logical NOT, no indication is required, because the new value is obvious at push back time. On the other hand, for the comparison operator, when pushed back to hold, it is sufficient to specify one operand to push back on, since it must be pushed back to the current value of the other operand. But, when comparison is pushed back to false, a value and operand must be specified, since, in general, there are many possible values that will make a comparison not hold. For logical AND, when pushed back to false, which operand to push on must be specified, since either will do, whereas to push logical AND to hold may require both its arguments to be changed. For the conditional expression operator, the condition may have to be changed and also the value of that side of the operator may have to be changed.

For model checking, the next state relation must be constructed from the executable rules. In this relation, a hidden input variable is created for every possible pushback, which is every safe value of every inout field. This is called a *pushback input*. Additional clauses are added to the next state relation to represent that at most one of the pushback inputs of each inout may hold at any one time, and that when it holds, the variables altered by that pushback have the constant values determined by the union of $pbind$ fields. The temporal logic assertions are checked at compile time for the bundle in isolation. In future, they will also be checked by the domain controller as it loads the bundle into a live DoP, or when a pair of DoPs are merged.

To ensure that all states are stable (and not oscillators), the system also implements an '*Idempotency Constraint*'. This states that any Pushlogic program will result in no further output changes if 'executed' more than once without change of any input field. (This rule is the basis for the loop unwinding in the source compiler.) A rule such as $track := (p)?track + 1 : track$ would not generally be admissable, because the number of increments executed while p holds is not defined, but carefully-constructed integration is allowed. For example, anything that is tantamount to the following differentiate/integrate rule pair is permissible:

$$d1 := d; \quad track := (d\&\&!d1)?track + 1 : track$$

The idempotency constraint is checked in our source compiler by symbolic evaluation (described below), and will be checked by the domain manager. The differentiation construction is required sufficiently frequently that it is built into the Pushlogic Source Compiler, described next. The above two object rules are then written as `if (↑d) track := track+1;` with the automatic allocation of a hidden variable to replace $d1$.

3 PUSHLOGIC SOURCE COMPILER

Although rules are frequently a useful way to express desired behaviour, many applications are most easily coded in an imperative programming style. Rather than expecting the user to manually convert his notions of application behaviour into Pushlogic object rules, a compiler for imperative-style expression of applications is used. We note that imperative programs deal essentially with sequential changes of state, whereas logical predicates over application programs deal in terms of the visible, accumulated results of these changes.

'Pushlogic Source' is a block-structured, imperative-like, programming language, but with no dynamic storage allocation. It is less fundamental to our approach than the object form, because a variety of source forms could be envisaged that would generate compatible object for various niche applications.

A Pushlogic Source program is an unordered list of declarations, function definitions and executable statements. The statements are all started in parallel when the compiled object bundle is loaded. A statement may be a sequential block, thereby providing an escape to the normal imperative programming paradigm.

It is our goal to support as many features found in common OO imperative high-level languages as possible, while still producing output that can be represented as Pushlogic object rules and checked automatically at load time. The currently available forms include integer arithmetic, without arrays, function call with compile-time unwind of all recursions, nested static object instances and the full C/Java set of imperative and control-flow constructs.

Executable sequences are composed in parallel. Each sequence may be considered to be enclosed in an infinite `while` loop that has its own thread that executes the rule as fast as possible, but with all such threads performing their assignments in synchronism. Sequential composition of behavioural statements is introduced with the block construct, denoted with C-like open and close braces. A further level of parallelism is possible inside a sequential block because parallel assignment is supported: e.g. $(a, b) := (e1, e2)$.

The internal flow of the compiler is shown in Figure 3. The input is parsed and converted to imperative intermediate code using conventional compiler techniques. Function calls are expanded in line. For each sequence in the source code a section of I-code is generated. I-code consists of labels, gotos, waits, assignments, resultis statements (used for returned values in the middle of inlined tasks and functions) and conditional branches. For each sequence, a run-time program counter is defined. At the object code level, these program counters act just like other local variables, and their values range over the labels in that sequence. Threads with fewer that two wait statements require a constant value in their run-time program counter and so these program counters are eliminated at compile time. There is no run-time spawning or joining of threads (although the illusion of this could be provided from a static set of threads using pre-processing techniques). Temporal logic assertions in the source code are split off and held separately. Safety assertions may be guarded by nested `if` statements and by the current value of the program counter.

The I-code is embedded in a binary-decision diagram (BDD) package by generating binary encodings of every variable (field), constant and operator. This then enables an equivalence checker to be used to compare any pair of expressions or check that a predicate is a tautology or invalid.

An entry point is defined as any entry point to a sequence of I-code or the location immediately after any wait instruction. Parallel symbolic evaluation is then conducted, until closure, or failure if more than 100 iterations is needed. This consists of starting in a null environment and evaluating from each entry point to collect symbolically the assigns to every variable, including program counters, up until a wait statement or the thread loops back to its initial entry point.

While more than one assign is made to a variable, by different threads, such as $v := e1;\ \ v := e2;$, the assignments are combined in pairs using the following rule

$v := (e1 = \perp)?e2 : e1;$
$check(e1 = e2 \vee e1 = \perp \vee e2 = \perp);$

This gives a single expression for every assigned variable. If the check fails, the compilation fails because the operations are incompatible.

After the first elaboration from all entry points, the process is repeated using the environment created by the first. Code guarded by differentiators will not have any consequences on the second or subsequent elaborations. After each elaboration, the equivalence checker is used to detect any changes in any symbolic value, and if there are, then another iteration is commenced. Before each new iteration, occurrences of $\perp$ in the expression for a variable in the environment are replaced with the symbolic value for that variable calculated on the iteration before. This exactly models the gating implemented by the nominal behaviour at runtime (which is the actual behaviour when interpreted and which is emulated by code structure when compiled native).

After a closed set of symbolic assignments has been computed, push back paths are created through the right-hand-side expressions from any field whose mode is 'inout'. For each safe value of an inout field, a path is traced backwards through the expression tree that will cause generation of that value. These paths extend back though local variables used as intermediate values in any computation. For all safe values of all bearing inouts, the same path must work for each local variable. This constraint can cause some novel error messages. The paths are stored in the push back indication section of each rule. Currently, the compiler chooses amongst various possible pushback options and writes its decision to a report file, but we might change this so that user pragmas must be provided to select behaviour when more than one method of compensation is possible.

The resulting object-level executable rules are optimised by spotting common subexpressions and inserting cross-references to allow the evaluations to be shared at runtime on the interpreter. The output code is stored in a bundle file, along with the assertions. It is also written to a C struct file that contains some initialised C arrays, for direct canning into ROM. In the future, the declarative byte code can also be converted to C to be run as native ROM code instead of being interpreted on the execution platform (thereby saving expensive RAM on embedded devices).

The BDD package is also used as a compile-time model checker to test the embedded assertions. Assertions that fail at compile time when a bundle is checked in isolation, or against a standard library and testbench should normally be corrected before attempting to load the code into a live DoP. The BDD package used is the original C code from SMV (McMillan, 2000) ported as a shared object to be loaded by Moscow ML. The fixed-point iterations used in model checking are all recoded in ML. The compiler amounts to 9K lines of ML and 15K lines of C.

Where a section of code does not intrinsically support a push back operation, it may be associated with a fuse variable by enclosing it in a fuse statement. For example, consider the following invalid code, that uses an enumeration type with one safe value and two unsafe values:

```
sort set mytype = { S: US1 US2 };
input x : mytype;
inout y : mytype;
y : = x;
```

The problem is that if y makes a unilateral change from US1, say, to S, which it is free to do, since it is an 'inout', then no push back is possible because x is an 'input' that cannot be changed from inside the bundle.

The solution is to enclose the rule inside a fuse. This fuse is able to 'blow' should y make a push back.

```
input x : mytype;
inout y : mytype;
fuse F1;
{ y : = x; } fuse F1;

forever { wait F1; sleep_secs(5);
       F1 := false; }
```

The fuse declaration defines a boolean variable with both values safe and to be set false on bundle load. The fuse statement is just syntactic sugar, because the line '{ y : = x; } fuse F1;' is rewritten during initial expansion as 'if (!f1) y : = x;'. The fuse declaration, however, does have a special effect: during pushback path creation, the fuse is chosen at last resort and only marked for push back update if

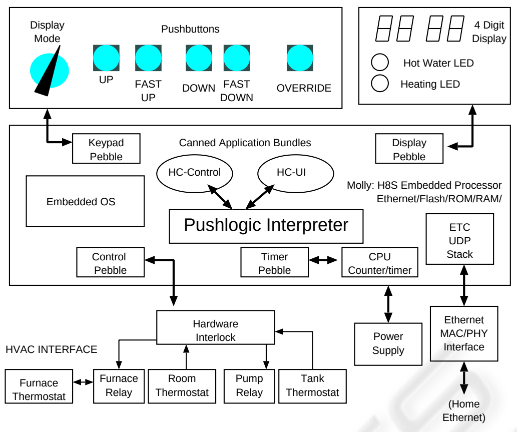


Figure 4: Heating Controller Components.

there is no other pushback path available. Only the inner-most fuse of any nested fuse blocks acts on the enclosed code.

The reset behaviour is enclosed inside a forever statement, equivalent to 'while (1)' and not needed since all push logic sequential sections are enclosed inside an implied forever. It resets the fuse five seconds after it has blown (see later for more detail of `sleep_secs`). If y refuses to accept the current value at this time, the fuse blows again. In general, other code can be sensitive to this fuse, to log or sound alarms and so on.

As mentioned, for checking, everything is currently converted to a binary encoding and a BDD checker is used, but in the future other forms of checkers can be tried, based on a mix of normal forms, Presburger Arithmetic (Presburger, 1929) or CVC (David Dill, 2004). Undecidable arithmetic and other uncheckable constructs must currently be manually partitioned out and placed in pebbles, so that they are not subject to checking. In the future, we may add additional markups to the source language to allow delineated embedding of undecidable code. The outputs from undecideable statements would be considered like other non-deterministic inputs to the system, but perhaps augmented with so called *fairness constraints* that force both options to be considered in liveness analysis.

4 EXAMPLES

We have not completed sufficient work to know finally whether Pushlogic meets all of its design goals, but we have implemented a number of full example devices, including an Alarm Clock, a DVD player, a Heating Controller, a Juke Box and various GUIs, physical keypads, lights and switches. These devices have been implemented as a collection of Pebbles with internal canned applications written in Pushlogic. A number of additional application scripts have been written that cause the devices to interact (e.g. play music from the DVD instead of sounding the alarm clock buzzer).

As a first example, Figure 4 shows the structure of our Heating Controller. This has been built and is about to be installed in a real house. It physically consists of a processor with ROM, RAM, Ethernet and Power Supply, a display and keypad, and a HVAC interface block. Architecturally, it consists of control, timer, display and keypad pebbles and a pair of Pushlogic object bundles that implement the functionality. The HVAC interface has solid state relays to control pump and furnace, input from tank and room thermostats.

The HC-Control bundle contains code to drive the output relays on and off at up to eight different programmed times, whereas the HC-UI bundle enables the programmed and current clock times to be inspected and edited via the front panel. Remote adjustment of the heating times is possible over the network, for instance, by running a second instance of the HC-UI on another platform, either with a second physical display and keypad, or under the GTK GUI. A remote process running on a server can be used to keep the time clock accurate, if desired, using the ETC protocol writes over the network.

There are multiple levels of interlock that ensure safe operation of the system. At the lowest level, the furnace thermostat is hardwired in series with the furnace gas valve, outside the controller. The controller hardware interlock contains logic gates that disable the furnace if both the tank and room thermostats are open. The control pebble (device driver) mirrors the interlock, causing a pushback on the furnace control field when both thermostats are open. The embedded application software, in the form of the canned Pushlogic script, can be seen not to operate the furnace until one or other of the thermostats is closed. Finally, the script contains the following saftey statement that is checked at compile time, that goes further than the hardware interlock, because it also asserts about the Pump relay.

```
 always
(Heating#Sense#RoomThermostat==0 &&
  Heating#Sense#TankThermostat==0) =>
  Heating#Control#Furnace==0 &&
  Heating#Control#Pump== 0;
```

When the two bundles of the heating controller are checked together, a number of small BDDs are formed and discarded during the elaboration phase of the compiler, which takes about two seconds on a 1GHz laptop running linux The consistency check generates a BDD that treats each executable rule as an assertion. This BDD has 58 primary inputs, and uses about fifty-thousand nodes. The next-state relation used for model checking has over 100 inputs because of the primed versions of each state variable, but is about the same size. These currently each take 5 seconds to form. Once formed, a number of liveness, safety and reachability assertions can be checked in rapid succession. A profile agent that handles both the ML and shared libraries has been implemented, so we have a firm grasp of where the time is being used.

We illustrate liveness checking using the following bundle that causes a variable called locked to be false for 5 seconds after a variable called button holds.

```
 def
bundle ButtonLock() {
  input v#keys#button : { false:true};
  output v#locks#unlocked : { false:true };
  forever {
    wait (button);
    unlocked := true;
    sleep_secs(5);
    unlocked := false;
    wait (!button);
    }
  local locked := !unlocked;
  live unlocked, locked;
}
```

It makes a call to the following timer library function, that blocks the thread for a period, using the timer pebble provided on all execution platforms. As explained, there is no notion of thread in the final bytecode because all function calls are inlined during compilation and all thread constructs are converted to executable rule form. The live statement is an assertion that the locked variable should never become stuck at one value permanenty.

```
fun sleep_secs(t)
{
  local until : { 0..59 };
  with (__local_timer)
  { until := (#time_now#second + t);
    wait(#time_now#second FQGT until);
  }
}
```

The timer code places the unblocking time in the local variable `until` and then blocks. The `FQGT` operator is builtin and performs a greater-than comparison that behaves sensibly as the arguments overflow in their field provided their initial difference is less than half the range. In the future, we would like to use a wider field than seconds (0 to 59) so that we can sleep, say, for many thousand milliseconds. However, larger fields consume more BDD primary inputs and BDD nodes, which are currently at a premium. We shall also consider automatic switching to a lifted form for modelling the sleep call, where it is held as a single wait statment on a fresh variable. This is simpler to model, provided there are few of these constructs, but complexity will eventually mount up in meta-constraints over the fresh variables that model the possible firing orders.

Here is a bundle that is incompatible with the ButtonLock: both cannot be loaded into the same DoP. To explain this, first we must mention that we have not fully implemented the re-hydration stage yet, and so hardcoded identifiers, such as the IP address of the other bundle's platform are currently hardcoded in the source files. The button variable was originally free to change at any time but becomes constrained by the second bundle to only change while the unlocked variable holds. The system cannot be unlocked without the button being pressed, and hence the live assertion in the Button listing fails. This will in future be spotted by the DoP manager, but currently can only be spotted by the compiler checking against precompiled bundles that are to hand.

```
def bundle B2() {
  pebble r = tup://128.232.1.45/v;
  input d#q : bool;
  r#keys#button := r#locks#unlocked && d#q;
}
```

5 CONCLUSION

This work was carried out under the CMI Goals/Pebbles project (Umar Saif, 2003). It has produced a strawman application scripting language that supports *code reflection*. The current interpreter runs on unix, bare PC motherboard, our embedded CPU cards and linux. A native-compiler that generates PIC assembler code and operates over the CAN bus (instead of Ethernet) is also being implemented. This will be less RAM hungry. We have completely implemented the top-level application code for several simple consumer devices (e.g. the alarm clock, DVD player, and so on). Work is ongoing on larger programs, such as TiVo PVR and voice mail, and in other areas, such as drinks machine, automotive (using CAN) and elevators. Our language has a number of novel features, including idempotent execution and the *mechanism* concept, where reverse execution is used to help handle network errors or device self-reset. Arrays and RPC are shortly to be tested out.

Future work is needed to analyse temporary error states during network races and to provide break-before-make form guarantees where Pushlogic is used to disable one server or device while enabling another.

The domain checker concept is well developed, but practical implementation is only just starting. We also plan to work on federation of DoPs based on known obligations and constraints of adjacent domains (Lupu E, 1997).

Our assertions currently do not contain quantifiers that range over devices or possible values of fields. As new devices and new versions of devices with extended variable domains can be inserted into a live DoP, certain negated existential forms will have to be restricted in order to preserve monotonicity.

Finally, we are seeking collaboration with an industrial partner where we can evaluate our ideas in practice and combine them with conventional safety-critical approaches, such as coverage testing.

REFERENCES

Chessell, M., Griffin, C., Vines, D., Butler, M., Ferreira, C., and Henderson, P. (2002). Extending the concept of transaction compensation. *IBM Syst. J.*, 41(4):743–758.

David Dill, S. B. (2004). CVC lite. Technical report, Stanford University.

Glässer, U. (1995). Systems level specification and modeling of reactive systems: Concepts, methods, and tools. In *EUROCAST*, pages 375–385.

Kaiser, J. and Mock, M. (1999). Implementing the real-time publisher/subscriber model on the controller area network (can). In *ISORC '99: Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 172, Washington, DC, USA. IEEE Computer Society.

Lupu E, S. M. (1997). Conflict analysis for management policies. In *5th Int Symp Integrated Network Management IM'97*. Chapman Hall.

McMillan, K. (2000). The smv language manual. Technical report, Carnegie-Mellon University.

Microsoft (2000). Universal plug and play device architecture, version 1.0. Technical report, Microsoft.

Monika Solanki, e. a. (2003). Introducing compositionality in webservice descriptions. In *Proceedings of 3rd ANWIRE workshop on adaptable services, DAIS-FMOODS.*

Pearson, J. (2005). Embedding systems at a higher level. *Electronic System Design*.

Presburger, M. (1929). Ober die vollstndigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. *Comptes Rendus du I congrs de Mathmaticiens des Pays Slaves*, pages 92–101.

ReactiveSystems (2003). Model-based testing and validation of control software with reactis. Technical Report 2003-1, Reactive Systems.

Schlingloff, B.-H., Martens, A., and Schmidt, K. (2005). Modeling and model checking web services. *Electronic Notes in Theoretical Computer Science: Issue on Logic and Communication in Multi-Agent Systems*, 126:3–26.

Umar Saif, S. W. e. a. (2003). A case for goal-oriented programming semantics. In *UbiComp 03, System Support for Ubiquitous Computing Workshop at the Fifth Annual Conference on Ubiquitous Computing*.