Pushlogic SPL1
Language Reference Manual (draft).
Updated - January 2007

DJ Greaves[1]
University of Cambridge, Computer Laboratory

April 20, 2009

[1]David.Greaves@cl.cam.ac.uk

Abstract

SPL1 Pushlogic is a scripting language for a dynamic population of devices (e.g. sensors, processors or actuators) and dynamic number of concurrent applications in a reliable or safety critical system. It is a constrained language, fully amenable to automated reasoning at various granularites. 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 of Pushlogic, complex embedded devices are partitioned into passive components known as ‘Pebbles’. API registration and reflection are 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. This report gives the syntax and semantics SPL1 Pushlogic. This work was carried out under the CMI Goals/Pebbles project [1].

This document is currently under preparation and is being changed every month or so...

Contents

Chapter 1

Introduction

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 exhibiting 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. Model checkers are good at exploring system behaviour over all possible behaviours of the undecidable subsystems.

In our approach, complex, autonomous or undecidable behaviour is partitioned and placed in ‘**pebbles**’ that interact using a constrained controlling language called Pushlogic. Pushlogic object level is a byte code, designed as an intermediate code for automated reasoning with respect to several execution models that vary in how fast a program can be checked and how accurately effects of message loss or delivery delay are included. Pushlogic source level looks like an unconstrained, imperative, multi-threaded OO-like language where the partitioning between decidable and undecidable constructs is not immediately apparent to the programmer.

Pushlogic object consists of bundles containing rules. Rules are either **consistency assertions** expressed in temporal logic or else **executable rules** that define a finite state machine or ‘*mechanism*.’ Bundles run inside a **domain of participation** (DoP). Dynmaic 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 be loaded, 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.

Pushlogic is a finite-state language, but the amount of state in a Pushlogic domain varies over time, as new bundles or pebbles enter and leave the DoP.

Instead of being executable, a bundle may be a **plant bundle** that models reactive and autonomous behaviour not programmed using Pushlogic. A world model mirrors the behaviour of the physical system or plant or other software agents. In many real systems, there are predictable effects from the output of actuators that may be detected by sensors. These feedback effects can cause undesirable effects, such as deadlock or oscillations, that Pushlogic can detect before they occur. Run-time monitoring of the conformance of the real system with its world model can also detect various faults and failures in sensors and actuators and so on.

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. Pushlogic implements a form of *compensation* where rules are executed in reverse. This is called a **pushback**. We believe this greatly reduces the effort required to handle errors and failures. To start with, less code needs to be written, but the real win is that error recovery procedures then add little overhead to the automated rule checking.

Pushlogic runs on real platforms under its own interpreter or compiled to native code (or in one extension, to .net bytecode). Two main execution platforms have been developed: a unix workstation application called **pusher** that also sports a GTK-based GUI if needed, and an embedded version that runs on Molly processor cards or on bare PC motherboards without OS. A geographical physical modelling system called **vworld** is currently being developed that enables a number of virtual, interacting pebbles to be visualised on a canvas.

For checking purposes, Pushlogic defines several execution models that successively decrease in level of modelling detail. All models are suitable for finite-state model checking. The aim is to enable a more-rapid check of a less-detailed execution model to be verified before proceeding to slower, yet more-detailed modelling.

1. **Message and Component Failure and Network Loss or Delay**
2. **Message and Component Failure and Network Delay**
3. **Message and Component Failure**
4. **Message Failure**
5. **Component Failure**
6. **Atomic, Reliable**

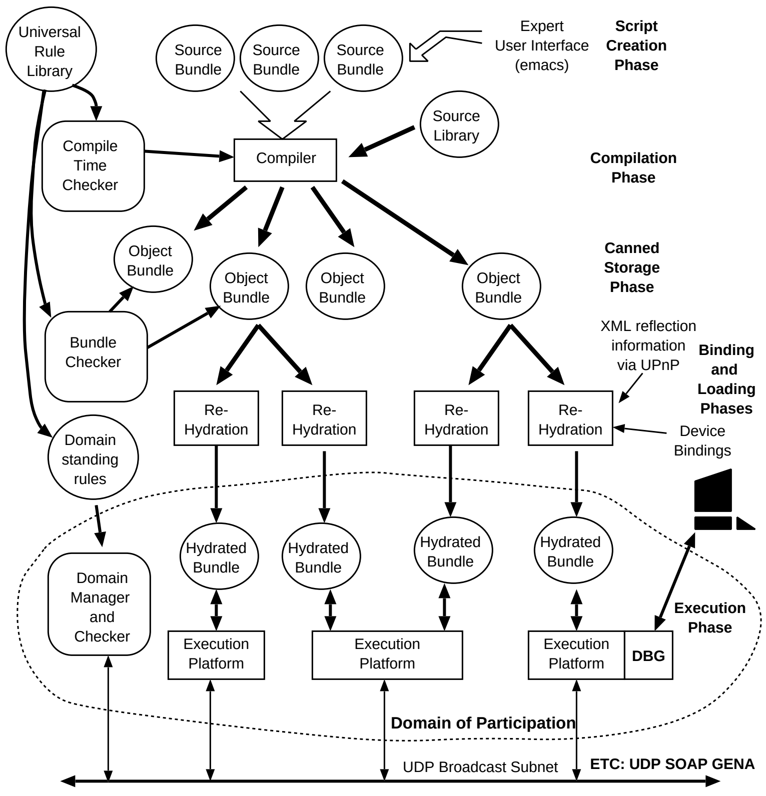


Figure 1.1: The write/compile/re-hydrate/execute toolchain for Pushlogic

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, that provides real-time checking of bundles joining the DoP, so that Pushlogic can provide a scripting language for safety-critical systems with a dynamic population of sensors, actuators and applications. We are also adding arrays and remote procedure call clients[1]

1.1 Toolchain Flow

Figure 1.1 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 re-hydration can be performed before canning the code to ROM, so that the code is bound to the local pebbles,

[1]Server-side RPC requires further study, because of the potentially unlimited number of concurrent activations that must be held as server state.

and part of it can be done later, for instance to bind to other devices encountered in the domain at run time.

Ubiquitous computing architectures, such as those based on XML, UPnP and XMLRPC have matured in terms of their support for automatic registration in directory services and description of the command APIs offered by devices for asynchronous eventing and RPC. However, until now there has been little emphasis on automated description of the *proactive* behaviour of applications running in such an environment. These applications invoke operations at various APIs available on the local device or over the network. Without these applications, the APIs would remain unused and pointless. However, when dozens of applications are running at once in an environment like a home, car or hotel room, they are bound to interact and occasionally disagree about the current correct setting of some value, such as brightness of room lights or locked state of a door.

A solution to this problem is being explored within our AutoHAN project[5]. In our approach, each device must be architecturally componentised into some number of passive '*Pebbles*' and a set of '*Pushlogic*' rules that control these Pebbles. A Pebble can represent a fairly large chunk of functionality: it could be a hardware component, such as a wall-mounted keypad or a fire alarm sounder, or it could be an entirely virtual device, hosted somewhere on the network, such as a speech recogniser. Each Pebble is able to describe itself using a reflection API, such as that provided by UPnP or XMLRPC, and register itself with dynamic directory services of the form required for ad hoc computing. Together with MIT Project O2S we have developed a complete architecture for this sort of behaviour. However, most importantly, no Pebble is able to interact with any other Pebble under its own volition. Instead, all proactive and interactive behaviour must be held outside the Pebbles in small application scripts that actually cause something to happen.

For instance, consider a DVD player designed to operate with our environment. As illustrated in figure **??**, each of its major, internal components is a separate Pebble (at the architectural modelling layer). The Keypad Pebble will not make direct contact with the Mechanism Pebble. Instead, when the 'play' key on the keypad is pressed, the application script that glues together the DVD components will convey the event to the Mechanism Pebble to commence playing. Using dedicated wiring to carry the output signals, as is common practice until now, the mechanism will be connected directly to the physical output sockets on the back of the unit. Accordingly, the application does not have to do anything to convey the multimedia stream to its destination. In the future, the media will also stream over a packet switched network using a virtual connection. It is then the job of the application to set up the connection parameters, although not to copy the data itself. In this scenario, the embedded application is having a proactive effect on other Pebbles in the local environment - for instance, it is routing audio and video to them. We use the term 'feature interaction' to describe the general situation where independent application scripts try to perform disagreeing operations on a Pebble at the same time. In the DVD example, feature interaction would arise when two DVD players try to route video to one display at once.

Using the Pushlogic approach, all application scripts within an environment of participating devices must either be implemented in Pushlogic, or else summarised in it. The various routes to using Pushlogic are illustrated in figure 1.1. A high-level language, such as Pushlogic Source, is compiled by the compiler to generate the Pushlogic object code. This is then canned in ROM for direct interpretation in the embedded target or further compiled to native code for a micro-controller. The latter approach may serve to use less RAM. In all routes, a Pushlogic form of each application is made available for checking in advance of the application being allowed to join the environment of participation. We use the term '*code reflection*' for the concept of application software being examinable in some for or another.

1.2 Bundles

Pushlogic programs consist of '*bundles*' of rules. A bundle must be checked before it is allowed to commence execution and may also be '*re-hydrated*' before checking. A bundle contains rules and meta information.

1.3 Binding and Naming

Bundles have access to several namespaces.

1. Their own local name space,
2. a namespace local to the platform they are running on,
3. a namespace local to the domain of participation (DoP),
4. a global, or outermost, namespace, accessed via URIs.

1.4 DoP and Checking Granularity

A Pushlogic program executes in a domain of participation (DoP). A DoP typically encompasses some number of Pushlogic execution engines. DoPs may become merged or federated, as explained in section 8.3. Various agents may attempt to insert a bundle into a domain of participation. For instance, a starting bundle may be loaded by the agent that first set up the domain. Equally, when new devices are added to the system, an associated bundle is also typically offered. The presence of certain combinations of pebbles and bundles in a domain may trigger loading of further bundles: eg a fire alarm system might be automatically implemented whenever there both a smoke sensor and an alarm klaxon are present in the domain.

A bundle must be checked against all other rules already in the domain and if all rules hold, then the bundle joins and its rules start to take effect as well. A bundle may also be unloaded by deleting all of its rules at any time, provided it is not critical to ensuring a consistency rule that is not also unloaded at the same time.

A bundle might require parts of it to be loaded onto different execution platforms for efficiency, but the underlying tuplecore implementation means this is not strictly necessary, since all points of the tuplespace are nominally accessible from anywhere.

... operational model ...

1.5 Re-Hydration

A bundle of rules may be '*re-hydrated*' before being offered to the domain. This means expanding various hierarchic macros in the rules to produce flat Pushlogic Object code. The number of rules in a bundle may be increased or decreased during re-hydration. For instance, a complete bundle of standard rules may be copied out from a store where they are held in a 'canned' form. Macro-generation is needed because Simple Pushlogic (SPL1) has no bindings, pointers, objects or dynamic storage allocation. The macro-generation ameliorates this by generating rules with static fields.[2]

Pushlogic rules may directly refer to fields with absolute path names (as part of a global tuple space), or to fields local to the current execution platform (generally the current device) or to the current domain of participation. In a general situation, this is not sufficient. Rules need to refer to locally bound devices, such as 'front-door-bell'. Re-hydration provides such binding. It also provides guaranteed uniqueness for certain field names needed in applications that require the equivalent of 'local variables'.

Research question: is it a good idea that the rules for rehydrating bundles to be implemented in the same language that is used within the bundles themselves? How do they need to differ and why? One clear distinction is that Pushlogic SPL1 is a finite-state language whereas dynamic rehydration changes the amount of state. This separation might be also most helpful in maintaining amenability to automated correctness checking.

[2]In the future, we may define forms of Pushlogic that are not called Simple Pushlogic. These may support dynamic state creation, e.g. to implement RPC server sides.

Chapter 2

Ontology

The Pushlogic approach defines its own ontology. This covers device structure and also domains of participation.

2.1 Glossary

Pebble. A pebble is a basic entity that can register its existance in a domain, broadcast events and be controlled over the network. A pebble does not interact directly with any other pebble.

Bundle. A bundle of Pushlogic rules consists of software, temporal logic assertions and meta information. It may be interpreted by a Pushlogic exection platform or be natively compiled. It is the only class of entity that provides communication between Pebbles.

Canned Bundle. A canned bundled is stored in a file on a server or in ROM, ready for re-hydration. It may contain formal variables that are replaced in macro-expansion style to actual variable (field/tuple) references.

Soft Pebble. A soft pebble is one that can run on any networked execution platform. It has no specialised or associated hardware.

Device. A device contains Pebbles and Bundles, execution resources and other system services that are neither Pebbles or Bundles.

Service. A service is the same as a device. However, the term is used for software-only devices. These can generally be dynamically instantiated whereas hardware devices are instantiated by a user introducing one into the domain.

Domain of Participation. A DoP is any space in which a quantifier ranges. Typically this covers a physical or logical space. Most-trivially, it is the local UDP broadcast subnet. An example use of a DoP is a rule which says ’all lights must be off when the master switch is off’: in this example, the word ’all’ is interpreted with respect to the current DoP. Devices and Pebbles can participate in several domains at once (Atif is working on this).

Chapter 3

Pushlogic Constants

Pushlogic source and object use the same constant forms.

3.1 Atomic Constant Values.

The constant values available are integers and strings.

Strings that are defined as part of an enumeration or the reserved strings ‘true’ and ‘false’ do not need quote marks when they appear at source level, but at object level there is no concept of whether a string was in quotes or not at source level.

Constant value strings containing spaces must be enclosed in quotes at the source level. They should be avoided for everyday device control.

Integers are rendered in base ten and have no leading zeros, except for zero itself. In all respects, an integer behaves equivalently to its corresponding base ten string. The runtime system (e.g. Pushlogic interpeter) may freely convert between ASCII string and binary representations of integers as it wishes.

Pushlogic object also uses the special constant bottom (also known as backstop) ($\perp$).

The null string, the string ‘false’ (whether in quotes at source level or not), the integer zero and any strings containing only zeros are the values that represent logical false in the evaluation of Boolean operators.

The null string is defined as the string of length zero. There is no separate ’null pointer’ version of the null string and string variables cannot be ’null’. Any platform that uses the null pointer to represent the null string must ensure the two are fully identified, under comparison and so on.

3.2 Event Constants

Certain fields or variables range over events. The event name is the same as the name as the field or variable that relays it. Events may be parameterised with an event constant that is a string or integer. An event type is a set of unique strings and/or integers that are the event constants possible for a given event. The null event constant denotes that an event is not currently occurring.

3.3 Pushlogic Types

Types in SPL1 are sets containing enumerations of constants and/or integer ranges. Types are designated using type expressions. A type expression is the name of a type or a list of constants enclosed in braces. A colon may partition the list: this defines the values before the colon as safe values. The constants are strings, which do not need quotes in this context, unless they contain spaces, or integer ranges of the form '`nn .. mm` ' where nn and mm are integers. The word '`event`' may be placed in front of the opening brace to declare an event type. An event type has an implied safe value that is the empty string and the given constants are unsafe values.

```
type-expression ::= <id> |
             {  u1 u2 ... } |
             {  s0 s1 ... : u1 u2 ... } |
             event { e1 e2 ... } |
             fuse | bool | lock
```

Named types are defined with the '`sort set`' statement but the right-hand side type expressions can be used inline as anonymous types. Example:

```
type-declaration ::= sort set <id> = <type-expression>
```

```
sort set <typename> = {  s0, s1, ... : u1, u2, ... };
```

3.3.1 Lock Type

The lock type is built in to Pushlogic and ranges over any string constant. Its safe value is the null string. Only the bundle that last set a lock type to a non-null value can set it to any other value.

3.3.2 Fuse Type

The fuse type is built in to Pushlogic and defined as follows:

```
sort set bool = {  true : false };
```

3.3.3 Boolean Type

The Boolean type is built in to Pushlogic and defined as follows:

```
 sort set bool = {  true, false };
```

The sort ‘`bool`’ is already defined by the system and contains the reserved constants ‘`false`’ and ‘`true`’, both as unsafe values.

It is an error to define any field to range over the reserved constants ‘`false`’ or ‘`true`’, unless the field ranges over exactly these two fields. They may be safe or unsafe.

Examples:

```
  sort set  d0 = bool;               // OK - making use of predefined sort.
  sort set  d1 = { true : false };   // OK - true is safe.
  sort set  d2 = { true false };     // OK - both unsafe, same as boolean.
  sort set  d3 = { false : 0..9 };           // Illegal.
  sort set  d3 = { false : true red brown }; // Illegal.
```

Chapter 4

Pushlogic Object Level (VM Execution)

Pushlogic originally generated its own native object-level bytecode. This was considered the reference standard for code reflection. Anything that generated this form of code could particpate in the system.

Now (2008), the Pushlogic compiler also generates .net IL code. The code can be checked whether conformant to one or more *checkablity profiles* and the domain manager will reject bundles that are outside its supported set of checkability profiles.

4.1 Code Reflection Schema

Pushlogic native object level is considered the primary form of represenation. It exists in bytecode and XML forms.

The XML schema for Pushlogic code reflection is as follows. A separate schema is used for data reflection via the tuplecore (todo: where is it listed?).

```
<?xml version="1.0" encoding="UTF-8"?> <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="bundle">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="PK_subxw"/>
        <xs:element maxOccurs="unbounded" ref="PK_domain"/>
        <xs:element maxOccurs="unbounded" ref="PK_rule"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="PK_subxw">
    <xs:complexType mixed="true">
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="PK_fieldr"/>
      </xs:sequence>
      <xs:attribute name="no" use="required" type="xs:integer"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="PK_fieldr">
    <xs:complexType>
      <xs:choice>
        <xs:element ref="PK_fielda"/>
        <xs:element ref="PK_tup"/>
      </xs:choice>
      <xs:attribute name="arg1s" use="required" type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="PK_domain">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="PK_s"/>
        <xs:element minOccurs="0" ref="PK_ellipsis"/>
        <xs:element ref="PK_fieldd"/>
      </xs:sequence>
      <xs:attribute name="mode" use="required" type="xs:NCName"/>
      <xs:attribute name="safecount" use="required" type="xs:integer"/>
      <xs:attribute name="spare" use="required" type="xs:NCName"/>
      <xs:attribute name="unsafecount" use="required" type="xs:integer"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="PK_ellipsis">
    <xs:complexType/>
  </xs:element>
  <xs:element name="PK_fieldd">
    <xs:complexType>
      <xs:choice>
        <xs:element ref="PK_fielda"/>
        <xs:element ref="PK_tup"/>
      </xs:choice>
      <xs:attribute name="arg1s" use="required" type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="PK_rule">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="PK_fieldw"/>
        <xs:choice>
          <xs:element ref="PK_query"/>
          <xs:element ref="PK_s"/>
        </xs:choice>
        <xs:element maxOccurs="unbounded" ref="PK_skip"/>
      </xs:sequence>
      <xs:attribute name="timer" use="required" type="xs:integer"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="PK_fieldw">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="PK_fielda"/>
      </xs:sequence>
      <xs:attribute name="arg1s" use="required" type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="PK_tup">
    <xs:complexType>
      <xs:attribute name="arg1tup" use="required" type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="PK_fielda">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="PK_tup"/>
      </xs:sequence>
      <xs:attribute name="arg1s" use="required" type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="PK_s">
    <xs:complexType>
      <xs:attribute name="arg1s" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="PK_query">
    <xs:complexType>
      <xs:sequence>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="PK_deqd"/>
          <xs:element ref="PK_query"/>
          <xs:element ref="PK_s"/>
          <xs:element ref="PK_subxr"/>
          <xs:element ref="PK_and"/>
        </xs:choice>
        <xs:element minOccurs="0" ref="PK_backstop"/>
        <xs:element ref="PK_skip"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="PK_and">
    <xs:complexType>
      <xs:sequence>
        <xs:choice maxOccurs="unbounded">
          <xs:element ref="PK_deqd"/>
          <xs:element ref="PK_subxr"/>
        </xs:choice>
        <xs:element ref="PK_skip"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="PK_backstop">
    <xs:complexType/>
  </xs:element>
  <xs:element name="PK_skip">
    <xs:complexType/>
  </xs:element>
  <xs:element name="PK_deqd">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="PK_query"/>
        <xs:element minOccurs="0" ref="PK_subxr"/>
        <xs:element ref="PK_s"/>
        <xs:element ref="PK_skip"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="PK_subxr" type="xs:NMTOKEN"/>
</xs:schema>
```

This schema is partial since it does not currently list all of the diadic operators. The PK_skip statement is shown instead of the pushback info in this version.

4.2 Virtual Machine

The bytecode is defined in this table:

```
Pushlogic Byte Code Descriptions : DRAFT.

For the bytecode representation, self-describing args preceed
the bytecode and fixed-field args follow the byte code.

For the XML representation, self-describing args are included as
sub-elements of the element and the fixed-field args are put
as attributes.


OPERATORS

--
CODE PK_and    2

Logical and function.

Three stack args : left, right and pushback info.
No postfix attributes.

--
CODE PK_or     3

Logical or function.

Three stack args : left, right and pushback info.
No postfix attributes.
--

CODE PK_not    4

Logical not function
One stack arg.
No postfix attributes.

--
CODE PK_inv    5

Bitwise complement.

One stack arg.
No postfix attributes.
--
CODE PK_xor    6

Logical xor function.
Three stack args : left, right and pushback info.
No postfix attributes.

---
CODE PK_divide 32

Division operator.
```

```
Three stack args : left, right and pushback info.
No postfix attributes.
---

CODE PK_mod 33

Modulus operator.
Three stack args : left, right and pushback info.
No postfix attributes.
---


CODE PK_fqgt 34


Four quadrant greater than operator.
Three stack args : left, right and pushback info.
No postfix attributes.
---

CODE PK_multiply 35

Multiplication operator.
Three stack args : left, right and pushback info.
No postfix attributes.

---
CODE PK_query  14

Conditional expression operator.
Four stack args: guard, true-exp, false-exp, pushback info.
No postfix args.

---
CODE PK_deqd  15

Equality comparison operator.
Three stack args : left, right and pushback info.
No postfix attributes.

---
CODE PK_dgtd  16

Greater than comparison operator.
Three stack args : left, right and pushback info.
No postfix attributes.

---
CODE PK_dged  17

Greater than or equals comparison operator.
Three stack args : left, right and pushback info.
No postfix attributes.

---
CODE PK_plus 20

Addition operator.
Three stack args : left, right and pushback info.
No postfix attributes.
---

CODE PK_minus 21

Subtraction operator.
Three stack args : left, right and pushback info.
No postfix attributes.
---

CODE PK_tup 22

Reference to a tuple in the current domain of participation -
sometimes currently used instead of localroot.
No stack args.

One postfix attribute: the tuple name.




CONSTANT VALUES


CODE PK_s      1

Defines a string constant

No stack args.
One postfix attribute: the string itself.
```

```
---
CODE PK_backstop 19

A constant value which when assigned does not change the current value.
No stack args.
No postfix args.


---
CODE PK_int 25

No stack args.
In bytecode, a representation of integers in base 256.
In XML, integer elements appear directly.

---
CODE PK_nint 26

No stack args.
In bytecode, a representation of integers in base ???.
In XML, integer elements appear directly.

---
CODE PK_ellipsis 28

A constant value which denotes other possible values in a domain declaration.
No stack args.
No postfix args.

---
CODE PK_range 29


Declares a subrange of the integers.
Two stack args: from and to inclusively.
No postfix args.

---

TOP LEVEL ITEMS

--
CODE PK_rule   7

Defines an executable rule

Six stack args are left-hand side tuple, left-hand field, right-hand side, pushback, spare and spare.
In XML form, the left-hand side tuple and field are combined and use a single element.
One postfix attribute: nominally called timeout but currently unused.

--
CODE PK_subxw   8

Define a subexpression.
One stack arg: any expression
One postfix attribute: the subx number.
---
CODE PK_subxr   9

Read a subx defined by subxw.
Zero stack args.
One postfix attribute: the subx number.


CODE PK_domain 30

Declares the range of values for a field.
Stack args: the safe values followed by the unsafe values.
Three postfix args: type, safe values, unsafe value.

----
CODE PK_safetylive 24 // not used now

An Oldform ?

----
CODE PK_sl      40

Safety/live rule definition.

Three stack args : guard, true/false and message.
No postfix attributes.

The middle value is true for a safety rule and false for a live rule.
The message should be reported if the rule fails.
Fairness constraints are to be added!

----
CODE PK_fieldl 41

not used anymore?
```

```
FIELD AND TUPLE ACCESS
---
CODE PK_localroot   13

Explicit name for the local root tuple - sometimes currently omitted by implication.
No stack args.
No postfix args.


---

CODE PK_uri 27

Reference to a tuple on another execution platform.

No stack args.
One postfix attribute: the uri.

----
CODE PK_fielda 23

A field reference where the field contains a tuple.
One stack arg: a tuple.
One postfix arg: the field name.

---
CODE PK_fieldw  10

A field reference for writing.
One stack arg: a tuple.
One postfix arg: the field name to be written.

---
CODE PK_fieldr 18

A field reference for reading.
One stack arg: a tuple.
One postfix arg: the field name to be read.

----
CODE PK_fieldlk 39

A field declaration where the field will be a lock.

One stack arg: a tuple that will contain the field.
One postfix arg: the field name.


---

CODE PK_fieldd 36

A field declaration.
One stack arg: a tuple that will contain the field.
One postfix arg: the field name.

----





MISCELLANEOUS

CODE PK_eventr 42

event read: pushlogic interepreter internal use only.
Does not appear in bytecode currently - inferred by bcload.

---
CODE PK_pbind 37

Pushback info.

??? currently being changed sligthtly

---
CODE PK_pbval 38


Pushback info.
```

```
??? currently being changed sligthtly


---
CODE PK_meta   11

Not used.

---
CODE PK_skip   12

A nop.
No stack args.
No postfix args.

----

CODE PK_eob 31

End of bundle marker - not used in XML form.
No stack args.
No postfix args.


EOF
```

The XML schema essentially follows the bytecode, in that a very simple tree walker is all that is needed to convert XML form to bytecode. The current push-logic interpreter indeed converts it in this way.

Pushlogic has three forms of representation: source, canned object and re-hydrated object. In this section we give the semantics of re-hydrated object, which is regarded as the primary form of Push Logic. It is actually a bytecode suitable for automated checking, distribution and loading into an execution platform (bytecode interpreter).[1]

A Pushlogic program, at the object level, is a set of rules known as a '*bundle*'. Pushlogic programs are aggregated into a domain of participation by disregarding bundle boundaries and forming the union set of rules, but a bundle may not be admitted if automated formal tests fail for any rule within it. These tests are given below. Each one is called a '*constraint*'. A race can arise if an attempt is made to load a pair of incompatible bundles at once: the system will accept only the rules from the first bundle.

Pushlogic object rules have three forms: executable, liveness and safety. The source compiler can convert more complex temporal constraints into a conjunction of live and safety checks for the object form to handle. Liveness and safety rules are assertions to be checked by a model checker invoked by the loader. The loader also has certain built-in constraints, described below, that require model checking. (The model checker can be run as a network service or as a process on the target execution platform.) The combination of the current bundle and all other rules in the domain of participation is model checked. If all assertions still hold, then the new bundle is acceptable to the domain and becomes part of it. Execution of its executable rules is then allowed.

[1]We also plan to implement a compiler that generates native C code from the bytecode, because the current byte code interpreter is RAM-hungry and slow.

4.3 Pushlogic Expressions

Pushlogic atomic object expressions are the constant values (strings, integers and bottom ($\perp$)) and the values of fields.

More complex expressions are built up using operators. The operators are any deterministic, referentially-transparent functions, including the normal Boolean connectives and the conditional expression construct (query-colon). The full core set of operators found in Java and C++ is supported, along with string catenation. Function calls could be provided for brevity, but these are currently fully inlined by the compiler and so do not appear in the object code.

Expressions are either level or event expressions. A level expression is a function of state whereas an event expression will only have a non-null value momentarily.

Arithmetic and comparison operators are provided using the normal coercion rules that allow strings and integers to be interchanged, as found in languages such as Perl. $\perp$ is represented with a unique byte code that is disjoint from any string: $\perp$ does not need to be explicitly written in the Pushlogic source level.

A full list of operators should be given here. Note that string cat uses dot and $\uparrow$ is xor.

A `FQGT` operator is provided that performs a comparison in the style of greater-than but with the assumption that a pair of integers in a adjacent quadrants of the number space have not lapped each other. Example needed here ...

4.4 Fields and their Declaration.

Our current, main Pushlogic implementation is part of a distributed tuple space platform and hence variables are called fields. Most fields are either events or range over the constant string/integer values defined above. Apart from strings, integers and events, the tuple space platform supports some other types, including level, primary key, credit, URI and so on. Their definition is beyond the scope of this document and so only mentioned in passing.

A tuple is a collection of fields, each named with a tag string. A field may contain a nested tuple.

Every tuple has a field called '*level* (i.e. the tag string for that field is 'level'). It contains a negative integer.

$$lev(v) \in Z^-$$

All fields and tuples are part of a global shared address space, and in principal, can be accessed from any bundle. However, the concepts of domain, device, pebble and bundle each impose overlays of name aliasing and access control on the global space.

Certain tuples are associated with the current DoP, certain with the current device or execution platform and certain exist only for private, local access by the current bundle. Like a bundle, a pebble also possesses a set of associated local fields, but these are not private: indeed they are used as shared variables for communication with the Pebble.

As a coding convention, users of Pushlogic are invited to use first-letter capitalisation for then names of fields that contain tuples and lower case for fields that are variables.

Fields that are local to the current bundle are declared with the keyword '`local`'. Fields that are shared and provide input or output to the current bundle are declared with the keywords '`input`', '`inout`' or '`output`'. Actually, at object level, the keywords are replaced with Pushlogic bytecodes.

The built in types, such as '`lock`', '`bool`' or '`fuse`' can be used at source level as a short hand to declare local fields. For example, the following two lines show equivalent ways of defining booleans.

```
    local var1, var2 : bool;
    bool var3, var4;
```

The definition of fields and tuples is identical at source level and object level and is unchanged by the compiler.

When a field is declared, a range of values for it may also be declared. If the range is to be used frequently, it can be named and represented as a '`type`' and then refered to by its name. The range is denotation of a list of constant strings, but integer subranges can be used to specifiy parts of this list.

The range of values is partitioned into *safe* and *unsafe* values. The the list of safe values is definitive, whereas code should operate correctly and all assertions pass if further unsafe strings or integers occur during run time. This supports dynamic system extensibility. The elipsis construct is allowed in list of unsafe value at the the pushlogic source level, but serves only to alter behaviour with respect to compile-time warnings.

One of the safe values, the first listed in any list, is known as the nominal starting value for that field, and can be used as such in offline checks that involve reachable state analysis.

When a bundle (or pebble) is introduced to a DoP, if the nominal starting value of any of the the newly created fields associated with the new bundle (or pebble) is inconsistent with extant domain values, fields may be set to **any** value that makes the rules consistent, but with priority being given first to nominal starting values, then to other safe values, then to listed unsafe values.

If insertion of a new bundle (or pebble) into a DoP would immediately cause a pushback inside the domain, this should perhaps be flagged at a meta-level to the operative attempting the insertion.

4.5 Level and Event Expressions

Expressions are either level or event expressions. It is possible to define the distinction in a syntax-directed way, as given shortly. However, the current and preferred implementation is not to rely on syntax but to use the elaboration rules in the compiler to determine whether a given expression is acceptable in a given context. The rules deny the conjunction of events from separate sources and require that all code reaches closure under repeated symbolic elaboration (§5.14.2). The elaboration rules offer a richer language since the syntax-directed form cannot easily encompass concepts of an differentiator enclosed in an integrator.

4.5.1 Syntax-directed guide to level and event expressions

The following rules would be broadly sufficient to distinguish level expressions from event expressions, but they are intended as a guide to programmers and are not a definition.

A level expression is essentially an expression with no leaf variables of type '`event`' and which does not contain the differentiation operator. It maintains its value unless any of the supporting variables changes their current value (their level in hardware terms).

An event expression is a function of one or more variables of type '`event`' or which contain the differentiation operator. Not all operators support event arguments: those that do are conjunction, disjunction and conditional expression (query-colon construct); negation of events is forbidden. Disjunction of event expressions (logical OR) is always allowed but conjunction requires that both arguments stem from a single external event delivered to the current bundle. The basis of this is that simultaneous delivery of external events is not meaningful.

4.5.2 Assertions on Level and Event Expressions

The '`always`' assertion asserts that a condition must always hold and its argument must be a level expression. The '`never`' and '`live`' assertions may apply to either type of expression.

Further temporal logic operators, such as '`until`' and '`w-until`' and general CTL expressions, are not currently supported, but will have the requirements on their arguments that would arise from decomposition into safety and liveness form.

4.6 Executable Rules

Each executable rule is an assignment of the form

$$f := exp : pbind$$

where f is a field (a scalar variable name) in a global name space and exp is a Pushlogic object expression and $pbind$ is compensation information that asists in reversing the operation of the rule. This information also identifies certain of the expression support (i.e. certain of the free variables occuring in exp) as the *sensitive parents* of the rule.

All pushlogic rules, at the object level, are composed in parallel and execute simultaneously.

D1: Rule Execution: The reference execution model for an executable Pushlogic rule is that all subexpressions 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 the same execution platform as the Pushlogic that arise owing to 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.

An event nominally holds for one gated cycle. All event fields are re-set to the null string after the cycle where they were processed.

4.6.1 Nominal Meaning of a Rule

A Pushlogic rule is to be thought of as holding at all times, except when it is assigning $\perp$, at which times the assigned field retains its current value or is controlled by another rule. Push Logic is designed such that each rule may also be interpreted as an assertion about the current state of the assigned field and the conjunction of all such assertions should hold at all times. However, because fields may be held on physically separate devices, and Push Logic interpreters interchange network messages when fields need to be changed, this conjunction will not globally hold for brief periods while network messages are in flight or during network disconnect.

4.6.2 Event and Level Constraints

The occurance of event expresions in executable rules is restricted by the following schema. The executable rule must be factorisable into one of three forms, where ee denotes an event expression, le denotes a level expression, lv denotes a level variable and ev denotes an event variable. The four forms are:

$$lv \quad := \quad (le)?le : \perp;$$

$$\begin{aligned} ev &:= (le)?ee : \bot; \\ lv &:= (ee)?le : \bot; \\ lv &:= (le)?ee : \bot; \end{aligned}$$

The second form is known as an ’emit’ assignment (§5.10.1).

The third and fourth forms are known as integrations.

4.6.3 Unilateral Reset to Safe Value

Unilateral changes to shared fields can occur for various reasons. For instance, a device may be disconnected from the network and an application that was relying on it can no longer function. As a second instance, consider a Pebble that controls the drainage valve on a cistern. A Pushlogic script may make the assignment that opens the valve, but the Pebble may make autonomous (unilateral) action to reset the valve to its closed state (a safe state) once the cistern is empty. This is allowed within our definition of a Pebble since the field is stored within the Pebble.

Local fields will not suffer unilateral changes: all changes will accrue from executable rules in the associated bundle.

4.6.4 Pushbacks: Simple and Complex Undo

When an executable rule makes a change to a field that fails (a failed update), the rule is obliged to compensate with corrective action. Otherwise, the rule could no longer be viewed also as a universally quantified assertion about the state of the domain. The update may fail straightaway or else the field may revert to some safe value after an interval of time. Either way, the system must perform a ‘*pushback*’ after a failure. A ‘*simple undo*’ involves changing a sensitive parent of the rule to a safe value that again makes the rule hold (when viewed as an assertion). An undo must also be performed by the rule if another rule or an external agent changes its driven field.

Push back information must be provided where there would otherwise be more than one possibility for compensating. If an external agent (or other rule) changes the value of the assigned field to one of its safe values, the push back indication uniquely identifies fields occurring in the associated expression that can be changed to make the rule hold. For information loosing operators, values must also be provided. For example, with logical not, no indication is required, because the new value is obvious at push back time. For comparison when pushed back so it holds, then 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. For comparison, when pushed back to false, a value and operand must be specified, since, in general, there are many possible vaues that will make a comparion fail. For conjunction, when pushed back to false, knowning which operand to push on is required, since either will do, whereas to push conjunction 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.

The pushlogic source '`cut`' function, (or whatever it is currently called!) does not appear in the object code - it serves only to influence the push back information.

4.6.5 Complex Undo

The complex undo is expanded at compile time using extensions of the `fuse` statement. No object-level representation is needed.

4.7 Inter-Bundle Communication

All communication between entities is through shared variables. (Multimedia is supported using the notion of third-party setup, where a field in a source pebble is set to the same value as a field in a sink pebble, where the value acts as a virtual circuit identifier).

Where a bundle alters the value of a field held on a remote resource, the run time system generates network traffic, such as a SOAP RPC [16]. Where a bundle is sensitive to changes on remote resources, it uses a soft-state registration protocol (a UDP version of UPnP's GENA) that causes it to receive notification of changes. An inout field may be set to one of its non-safe values by at most one bundle.[2]

Changes in local or remote field values are notified to the Pushlogic execution engine because it registers for appropriate notification callbacks.

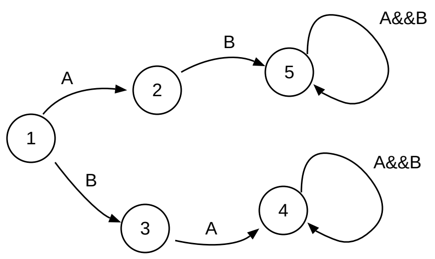
$$v := [\![\, exp \,]\!]_\sigma$$

4.8 Standing Constraints

Apart from embedded safety and liveness constraints coming from bundles, every DoP is subject to the following standing system constraints. A bundle cannot be loaded if it would violate any of these constraints.

PLC1: Level Ordering Constraint: For every rule, the level of the assigned field must be less than any level found in the supporting parents (free fields of) its

[2]Currently we have implemented our own protocol, running over UDP, called ETC, that implements the field writes, remote registrations for events and code and API reflection, but there is little reason not to use the standard protocols.



```
if (s==1 && A) s := 2;
if (s==2 && B) s := 5;
if (s==1 && B) s := 3;
if (s==3 && A) s := 4;
```

Figure 4.1: Example of a Race Hazard

assigned expression

$$\forall f' \in sup(exp).lev f < lev(f') \vdash f := exp$$

Fields may be allocated new levels by the system as part of the process of loading a bundle. However, this is not always successful because a pair of rules in different bundles may have contradictory field level requirements. Where the fields are held on different execution platforms, a distributed algorithm may be run to establish their levels. If this constraint cannot be met, the bundle is not loaded.

The level ordering constraint does not apply to local variables, including lock or fuses or compiler-generated program counters.

PLC2: Consistency Constraint: A Pushlogic object expression may be deterministically evaluated in an environment, σ, to produce a string or else the special value bottom, $\perp$. When a rule produces $\perp$, it has no affect on the assigned field: $\sigma[\perp/v] = \sigma$.

To ensure consistency (i.e. to avoid fighting between rules), all rules that assign a given field must evaluate to a common non-bottom value or bottom itself in all possible environments. Each rule of a new bundle is checked against the existing participating rules and the bundle is not loaded if any violates this condition.

PLC2b: No race hazards: Any program that contains a race of the nature shown in figure 4.1 will not unwind during compilation since the final result depends on the order of interleaving.

PLC3a: Safe Value Constraint A: A pair of bundles is incompatible if they disagree on their safe value declarations for any field.

PLC3b: Safe Value Constraint B: There must exist at least one setting of the fields such that all executable rules hold and all fields that have safe values defined are set to one of those safe values.

PLC3c: Safe Values Constraint C: The safe values of a field must be a subset of the safe values that any expression assigned to that field can safely generate, or else the expression must be able to safely generate bottom. By safely generate, we mean that the expression generates that value when all of its supporting fields are set to their safe values. This constraint ensures that when a pushback occurs, and an assigned value is set to one of its safe values, the assigned expressions can

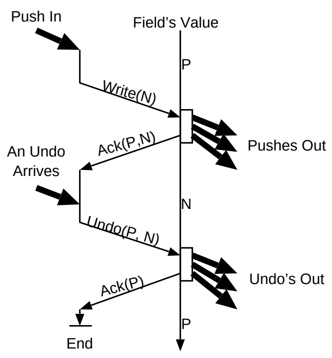


Figure 4.2: Pushlogic Undo Sequence (Failed Update)

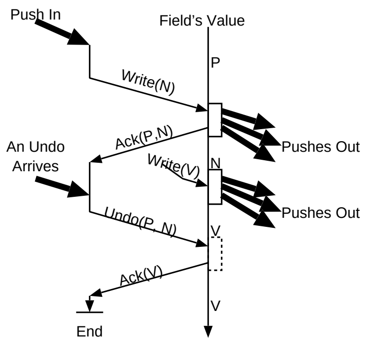


Figure 4.3: Undo Sequence with Race Condition

push back on their support, safely, so that the rule holds. (This constraint is stricly stronger than PLC3b ?).

PLC3d: Safe Live Insertion Constraint: An insertion into a DoP that causes an immediate pushback may be considered an error in some applications. At least a warning or deferral should be offered to the operative.

PLC4: Push Back Uniqueness: Only one possible pushback procedure should be possible for each possible pushback circumstance. If there is more than one, then the $pbind$ information is ambiguous. The constraint implies that where a parent is shared between multiple rules of a bundle there is just one change to the parent that is acceptable with respect to this constraint for all rules sharing that parent field. [3]

D7: Undo Race Avoidance: Where an undo operates over the network between different execution platforms, a race can arise, where another independent change to the driven field has been performed by another rule or external agent. This normal course of events is illustrated in figure 4.2, whereas figure 4.3 shows the race condition. To overcome races, a simple undo ... TBD. Needs work.

Declarative programming languages aim to consist of a set of declarations that all

[3]This constraint is concave, in that two parts of an admissable bundle, considered as separate bundles, might be inadmissable in isolation.

hold at all times. On the other hand, many useful control sequences cannot be expressed entirely declaratively. For instance, when we press the ‘skip forward’ key on a CD or DVD player, we expect it to jump to the next track or index point. When we press the button again, a new definition of ’next’ now pertains and so the operation is not idempotent. Implementation of this feature requires both a differentiation to detect the active edge of the button push and an integration to accumulate the increment to the track number.

PLC5: Idempotency Constraint: Any Pushlogic program will result in no further output changes if ‘executed’ more than once without unilateral (external) change of any field.

Idempotency is assured if every integration is uniquely associated with a differentiation. In addition, an integration of the form $a := a + 1$ breaks the Level Ordering Constraint in that a occurs on both side of the rule and hence the level on both sides is equal.

The solution to this dichotomy is provided by the Gated Update mechanism.

The gated nature of updates to fields all held on a common platform allows that certain rule combinations to operate deterministically when they would not otherwise. Consider the following pair of rules where d1 is a tightly-coupled field:

$$\begin{aligned} d1 &:= d; \\ d2 &:= (d\&\&!d1)?1 :\perp \end{aligned}$$

This pair will reliably set d2 to one whenever d goes from false to true. Without the gated-update constraint, the second rule might always be executed after the first rule and hence the guard would never hold. In our current implementation, a tightly-coupled field is any field held on the same execution platform and these are readily determined because their path name starts with the local root.

The int-diff constraint provides that an infinite `while` loop wrapped around a parallel statement, that contains no internal `waits`, is not sensitive to the additional loops made by the thread.

PLC6a: Montonicity constraint: All rules must meet the other constraints with any amout of extension to the range of unsafe values in any field.

The constraints on allowable Pushlogic programs (union of unbundled rule bundles) have just been presented. These constraints can be checked without full possession of the Pushlogic object code. This could be very useful in large systems or where IP or policy needs to be protected. A summary form for a bundle may be defined that lists for each assign field the level constraints and the possibly assigned values. Where the Pushlogic Timer or other common resource is frequently depended on, then the summary can helpfully mention this explicitly. For instance, a set of different bundles might control a common field at different times of day and these would appear incompatible if time were ignored, but by including it in the summary, the bundles become compatible.

PLC6b: Montonicity constraint: Any bundle, whose assertion guarantees rest on the presence of other bundles that might become unloaded, is not allowed. (This is checked using a non-deterministic presence guard for each unloadable bundle by the domain manager).

PLC7: No oscillations constraint: Oscillation is created by inverting loops. Any bundle that contains an internal oscillating loop cannot be elaborated by the compiler and causes a compile-time error. When bundles are brought together with each other or world models any oscillators then formed are detected by the domain manager and the union is not allowed.

For example, the following two lines are inconsistent since they form an oscillator. If the lines are in the same bundle, a compile-time unwind fail error is flagged. If they are in different bundles but the variables are bound to form a distributed inverting loop then the oscillation error is flagged by the domain manager.

```
 a := b;
 b := !a;
```

Many device control loops are oscillators, such as those used in thermostats. In order to implement these, the loop must be broken using a time delay at some point to regulate the maximum frequency of oscillation. For example:

```
with Timer#Countdown if (#atimer == 0)
{
   #atimer := 1000; // Delay for one second
   b_delayed = b;
}
 a := b_delayed;
 b := !a;
```

Resynchronisation Constraint: A liveness predicate expresses that two supposedly-coupled systems will eventually become re-synchronised after network distruption has finished.

4.9 Temporal Logic Assertions

A bundle may contain a number of safety or liveness assertions. These take expressions as arguments and the expressions have the form defined in §4.3.

Object-level safety and liveness assertions are checked by the Domain Manager at bundle load time. They are retained by the Domain Manager for further checking against newly arriving bundles. Monotonicity constraints imply they cannot be violated by bundle or pebble departure.

Chapter 5

Pushlogic Source Language

NB: There is a user manual for the source compiler in html on the website. This chapter discusses the source language per se, rather than how to compile it.

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 and currently no arrays. 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. The Pushlogic constraints on a bundle are implemented at bundle load time, but, as far as possible, are also implemented by the compiler to give advanced warning.

5.1 Concrete syntax tree

The concrete syntax tree is following yacc file:

```
/*
 *  $Id: pushlogic.yy,v 1.35 2008/07/11 08:15:04 djg11 Exp $
 *
 * Bigtop.
 # CBG Badger Tuplers Project
 # University of Cambridge
 # Computer Laboratory
 # (C) 2004 David J Greaves
 #
 # Mostly Written Romsey - Dec 2004.
 #
 *
 * Pushlogic Grammar
 */
```

```
%{
#include "cbglish.h"
#include <stdio.h>

#define PL_LINEPOINT(X) add_linepoint("pl_linepoint", X)

%}

%union {
builder *auxval;
        }
%type <auxval> pl_assoclist pl_associtem bytecode
%type <auxval> pushl_decls prog module value value_list value_list_or_nil
%type <auxval> pushl_def pl_formals pl_actuals pl_formal pushl_defstyle
%type <auxval> pushl_statement pushl_statement_list case_item case_item_list pl_type
%type <auxval> pushl_statename pushl_statechartlist pushl_state pushl_stateitemlist pushl_stateitem pushl_statespec
%type <auxval> sd_id_comma_list pushl_field_decl pushl_macro_decl pushl_macro_decl_list pushl_field_decl_list
%type <auxval> exp exp1 exp2 exp3 exp4 exp5 exp6 exp7 exp8 exp9 exp10 exp11 exp12 exp12a exp13 exp13a
%type <auxval> string_option
%type <auxval> formal_comma_list exp_comma_list exp12_comma_list
%type <auxval> xml_file xml_element xml_element_list xml_att_list
%%


prog:
  pushl_decls
{
builder *r = $1;
results = LISTCONS(r, results);
$$ = freverse(results);
}
;

bytecode:
 /* nothing */ { $$ =
LISTEND(0);
}
| sd_string sd_number bytecode { $$ =
LISTCONS(TREE2("pl_stringtab", $1, $2), $3);
}

| sd_number bytecode { $$ =
LISTCONS(TREE1("pl_bytecode", $1), $2);
}
;

pushl_decls:  /* nothing */ { $$ =
LISTEND(0);
}
| module pushl_decls { $$ =
LISTCONS($1, $2);
}
;

pushl_statement_list:  /* nothing */ { $$ =
LISTEND(0);
}

| pushl_statement pushl_statement_list { $$ =
LISTCONS($1, $2);
}
;




/* A simple xml parser here for reading in compiled bundles and code reflected fragments */

xml_qbody:  /* It would be better to handle this xmlq in the lexer ? */
  sd_id xml_qbody
| sd_string xml_qbody
| ss_equals xml_qbody
| ss_minus xml_qbody
| ;

xml_element_list: ss_dltd ss_slash sd_id ss_dgtd { $$ =
LISTEND(0);
}

| xml_element xml_element_list { $$ =
LISTCONS($1, $2);
}
;

xml_att_list: { $$ =
LISTEND(0);
}

| sd_id ss_equals sd_string  xml_att_list { $$ =
LISTCONS(TREE2("xml_att", $1, $3),  $4);
}
;
```

```
xml_element:
  ss_dltd sd_id xml_att_list ss_dgtd xml_element_list
{ $$ = TREE3("xml_element", $2, $3, $5); }

| ss_dltd sd_id xml_att_list ss_xml_singleton
{ $$ = TREE3("xml_element", $2, $3, LISTEND(0)); }

| sd_id  { $$ = TREE1("xml_chars", $1); }
| sd_number  { $$ = TREE1("xml_int", $1); }
;


xml_file:
  ss_xmql xml_qbody ss_xmqr xml_file  { $$ = $4; }
| xml_element  { $$ = $1; }
;

module:
   pushl_def pl_semicolon_opt { $$ = $1; }

|  pushl_statement { $$ = $1; }

|  ss_pling bytecode {
        $$ = TREE1("pl_compiled_bundle", $2);
}

|  ss_xmql xml_qbody ss_xmqr xml_file  { $$ = TREE1("pl_xml", $4); }
;


pl_semicolon_opt: ss_semicolon | /* nothing */
;


pl_formals:
  ss_lpar ss_rpar
{ $$ = NULL; }
|  ss_lpar formal_comma_list ss_rpar
{ $$ = $2; }
;

pl_actuals:
  ss_lpar ss_rpar
{ $$ = NULL; }
|  ss_lpar exp_comma_list ss_rpar
{ $$ = $2; }
;


pushl_defstyle:
   sd_id { $$ = $1; }
|  s_pebble { $$ = mkstring("pebble"); }
;

pushl_def:
  s_def pushl_defstyle sd_id pl_formals ss_lsect pushl_decls ss_rsect
     { $$ =
TREE4("pl_def_bundle", $2, $3, $4, $6); }
;


pushl_macro_decl:
  exp12 ss_equals exp { $$ =
   TREE2("", $1, $3);
}

;

pl_type:
  ss_lsect value_list ss_colon value_list ss_rsect  { $$ =
   TREE2("pl_safeunsafe", $2, $4);
}

| ss_lsect value_list ss_rsect { $$ =
   TREE1("pl_unsafe", $2);  /* depreciated form */
}

| s_event  { $$ =
TREE1("pl_event", LISTEND(0));
}

| s_event ss_lpar value_list_or_nil ss_rpar { $$ =
TREE1("pl_event", $3);
}

| s_fuse{ $$ =
TREE0("pl_fusetype");
}

| s_lock{ $$ =
TREE0("pl_locktype");
}

| sd_id { $$ =
TREE1("pl_tid", $1);
}
;

pushl_field_decl:
  exp12_comma_list ss_colon pl_type { $$ =  /* preferred form for now on */
   TREE2("pl_fielddec", $1, $3);
}

| exp12_comma_list  { $$ =
   TREE1("pl_fielddecn", $1);  /* depreciated form - no type given */
}
;

pushl_field_decl_list:
  pushl_field_decl { $$ =
LISTCONS($1, LISTEND(0));
}

| pushl_field_decl ss_comma pushl_field_decl_list
{ $$ =
LISTCONS($1, $3);
}
;


pushl_macro_decl_list:
  pushl_macro_decl { $$ =
LISTCONS($1, LISTEND(0));
}

| pushl_macro_decl ss_comma pushl_macro_decl_list
{ $$ =
LISTCONS($1, $3);
}
;



case_item:
  s_case exp_comma_list ss_colon pushl_statement { $$ =
TREE2("pl_case_item", $2, $4);
}

| s_default ss_colon pushl_statement { $$ =
TREE1("pl_case_default", $3);
}
;



case_item_list:
 case_item { $$ =
LISTCONS($1, LISTEND(0));
}

| case_item case_item_list { $$ =
LISTCONS($1, $2);
}
;

pl_assoclist:
 pl_associtem { $$ =
LISTCONS($1, LISTEND(0));
}

| pl_associtem ss_comma pl_assoclist { $$ =
LISTCONS($1, $3);
}
;


pl_associtem:
sd_id ss_equals exp { $$ =
TREE2("pl_associtem", $1, $3);
}


pushl_statement:
  s_sort s_set sd_id ss_equals pl_type ss_semicolon { $$ =
   PL_LINEPOINT(TREE2("pl_sortset_dec", $3, $5));
}


| ss_lsect pushl_statement_list ss_rsect s_fuse exp ss_semicolon { $$ =
  PL_LINEPOINT(TREE2("pl_fused", TREE1("pl_block", $2), $5));
}
```

```
| s_pragma sd_id ss_equals exp ss_semicolon { $$ =
  PL_LINEPOINT(TREE2("pl_pragma", $2, $4));
}

| s_fun sd_id pl_formals ss_lsect pushl_decls ss_rsect pl_semicolon_opt { $$ =
   PL_LINEPOINT(TREE3("pl_fun", $2, $3, $5));
}

| s_input pushl_field_decl_list ss_semicolon { $$ =
   PL_LINEPOINT(TREE2("pl_dv", TREE0("dv_input"), $2));
}

/* | s_lock pushl_field_decl_list ss_semicolon { $$ =
 *  PL_LINEPOINT(TREE1("pl_lock", $2));
 * }
 */

| s_output pushl_field_decl_list ss_semicolon { $$ =
   PL_LINEPOINT(TREE2("pl_dv", TREE0("dv_output"), $2));
}

| s_inout pushl_field_decl_list ss_semicolon { $$ =
   PL_LINEPOINT(TREE2("pl_dv", TREE0("dv_inout"), $2));
}

| s_local pushl_field_decl_list ss_semicolon { $$ =
   PL_LINEPOINT(TREE2("pl_dv", TREE0("dv_local"), $2));
}

| s_facet exp ss_equals exp ss_semicolon { $$ =
   PL_LINEPOINT(TREE2("pl_facet_instance", $2, $4));
}

| s_macro pushl_macro_decl_list ss_semicolon { $$ =
   PL_LINEPOINT(TREE1("pl_macro", $2));
}

| s_const sd_id ss_equals exp ss_semicolon { $$ =
   PL_LINEPOINT(TREE2("pl_const", $2, $4));
}

| s_pebble sd_id ss_equals exp ss_semicolon { $$ =
   PL_LINEPOINT(TREE2("pl_pebble", $2, $4));
}

| s_with exp pushl_statement { $$ =
PL_LINEPOINT(TREE2("pl_with", $2, $3));
}

| s_skip ss_semicolon { $$ =
PL_LINEPOINT(TREE0("pl_skip"));
}

| s_break ss_semicolon { $$ =
PL_LINEPOINT(TREE0("pl_break"));
}

| s_continue ss_semicolon { $$ =
PL_LINEPOINT(TREE0("pl_continue"));
}

| sd_id  ss_colon { $$ =
PL_LINEPOINT(TREE1("pl_label", $1));
}

| s_wait  exp ss_semicolon { $$ =
PL_LINEPOINT(TREE1("pl_wait", $2));
}

| s_goto sd_id ss_semicolon { $$ =
PL_LINEPOINT(TREE1("pl_goto", $2));
}

| s_return  exp ss_semicolon { $$ =
PL_LINEPOINT(TREE1("pl_return", $2));
}

| s_switch exp ss_lsect case_item_list ss_rsect { $$ =
PL_LINEPOINT(TREE2("pl_switch", $2, $4));
}


| s_disable exp_comma_list ss_semicolon { $$ =
PL_LINEPOINT(TREE1("pl_disable", $2));
}

| s_meta  pl_assoclist ss_semicolon { $$ =
PL_LINEPOINT(TREE1("pl_meta", $2));
}

| s_live string_option  exp_comma_list ss_semicolon { $$ =
PL_LINEPOINT(TREE2("pl_live", $2, $3));
}

| s_emit exp ss_semicolon { $$ =
PL_LINEPOINT(TREE1("pl_emit", $2));
}

| s_never string_option  exp_comma_list ss_semicolon { $$ =
PL_LINEPOINT(TREE2("pl_never", $2, $3));
}

| s_always  string_option exp_comma_list ss_semicolon { $$ =
PL_LINEPOINT(TREE2("pl_always", $2, $3));
}

| s_while ss_lpar exp ss_rpar pushl_statement { $$ =
PL_LINEPOINT(TREE2("pl_while", $3, $5));
}

| s_forever  pushl_statement ss_semicolon { $$ =
PL_LINEPOINT(TREE1("pl_forever", $2));
}

| s_if ss_lpar exp ss_rpar pushl_statement { $$ =
PL_LINEPOINT(TREE2("pl_if", $3, $5));
}

| s_if ss_lpar exp ss_rpar pushl_statement s_else pushl_statement { $$ =
PL_LINEPOINT(TREE3("pl_ife", $3, $5, $7));
}

| ss_lsect pushl_statement_list ss_rsect { $$ =
PL_LINEPOINT(TREE1("pl_block", $2));
}

| ss_lpsect pushl_statement_list ss_rpsect { $$ =
PL_LINEPOINT(TREE1("pl_parblock", $2));
}

| exp ss_colonequals exp ss_semicolon { $$ =
PL_LINEPOINT(TREE2("pl_assign", $1, $3));
}

| exp ss_plusequals exp ss_semicolon { $$ =
PL_LINEPOINT(TREE2("pl_assign", $1, TREE3("pl_diadic", YYLEAF("pl_plus"), $1, $3)));
}

| exp ss_minusequals exp ss_semicolon { $$ =
PL_LINEPOINT(TREE2("pl_assign", $1, TREE3("pl_diadic", YYLEAF("pl_minus"), $1, $3)));
}

| exp ss_semicolon { $$ =
PL_LINEPOINT(TREE1("pl_e_as_c", $1));
}

| s_stategraph sd_id pl_formals ss_lsect pushl_statechartlist ss_rsect ss_semicolon  { $$ =
TREE3("pl_stategraph",  $2, $3, $5);
}
;


pushl_statechartlist:
/* nothing */
{ $$ =
LISTEND(0);
}
| pushl_state pushl_statechartlist { $$ = LISTCONS($1, $2); }
;

pushl_statename:
   sd_id { $$ = $1; }
|  s_disable { $$ = mkstring("disable"); }
;


pushl_state:
  s_state pushl_statename  pl_formals  ss_colon pushl_stateitemlist s_endstate { $$ =
TREE3("pl_state_def",  $2, $3, $5);
}
| s_state pushl_statename ss_colon pushl_stateitemlist s_endstate { $$ =
TREE3("pl_state_def",  $2, 0, $4);
}
;

pushl_stateitemlist:
/* nothing */
{ $$ =
LISTEND(0);
}
| pushl_stateitem pushl_stateitemlist { $$ = LISTCONS($1, $2); }
;
```

```
pushl_stateitem:
  pushl_statement { $$ = /* implied body: */
TREE2("pl_state_action", mkstring("body"), PL_LINEPOINT($1));
}
| sd_id  ss_colon pushl_statement { $$ =
TREE2("pl_state_action", $1, PL_LINEPOINT($3));
}

| s_exit  ss_colon pushl_statement { $$ =
TREE2("pl_state_action", mkstring("exit"), PL_LINEPOINT($3));
}

| pushl_statespec ss_rarrow1 pushl_statespec ss_semicolon { $$ =
TREE3("pl_transition", $1, $3, TREE0("NONE"));
}

| pushl_statespec ss_rarrow1 pushl_statespec ss_colon pushl_statement { $$ =
TREE3("pl_transition", $1, $3, TREE1("SOME", PL_LINEPOINT($5)));
}
;

string_option:
/* nothing */  { $$ = TREE0("NONE"); }
| sd_string ss_colon { $$ = TREE1("SOME", $1); }
;


pushl_statespec:
  exp { $$ =
TREE1("pl_state_exp", $1);
}
| s_exit{ $$ =
TREE0("pl_state_exit");
}
| s_exit ss_lpar sd_id ss_rpar{ $$ =
TREE1("pl_state_tagged_exit", $3);
}

;


/*
 *
 */



sd_id_comma_list:
/* nothing */
{ $$ =
LISTEND(0);
}
| sd_id { $$ = LISTCONS($1, LISTEND(0)); }

| sd_id ss_comma sd_id_comma_list { $$ = LISTCONS($1, $3); }
;


exp12_comma_list:
  exp12 { $$ = LISTCONS($1, LISTEND(0)); }

| exp12 ss_comma exp12_comma_list { $$ = LISTCONS($1, $3); }
;



exp_comma_list:
  exp { $$ =
LISTCONS($1, LISTEND(0));
}
| exp ss_comma exp_comma_list
{ $$ =
LISTCONS($1, $3);
}
;

pl_formal:
  exp ss_colon exp { $$ =
TREE2("pl_formal", $1, $3);
}

|  exp { $$ =
TREE1("pl_formal_nt", $1);
}
;

formal_comma_list:
  pl_formal { $$ =
LISTCONS($1, LISTEND(0));
}

| pl_formal ss_comma formal_comma_list
```

```
{ $$ =
LISTCONS($1, $3);
}
;


value:
  sd_number ss_dotdot sd_number { $$ =
TREE2("pl_num_range", $1, $3);
}

| sd_number ss_minus sd_number { $$ =
TREE2("pl_num_range", $1, $3);
fprintf(stderr, " Syntax change: Please use .. instead of - for integer ranges\n");
}

| sd_id { $$ =
TREE1("pl_id", $1);
}


| ss_ellipsis { $$ =
TREE0("pl_ellipsis");
}

| sd_number { $$ =
TREE1("pl_num", $1);
}

| sd_string
    { $$ =
TREE1("pl_string", $1);
}
;


value_list_or_nil:
  /*  */ { $$ = LISTEND(0); }
| value_list { $$ = $1; }
;


value_list:
  value
{ $$ =
LISTCONS($1, LISTEND(0));
}
| value value_list
{ $$ =
LISTCONS($1, $2);
}
;



exp:
  exp1 { $$ = $1; }

;


exp1:
  exp2 ss_query exp1 ss_colon exp1 { $$ =
TREE3("pl_query", $1, $3, $5);
}

| exp2 { $$ = $1; }
;

exp2:
  exp3 { $$ = $1; }

| exp3 ss_rarrow2 exp3 { $$ =
TREE3("pl_diadic", YYLEAF("pl_implies"), $1, $3);
}
;


exp3:
  exp4 { $$ = $1; }

| exp3 ss_rarrow2 exp4 { $$ =
TREE2("pl_implies", $1, $3);
}

;


exp4:
  exp5 { $$ = $1; }
| exp4 ss_logor exp5 { $$ =
TREE3("pl_diadic", YYLEAF("pl_logor"), $1, $3);
}
| exp4 ss_disj exp5 { $$ =
TREE3("pl_diadic", YYLEAF("pl_logor"), $1, $3);
}
;


exp5:
  exp6 { $$ = $1; }
| exp5 ss_logand exp6 { $$ =
TREE3("pl_diadic", YYLEAF("pl_logand"), $1, $3);
}
| exp5 ss_caret exp6 { $$ =
TREE3("pl_diadic", YYLEAF("pl_xor"), $1, $3);
}
| exp5 ss_conj exp6 { $$ =
TREE3("pl_diadic", YYLEAF("pl_logand"), $1, $3);
}
;


exp6:
  exp7 { $$ = $1; }

| exp7 ss_deqd exp7 { $$ =
TREE3("pl_diadic", YYLEAF("pl_deqd"), $1, $3);
}
| exp7 ss_dned exp7 { $$ =
TREE3("pl_diadic", YYLEAF("pl_dned"), $1, $3);
}
| exp7 ss_dltd exp7 { $$ =
TREE3("pl_diadic", YYLEAF("pl_dltd"), $1, $3);
}
| exp7 ss_dled exp7 { $$ =
TREE3("pl_diadic", YYLEAF("pl_dled"), $1, $3);
}

| exp7 ss_dgtd exp7 { $$ =
TREE3("pl_diadic", YYLEAF("pl_dgtd"), $1, $3);
}
| exp7 s_FQGT exp7 { $$ =
TREE3("pl_diadic", YYLEAF("pl_fqgt"), $1, $3);
}
| exp7 ss_dged exp7 { $$ =
TREE3("pl_diadic", YYLEAF("pl_dged"), $1, $3);
}
;



exp7:
  exp8 { $$ = $1; }
| exp9 ss_stile exp8 { $$ =
TREE3("pl_diadic", YYLEAF("pl_binor"), $1, $3);
}


;

exp8:
  exp9 { $$ = $1; }
| exp8 ss_ampersand exp9 { $$ =
TREE3("pl_diadic", YYLEAF("pl_binand"), $1, $3);
}

;

exp9:
  exp9 ss_plus exp10 { $$ =
TREE3("pl_diadic", YYLEAF("pl_plus"), $1, $3);
}

| exp9 ss_minus exp10 { $$ =
TREE3("pl_diadic", YYLEAF("pl_minus"), $1, $3);
}

| exp10 { $$ = $1; }
;


exp10:
  exp10 ss_star exp11 { $$ =
TREE3("pl_diadic", YYLEAF("pl_multiply"), $1, $3);
}

| exp10 ss_slash exp11 { $$ =
TREE3("pl_diadic", YYLEAF("pl_divide"), $1, $3);
}

| exp10 ss_percent exp11 { $$ =
TREE3("pl_diadic", YYLEAF("pl_mod"), $1, $3);
}
```

```
| exp11 { $$ = $1; }
;

exp11:
  ss_caret exp12 { $$ =
TREE1("pl_differentiate", $2);
}

|  exp12 { $$ = $1; }
;


exp12:
  ss_hash exp12a {
$$ = TREE1("pl_wfield", $2);
}
| exp12a { $$ = $1; }
;


exp12a:
  exp12a ss_hash exp13 {
$$ = TREE2("pl_raw_field", $1, $3);
  }

| exp13 { $$ = $1; }
;




exp13:
  exp13a ss_lpar ss_rpar     { $$ =
TREE2("pl_apply", $1, NULL);
    }

| exp13a ss_lpar exp_comma_list ss_rpar     { $$ =
TREE2("pl_apply", $1, $3);
    }

| ss_lpar exp ss_comma exp_comma_list ss_rpar  { $$ =
TREE1("pl_tuple", LISTCONS($2, $4));
    }


| exp13a { $$ = $1; }

| ss_lpar exp ss_rpar { $$ = $2; }

| ss_query exp13
    { $$ =
TREE1("pl_channel_read", $2);
}

| sd_number
    { $$ =
TREE1("pl_num", $1);
}

| sd_string
    { $$ =
TREE1("pl_string", $1);
}

| ss_tilda exp13
{ $$ =
TREE1("pl_binnot", $2);
}

| ss_pling exp13
{ $$ =
TREE1("pl_lognot", $2);
}
;


exp13a:
  sd_id { $$ = TREE1("pl_id", $1); }

| sd_string    { $$ = TREE1("pl_string", $1); }


| exp13a ss_dot exp13a { $$ = TREE3("pl_diadic", YYLEAF("pl_cat"), $1, $3); }

| ss_dollars { $$ = TREE0("pl_dollars"); }
;


%%

#include <stdio.h>
```

```
#include <ctype.h>

extern FILE *stderr;
int yyerror (const char *s)
{
  extern builder *lishlex();
  extern void exit(int);
  extern void givesrcPoint();
    givesrcPoint();
    fprintf(stderr, "Syntax error: %s: \n", s);
fprintf(stderr, "Next symbol: %s\n", atom_to_str(lishlex()));
exit(1);
  return 1;
}


void yydebug_on()
{
#ifdef YYDEBUG
// extern int yydebug;
//    yydebug = 0;
#endif
}

char *smllib = "plgram";

int yylex()
{
  char s;  int v=0;
  extern void givesrcPoint();
  extern builder *lishlex();
  extern int lextracef;
  builder *b, *a = lishlex();
  if (!a) return 0;
  yylval.auxval = a;

  if (fnumberp(a)) return sd_number;
  if (fstringp(a)) return sd_string;

  b = fgetprop(a, smacro);
  if (lextracef)  printf("Lex %i %p %s\n", v, b, atom_to_str(a)); fflush(stdout);
  if (b) return atom_to_int(fcdr(b));
  s = atom_to_str(a)[0];
  if (isalpha(s) || s=='_') return sd_id;
  givesrcPoint();
  cbgerror(cbge_fatal, "Illegal input token %c", s);
  return 0;
}

/* eof */
```

5.2 Abstract syntax tree

The abstract syntax tree is defined using the following SML datatype:

```
(*
 * $Id: plgram.sml,v 1.27 2008/07/11 08:15:04 djg11 Exp $
 # CBG Badger Tuplers Project
 # University of Cambridge
 # Computer Laboratory
 # (C) 2004 David J Greaves
 #
 # Mostly Written Romsey - Dec 2004.
 #
 #  1-Oct-05 added parblock and fuse
 #
 *)


open linepoint;

datatype pl_diop_t = pl_deqd | pl_dned | pl_dltd | pl_dled | pl_dged | pl_dgtd | pl_logand | pl_logor | pl_binor | pl_binand | pl_plus | pl_
;


datatype pl_type_t =
  pl_safeunsafe of pl_exp_t list * pl_exp_t list
| pl_unsafe of pl_exp_t list
| pl_safe of pl_exp_t list
| pl_event of pl_exp_t list
| pl_fielddec of pl_exp_t list * pl_type_t
| pl_fielddecn of pl_exp_t list  (* depreciated form *)
|       pl_tid of string
|       pl_fusetype
|       pl_locktype

and pl_formal_t =
  pl_formal of pl_exp_t * pl_exp_t
| pl_formal_nt of pl_exp_t

and pl_exp_t =
pl_query     of pl_exp_t * pl_exp_t * pl_exp_t
| pl_raw_field of pl_exp_t * pl_exp_t (* comes in this way from parser *)
|       pl_string    of string
| pl_wtuple    of pl_exp_t list (* no longer generated *)
| pl_tuple     of pl_exp_t list
| pl_id        of string
| pl_ellipsis
| pl_backstop
| pl_dollars
| pl_differentiate of pl_exp_t
| pl_wfield    of pl_exp_t
| pl_diadic    of pl_diop_t * pl_exp_t * pl_exp_t
| pl_catenate  of pl_exp_t * string
| pl_num       of int
| pl_num_range of int * int
| pl_lognot    of pl_exp_t
|       pl_apply     of pl_exp_t * pl_exp_t list
|       pl_par       of pl_exp_t * pl_exp_t

(* The following do/does not occur in yacc parse tree *)
|       pl_raw_fieldl of pl_exp_t list


|       pl_filler1  (* Used only to supress the unused cases warning in my other match traps *)
;

(* Old form, compiled bytecode files: *)
datatype pl_comp_t =
        pl_stringtab  of string * int
|       pl_bytecode   of int
;

datatype xml_t =
   xml_element of string * xml_att_t list * xml_t list
|  xml_meta of string
|  xml_int of int
|  xml_chars of string

and xml_att_t = xml_att of (string * string)
;

(* A transitioner should be one of entry | body | exit *)
type pl_transitioner_t = string;


datatype dv_t = dv_output | dv_input | dv_inout | dv_local | dv_fuse | dv_lock | dv_tuple | dv_other;

datatype pl_case_item_t =
pl_case_item of pl_exp_t list * pl_cmd_t
| pl_case_default of pl_cmd_t

and pl_cmd_t =
pl_if        of pl_exp_t * pl_cmd_t
| pl_while     of pl_exp_t * pl_cmd_t
| pl_forever   of pl_cmd_t
| pl_ife       of pl_exp_t * pl_cmd_t * pl_cmd_t
|       pl_switch    of pl_exp_t * pl_case_item_t list
| pl_assign    of pl_exp_t * pl_exp_t
| pl_pragma    of string * pl_exp_t
| pl_block     of pl_cmd_t list
| pl_parblock  of pl_cmd_t list
| pl_with      of pl_exp_t * pl_cmd_t
| pl_fused     of pl_cmd_t * pl_exp_t
| pl_wait      of pl_exp_t
| pl_goto      of string
| pl_label     of string
| pl_break
| pl_continue
| pl_emit      of pl_exp_t
| pl_return    of pl_exp_t
| pl_assert_sl of string * bool * pl_exp_t
| pl_never     of string option * pl_exp_t list
| pl_always    of string option * pl_exp_t list
| pl_live      of string option * pl_exp_t list
| pl_disable   of pl_exp_t list
| pl_skip

|       pl_dv        of  dv_t * pl_type_t list
|       pl_macro     of (pl_exp_t * pl_exp_t) list
|       pl_linepoint of linepoint_t * pl_cmd_t
|       pl_pebble    of string * pl_exp_t
|       pl_const     of string * pl_exp_t

|       pl_e_as_c    of pl_exp_t
|       pl_sortset_dec  of string * pl_type_t
|       pl_meta      of pl_associtem_t list
|       pl_bundle    of  string * pl_cmd_t list

|       pl_def_pebble of string * pl_formal_t list * pl_cmd_t list
|       pl_def_bundle of string * string * pl_formal_t list * pl_cmd_t list
|       pl_facet_instance of pl_exp_t * pl_exp_t
|       pl_fun       of string * pl_formal_t list * pl_cmd_t list

|       pl_stategraph of string * pl_formal_t list * pl_state_t list

|       pl_compiled_bundle of pl_comp_t list
|       pl_xml of xml_t

|       pl_filler2  (* Used only to supress the unused cases warning in my other match traps *)

and pl_associtem_t =
        pl_associtem of string * pl_exp_t

and pl_state_t =
        pl_state_def of string * pl_formal_t list list * pl_stateitem_t list

and pl_stateitem_t =
        pl_transition of pl_statespec_t * pl_statespec_t * pl_cmd_t option
|       pl_state_action of pl_transitioner_t * pl_cmd_t

and pl_statespec_t =
        pl_state_exp of pl_exp_t
|       pl_exit_state_exit
|       pl_exit_state_tagged_exit of string
;

datatype metainfo_t =
  MIA of string * string
| MIA_filler
;


(* eof *)
```

5.3 Program File

A Pushlogic Source program is an unordered list of declarations, bundle, pebble and function definitions. A simple program file contains just one bundle definition that contains all of the rules as well as further declarations.

Comments are defined with the BCPL-style double slash.

5.4 Bundle Declaration

```
def bundle mybundlename()
{
    // contents go here.

}
```

A bundle declaration uses the keyword sequence '`def bundle`'. The bundle content is a list of declarations and statements. The bundle name should commonly be the same as the program file name on the storage media.

The statements in a bundle are all started in parallel when the compiled object bundle is loaded. A statement may be a sequential block, thereby providing the normal imperative programming paradigm.

Each item within a bundle definitions is either a declaration, a first-order or temporal logic assertion or an executable sequence of imperative code. 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 of the bundle performing their next 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. $(a, b) := (e1, e2)$.

Bundles may also be declared using '`def pebble`', '`def world`' and '`def plant`'. The pebble declaration allows soft pebbles to be defined fully using Pushlogic. Pebbles have slightly different rules over binding of shared fields from bundles ... details to follow.

5.5 Constant Values

Constants may take the same forms as those define in §3.1 for object level, except that bottom is not allowed at source level.

5.6 Identifiers

Identifiers appear alone or as part of an heirarchic field name.

Identifiers appearing alone must be either a member of one or more field range enumerations, the last component of an heirarchic field name or the reserved identifiers '`true`' or '`false`'.

Where the last component of an heirarchic field name is shared over more than one field, or is also a constant value from a field range enumeration, the identifier cannot be used alone: it must be placed in double quotes to imply the constant value, or to refer to a field, must be disambiguated by providing further trailing portions of the heirarchic field name or using by using a '`with`' statement.

The alone use of identifiers is not currently working in the compiler always.

5.7 Field Declarations

Variables are known as fields.

Field declarations define the heirarchic name of the field (its name) and then (partially) enumerate the ranges of safe values and unsafe values, delimited with colons. The safe and unsafe enumerations for must be disjoint for any one field.

If the colon between the safe and unsafe values is ommitted, all listed values are unsafe.

Field values can be enclosed in quotes when they need to contain non-alphanumeric characters for any reason. An integer range is specified with a two dots. Elipsis is allowed as an unsafe value, represented with three dots.

Declarations may be introduced with the following keywords: `input, output, inout, lock, local, macro` and `fuse`. Any number of the same sort may be declared in one statement using commas to separate them.

Some examples are

```
  input  a#b#c1 : { s0, s1 : v1, v2 };
 output  a#b#c2 : { 0 : 1..99 };
  inout  e#c3   : { off : 1..9, ... };
  local  david  : { contented : happy, sad };
```

Because Pushlogic is designed to operate in a dynamic, extensible environment, further values of the fields may occur at run-time, beyond those defined and checked against at compile time. Not every field name needs be declared, provided the compiler has sufficient information overall to work out the sensitive parent list to put in each object level rule.

The '`input`' declaration defines a field that is only read by the bundle. The '`output`' declaration defines a field that is only written by the bundle. The '`inout`' declaration defines a field that is both read and written. The writes to an inout field are frequently not explicit in the source code because they arise only during a pushback. Definition of a field as input instead of inout can lead to a 'no suitable sensitive parent setting' error from the compiler.

The '`local`' declaration defines a field that is allocated space in the bundle's local tuple.

The '`macro`' declaration defines a name of a subexpression that is macro expanded before use. This might be deleted in future.

The '`lock`' modifier defines a field that supports atomic operations. See §5.10.15.

The '`event`' modifier defines a field that communicates an event. See §3.2.

The '`fuse`' modifier defines a field of Boolean sort that is given low priority for pushback. See §5.10.16. The range declaration for a fuse is optional, and if provided, must be '{ `false :  true` }'.

When a modifier appears without a preceeding declaration keywork then a local declaration is made.

5.7.1 Sort Statement

The ‘`sort set`’ statement associates a user’s idenfitier with a pair of safe and unsafe range enumerations. It can then be used in declarations, as shown in this example:

```
 sort set mysort = { s0, s1 : v1, v2 };
output  x#b, x#c : mysort;
 inout  x#d      : mysort;
```

5.7.2 Namespace Binding

All tuples exist in a global name space, but aliases or handles for certain points are provided for ease of reference. The available handles are denoted with

Leading Symbol	Meaning
dollars ($)	Local Bundle Private Namespace
<blank>	Device/Platform Namespace
hash (#)	Device Namespace or `with` context
slash (/)	DoP Root
`tup://`	URI - remote tuple access

Hash and slash are inter-changeable as delimiters between the parts of an heirarchic field reference but have special meaning at the start of a field name. Dot may also be used, but is intended for access to components within the OO parts of the language.

A field name starting with a dollars sign is a local field name. These need not be used, since the compiler provides the special ‘`local`’ keyword that defines aliases so that local variables may be stored in the bundle’s private area. In an implementation, the dollars symbol is mapped during re-hydration to a fresh tuple stored on the local execution platform whose primary key is the bundle instance identifier.

When the name starts with a hash it is interpreted with respect to the field prefix given in a textually surrounding ‘`with`’ statement. If there is no surrounding ‘`with`’ statement then the reference is to the namespace of the local device. This is also the default namespace when no leading character is given and the field reference starts with a tuple name.

A field name starting with a slash refers to fields provided and stored within the current DoP.

Field names may also start with a URI or a symbolic name that is converted to a URI during re-hydration. A more detailed description of heirarchic names is outside the definition of Pushlogic and are defined in the Tuplecore document and web pages.

The ‘`pebble`’ alias statment establishes a pointer to part of the namespace. It is frequently used to provide a Pushlogic bundle with access to pebbles instantiated on the same device (platform), hence its name, but it can also be used for access to fields shared by other bundles.

Here we present a simple example using hardwired IP address, but actual device addresses should not normally appear in the source code: they should be supplied during re-hydration:

```
def bundle simplelink()
{
   pebble my_keyboard = Pebbles#Front_panel#Keyboard;
   pebble your_devices = tup://128.232.1.10/Pebbles#Devices;
   // field declarations omitted
   your_devices#front_panel#on_led := my_keyboard#on_switch;
}
```

Pebble aliases can be defined inside a bundle statement (as shown) or outside.

5.8 Pragmas

The ‘`pragma`’ statment enables control flags to be passed to the compiler. These may occur inside a bundle declaration or at the top level of a file. See the compiler manual for details of the supported pragmas.

5.9 Operators

The operators available at source level include all those defined in for object level in §4.3. Each is denoted with its usual symbol or digraph.

In addition, the differentiate operator is provided, denoted with uparrow ($\uparrow$ exp).

The currently available forms are summarised in Figure 5.1.

It is our goal to support as many features found in common OO imperative HLLs as possible, while still producing output that can be represented as Pushlogic object rules and checked automatically at load time.

5.9.1 Function Call

Function calls are expanded fully at compile time and so must be have statically bounded recursion.

- String constants (quotes optional if part of an enum): e.g. `"hello"`,
- Local hierarchic field names: a#b#c,
- Remote hierarchic field names: tup://128.232.1.22/a#b#c,
- Function call: f(a,b,c...)
- Vector of expressions (, ,)
- Comparison predicates: $<><=>===$ $!=$
- Integer arithmetic: + - * /
- Differentiation: $\uparrow$ exp,
- Conditional expression: $(g)?S_t : S_f$,
- Blocking remote procedure call: e.g. `rc=device!(...)`,
- String catenation operator: e.g. `"nice" . "girl"`,
- Attribute access with constant tag string e.g. `var.ID`,

Figure 5.1: Pushlogic Source Operators

5.10 Pushlogic Statements

5.10.1 Emit Statement (SOAP and GENA too)

The ‘`emit`’ statement delivers a Pushlogic event. This may be mapped to a non-Pushlogic GENA event or device RPC during rehydration (§5.11.1).

```
   if (<ee>) emit <event-name>;
   if (<ee>) emit <event-name>(args, ...);
```

The ‘`emit`’ statement is shown in the context of an ‘`if`’ statement that is guarded by an event expression. Such a guard must normally exist within the surrounding program flow control in some form or another.

If the guard is a level expression, this would allow cause a nominal, continuous stream of back-to-back events to be emitted and would tend to violate idempotency.

The guarding context may be a level expression if the event being emitted is entirely local and the nominal stream of events is local to the current bundle and is integrated back to being a level expression in all places where it is used.

In the future, it is envisaged that closer integration with UPnP, SOAP and other device control languages will be implemented, and hence the emit statement will be implied by constructs such as

```
 if (<ee>)
    house.livingroom.curtains.setto(halfway);
```

This need arises since many devices have an event-driven API and integrate the received event stream to set their internal state; these are conventional models of commanding over an asynchronous packet network.

5.10.2 Pebble Statement

The `pebble` statement is used to hard-code the address of a network resource, such as the IP address and port number of a remote pebble. This statement should not normally be used and should not occur in portable code. Binding is normally performed at rehydration time, mapping symbolic constants in the source code to active network addresses.

```
 def bundle displayecho()
 {

   pebble PushClock = "tup://169.254.25.32:1200";
   input PushClock#Pebbles#ClockDisplay#Leds#hour : {0..23};
   input PushClock#Pebbles#ClockDisplay#Leds#minute : {0..59};


   pebble DisplayPanel = "tup://169.254.25.192:1202";
   output DisplayPanel#Pebbles#Display#value : { "No message yet" : ..


   // Use the string cat operator, dot, to make the output message.
   value := "Time now:" . hour . " " . minute;

}
```

5.10.3 Input and Output Statements

5.10.4 Assignment Statement

Assignments are denoted with the colon-equals assignment operator.

Parallel assignment is supported where the same number of comma-separated expressions appear on both sides of the assignment : e.g. $(a, b) := (e1, e2)$.

All executable rules are placed in parallel by default at the top-level but sequential composition occurs inside nested blocks, unless explicitly written in this parallel form.

Event and Level Constraints

The constraints on the object-level assignments between level and event variables and expressions, given in §4.6.2, is reflected at the source level, but with one relaxation, described in §5.10.1.

The object-level constraints allow the following four basic source forms, or anything tantamount to them:

```
   if (le) lv := le;
   if (le) ev := ee;
   if (ee) lv := le;
   if (le) lv := ee;
```

Rather than assigning to an event variable, emitting an event is possible, described in §5.10.1.

The third and fourth forms are known as integrations.

5.10.5 Sequential composition

Statements separated by a semicolon inside a ‘`bundle`’, ‘`def mod`’ or top-level ‘`with`’ with statment are still considered top-level and composed in parallel. All other statements separated by a semicolon are composed in series, corresponding to normal imperative programming.

An example:

```
def bundle s()
{  // Bundle braces do not force seq
  ...
a := b; // Three statements in parallel (two assigns and an if).
c := d;
if (e)
  {
    f := h;   // A seqeuence of two statements.
    g := f+j; // Assign to f above has immediate effect on this rhs.
  }
}
```

5.10.6 With Statement

The ‘`with`’ statement sets up a textually-scoped alias for part of tuple space. Field references that start with an hash sign inside the statement are refered to this alias. An example:

```
with (/devices#book)
{
   x1 := /devices#book#page#number;
   x2 := #page#number;
   // x1 and x2 have actually been assigned the same.
}
```

The ‘`with`’ statement does not cause sequential composition of its contents.

5.10.7 If/Then/Else Statement

The ‘`if/else`’ statement, as found in the C language, is supported.

5.10.8 Switch/Case/Default Statement

A form of ‘`switch`’ statement, is supported. Multiple tags per statement block are allowed using comma separation. Unlike the C language, flow does not fall from one branch into the next and so there is no associated binding for the ‘`break`’ statement.

A tag called ‘`default`’ may be used to catch any, otherwise unmatched conditions.

5.10.9 Stategraph Statement

The stategraph defines a finite state machine, where each state has a state name. A top-level stategraph is always active, meaning it is in exactly one state. On the other hand, a stategraph that is instanced as a child stategraph within a state in another stategraph is inactive (not in any state) unless its parent is in that instantiating state. A state may instantiate any number of child stategraphs but recursion is not allowed.

The stategraph general form is:

```
stategraph graph_name()
{

  state statename0 (subgraph_name, subgraph_entry_state), ... :

        entry:  statement;
        exit:   statement;
        body: statement;

        statement;
        ...            // implied ’body:’ statements
        statement;

        c1 -> statename1: statement;
        c2 -> statename2: statement;
        c3 -> exit(good);
        ...

        exit(good) -> statename3: statement;
        exit(bad) -> statename4: statement;

        ...


   endstate


   state statename2:
...
...
   endstate


   state disable: // A special state that can be
                  // forced remotely

   ...

}
```

A state may contain tagged tatements, each of which may be a basic block if required. They are distinguished using three tag words. The ‘entry’ statement is run on entry to the state and the ‘exit’ statement is run on exit. The ‘body’ statement is run while in the state. A ‘body’ statement must contain idempotent code, so that there is no concept of the number of times it is run while in the state. Statements with no tag are treated as body tagged statements. Multiple occurrences of statements with the same tag are allowed and these are evaluated as though executed in the textual order they occur or else in parallel (current implementation is serial but this will be change to parallel, so watch out!).

A state contains transition definitions that define the successor states. Each transition consists of a boolean guard expression, the name of one of the states in the current stategraph and an optional statement to be executed when taking the transition. In situations where multiple guard expressions currently hold, the first holding transition is taken.

The guard expressions range over the inputs to the stategraph, which are the variables and events in the current textual scope, and the exit labels of child stategraphs.

When a child stategraph becomes active, it will start in the starting state name is given as an argument to the instantiation, or the first state of no starting name is given.

A child stategraph becomes inactive when its parent transitions, even if the transition is to the current state, in which case the child stategraph becomes inactive and active again and so transitions to the appropriate entry state.

A child stategraph can cause its parent to transition when the child transitions to an exit state. There may be any number, including zero, of exit states in a child stategraph but never any in a top-level stategraph. The parent must define one or more transitions to be taken for all possible exit transitions of its children. An exit state is either called ’exit’ or ’exit(id)’ where ’id’ is an exit tag identifier. Exit tags used in the children must all be matched by transitions in the parent, or else the parent must transition itself under the remaining exit conditions of the child or else the parent must provide an untagged exit that is used by default.

A stategraph may be wholly enclosed inside any conditional statement, such as an ‘`if`’ or ‘`case`’ statement, in which case it is as though all of its internal activity is guarded by that condition: the condition is simply folded inside every construct to the point where a conditional is allowed. The stategraph does not reset to its starting state when this guard does not hold.

A stategraph with a state called disable may be disabled from elsewhere in the same bundle using the ‘`disable`’ statement. Please see §5.10.10.

The stategraph general form is sufficient to encompass the SysML state machines.

5.10.10 Disable Statement

The ‘`disable`’ statement is used for a remote disable of a stategraph.

Syntax:

```
  if (g) disable stategraph_name1, stategraph_name2, ...;
```

The disable statement must be conditional, otherwise the stategraph would never leave its disable state, and the disable guard, g may either be an event or level expression. When the disable guard is a level expression it takes precedence over any transitions in the stategraph that lead from the disable state.

```
  if (g) disable stategraph_name1, stategraph_name2, ...;
```

5.10.11 While/For/Break/Continue Statements

The ‘`while/continue/break`’ statement, as found in the C language, is supported.

The ‘`for/continue/break`’ statement, as found in the C language, is supported.

The ‘`do/while/continue/break`’ statement, as found in the C language, is not yet implemented.

The ‘`forever`’ statement is an alias for ‘`while(1)`’.

5.10.12 Procedure Call Statement

A procedure call statement has no keyword. A call consists of a function name followed by its arguments in parenthesis. Procedure calls are expanded at compile time and hence must have a compile-time determined upper-bound on recursion depth.

5.10.13 Return Statement

The ‘`return`’ statement is for use in functions only.

An example:

```
def fun add(a, b, c)
{
  if (a=red) return b;
  return c+d;
}
```

5.10.14 Wait Statement

```
while(1)
{
  ...
  wait expr;
  ...
}
```

The ‘`wait`’ statement cause the current ‘thread’ to wait until a condition holds. There are no threads in Pushlogic object code and so the wait statement is implemented by splitting the source code into separate basic blocks that are guarded by values of an automatically-defined enumeration that chains control from one block to the next. The enumeration variable acts rather like a program counter. At runtime it is stored in the field ‘$local#pcnnn’ where nnn is an integer that is unique to a bundle. To avoid inter-bundle name-space clashes, all fields in starting with ‘$local’ are renumbered to be disjoint at bundle load time. The same mechanism is used to create a hidden variable to store the old value of a field by the differentiation operator denoted with the uparrow $\hat{()}$.

SOME REPETITION HERE!

The `wait` statement blocks the current thread until the condition holds. If the thread loops around and enters the same wait statement while the condition still holds, the thread is not blocked.

Where more than one `wait` statement exists for a thread, a program counter variable is created and stored as a field in the bundle’s local tuple.

5.10.15 Lock Statement

The `lock` statement declares a field that supports atomic test-and-set operations.

A field of sort lock has as its safe value the empty string.

Any bundle may store a value in a lock field, but only if it is currently null. If it is not null, the write will fail and a pushback occurs. A bundle typically stores its own private identifier (accessible from the metainfo tuple §6.1).

A bundle is responsible for clearing the lock back to the empty string when it has finished with the guarded resource.

5.10.16 Fuse Statement

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:

```
{
    input X#x : { S: US };
    inout Y#y : { S: US };
    y : = x;
}
```

The problem is that if $Y\#y$ makes a unilateral change from US to S, which it is free to do, since it is an 'inout', then no push back is possible because $X\#x$ 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 $Y\#y$ make a push back.

```
    input X#x : { S: US };
    inout Y#y : { S: US };
    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;'. During pushback path creation, the fuse is chosen as the last option and only marked for push back update if 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. If $Y\#y$ refuses to accept the current value at this time, the fuse blows again. Other code can be sensitive to this fuse.

5.11 Pushlogic RPC

Currently RPC is not used and all comunication between platforms is implemented via the shared-variable illusion implemented by the tuplecore 'ETC' protocol. In the future, Remote procedure call (RPC) may be used between Pushlogic bundles, or between a Pushlogic bundle and a non-pushlogic entity.

5.11.1 Foreign RPC (SOAP and GENA)

Pushlogic may make calls directly over the network using XML RPC (and in the future SOAP RPC). Details to be added...

Pushlogic can also send and receive GENA events by setting up simple mappings between Pushlogic events and GENA events. Details to be added...

5.11.2 Native RPC

Native remote procedure call is provided for communication between Pushlogic bundles on the same or different execution platforms. implemented by expansion to other statements. Owing to the dynamic storage restiction limitations of SPL1, a bundle must be re-hydrated for each concurrent service operation. Blocking RPC is currently being developed - the blocking aspect is implemented by expansion to the '`wait`' statement.

Non-blocking RPC does not return a result and is denoted with '`device!(...)`' where the ellipsis is replaced with a list of assignments to mutable fields of that device. This is translated to a conjunction of assertions that the appropriate tag fields of the indexed device have the values being passed. Blocking RPC is implemented by the compiler as a combination of non-blocking RPC and a `wait` statement.

Details to be added here ...

5.12 OO Structures

Some basic syntactic sugar is implemented to enable objects to be defined and instantiated. All instantiations are performed at compile time using abstract interpretation and so must be statically determinable.

This part of the compiler is currently a bit broken, but contains nothing novel.

5.13 Temporal Logic Assertions

Assertions may be included in the Pushlogic source code and checked by the system model checker as well as at compile time or at load time, as appropriate. Each assertion can have a textual name.

A '`live`' assertion asserts that a condition must reoccur infinitely often. A '`never`' assertion asserts that a condition must never occur. An '`always`' assertion asserts that a (level) condition must always holds and is equivalent to a never statement with negated condition. In these assertions, any number of conditions may be listed, separated by commas, and these have the same meaning as if provided in a separate statements. Live statements may occur inside if/then/else and other control flow statements, in which case each condition is guarded by (conjuncted with) the enclosing conditional statement guards.

In the future, these simple assertions will be augmented with richer assertions that span the ground between liveness and safety: i.e. until assertions and assertions that specify quantitative maximum and minimum valuations on resource use.

```
    always [ string : ]  <exp>, <exp> ...;
    never [ string : ]  <exp>, <exp> ...;
    live [ string : ]  <exp>, <exp> ...;
```

The string is the rule name that is carried forward for output by logging or monitoring code.

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 "Door Unlockable Assertion" 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 pre-compiled 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;
}
```

5.14 Compiler Operation

It is helpful to briefly present the internal operation of the compiler. The internal flow of the compiler is shown in Figure 5.2.

5.14.1 Conversion to I-Code

The input is parsed and converted to imperative intermediate code using conventional compiler techniques. Function calls are expanded in line. For each sequence

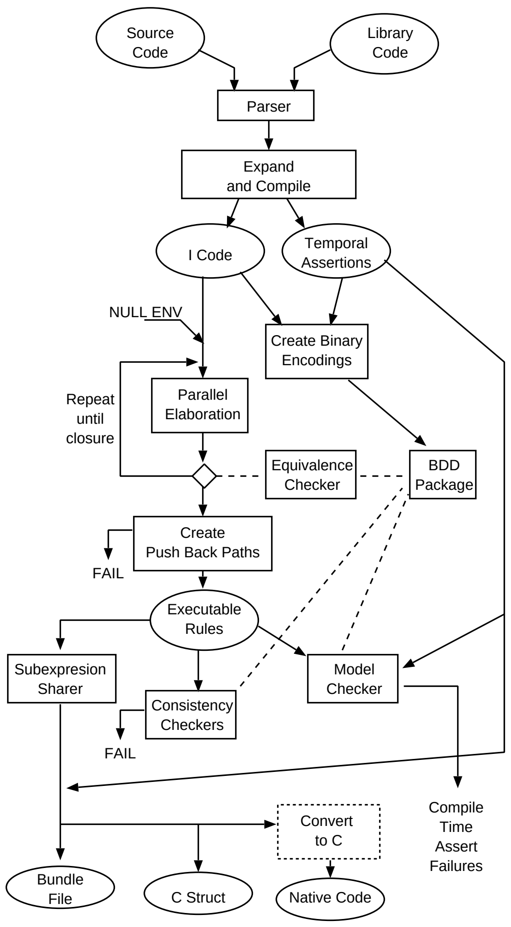


Figure 5.2: Internal structure of the Pushlogic Compiler

in the source code a section of I-code is generated. I-code consists of labels, gotos, waits, assignments and conditional branches. For each section, 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 section. There is no run-time spawning or joining of threads (although the illusion of this can 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. Liveness assertions may be guarded by nested `if` statements and by the current value of the program counter.

Each I-code instruction is stored in an array, indexed by compile-time program counter, and each has one of the following forms:

```
(* Intermediate, imperative assembler code form *)
and icode_t =
  I_assign of bc_t * bc_t
| I_resultis of bc_t
| I_goto of int
| I_wait of bc_t
| I_if of bc_t * int
| I_eof
| I_skip
| I_safetylive of bool * bc_t (* safe is true *)
```

Runtime program counters range only over the entry point to a thread and the points immediately following an `I_wait`.

The I-code is embedded in a 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.

5.14.2 Repeated Elaboration from each Entry Point until Closure

An entry point is defined as any entry point to a section 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. Function calls are expanded in line.

Elaboration of assignment

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

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

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

Elaboration of Sequential Composition

Sequential composition of statements is implemented by forming a conjunction of their translations but where any assignment is implemented as as symbolic substitution before translating a next statement.

$$\begin{array}{rcl} [\![v = e; C_2]\!] & \rightarrow & [\![v = e]\!] \wedge [\![C_2[e/v]]\!] \\ [\![C_1; C_2]\!] & \rightarrow & [\![C_1]\!] \wedge [\![C_2]\!] \end{array}$$

Elaboration of WAIT

The '`wait`' statement essentially divides an infinite thread circuit into a number of arcs. Each arc commences with a different setting of a synthesized program counter that is generated for each parallel statement containing waits. These program counters are stored in the local tuple of the execution platform and renamed to be unique at bundle load time. The program counter may be set to one of a number of new values at the end of each arc, depending on conditional execution paths within the arc. A program counter must be classed as integrator (its next value depends on its current value). The guard conditions present in `wait` statements must accordingly, somehow achieve the differentiator property when the bundle is model checked as a whole. Some examples will be added here.

Elaboration of IF/THEN/ELSE

The if/then/else construct is converted to an object form conditional expression

$$[\![\texttt{if}\ c\ \texttt{then}\ T\ \texttt{else F}]\!] \rightarrow [\![c]\!]\ ?\ [\![T]\!]\ :\ [\![F]\!]$$

that is then expanded as usual:

$$(c)?t : f \rightarrow (c \wedge t) \vee (\tilde{}c \wedge f)$$

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 behaviour of the runtime interpreter, which holds (or *gates*) all assignments until every subexpression has been recomputed, and then performs a commit.

5.14.3 Compensation Path Determination

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.

Sub-expressions are generated by spotting common subexpressions using a hashing technique. Where a pair of rules use a common subexpression, this sharing is noted by a re-writing phase before code generation.

5.14.4 Compile Time Assertion Checking

The model checker constructs a next state relation from the executable rules. For the purposes of the 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 during pushback calculation.

5.14.5 Code Output

The output code bundle, containing executable code, field definitions and assertions, is written to four output files that all contain the same information:

- a bytecode bundle file (list of integers in ASCII/comma format),
- a C struct file that contains some initialised C arrays, for direct canning into ROM,

- a dot net version (CIL assembly file),
- an XML encoded version.

The dot net version can be canned to ROM by compiling it with the `ilasm` assembler from the mono project and then using the `monos` utility program on the resulting bytecode.

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).

5.15 Model Checking

The pushlogic compiler contains a symbolic model checker that uses a BDD package. This is the same BDD package as used by the compiler for equivalence checking when it is searching for idempotent closure (§5.14.2).

The model checker in the compiler can operate on more than one bundle at once, checking inter-bundle interactions. Since the compiler can accept, on its command line, at most one source file and any number of object files, there are three ways the model checker might get invoked:

- With one source bundle it checks the assertions in that bundle are consistent with the logic in that bundle.
- With one source bundle and one or more object bundles, it checks that the current source bundle will be compatible when run alongside the object bundles in a domain.
- With a list of object bundles, it checks they are mutually compatible.

The third way enables the compiler to serve as a checker over a set of rehydrated bundles. Hence it can serve as the **domain checker**.

Scalability is a big problem with BDD-based model checking. A fair bit of time is used up finding variable orderings that lead to a compact BDD. The compiler writes out the order it finally selected to a hidden file, `.bdd.xml`, and reads it in again, if present in the current directory. Since the filename is currently fixed, it is important to do widely differing runs in different directories.

Current research is developing an incremental model checker so that scalability restrictions are greatly reduced.

5.16 Bundle Meta Info

The compiler generates a small amount of meta information and stores this in a dedicated tuple in the local space.

5.17 Binding Hooks

Before execution and insertion into a DoP, a bundle is re-hydrated using operations akin to macro-language rewrites ...

Chapter 6

Standard Environment

Pushlogic programs may rely on the presence of certain libraries at compile time and certain Pebbles at run time.

6.1 Bundle Meta Info

Object code from the compiler automatically contains assignments to a tuple called ‘BundleInfo’. This includes meta-information regarding the compiler version, source file name and so on.

Code in the bundle may also typically store additional meta info, for example

```
    BundleInfo#Company := "Acme Limited";
    BundleInfo#Release := "Version 21.2";
```

The execution platform stores the BundleInfo tuple in the local tuple for the bundle. The local tuple is stored in the Bundles tuple of the hosting execution platform under a field name that serves as a unique instance identifier for the re-hydrated bundle.

Additional information can be passed to the BundleInfo tuple using the ‘`meta`’ statement. This statement takes a comma-spearated list of tag/value pairs. For example

```
    meta Subassembly= "Motioncontrol", Release= "Version 21.2";
```

Currently, the `meta` statement does not do anything other than create a field in the BundleInfo and store a constant expression in it.

6.2 Local Variable Store

Pushlogic places its local variables in a tuple called ‘Local’.

6.3 Pushlogic Timer

Every Pushlogic platform provides access to a timer. The timer has a real-time clock and also provides any number of countdown timers.

The local real-time clock may be accessed by first declaring the following fields in the Pebbles tuple of the local platform and then reading them as needed:

```
input Pebbles#Timer#Timenow#hour : {0..23},
      Pebbles#Timer#Timenow#minute : {0..59},
      Pebbles#Timer#Timenow#second : {0..59};
```

If the local bundle also sets the real time clock, then it should declare these fields using the '`inout`' keyword.

Countdown timers are created by declaring a field inside Timer#Countdowns. Two different bundles on the same platform will interfere if they both declare the same count down counter - this needs blocking. The field must be set to an integer number of milliseconds and it counts down to zero by itself. The following code fragment illustrates how to make a light flash at five hertz.

```
inout Pebbles#Timer#Countdown#mytimer : { 0..100 };
output Somedevice#lights#light : { off: on };

with Pebbles#Timer#Countdown if (#mytimer == 0)
{
   #mytimer := 100; // Half cycle every 100 milliseconds
   if (light==on) light:=off; else light:=on;
}
```

The rest of this section is obsolete.

Frequently, rules must fire at a particular time of day, or describe constraints that apply only for specific periods of time. Examples are, respectively, "Turn the light on at 6:30 pm" and "All lights must be off between 01:00 and 06:30". As a basis for execution of temporal rules, a clock device is provided as a local resource at each Pushlogic interpreter. At the object level, it accessed in the same way as any other Pebble, but hooks for handling time are built in to the Pushlogic source language. Time encompasses both a linear, infinite sequence and a set of finite periodic schedules, hourly, daily and weekly. We refer to each of these as a base temporal extent. Any constant time expression mentioned in the rules can be seen as a partition of a base temporal extent into two temporal extents (or epochs): before the mentioned time and after the mentioned time.[1] Taking the union of all partitions on temporal extents leads to a finite number of epochs. It is simple to statically evaluate whether any expression referring to time is true or false in a

[1] An exception is that if only one time expression exists and it refers to a periodic temporal extent then it has no effect.

```
with Pebbles#Keypad
   if (#now == stop)
    { #(playled, pauseled, stopled)  := (0,0,1);
      Works#cmd  := stop;
    }
    else if (#now == play)
    { #(playled, pauseled, stopled)  := (1,0,0);
      Works#cmd  := play;
    }
    else if (#now == pause && local#keypad_old != pause)
    { if (Works#cmd == play)
        { #(playled, pauseled, stopled)  := (1,1,0);
           Works#cmd  := pause;
        }
      else
        { #(playled, pauseled, stopled)  := (1,0,0);
           Works#cmd  := play;
        }
    }
    else if (#now == eject)
        { #(playled, pauseled, stopled)  := (0,0,0);
           Works#cmd  := eject;
        }
```

Figure 6.1: Pushlogic Source Code: fragment from our DVD player demo.

given epoch. The rule validator must essentially collate the epochs and then check for rulebase consistency in every epoch. It can do this as an explicit datastructure but better would be to do it symbolically.

The tagged fields refer to components of the worldview. The other operators all have their normal meanings. In the future, arithmetic can be supported using Presburger [8] or CVC Lite [7].

A fragment of Pushlogic Source for our prototype DVD player is listed in Figure 6.1.

6.4 Assistance with Network Race Conditions

A number of problems arise when a program is distributed over the network and are well-known in concurrency theory: e.g. the Dining Philosophers, the Byzantine Generals problem, and more general problems in load balancing. Any language design must partition the work of solving these problems between compile-time and run-time and between language-level features, libraries and application code. A distributed implementation of Esterel, for instance, would still maintain the Esterel concept of the atomic event, although there is a heavy penalty in its implementation. Esterel does not have a native, looser, more-efficient, network paradigm. In Pushlogic, however, the basic operation, a write to a remote field, is not itself an atomic test-and-set and hence we need to consider the further support required to avoid problems from network races and unreliability.

Any program that contains a race of the nature shown in figure 4.1 will not unwind during compilation since the final result depends on the order of interleaving.

Races: As fields are names in a global name space, there may be network delays in making access to their values for read or write. Races may also arise in that no synchronisation between reads and writes is implemented at execution time.

The binding part of resource allocation is handled in Pushlogic at re-hydration time. All instances of a given type of device must be given unique identifiers during binding, and, because currently there are no arrays in Pushlogic, each identifier must textually exist in the Pushlogic source program.

6.4.1 Test and Set Facility

Consider a pair of DVD players that are commanded at once to send their output to a single display, where that display can only handle a single stream. If the display has a field that says which stream it is currently receiving, then the most simple form of resource allocation is to allow this to set at any time and the most recent write to it is the current winner. This gives the familiar radio-button method of source selection.

Where it is important that a resource is claimed for a specific interval, terminated by an explicit release, then concurrency theory tells us that an atomic test-and-set is required at some level in the system or else we must essentially have a set of locks, indexed by requester name, that implements a variant Dijkstra's solution.

Pushlogic allows a field to be declared as a '`lock`', whose safe value is the null string. More than one bundle may store a non-safe value in a lock field (relaxing the normal rule) but such a store is only successful if the previous value was the null string. In a network race, the second client that attempts to store a non-null value will experience a pushback.

6.4.2 Make/break Issues

Where an event triggers the change of two or more fields, sometimes the order in which they are changed is critical. For instance, electrical changeover switches come in make-before-break and break-before-make varieties.

Since Pushlogic is a relatively high-level language, built-in support for make-break ordering should perhaps be provided for the common cases, rather than forcing the programmer to implement his own sequencers. However, this is for future study. What is implemented is checking over various message delivery skew and loss scenarios (§1.4).

6.5 Low-level Parallel Composition of Tuples.

Both the '`pebble`' statement in Pushlogic and the binding performed at rehydration implement mappings between identifiers in a Pushlogic program and other parts of the Tuplespace, typically tuples in Pebbles. Sometimes it is helpful for there to be more than one, concurrent binding in place. For instance, we might want to address all the klaxons on the floor of a building or throughout the whole building. Also, different Pushlogic bundles might want to use different parallel compositions, such that a pair of pebbles that are in parallel from the point of view of one program are separate from the point of view of another.

There are two basic ways in which a number of pebbels can be addressed at once from a single Pushlogic read or write of a field: either the pushlogic code is macro expanded in some way, such as being rehydrated more than once, or else the pebbles are 'hardwired' in parallel at a lower level, such as with additional mechanisms in the tuplecore.

We require that any pair of fields composed by low-level parallel composition have the same tag name and safe value set. The meaning of low-level parallel composition is slightly different for read and write operations. We also need to define the behaviour under unliateral change, pushback and under message delivery errors.

When fields that are paralleled are written by a Pushlogic program the writes are simply sent to all of the fields. This is useful, for instance, when wishing to mimic the status of a hardware Pebble with a software GUI that reflects the most recent fields written to the Pebble. Each write is sent to each of the two destinations. If one of the writes fails or is pushed back to a safe value, then the run-time system must push back the other fields mapped in parallel accordingly. These push backs appear as undo's to code in other bundles that drives the mapped fields and to new writes to code that references the mapped fields.

For fields written unilaterally by external devices, or fields that refuse to accept a pushback, the value of their parallel composition is defined to be the most recent value written to any one of the fields. This provides suitable behaviour when a number of very simple push button pebbles are paralleled: the push logic program receives the up and down strokes of each key with no demarcation as to which button was pressed. However, it is expected that even the most simple push buttons, whether implemented physically or as part of a GUI, will be momentary at the point of operation, but with built-in status indicators as latches, and therefore able to accept a push back. If the off state is safe and the on state not, then ... and any pushback to off will turn off all the indicators. These ideas extend easily to other forms of widget.

Chapter 7

Plant Model

Instead of being executable, a bundle may be a plant model. A plant model mirrors the behaviour of the physical world system or plant. In many real systems, there are predictable effects from the output of actuators that may be detected by sensors. These feedback effects can cause undesirable effects, such as deadlock or oscillations, that Pushlogic can detect before they occur. Run-time monitoring of the conformance of the real system with its world model can also detect various faults and failures in sensors and actuators and so on.

A plant (world) model declaration uses the keyword sequence ‘`def world`’. The bundle content is a list of declarations and statements, like any other bundle.

For example

```
def world name()
 {
   input plant#heater#setting : { off: lo, hi };
   output plant#ambient#temperature : { -273 .. 1000 };

   forever
   {
     sleep_seconds(1);
     if (setting==hi && temperature < 90) temperature += 3;
     else if (setting==lo && temperature < 90) temperature += 1;
     else if (setting==off && temperature > 0) temperature -= 1;
   }

 }
```

A plant model generates bytecode that does not execute on any platform, but which is used for bundle consistency checking.

The sequence ‘`def plant`’ can be used instead of ‘`def world`’ to define a world model: it makes no difference at the moment.

Run-time checking of the real plant's consistency with world models will be implemented.

Chapter 8

Domain Manager

The Domain Manager is an aggregation of services that

- delimit the domain boundary,
- manage domain name spaces, including network addresses and device names,
- allow new devices, bundles and pebbles to join the domain,
- fetch canned bundles in respsonse to trigger actions, re-hydrate them and attempt to insert them into the domain,
- allocate interpreted bundles to execution platforms,
- provide compute resources for soft pebbles,
- provide model checking over all rules in the domain to ensure all properties are satisfied, and
- support removal of devices, bundles and pebbles when no longed wanted.

The Domain Manager provides a few standard pebbles, used by Pushlogic in read-only mode, that provide domain status information.

Bundles run inside a domain of participation (DoP). Dynmaic 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 be loaded, 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.

8.1 Using the compiler to check domains

It is planned that a number of compiled bundles can be read in during a compilation and the bundles being compiled are checked against them. Indeed, no source-level bundles are provided, the compiler will act as a static checker for a collection of object bundles. This has been implemented but no examples written up.

8.2 Incremental and Real-time Model Checking

We wish to design an object bundle file format that is as amenable as possible to rapid incremental model checking or assume/guarantee-style automated reasoning.

Real-time Model Checking ... is one of our main challenges being explored ...

8.3 Federation of Pushlogic DoPs

Pushlogic rules hold within a domain of participation (DoP). The DoP may cover multiple execution platforms, but all rules are shared in terms of consistency checking.[1]

[1]Variations on this model are required in practice, to provide localised behaviour and assurances, to dynamic allow merging and dividing of domains and to provide federation of domains where knowledge about peer domains is available in summary form only.

Chapter 9

Pebbles and Pebble Formal Model

Pebbles themselves are self-contained hardware or software objects that fulfill a certain task. Examples are a numeric entry keypad, an electronic piggybank or a speech recognition engine. Our vision is that all such devices shall, in the future, share a common middleware and reflection API.

A pebble that is compatible with the Pushhlogic/Tuplecore system is defined formally here. The XML schema for such a Pebble follows exactly the same structure.

An uninstantiated_pebble is a quad of a type_name, pebble_dataplane and pebble_behaviour.

A type_name is a string that is unique name for an uninstantiated pebble. It is sensible to use use URI's as type_names, since these can ensure uniqueness, but other mechanisms can also be used.

A pebble_dataplane is a set of (field_name, field_domain, field_type, field_value) quads where the field_names are disjoint.

A field_name is a list of strings. The last string is the final_field_name and the and others are tuple_names. By convention, field names are in lower cases and tuple names are capitalised (first letter is a capital letter). Field_names that are only different in their final_field_name are said to be part of the same tuple. Using Pushlogic, field_names are written with a hash sign between each string, for instance Pebbles#Lamp#status.

A field_domain is a disjoint union of safe and unsafe lists. Each member of the safe and unsafe list is a field_domain_specifier. A field_domain_specifier is a string constant or an integer, an integer range or an ellipsis. (Please see §5.7 for the concrete syntax for field declarations used in Pushlogic.)

A field_type is one of the following values (fluent, event, read_only, lock, money). Most pebbles only use the fluent type for all fields. A fluent is a conventional variable that retains its most-recently-written value and returns this value when read.

A field_value is a constant string that conforms to the field_domain. It is set to the first item on the safe list when the pebble is first created or reset. A field value may be interpreted to have integer or real semantics in some contexts, but it is primarily a string. (There is no NULL string but the string of zero length is allowed. When interpreted as a boolean, all values are true except for "", false, FALSE and 0.)

An instantiated_pebble is the conjunction of an instantiated_pebble and an instance_name.

The instance_name is a globally unique list of strings of which the first may generally be a URI string (ie has the prefix 'tup:').

The field_names of an instantiated_pebble are logically assimilated into the global address space by prepending the instance_name.

The pebble information (metainfo) for a pebble is held as part of the pebble_dataplane as fixed-value fields with pre-assigned field names whose first element is 'info'. Therefore there is no specific part of the pebble XML schema that corresponds to metainfo. These fields have read_only type; their domain safe list consists of a single string that is identical to their fixed-value; their unsafe domain list is empty.

Most of the metainfo is missing until a pebble is instantiated, but the type_name should be present in the field with name Info#type.

Pebbles do not act on other pebbles but can have internal reactive behaviour in the form of various unilateral interlocks and releases. For instance, a hardware interlock may prevent a furnace control field to be set to ON if a thermal cutout field is registering OVERHEAT. This is an example of an interlock. An example of internal reactive behaviour would be if the pebble set its own furnace control field to OFF when the thermal cutout field is reading OVERHEAT. None of the internal reactive behaviour is allowed to set any field to an unsafe value or to prevent a field from being externally set to a safe value. (See also §4.6.3.) The internal reactive behaviour of a Pebble is described (currently) using Pushlogic rules, that have their own schema described §4.1.

9.0.1 Platform Metainfo: Reflection via Pebble Dataplane

An instantiated pebble sits on platform which is either an embedded system or server. In terms of registration and metainfo, platforms have the same structure as Pebbles and hence do not show up as an explicit structure in the XML schema.

A platform has a platform_name that is a globally unique list of strings of which the first might be a URI string (ie. has the prefix `'tup:'`).

For embedded system platforms, quite frequently, there is only one instance of a given pebble ever present on the platform and because all execution platforms have unique names, the plaform's name can be extended with the string 'Pebbles' and then the pebble's type_name is appended on to that to form the unique instance name for the pebble.

For instance, we might have a platform called ‘`tup://192.168.1.100`’ and a pebble type_name of ‘`ThermalCutout`’ giving a pebble instance name of ’`tup://192.168.1.100#Pebbles#ThermalCutout`’.

The metainfo for this example pebble would be found at

```
tup://192.168.1.100#Pebbles#ThermalCutout#Info
```

and we would expted the following field to also have the value ‘`ThermalCutout`’:

```
tup://192.168.1.100#Pebbles#ThermalCutout#Info#type
```

Where there are several pebbles of the same type_name on one platform, these must have intance names that differ somehow. For instance, the pusher platform just adds a decimal number on the end of the type_name.

```
tup://192.168.1.100#Pebbles#ThermalCutout0
tup://192.168.1.100#Pebbles#ThermalCutout1
tup://192.168.1.100#Pebbles#ThermalCutout2
```

Platforms also have metainfo. This is held using field_names that begin with the word Platform. One typical field for a platform is an IP address. For instance, we would expect the following field

```
tup://192.168.1.100#Platform#ipaddress
```

to have value ’`192.168.1.100`’

9.0.2 Bundle Metainfo: Reflection via Pebble Dataplane

Platforms host bundles of running software. The local variables and metainfo for such bundles are readable using the pebble_dataplane XML schema. Like an instantiated_pebble, a running_bundle has a type_name and an instance_name and the same naming conventions are used to create instance_names as are used for pebbles, except the string ‘`Bundles`’ is used instead.

For instance, a bundle instance name might be

```
tup://192.168.1.100#Bundles#ClimateControl
```

and its metainfo would be stored in fields such as

```
tup://192.168.1.100#Bundles#ClimateControl#BundleInfo#version
tup://192.168.1.100#Bundles#ClimateControl#BundleInfo#copyrightMessa
```

and its (Pushlogic) local variables are placed in

```
tup://192.168.1.100#Bundles#ClimateControl#Locals#var1
tup://192.168.1.100#Bundles#ClimateControl#Locals#var2
```

Chapter 10

Execution Platforms

A number of execution platforms are envisaged, but they share the same API when viewed from the network.

Currently, there is an interpreter for the bytecode in the badger/pushlogic directory that can be compiled for embedded systems or workstation use. When compiled for a workstation it is called **pusher**. A native compiler for the CAN/PIC platform is also planned (has been envisaged).

The pusher interpreter may also pretend to be an execution platform pebble, meaning that it might beacon some metainfo so that a domain manager gives it some bundles to run.

Currently, all platforms speak the ETC protocol over the network.

There is a utility called `cmdline` that enables one to manually send low-level ETC commands over the network.

The `cmdline` program allows platform reboot and field read, write and subscribe operations.

Arg syntax is

```
 cmdline [-nNONCE] [-pNNN] [-dURI] reboot
 cmdline [-nNONCE] [-pNNN] [-dURI] w tag v [ tag v ...]
 cmdline [-nNONCE] [-pNNN] [-dURI] r tag
 cmdline [-nNONCE] [-pNNN] [-dURI] s tag
```

For example, to change the value on the display panel pebble one can use something like to set a remote field

```
 cmdline -d tup://169.254.25.193:253 w Pebbles#Display#value 'Hello Wo
```

Several field, value pairs can be supplied to one write command.

There is also a pushdown operation for tuplecore use, but that is not currently used.

10.1 Registration

A platform must announce its presence and register with one (or more) domain manager(s).

We have had various registration technologies, including O2S. Atif is currently implementing UPnP and RDF registration.

Device API reflection is currently achieved through the code reflection interface.

10.2 Code reflection

A platform exports the source code of its running bundles, including the assertions about the operation of the enclosing domain that its bundles have made. This is called *code reflection*.

Code reflection is achieved using an HTTP GET and the code is encoded in XML.

10.3 Web Interface

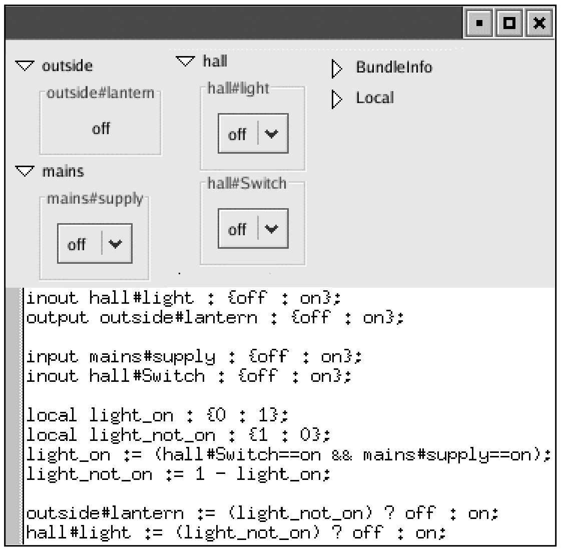
Many platforms implement a web interface that allows web-based viewing of internal state and a certain amount of commanding. The web server provides XML and also a canned CSS style sheet for easy viewing.

10.4 Pusher: Command Line and GUI Tool

The interactive interpreter for pushlogic on workstations is called **pusher**.

Pusher can be run standalone, on a workstation, with a number of bundles loaded from the command line. Figure 10.1 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 variables ‘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 sub-tuple 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.

The GTK GUI allows the user to view and manipulate the TupleCore tree in real time. It uses the tuplecore library to access the TupleCore and it needs a function ’get_domain’ which is included in the PushLogic library. It needs to be initialized using ’init_gtk()’, and ’gtk_tree(root_tup, NULL)’ builds the user interface. A



```
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 10.1: Lanterns - An Example of Pushlogic under GTK GUI

thread needs to be created which runs 'do_gtk()' to exercise the GUI. It creates a gtk_label for output and local variables, which are updated with an upcall from the tuple substrate. For inputs it creates either a gtk_scale if the domain is an integer range, or a gtk_combo_box (i.e. a menu) for an enumerated type.

Th pusher interpreter may also pretend to be an execution platform pebble, meaning that it might beacon some metainfo so that a domain manager gives it some bundles to run.

10.4.1 Pusher Command Line Arguments

Command line arguments are compiled bytecode bundles or options.

The '-nogui' command line option disables the GUI and directs all output to the console.

The '-cycles=nn' command line option makes the platform exit to the OS after so many seconds.

The '-tupdump' command line option prints the internal tuplespace to the console every few seconds.

Bundles should have suffix '`.plc` ' and this is added if no dot is present in the filename. Alternatively, bundles should have suffix '`.plcx` '.

Bundles with a `.x` suffix are XML coded bundles, as reflected from execution platforms. NB: the compiler, pushcomp, generates four object files from each compilation, two of which are loadable by pusher and the remainders are for canning to device ROMs (pushlogic bytecode or .net bytecode).

Bundles are loaded from the current directory or any directory listed on a colon-separated list of directories stored in the PLPATH environment variable.

Bundles can be listed on the command line along with an instance name, separated by an equals. This allows two instances of the same bundle to be loaded. Each will put its local fields in its own local tuple, named with the instance name: for example ‘`ins1=ding.plc ins2=din.plc `’.

10.5 Console Output

Under the debugger, it is possible for a Pushlogic program to write strings to the console and to exit with a return code. This is done by sending events to certain fields of a local tuple called Platform#System. Library functions to assist with this are provided.

Any constant assigned to Platform#System#sysprint is displayed on the console.

Any integer assigned to Platform#System#sysexit causes the platform to return to the OS using the integer as the return value.

```
  output Platform#System#sysprint : event;
  output Platform#System#sysexit : event (0..255);
```

Chapter 11

Other Issues

Parasitic Feedback: Although Pushlogic rules may not cause a higher level field to change value as a result of a lower field changing in value (apart from via undo's) there is nothing to stop a Pebble making a cause/effect connection between a pair of fields in this way. Such interactions can lead to oscillation and violate Pushlogic principals.

Multi-media: The Pushlogic interpreter is not envisaged as having sufficient throughput to directly manipulate multi-media streams. Instead, multi-media streams are started, stopped and routed using Pushlogic as a 'control plane'. The Pushlogic sets fields held on media Pebbles, such as cameras, speakers, fileservers and so on to circuit identifier values. The Pebbles then send the media streams amongst themselves until again commanded by the Pushlogic. If a field naturally ends, then its source or destination Pebble can set the controlling field back to its safe value and the Pushlogic receives an undo that can serve in the same ways as a conventional 'interrupt'.

Asynchronous Eventing: Pushlogic may be thought of as an algebra over asynchronous events, and hence has much in common with the work of our Opera group. Our fields implement what is now known as a publish/subscribe mechanism. The existence of fields is published in the reflection information of a device and they are bound at rule re-hydration time. When a field changes, our wire protocol sends asynchronous notifications to the subscribers.

Index

References

[1] 'A Case for Goal-oriented Programming Semantics' in System Support for Ubiquitous Computing Workshop at the Fifth Annual Conference on Ubiquitous Computing (UbiComp '03). Umar Saif, Hubert Pham, Justin Mazzola Paluska, Jason Waterman, Chris Terman, Steve Ward. http://o2s.csail.mit.edu/goals.html

[2] S. Kambhampati. 'A comparative analysis of partial-order planning and task reduction planning.' ACM SIGART Bulletin, Special Section on Evaluating Plans, Planners and Planning agents, Vol. 6., No. 1, January, 1995. citeseer.nj.nec.com/kambhampati95comparative.html

[3] 'Service Composition for eHome Systems: A Rule-based Approach' M Kirchhof, P Stinauer: Aachen eHome Group. www-i3.informatik.rwth-aachen.de/tikiwiki/tiki-index.php?page_ref_id=206

[4] Universal Plug and Play. www.upnp.org

[5] Greaves et al. '*The AutoHAN project.* www.cl.cam.ac.uk/Research/SRG/netos/han/AutoHAN

[6] Uwe Glasser. '*Systems Level Specification and Modelling of Reactive Systems: Concepts, Methods, and Tools*', citeseer.nj.nec.com/258.html

[7] CVC Lite http://verify.stanford.edu/CVCL/

[8] M. Presburger: 'Ober die Vollstndigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt'. In Comptes Rendus du I congrs de Math maticiens des Pays Slaves, Warszawa, 1929, pp.92-101.

[9] Bierman and Sewell. 'Iota: A concurrent XML scripting language' Technical Report Number 557 Computer Laboratory. ISSN 1476-2986 Iota: !A concurrent XML scripting language with applications to Home Area' www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-557.pdf

[10] 'Towards Ubiquitous End-User Programming' Rob Hague, P Robinson, A Blackwell. ACM Conference on Ubiquitous Computing, Seattle, October 2003. www.cl.cam.ac.uk/ pr/publications/ubicomp03

[11] Lupu E, Sloman M '*Conflict Analysis for Management Policies*' In 5th Int Symp Integrated Network Management IM'97 San-Diego 97 (Chapman Hall).

[12] J Bacon et al. 'Cambridge Event Architecture(CEA)' www.cl.cam.ac.uk/Research/SRG/opera/projects

[13] Monika Solanki et al. 'Introducing Compositionality in Webservice Descriptions'. Proceedings of 3rd ANWIRE workshop on adaptable services, DAIS-FMOODS, November 2003, Paris.

[14] Trolltech, Creators of QT. www.trolltech.com.

[15] 'Paramodulation and theorem-proving in first-order theories with equality.' G. Robinson and L. Wos. In D. Michie and R. Meltzer, editors, Machine Intelligence, Vol. IV, pages 135-150. Edinburg U. Press, 1969.

[16] W3C: Simple Object Access Protocol (SOAP) www.w3.org/TR/SOAP

Many thanks to Daniel Gordon who implemented much of the first Pushlogic system and who contributed greatly to its definition.