Copy Number .....

Waveform Description Language

P6957-11-014
13-April-2000 Issue 1

Written by

.....

E.D.Willink

Reviewed and Approved by

Project Manager

.....

C.Waugh

Authorised by

Project Director

Date ....

C.P.Ash

DERA Portsdown
Portsdown Hill Road
Fareham
Hants
PO17 6AD

Configuration Management	
CM Tool/Version	Visual SourceSafe v5.0 and v6.0
Library Location	Q:\P6957\Vss\…
Library Project	$/P6957/Project Library/11 Technical Reports & Notes/ 014 Language
Baseline	
Elements	`Language.doc` - words `Language.vsd` - pictures `WdlSyntax.y` - syntactic grammar `WdlLexer.l` - lexical grammar `Language.wdl` - syntax checked examples `Language.wsg` - picture icon templates

Modification History			
Author	**Ident**	**Date**	**Description**
EDW	25 February 2000		Very preliminary draft made available to interested parties
EDW	Issue 1	13-April-2000	End of phase 1.

Table of Contents

List of Figures

List of Tables

1 Introduction

This document will eventually become the WDL language definition. It is currently in a rather transitional state from a document that started as a discussion of the requirements of the WDL type system and evolved into a discussion of the problems of representing statecharts, hierarchical flows and leaf behaviour consistently.

Until suitable tutorial material is available, readers may find it helpful to refer to the PDR Library Primitives (P6957-11-004) for a description of associated library support and to the FM3TR Decomposition (P6957-11-005) for an extensive example application.

The Waveform Description Language (WDL) has two distinct and exactly equivalent syntaxes:

- Wdl is a conventional textual language like C or Ada.
- WdML is an XML dialect.

1.1 Conventions

Type names are in `CapitalisedMixedCase`.

- Types
  - Enumerations
  - Records
  - Unions
- Entities (class names)
- Messages (type names)

Instance-names are in `lower_case`.

- Entity instances (instance names)
- Messages (flow names)

Manifest constants are in `UPPER_CASE`

- Enumerators (including state names)

Library entity names are nouns rather than verbs. Since entities are a form of type (they can be instantiated), entity names should follow a `CapitalizedMixedCase` spelling convention.

`typewriter` font is used for code examples, with structuring keywords often shown in **`bold`**. This does not imply that the keyword is reserved; there are no reserved words in Wdl.

1.2 History

(CORBA [CORBA98]) IDL was used as a starting point for the number and type concepts within Wdl. However IDL specifies an exact implementation, whereas WDL seeks only to specify the minimum requirements to be satisfied by a conformant implementation. Very little of IDL remains directly visible in WDL, however once a compiler has chosen implementation types for each specified type, translation of WDL to IDL should not prove as hard as might first appear.

MoML [Lee00] (the XML [XML98] dialect used by Ptolemy II [Davis99]) provided some inspiration for the message flow syntax and the overall syntactical structure. However MoML leaves most syntax to be resolved from unspecified text strings, which is rather

unsatisfactory for a complete language. The 10 or so language elements of MoML therefore become 100 or more, once types and statecharts and expressions are all fully defined.

2 Further Work

The language described in this document is not fully defined, and the description is not an adequate definition. Language constructs are defined by way of example, rather than by precise exposition of syntax alternatives, which the reader may deduce from the grammar in Section 9.2.

Further work must therefore involve

- more detailed consideration of the language
- a detailed language specification
- a good language tutorial (elaborating the accompanying FM3TR decomposition)

No language is ever complete, so changes must be expected as

- experience is acquired from specifiers
- compilation problems and opportunities arise
- implementers provide feedback
- related tools and standards appear
- compilers improve
- target processors evolve

Successful code generation from a WDL specification involves a number of advanced transformations not present in conventional compilers. Some of these are outlined in Section 9.3. Some must be automated, while others can be performed manually in order to minimise the complexity of a first compiler. Thereafter some of the manual transformations can be automated, while others can be assisted. It is unlikely that transformations that involve fundamental system design decisions can be fully automated.

Throughout this document a 10% grey background is used to indicate particularly tentative areas. Readers should be aware that usage and review of the language has been fairly limited, and so some revisions should be expected as experience is acquired. The remainder of this section comprises a note form list of some of the outstanding issues.

2.1 Syntax

Reserved words

Is no reserved words really a good idea?

TextLiteral

Should Java inserts be a distinct lexeme with unique surrounding context?

Grammar

doc constructs everywhere

property constructs more consistent, what is a property?

some constructs such as statement cover three or more contexts

2.2 Names

What are the name visibility rules?

How are namespaces declared and used?

Where are useful constants such as pi defined?

Is a reserved word such as this necessary? Examples currently use this, value, type-name rather arbitrarily.

2.3 Expressions

More generality? Can an if be an expression without a dangling else problem?

Constructors and Destructors

2.4 Types

Built-in types

Full definition of built-in attributes

Set/Bag/Dictionary

Flows

How is the Nyquist rate specified?

Unions

How does a composite discriminator type get created for a discriminated union of discriminated unions particularly when the innermost ones use generic enumerators?

Strings

What are they?

Arrays

Define rest of data parallelism?

Do we want strong shape checking or minimum shape selection? Strong checking catches errors. A minimum shape could be supported by an appropriate coercion.

Do we need l-value data parallelism?

How do variable-sized arrays really work? Want to declare an upper-bound but construct an actual size somehow.

Consistency of continuous arrays

Are list concepts necessary for flexible construction?

Classes

How are class and instance concepts distinguished?

Entities/Instances

The same keyword seems to be being used for entity instances as well as entity definitions. One must be given a distinct name.

Overloading

How is multiple dispatch resolved for add?

Generics

How is a generic type that must comply with an interface specified?

Is the semantics of generics that are naturals consistent with those that are types?

Perhaps generic that is natural should be: generic X : value = natural;

and generic that must comply with Interface is: generic X : type = Interface;

2.5 Concepts

Inheritance

Do we need it? If not why not? Yes for types, Maybe for entities

What about multiple inheritance? Perhaps Java-like, one primary, additional secondary.

Time interpolators

Definition of continuous time syntax

Library blocks to convert to continuous time.

State change

Blank transitions seem to be pragmatically interpreted as automatic/on child exit transitions. Blank transitions should be automatic. On child exit transitions must therefore use a reserved exit transition name.

Dynamic configuration

New/Delete of processes/threads

New/Delete of objects?

Meta-level

constraints define meta-level execution.

let constructs are a little ambiguous as to the meta/non-meta context.

Provision of anEntity._ports for a constraint iteration is very meta.

Shouldn't there be a clearer perception of meta?

Refinement

The semantics of refinement, extension, composition need much more detail.

2.6 Constructs

Constraint expressions

collect and select

. or -> as set operator

are types/values used consistently

Exceptions

What are they?

2.7 Code generation

Code emission

Generation/encapsulation of reference Java

Ditto other languages.

Partitioning

Declaration of processors, resources.

Can resources be shared? If so how is deadlock avoided? Probably adequate and easier to require full exclusive acquisition prior to starting.

Constraints to direct entities to available resources.

Procedures for sequencing external utilities/writing makefiles.

2.8 Versioning

Where is a WDL language version defined and used?

How is a WDL version mismatch accommodated?

How is application version control supported?

How is a build state documented?

How are build dependencies maintained?

2.9 Formality

Can any analysis show where WDL is unsound?

Is a WDL specification complete and deterministic?

2.10 External Review

A preliminary external review of the language concluded that the ideas were appropriate for the target application domain, and suggested that there may be scope for claiming a degree of formality and soundness.

It was suggested that multi-dimensional arrays may be unnecessary given the support for nested arrays, and that eliminating multi-dimensional arrays may avoid a syntax ambiguity with respect to empty vectors of empty vectors. This needs investigating and probably adopted.

A more general type was suggested to unify records and discriminated unions. Each field would have a not valid characteristic. This provides a much more general capability, but it is unclear that the extra functionality addresses any practical problems, and it is doubtful that using a concept further removed from practical problems would be acceptable to users. This can be considered further, but should probably be rejected.

A number of more minor comments have been incorporated into the main text.

3 Language

WDL is an implementation-neutral language in which the externally observable behaviour of a (sub-) system can be specified unambiguously[1].

> The decision to avoid ambiguity requires behaviour to be deterministic, and consequently excludes many models of computation. This is unfortunate, but may be alleviated at the specification level by explicitly enumerating alternative deterministic behaviours. An implementation may then choose which alternative to adopt.
>
> Many WDL characteristics are direct corollaries of the need for deterministic behaviour. Characteristics that lead to indeterminacy must be revised.
>
> Everything can be specified in a machine readable form so that tools may gradually check more and more. However some specifications such as case colour may never merit automated checking. Other specifications such as power consumption may be inappropriate or too difficult to automate.
>
> The philosophy is to introduce a mechanism that allows an over-Utopian specification to be made more practical, rather than to have a specification that is practical today, but becomes restrictive as better tools become available.

3.1 Principles

A WDL specification uses the ideal behaviour of an infinitely fast implementation as a reference, and requires practical implementations to satisfy tolerances with respect to this reference.

- inertia to bound tardy behaviour
- runaway to bound premature behaviour
- latency for clock jitter
- spurious energy for signals

Practical implementations may use any algorithm or computation order that satisfies the tolerances on the ideal observable behaviour.

Specifications may restrict implementations by imposing bit-truth on selected computations or incorporating some internal nodes as external observation points.

Although implementation details should be omitted from an abstract specification, refinements may be supplied in order to specify a particular implementation. The degree of refinement necessary depends on the capabilities of compilation tools.

The over-specification implicit in most implementation languages is avoided. As much parallelism as possible is preserved without introducing indeterminacy. The concepts of data flow [Bhattacharyya96] and data parallelism are used to specify parallel computations.

---

[1] Two forms of indeterminate behaviour can be specified.

- A specification involving a random process may produce results that cannot be reproduced without knowledge of the random process.
- A specification of a chaotic process in which calculations are excessively precision sensitive may lead to unrepeatable results unless bit-true arithmetic is also specified.

Each of these can be made determinate by defining the introduced error as an observable input.

The behavioural specification of an entity is composed from a hierarchy of the (sub-) specifications of (sub-) entities and the specification of their interactions. The behavioural specification of leaf entities is defined directly.

Interaction occurs only through message flows between entities. Computation occurs in response to rendezvous between message flows, which can be continuous (signals, values or constants) or discrete (events or tokens). Computation is performed in an Object-Oriented style using the data types of messages. Entities may be polymorphic with respect to data element type, array dimensionality and message flow behaviour. Deterministic behaviour is ensured by strict rules concerning event concurrency and state update.

Observable

An observable behaviour is

- the occurrence of an external event and its associated information
- the information carried by an external signal or value flow

Token flows are not observable.

The Nyquist rate of a signal flow is not observable.

The relative timing of coincident events, signals and values is not observable.

Should there be an `observable` keyword to annotate observable internal flows?

3.2 Numbers

The WDL number system supports conventional number concepts such as `1.23e-4` and `0x5678`. It adds support for dimensions, number ranges, statistical distributions and optimization criteria.

3.3 Types

The WDL type system has many similarities but also significant differences to conventional languages.

The built-in types define behaviours with potentially infinite precision. Practical types are tailored from these by specifying their required representational range, accuracy, overflow and rounding characteristics.

Enumerations and records (classes) are fairly typical of Object Oriented languages.

Unions are discriminated - the type (rather than just the programmer) knows the content.

Arrays are of arbitrary dimensionality and scalars are degenerate arrays. Arrays may be distributed over multiple connections, or in time. Arrays distributed in time need not have discrete indexes.

Bit-truth is supported by layout types and overlays.

3.4 Entities

Types define the low level computational datums. Entities define a highly encapsulated high level with precise but flexible scheduling semantics.

Any system may be described by exactly one entity.

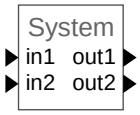


Figure 1 Single Entity System

The entity has a number of input and output ports to support interaction with its environment. Each interaction takes the form of a message that flows into an input port or out of an output port in accordance with some communication protocol. `inout` ports support bidirectional[2] and less deterministic flows.

Few systems are sufficiently simple to permit direct specification or implementation. It is convenient to decompose the system into sub-systems and provide sub-specifications for each sub-system.

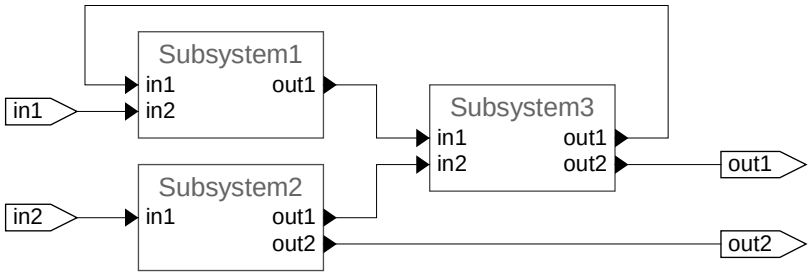


Figure 2 Decomposed System

> The graphical presentations in this document are intended only to indicate the information that could be conveyed by a well-configured schematic editor. It is not intended that a practical system should be so lacking in mnemonic quality. The pictures are the result of a pragmatic configuration of Visio.

The decomposition into three subsystems and four hierarchical ports introduces extra message flow paths, but preserves the concept that a specification comprises entities and message flows between ports. Some flows may traverse the decomposition hierarchy.

The sub-specification of a sub-system may be recursively decomposed into sub-sub-specifications. Eventually decomposition reaches a level where further decomposition does not assist the specification activity; it is necessary to specify the behaviour directly.

Any entity comprises

- external ports to interact with other entities

In principle any entity, but in practice just leaf (or statechart) entities may comprise

- internal attributes to define entity state
- internal operations to share response functionality
- responses to define behaviour

Each response may have its own private state

[2] Bidirectional flows are very rarely necessary, since a pair of unidirectional flows is usually clearer, however antennas and multiplexed buses are bidirectional.

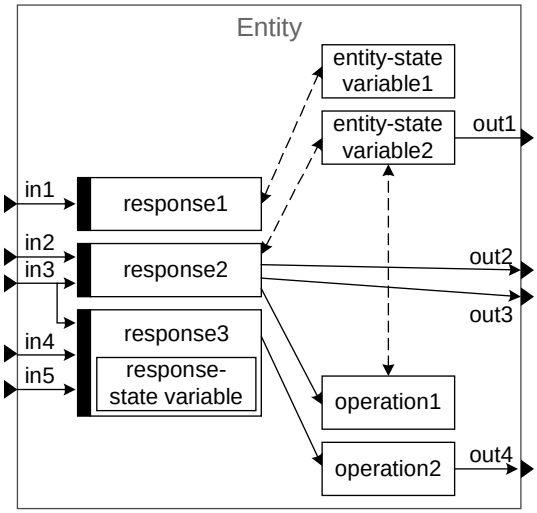


Figure 3 Anatomy of an Entity

Response

Each response performs a rendezvous of some subset of the inputs, before computing results using the rendezvoused inputs and internal state. The computation may involve generation of message flows[3] to output ports and update of internal state.

The semantics of rendezvous and state update, to be described shortly, ensure that state update is deferred until completion of the response processing.

Operation

Operations define functionality for use by the response code. Operations are accessible only from the responses or other operations of the entity.

Entity State

Entity state variables are accessible only to the operations and responses of the entity. A limited form of external visibility may be provided by defining that an output publishes a value computed from one or more state variables.

Update of entity state variables is deferred until computation within the entity has completed.

Response State

Response state variables are accessible only to the response.

Update of response state variables is deferred until computation of the response has completed.

Integrity

The rendezvous ensures the integrity of response state.

The constraint on event concurrency and deferred entity state update ensures the integrity of event state.

[3] The term *flow* rather than *message* is used, to avoid the implication that communication occurs at discrete times rather than continuously.

3.5 Flows

Interaction between entities occurs as a result of (message) flows. Each flow has a data type to define its content, and a flow type to define its communication strategy. Continuous communication may use a signal, value or constant flow type. Discrete communication may use an event or token flow type.

Signal

A signal flow specifies a continuous communication. It may be implemented as either a continuous time (analogue) signal or by a discrete time (sampled) signal. Discrete sampling satisfies a Nyquist criteria and so the continuous signal can always be hypothetically reconstructed using an ideal rectangular filter in order to validate the specified behaviour. This ideal reconstruction filter has no delay, no attenuation within Nyquist bandwidth and infinite attenuation outside.

Specification of a realistic sample rate for many practical systems necessarily introduces a small degradation through loss or aliasing of energy outside the Nyquist bandwidth. WDL therefore supports specification of the Nyquist rate of a signal as a hint to an implementer, not as a permission to ignore any associated degradation. Refinement of a specification to use a practical sample rate is a system design trade-off in which the (small) distortions introduced by a reduced rate are accounted for in a total implementation loss budget.

Value

A value flow also specifies a continuous communication, however a value flow is not inherently time varying. A value flow typically defines asynchronous control. It therefore changes at irregular times in response to the irregular changes of its source. The Nyquist rate for a value flow is infinite.

Since a value flow is asynchronous, it is beneficial to specify a fairly large inertia tolerance to maximise the scheduling alternatives available to an implementation.

Constant

A constant flow is a restrictive form of value flow that is guaranteed not to change within the life-time of its recipients. The Nyquist rate for a constant flow is therefore zero. The constant assertion provides for greater clarity in a specification, may assist an automated code generator, and may constrain the selection of candidate implementations. A constant flow does not however differ fundamentally from a value flow and so constant flows are not distinguished when the differing semantics of event, token, signal and value flows are discussed.

> The stronger constraint is intended primarily to allow practical implementations of library entities to declare their limitations. WDL does not distinguish between inputs, parameters or properties and so the size of an FFT is an input. It is difficult to provide an efficient implementation that dynamically supports any FFT size, and so an implementation may declare that its size is a constant input. Since most practical applications require deterministic FFT sizes, a WDL compiler should find that the restrictive implementation is adequate.

Constants inputs may be loosely referred to as parameters or properties.

Event

An event flow is the primary form of discrete message flow. It is the sole stimulus for a discrete computation. Concurrent computation of more than one event is prohibited in order to satisfy the requirement for specification of deterministic behaviour; events are serviced sequentially. This is a highly prescriptive model of computation that hides any

parallelism that could be exploited by an implementation. However this strict foundation enables a compiler to analyze the dependency of events and so remove unnecessary sequentialization.

Token

An alternate and more flexible form of discrete message flow is supported by token flows using the dataflow paradigm. A token flow computation occurs in an entity as soon as the requisite set of input tokens is available. Completion of a computation typically produces further output tokens. Token computations can be performed in any order with any degree of parallelism, since a computation cannot be performed before its inputs are available, and since subsequent computations cannot proceed until prior outputs are produced. A naive implementation of token flows requires a potentially infinite FIFO to buffer each token flow. Compile-time analysis can eliminate most FIFOs and severely bound others.

3.6 Concurrency

Concurrent processing of multiple events may lead to non-deterministic behaviour. Therefore concurrent events are forbidden in the ideal reference, although a practical implementation may of course introduce concurrency provided the observable behaviour remains within tolerance.

Prohibition of concurrent events requires that a deterministic sequence be established for all co-occurrences.

External concurrency

An arbitrary precedence between external ‘concurrent’ events can be imposed since no practical system can achieve sufficient precision to guarantee that events should be treated concurrently. An external requirement for genuinely concurrent events should be changed into a requirement for a single compound event.

Child concurrency

Response to an event may generate a child (sub-)event. This sub-event cannot occur before or concurrently with its parent, so it must occur after its parent. In order to ensure deterministic behaviour for an arbitrary hierarchy, it must occur immediately after its parent so that sub-event and its sub-sub-events appear as extension to the original parent event. All token-based computation and all child events are completed for one event before any further sibling or ancestral event is processed.

Precedence

A safe response generates fewer event outputs than it has event inputs (at most one). An unsafe response may generate co-occurring events. A precedence must be defined for these to avoid a potentially non-deterministic concurrency.

Events may be generated by

- `Clock` entities
- `When` entities performing signal threshold detection
- `Event` entities (a token rendezvous generating events)

A precedence order for such entities is established by using the `_after` parameter to specify that the concurrent events of one entity occur after those of another. It is impractical to provide such specifications directly in a hierarchical design, and so every entity implicitly generates events after its hierarchical parent. The precedence between nested entities is established by specifying `_after` for appropriate parent entities.

Expressions

A `seq{}` expression may involve multiple sequential computations, some of which may be event generating. `seq` defines an execution and consequently a generation order.

A `par{}` expression may involve multiple potentially concurrent computations some of which may be event generating. `par` does not define an execution order. A `par` that generates no events is safe, a `par` that generates one event is safe if it is the only event generating response to an event, a `par` that generates multiple events is unsafe and must be prohibited.

> Requiring the compiler to detect safety violation is restrictive but reliable. A specification should be manually restructured to eliminate or sequence the multiple event generation.
>
> The compiler could be required to impose an order, possibly by converting a `par` as a `seq` for the purposes of side-effect sequencing. This is an automated restructuring that could be difficult to implement since it could involve difficult remote synchronisations. However this violates the principle of `par`: expressions that can be executed in parallel. If such execution involves unpredictable side-effect order, then execution cannot be in parallel.

loops

3.7 Life-time

An alternate form of decomposition is provided when a UML [UML99] statechart [Booch99] is used to define the behaviour of an entity, and the UML statechart syntax is extended to support consistent hierarchical decomposition.

For instance, consider a simplified receiver subsystem that responds to a received signal to extract an initial synchronisation data packet followed by successive data packets. If data integrity is lost, the subsystem can be sent a message instructing it to reacquire.

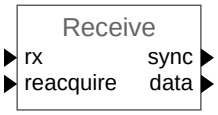


Figure 4 Receiver entity

This behaviour may decomposed into two states, the first of which performs acquisition algorithms to acquire the signal before emitting a message to publish the discovered synchronisation packet. The second state tracks the signal and emits each data packet.

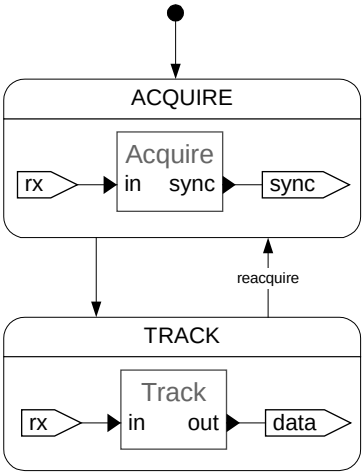


Figure 5 Receiver statechart

The outer boxes and transitions in Figure 5 use UML statechart syntax and show two states and an initial state transition to the ACQUIRE state. The `reacquire` input provokes a transition back to the ACQUIRE state. The transition from ACQUIRE to TRACK has no associated triggering event and so occurs when the ACQUIRE state (or any non-concurrent part of it) exits.

The inner boxes extend UML by allowing the internal state behaviour (the do action) to be defined by a nested message flow diagram. The nested diagrams in the example are very simple, just establishing the connectivity of the external `Receiver` connections to the internal `Acquire` and `Track` entities.

The exclusive nature of statechart states precisely defines the observable life-time of their contents and corresponding flows. The `Acquire` and `Track` entities therefore have an exclusive existence and so are unable to communicate or share context with each other, or successive incarnations of themselves. The flows experience corresponding control of their connectivity. The `rx` flow is connected in all states, so is permanently connected and experiences an instantaneous change of connection at the state transition. The `sync` and `data` outputs are only connected in some states, which is not a problem for discrete flows. The `reacquire` input is used by the state machine and so must be a discrete flow.

> The instantaneous context changes define the infinitely fast ideal reference behaviour. Practical systems may involve buffering or blocking delays and so may have to communicate some form of timing to accompany the state transition. The presumption of infinite speed avoids this very significant but highly implementation dependent detail appearing in the specification.

3.8 Computation

Continuous computation (analogue or digital signal processing) is defined by the required time-dependent behaviour of a signal flow.

A Nyquist rate may be specified for a signal flow to instruct a compiler how to implement the processing in discrete time. An ideal rectangular bandlimiting filter is applied to reconstruct the continuous flow for specification tolerance analysis.

> This can give deterministic behaviour for non-linear operators such as a squarer, provided such operators correctly constrain the relationship between output and input Nyquist rates.

A token rather than signal flow may be specified when the option of an analogue implementation can or must be excluded.

Discrete computation occurs only at events. Deterministic behaviour implies:

- discrete computation completes during an event
  - token based computation proceeds to exhaustion
- no concurrent events
  - child events are serviced at the end of their parent event
- state stable during computation
  - entity state updated after event
  - response state updated after response
- continuous flows are stable during an event
  - continuous flow inputs are sampled at the start of an event
  - continuous flow outputs are updated at the end of an event

3.8.1 Responses

Computation is performed by entities that respond to input flows. A leaf entity or statechart may have multiple responses, each using some subset of its inputs. Each response occurs after a rendezvous has been established between all inputs of the response.

A state machine is a particular useful stylised behaviour for which a subset of the possible responses are enabled according to the prevailing state. Each enabled response performs a rendezvous between its triggering flow and any other flows used to evaluate the guard condition. The subsequent action code is executed and the state is updated as appropriate.

3.8.2 Rendezvous

The conventional rendezvous synchronization concept delays computation until all parties to the rendezvous are ready. In WDL the parties are the inputs, and the distinct flow types of inputs lead to distinct forms of rendezvous.

A continuous rendezvous involves only continuous flows and occurs continuously. This is a null operation.

A discrete rendezvous involves at least one discrete flow and consequently occurs at the same time as an event. Continuous inputs to a discrete rendezvous are sampled at the event instant. There can be at most one active event, and so there are exactly two forms of discrete rendezvous. An event rendezvous for which there is an event input, and a token rendezvous for which there are no event inputs.

A token rendezvous occurs conditionally depending upon whether all the required token inputs are present at the event instant.

An event rendezvous occurs unconditionally and so an error may occur if required token inputs are missing. This potential error is a compile-time error unless it can be established that the error never occurs. Using a `Poll` library entity rather than an implicit rendezvous or explicit `Synch` library entity can circumvent the error.

It is unsatisfactory to have a language definition that depends upon the proof capabilities of a compiler. It may therefore be necessary to allow manual assertion that a token input will never be missing.

The outputs of discrete rendezvous are tokens unless specified to be events. Multiple event outputs are sequenced so as to be non-concurrent. Any events generated by a token rendezvous are sequenced with respect to their parent context.

3.8.3 State change

The relative ordering of token flow computations is unpredictable, so token computations cannot have non-token results. A token flow cannot update an entity state variable. Only an event flow can update entity state.

Update of an entity state variable must occur at the end of an event to avoid indeterminacy in the use of prior/post state by concurrent computations.

Update of a response state variable must occur at the end of the response to avoid indeterminacy in the use of prior/post state by the response, and to allow multiple firings of the response during a single event.

Multiple updates of a state variable must have a deterministic final state. Therefore the potential updates are queued but only the final update is applied.

3.8.4 Computation Order

A precise computation order is defined to ensure that the behaviour is deterministic. A practical implementation may use any order that exhibits the same observable behaviour. A flexible specification should use token rather than event flow wherever possible to maximise the opportunities for concurrent or optimized execution.

Discrete event computation

A discrete event computation comprises the following sequential stages:

- immediately before an event, all work has been completed; nothing to do
- event occurs and is delivered to the event rendezvous of some response
- response is computed involving the potentially concurrent
  - generation and recursive computation of discrete token flows
  - queuing of child events in deterministic order
  - queuing of entity state variable updates (retaining last of multiples)
- all pending entity state variable updates are performed (concurrently)
  - entity state values are propagated for value flows
  - context of continuous signal generators is adjusted
- queued child events are computed recursively
- immediately after an event, all work has been completed; nothing to do

Discrete token computation

A discrete token computation occurs only within the context of an event computation, whether by direct invocation from the event computation or through recursive activation of token computations. It comprises the following sequential stages:

- token rendezvous of some response is satisfied
- response is computed involving the potentially concurrent
  - discrete token computations
  - queuing of child events in deterministic order
  - queuing of entity state variable updates
  - queuing of response state variable updates
- update of response state variables (concurrently)

State machine computation

A state machine computation fits within the framework just outlined for discrete event and discrete token computations.

Each state machine state is a state variable updated in the same way as any other state variable. The state is not used (and must not be allowed to be used) during state transitions, so the relative timing of state update and transition actions is not significant.

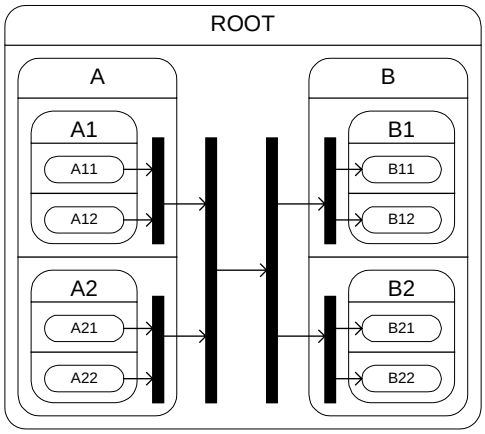


Figure 6 Trees of state changes

As shown in Figure 6, a state machine transition potentially involves exiting from multiple concurrent states (A11, A12, A21, A22) that form the leaves of a tree, followed by entry to multiple concurrent states (B11, B12, B21, B22) that form the leaves of another tree. The two trees share a common root within some unaffected parent state (ROOT). Transition action code may be associated with each branch, and entry/exit action code with each node. UML (and WDL) requires that action code appear to be executed in sequence along each path. WDL allows parallel paths to execute concurrently, but requires that there be no associated event concurrency violations. A WDL compiler must enforce this constraint.

Correct sequencing of the transition actions is ensured by creating a token flow from all old states to all new states, with transition, entry and exit action code activated as token responses. It is possible for multiple events to be generated during a complicated transition, and so the possibility of concurrent event generation must be resolved. Events that originate on a linear trace through the transition trees have a clearly defined order. Events occurring on parallel traces must be given an explicit precedence to ensure deterministic behaviour. This may be achieved by annotating states with the `_after` parameter.

This policy ensures that the entire transition occurs as a single event, followed by the state update, followed in a deterministic order by any child events generated by the transition. It does not seem appropriate to require that a child event generated along some branch of the tree be full serviced before processing of further nodes and edges of the transition graph, since such servicing could involve re-entrant state change. This may be regarded as a restriction on, or clarification of, UML semantics.

A state machine is normally considered to respond to events, however the semantics of WDL token and event flows permit a state machine that responds to tokens, provided every response is arbitrated by the same source token flow. In this case the state machine state can be maintained as a response rather than entity state variable, and the state machine can fire more than once per event, and specifications need not create spurious events just to sequentialize state machines that can run in parallel.

Continuous computation

A continuous signal computation occurs continuously, except that continuous computation is suspended during the zero time duration of each discrete computation. The result of a continuous computation is the fixed-point of the continuous time differential equations. The computation is performed so as to provide all required results.

The result of a continuous value computation is available whenever it is required.

Implementations can choose whether to use eager[4] or lazy[5] evaluation since evaluation must be free from side effects.

3.8.5 Open Circuits

General purpose entities, particularly library entities, tend to have a number of inputs and outputs that are not always needed. It is inconvenient to have to terminate all inputs with a zero `Constant` and all output with a `Sink`, however allowing missing connections eliminates an opportunity for diagnosing specification errors.

From a perspective of ensuring safe deterministic behaviour:

- An `event` input port may be left unconnected.
- Input non-`event` ports with a default value may be left unconnected.
- A non-`token` output port may be left unconnected.
- A `token` output port may be left unconnected, provided the same form of lack of connection occurs in all states. This allows the case of a hierarchical entity whose output is unconnected, but establishes a connection from an inner leaf entity to the hierarchical output. It does not allow a connection into a multiplicity of alternate states, all of which happen not to use the flow.

In practice, WDL requires its schematic editor to support customisation of icons to provide variable numbers of connections. This same support should allow relevant control inputs to form part of the icon, while irrelevant control inputs are omitted, receiving their values by default, or by textual definition. It would seem appropriate for this customisation to also support the concept of deliberately-not-connected to avoid the need for explicit `Sink` entities. Ports not appearing on the icon should then observe the above rules. Ports appearing on the icon should be explicitly marked as not connected.

When a token output is left unconnected in some but not all states, the lack of connection should be diagnosed, since an implementation may require an infinite FIFO to buffer tokens while in unconnected states.

The potentially infinite FIFO buffering token flows should be located so as to avoid loss of tokens. If a token is generated it must be delivered if at all possible. This may require a FIFO to be located near the source if its sink is dynamically connected, or near the sink if the source is dynamically connected. Tokens may only be lost if generated into a flow hat is never connected during the life-time of any part of the connection. This needs some formality to resolve pathological cases of dynamic partial connections along long paths.

3.8.6 Forks

When a message flow is connected to more than one destination the message flow forks to produce multiple forked flows.

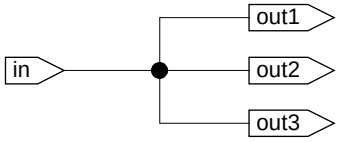


Figure 7 3 way message fork

Signal and value flows may fork. The same continuous information flow is made available to all forked flows.

[4] When inputs are available.
[5] When result is needed.

Token flows may fork by creating multiple copies of the token, so that the same sequence of tokens flows down each forked flow. However there is no implied synchronisation between the forked paths, so it may be necessary to provide independent infinite FIFOs for each forked flow. This generality is necessary to support distinct life-times for the recipients of each flow.

> The common use of forks to feed the same flow to a variety of alternate destinations can lead to a specification error unless all deselected flows incorporate a sink to discard unwanted input. In the absence of the sink, all input intended only for the selected destination is buffered awaiting selection of the unwanted one.

Event flows may not fork, since this would create two concurrent events. The `Event` library primitive supports the generation of multiple sequenced events.

3.8.7 Joins

When more than one message flow is connected to a single destination the message flows join.

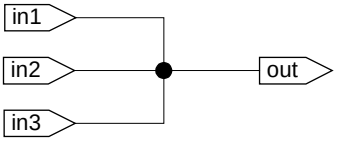


Figure 8 3 way message join

Signal and value flows may not join, since there is no mechanism to resolve the potentially conflicting coexisting values.

Token flows may not join, since there is no mechanism to interleave the tokens deterministically.

Event flows may join, since events are non-concurrent and so can be interleaved in order of occurrence.

3.8.8 Arrays

An array of mixed flow types is not permitted.

A conventional array of events violates concurrency constraints, therefore an array of events may be defined as the routing of an event from a source array element to a destination array element.

An array of tokens should all co-exist, there should therefore be no distinction between an array comprising a token for each element and a single token comprising the entire array value.

Is this too pragmatic? Is it unreasonable or inconsistent to exclude the routing of a single sporadic token from source to destination?

All elements of an array of continuous flows exist, so an array of continuous flows is the same as a continuous flow of the array.

3.8.9 Conversions

Each message flow type in a specification must be explicitly specified or deducible from the constraints imposed by the sub-specifications of the contributing entities. Some library entities permit a variety of flow types to be used, and so there may be occasions where a message flow needs conversion from one form to another. A few flow conversions can be

performed automatically. Most cannot.

To Value

Conversion of a non-value to a value is unsafe, since the time of conversion and consequently the value is not deterministic.

The `Store` library entity can be used to convert an event or token to a value.

A signal must first be sampled by an event before a `Store` can convert it to a value.

To Signal

Conversion of a non-signal to a signal is unsafe, since no other flow type has a well-defined continuous time value and a finite Nyquist rate. Generation of signals is performed by specialised `Generator` library entities.

A value can be converted to a signal, if it can be established that the flow satisfies the Nyquist rate. This conversion is always possible for a constant flow.

To Token

A signal or value may be converted to one or more tokens in order to satisfy the requirements of a discrete rendezvous. The rendezvous necessarily occurs during the processing of some event and so the conversion uses the value of the signal or value flow at that instant.

An event may be converted to a token when fed to an entity that explicitly requires a token input.

To Event

A signal or value cannot be converted to an event, since neither have a time of occurrence.

An event may be created from a signal or value, by establishing a rendezvous between some other event and the available signal or value.

The `When` library entity may be used to create an event at the instant one signal crosses over a threshold defined by another.

Token flows cannot be safely converted to events, since multiple tokens can occur concurrently, and multiple events cannot. The `Event` library entity is therefore provided to perform the conversion and support establishment of a precedence for concurrent events.

> A single conversion could be performed automatically, however this convenience could prevent successful re-use within a more complex specification.

The automatic flow conversions are summarised in Table 1

From \ To	event	token	signal	value
event	event	token	Illegal (use a `Generator` entity)	Illegal (use a `Store` entity)
token	Illegal (use an `Event` entity)	token	Illegal (use a `Generator` entity)	Illegal (use a `Store` entity)
signal	No event	As many tokens as required	signal	Illegal (use `Clock`, `Synch` and `Store` entities)
value	No event	As many tokens as required	Illegal (use a `Generator` entity)	value
constant	No event	As many tokens as required	signal	value

Table 1 Flow conversions

3.8.10 Return Flows

Discrete flows may have an associated return flow.

The reverse flow for an event supports synchronous communication. An outgoing event carries a command which is serviced instantly enabling a reply to be carried back to the sender. The reply is made available during a second phase of rendezvous operation....

The reverse flow for a token supports synchronised asynchronous communication. An outgoing token carries a command which may sit in some FIFO until it can be serviced. Eventually a recipient Receiver entity may accept the command and return a reply which is dispatched instantly to the original Sender which generates an event containing the reply....

3.8.11 Significant Library Primitives

`Clock` - time event flow generator

Generates time dependent events.

`When` - signal event flow generator

Generates an event one on signal crosses over another.

`Event` - token event generator

Generates an event from a token.

`Poll` - token existence event flow generator

Tests for the availability of a token at the start of an event.

`Synch` - token or event generator

Performs a rendezvous.

`Generator` - signal flow generator

A family of generators of signal flows.

`Store` - value flow generator

Updates a value flow at the end of an event from a value passed during an event.

`Constant` - constant flow generator

An invariant value.

`Delay` - token or signal flow delay

Imposes a timed delay on a signal flow.

Imposes a counted delay on a token flow.

`Sender` - returned flow sender and receiver

Initiates an outgoing returned flow, and receives its return.

`Receiver` - returned flow receiver and sender

Receives an outgoing returned flow, and generates its return.

4 Values

4.1 Names

WDL names are similar to modern languages, and very similar to Java.

4.1.1 Identifiers

An identifier in WDL follows the example of many modern computer languages. An identifier is an arbitrary-length case-sensitive sequence of alphanumeric characters and underscores, with the constraint that the first character is non-numeric.

```
alpha                   [A-Za-z$_\.]
symbol_char             [0-9A-Za-z$_\.\-]
identifier              {alpha}{symbol_char}*
```

More general languages such as XML provide escape mechanisms so that any character can be used in a name. This is flexible but unreadable. It supports ç in names, which is helpful, if you happen to be programming in French, but even there a lack of consistent support can cause more trouble than it is worth. In English, this generality appears to support spaces in names which again gives a superficial improvement in legibility at the expense of numerous side effect anomalies.

4.1.2 Reserved Words

Since Wdl should coexist with WdML and a graphics form, it is desirable to avoid reserving certain names. It is confusing to the user of a graphics environment if an input cannot be called `in`.

There are therefore no fully reserved words in Wdl. However it may be necessary to prohibit the use of subroutine like expression constructs such as `if` or `switch` as subroutine names.

The precise Wdl syntax is evolving on this point. In principle, it is possible to use `then` to disambiguate `if (expression) then` as a usage of the conditional keyword rather than a confusing subroutine call. However the syntax is then only just free from conflicts and requires undue amount of careful structuring.

4.1.3 Hierarchy

Hierarchical access to nested names is supported through use of the dot operator as in Java. Access to the root namespace is supported through use of a double dot prefix.

```
Package.Element
..Root.Package
```

Modern languages avoid over-use of a single communal name-space through classes, modules name-spaces or packages.

In C++ (and CORBA IDL) a hierarchical type name is specified through the use of `::` as a scope separator:

```
OuterModule::InnerName
```

The Java-style dot operator is more compact, readable and consistent. It also avoids the C++ syntax ambiguity for `A::B ::C`. C++ originally used dot and then changed, presumably to resolve the ambiguity created by making the `struct` keyword optional, thereby allowing a type and non-type to share the same name in the same name-space. The need to disambiguate these cases does not arise in WDL.

4.1.4 Namespaces

The visibility rules for names remain to be decided. It was originally envisaged that the full context of a (structurally and/or behaviourally) enclosing entity should be available in order to avoid passing excessive context. However this violated the integrity of state update. It appears that only constant definitions benefit from easy access; passing a bundle of configuration parameters for more dynamic values is not too onerous.

Declarations for importing names from a foreign namespace are undecided; the C++ namespace concepts may be a suitable starting point.

4.2 Numbers

The representation of numbers in modern languages is preserved in WDL, and extended to support

- dimensions
- ranges, distributions and optimization preferences
- array values

4.2.1 Decimal number

A decimal number follows conventional practice

```
1       1.0     +1e-57
```

More precisely, a decimal number is the longest sequence of characters satisfying the following lex [Levine92] grammar.

```
sign              [-+]
fraction          \.[0-9]*
exponent          [eE]{sign}?[0-9]+
simple_number     {sign}?[0-9]+{fraction}?{exponent}?
```

4.2.2 Complex number

Complex numbers are represented by the sum of the real and imaginary components, with the imaginary component identified by an i suffix. When necessary parentheses can be used to avoid confusion in an expression.

```
1+0.707i
```

Parsing conflicts are resolved by resolving the i suffix at the lexical level, revising the grammar to:

```
simple_number     {sign}?[0-9]+{fraction}?{exponent}?i?
```

4.2.3 Non-decimal number bases

Binary and hexadecimal numbers may be specified using a `0b`/`0B` or `0x`/`0X` prefix to the longest possible sequence of binary (`0` or `1`) or hex (`0` to `9`, a to `f`, A to `F`) characters.

```
bin_number          0[bB][0-1]+
hex_number          0[xX][0-9A-Fa-f]+
```

The C octal syntax is a nightmare, and visually unsatisfactory if replaced by 0o or 0O.

Special prefixes, such as `$` for hex and % or ^ for binary, use characters that may be needed elsewhere.

4.2.4 Dimensioned Number

Values may have an associated unit

```
10‘kHz              -5‘dB
```

More precisely, a dimensioned number is a number, name or parenthesised expression followed by a backquote and a scoped name.

> Backquote is the least offensive choice for a very high precedence operator. It would be possible to use the absence of whitespace to bind an identifier, but that conflicts with the widespread (standard) practice of separating dimensions by whitespace.

Dimensions or Units

Dimensions such as `kHz` or more fundamentally units such as `m` are not defined by WDL, although a library package could define the common SI units. Units and dimensions are user defined as part of the type system. The user may therefore define as many fundamental units as appropriate; not just the 6 physical ones.

```
dimension s;                    // New fundamental unit
dimension Time = s;             // Synonym
dimension hour = 3600 * s;
dimension Hz = 1 / s;
dimension kHz = 1000 * Hz;
```

Multiplication and division of dimensions may be used to create compound dimensions. Arithmetic of dimensioned numbers must be dimensionally consistent. Dimensioned numbers must be converted to a dimensionless form (by division by a unit dimensioned value) in order to be used by dimensionless operators such as `log`. Trigonometric operations such as `cos` are defined for angular values, for which `quadrant` is a built-in dimension against which radians or degrees may be defined.

> Angles are defined using quadrants rather than radians or degrees in order to choose a number range that may avoid the need for a floating point scaling in sensitive practical implementations.

Dimension names exist in a distinct (hierarchical) namespace used only when resolving a dimension name following a backquote. There is therefore no ambiguity that `100‘scope.s` is to be resolved as the dimension name `s` with the entity or type `scope`.

4.2.5 Constrained Number

A range of numbers may be specified:

```
range { minimum 50‘MHz; maximum 250‘MHz; }
```

4.2.5.1 Distributed Number

Some specification values, particularly those concerning subsystems, define probabilities, either as an outright specification on performance, or to define typical operating scenarios.

```
range { distribution normal; value 5‘dB; rms 0.1‘dB; }
range { distribution uniform; maximum 1‘s; confidence 0.99; }
```

Distributions such as `normal`, `poisson`, `rayleigh`, `rice` and `uniform` can be built-in. It should be possible to define a required behavioural interface to allow a user defined type to characterise a user defined distribution.

4.2.5.2 Optimized Number

Specifications may require or allow a parameter to take a range of values, either because operation throughout a range of values is required (from 10 MHz to 100 MHz) or because achievement of a specific value is unrealisable (less than 1 second). When an operational range is specified, all values are generally of equal import. When an operational range is tolerated, some values are generally better than others: "less than a second" often means get as close to zero as possible.

A WDL specification can capture this preference, in a way that offers an opportunity for a smart interpreter to optimize performance. Each value may be accompanied by an equation to support optimization. The value of the equation should be zero for an optimum setting, and nominally +1 or -1 for a marginally acceptable value.

An acquisition time with a quadratic preference for 0 may be specified by the 'number':

```
range
{
    distribution uniform;
    maximum 1's;
    confidence 0.99;
    let weight = this / maximum;     // temporary 'variable'
    optimize weight * weight;
}
```

An optimum system configuration may be identified as that with the minimum sum of the absolute values of its optimization criteria.

This syntax says nothing about how or if the optimum may be found. The system designer retains the option to bias the optimization by changing the weighting on some of the equations. Specification writers should be aware that an optimum is much more likely to be found if the optimization equations are linear.

4.2.6 Semantics

Dimensions must be consistent.

Only a single dimension may be specified.

Thus

```
(1.5`kHz)`MHz                // Bad multiple dimensions
range { minimum 10; maximum 100`MHz; }
                  // Bad - missing dimension
range { value 1.5`N; within 1`J; }
                             // Bad mix of SI units
range { minimum 1`foot; maximum 10`feet; }
    // Ok (if foot and feet defined from same fundamental unit)
range { value 1`inch; within 1`cm; }
    // Ok (if cm and inch defined from same fundamental unit)
```

Partial dimensions do not compose. Thus `(5`k)`Hz` is illegal.

If legal, `(5'm)'Hz`, could be 5 milli-Hz or 5 metre-Hz.

4.2.7 Compound Values

An array value may be specified using `[]` to delimit the elements of an array and () to group multiple elements of a dimension.

```
[ 0, 1, 2, 3 ]                   // 4 element vector
[(1,0),(0,1)]                    // Unit matrix
[((1,0),(0,0)),((0,0),(0,1))]    // Unit box
[[1,0],[0,1]]          // 2 element vector of 2 element vectors
[[0]]                  // 1 element vector of 1 element vector
[]                               // empty vector
[()]                             // empty matrix
[(())]                           // empty box
```

The C-style syntax using {} is only suitable for the unique context of C initializers. Wdl supports array constants within expressions and so must use punctuation that is not ambiguous for expression contexts. It is not practical to use {} as both number and statement grouping, if an expression is to be a legal statement.

Elements of a dimensioned compound value must be consistently dimensioned, either by

dimensioning each element:

```
[ 0‘Hz, 1‘kHz, 2‘MHz ]
```

or by dimensioning the compound value:

```
[ 0, 1000, 2000000 ]‘Hz
```

4.2.8 Constructed Numbers

A number may be created with an explicit type by construction from some other form

```
NewType(oldValue)
```

4.2.9 Queries

Can some missing dimensions be tolerated?

```
value 1‘kHz; within 1%;
```

is inconsistently dimensioned and must be specified as `value 1‘kHz; within 1%‘kHz;`.

5 Types

WDL types extend the capability of the type systems of other Object Oriented languages by adding

- ability to define mathematical behaviours
- multi-dimensional arrays
- discriminated unions

WDL types define a mathematical behaviour of unlimited precision. Practical specification types are defined by tailoring the abstract behaviour to match a target capability. Implementation types are chosen by a compilation system.

The abstract behavioural types in order of increasing seniority are:

```
boolean                 // false (0) or true (1, non-zero)
natural
integer
real
complex
```

A value of a more junior behaviour can be converted automatically to a more senior behaviour. The reverse conversion is never performed automatically. When behaviours have been tailored, the conversion of a junior type can only be converted to more senior types that exhibit compatible tailoring.

The **`void`** type may be used to define the absence of a value.

> The only IDL types supported directly in WDL are `boolean` and `void`. Types such as short and long are not provided, since in IDL they mandate specific precisions, which defines an implementation not a requirement. The one-word IDL types may be declared using WDL type declarations. Multi-word IDL types such as `long long` cannot be directly declared since all WDL type names are single word. Applications must therefore pick single word names as is done for `Corba::LongLong`.

5.1 User-defined type names

A user-defined type (such as `bool`) may be declared as:

```
type bool : boolean;
```

5.2 Simple Types

5.2.1 Tailored types

Practical types are defined by tailoring abstract or tailored types to specify their minimum or exact mathematical behaviour. A minimum specification preserves the implementation freedom to choose a more efficient machine-specific type with excess precision. An exact specification can enforce bit-true behaviour.

> The IDL built-in types are implementation types with defined precisions - suitable for specification of an implementation but too restrictive for specification of a specification.

The IDL built-in type `octet` could be specified as

```
type octet : natural
{
    bits 8;
```

```
        overflow none;
    };
```

Tailoring is performed with respect to the accuracy of number representation, the overflow and rounding behaviour. For integral values, the range of represented numbers may be shown using a number line.

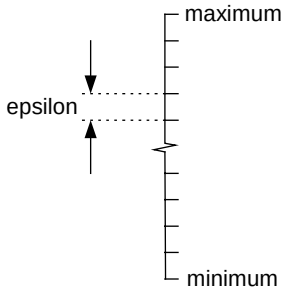


Figure 9 Number Line of Representable Values

maximum and *minimum* denote the inclusive extreme representable values, and *epsilon* the (maximum) separation between adjacent representable values. The number of representable values is (at least)

$$levels = 1 + \frac{(maximum - minimum)}{epsilon}$$

For floating point values, a distinct number line applies for each exponent value, *plus_huge* and *minus_huge* are the maximum and minimum values for the largest exponent value. *maximum*, *minimum* and *epsilon* are defined for the smallest exponent at which 1+*epsilon* is representable.

To avoid the need for redundant specifications the following defaults apply:

$$maximum = 2^{bits} - 1$$
$$minimum = 0$$
$$epsilon = 1$$
$$huge = default\_huge = \max(|maximum|, |minimum|)$$
$$plus\_huge = maximum \cdot \frac{|huge|}{default\_huge}$$
$$minus\_huge = minimum \cdot \frac{|huge|}{default\_huge}$$

`bits`

Specifies the number of bits in the representation of a bit-true natural number.

`epsilon`

Let α be the most positive non-zero representable value closest to 1 (normally 1).

Let β be the most positive representable value adjacent to α (normally 1+ε).

Then $\varepsilon = \left| \frac{\alpha - \beta}{\alpha} \right|$

`epsilon` defaults to `1` to suit integral specifications. A much smaller value such as `1.192092896e-07` might be specified for a `float` type.

`minimum, maximum`

The range of numbers representable while maintaining the `epsilon` resolution. Each value of $minimum + i * \varepsilon$ from `minimum` to `maximum` inclusive is representable for integer `i`.

`minimum` and `maximum` may often straddle zero, such as the range -32768 to +32767 for a `short`. It is permissible for both limits to have the same sign such as 1 to 31 for a day of the week.

`huge`

The largest (inclusive) representable non-infinite value. `huge` defaults to the larger of `maximum` and `minimum` to suit integral types. A much larger value such as `3.402823466e+38` might be specified for a `float` type.

`tiny`

The most positive non-zero representable value closest to zero. `tiny` defaults to `epsilon` to suit integral types. A much smaller value such as `1.175494351e-38` might be specified for a `float` type.

`+infinity`, `-infinity`, `nan`

The IEEE 754 (IEC 559) concepts of plus/minus infinity and not-a-number are (to be) supported.

`overflow`

Calculations performed with inadequate arithmetic precision may overflow. Behaviour under these conditions is ill defined in some computer languages, requiring the programmer to ensure that overflow does not occur.

A WDL specification can only leave overflow undefined, when such behaviour is guaranteed not to occur.

For DSP applications, it is common to exploit modulo overflow characteristics as part of the intended behaviour.

`none`

The conventional unspecified behaviour may be requested by specifying `overflow none`. This is an assertion that overflows do not occur and so an implementation may choose the most efficient form of arithmetic operation. If a compiler can detect a context in which an overflow contributes to an observable result, the compiler should report an error.

`infinity`

Overflows may be represented by the appropriate form of infinity by specifying `overflow infinity`.

`clip`

Overflows may be clipped (saturated) to the nearest representable (non-infinite) value by specifying `overflow clip`. Clipping occurs by selection of either *plus_huge* or *minus_huge*.

wrap

Overflows may wrap-around modulo (epsilon+plus_huge-minus_huge) by specifying `overflow wrap`. Wrap-around arithmetic is defined for integral and fixed-point types (those for which `huge` has the default value).

raise

An exception may be raised whenever an overflow occurs by specifying `overflow raise`.

The names and types and semantics of exceptions remain TBD.

round

Floating point calculations may require lower order result bits to be discarded. The selected rounding behaviour determines how this is performed.

convergent

larger

nearest

smaller

zero

5.2.2 Enumerations

A Wdl enumeration defines a numeric type with enumerators to name distinct values.

```
enumeration Color { RED, ORANGE, BLUE };
```

The representation of a value may be constrained by adding an assignment, to define an alias or to enforce an explicit value.

```
enumeration Constants
{
    DAYS_IN_WEEK = 7,
    PI = 3.14159,
    DAYS_PER_WEEK = DAYS_IN_WEEK
};
```

It is intended that explicit values should only be provided where external messaging format constraints define values. If internal values are left unspecified, a smart compiler may choose internal values that map conveniently to externally imposed constraints.

A repeated enumeration with distinct enumerators extends:

```
enumeration Color { YELLOW, GREEN };
```

A repeated enumeration with repeated enumerators constrains:

```
enumeration Color { ORANGE = RED + YELLOW };
```

The representation of unconstrained values is specified only in so far as requiring that each unspecified enumerator be distinct from all other enumerators in the same enumeration.

Sequential assignment starting at zero, or the previous value is not required.

This is an unfortunate incompatibility with C, but a presumption of sequential values has no place in a specification. It may be appropriate to provide a simple syntax such as =+ to

facilitate sequential specifications, however use of such a syntax would often occur where explicit values were necessary, and so explicit values rather than the weaker adjacency should be specified.

An implementation may choose any representation. Unspecified values are therefore an implied form of genetic value (an auto-generic) that must be resolved to locate a solution consistent with the specification and amenable to efficient implementation.

```
enumeration ProtocolA
{
    START,
    STOP,
    INITIALIZE
};

enumeration ProtocolB
{
    RESTART,
    START = ProtocolA.START,
    STOP = ProtocolA.STOP
};
```

Clearly RESTART and START must have distinct values, so the simplest implementation of RESTART =0 and START =0 is not legal; the compiler must choose the free values to avoid conflicts with constrained values.

5.3 Pointers and References

WDL has no pointers or references.

The nastier syntactical problems of C and C++ are thereby avoided.

There may be a need for references to have a transient existence to satisfy a generalisation of array arithmetic to l-values.

5.4 Arrays

Arrays in WDL have a number of generalisations compared to other more conventional languages

- arrays are multi-dimensional and homogeneous
  - uniform size for each dimension
- arrays of arrays are heterogeneous
  - each element may be a different shape
- scalars are zero dimensional arrays
- temporal arrays may distribute their elements over time
- multi-port arrays may distribute their elements over connections
- the ‘elements’ of an array may be continuous

An array is a self-describing object, and so attributes such as `_rank` and `_shape` may be accessed or constrained.

The implementation policy for an array is not specified, and so an implementation may use whatever representation is appropriate.

WDL avoids the indiscipline of C arrays and their pointer equivalence, WDL also avoids the degenerate C++ templates such as `sequence` found in IDL.

5.4.1 Multi-dimensional arrays

WDL supports arrays of arbitrary dimensionality and generalises computations to operate element-wise upon those arrays.

An array is declared by suffixing an optional dimension within `[]`. Multi-dimensional arrays may be specified by providing a list of scalar dimensions

```
... ElementType[10,5] ...
```

Alternatively, an explicit 10 by 5 dimension can be configured

```
constant VectorShape = [10,5] : natural[2];
```

and used to define the dimension indirectly

```
... ElementType[VectorShape]
```

The index of or size of each dimension is an index starting from zero: a natural. One or more dimensions may be left unspecified by using `*` as a placeholder. An arbitrary number of dimensions may be left unspecified by using `**`. Therefore:

```
MyType            // possibly scalar, but could be generic shape
MyType[]          // explicitly scalar
MyType[1]         // one element vector
MyType[1,1,1]     // one element box
MyType[*]         // vector of unspecified size
MyType[2,2]       // two by two matrix
MyType[**]        // array of unspecified dimensionality
MyType[*,*]       // matrix of unspecified sizes
MyType[*][2]      // vector of 2 vectors of unspecified size
```

A generic parameter may be used as a dimension in order to support shape polymorphism for any array dimensionality including scalars.

```
type MyMatrix : natural[2,2];
generic MyShape : shape;
type MyArray : MyMatrix[MyShape]; // Generic array of matrices
```

The **rank** of a datum is its dimensionality:

- 0 for a scalar
- 1 for a vector
- 2 for a matrix
- 3 for a box
- etc.

The **shape** of a datum is a vector of the sizes of each dimension.

- `[]` for a scalar
- `[ 4 ]` for a 4 element vector
- `[ 2, 3 ]` for a 2 by 3 element matrix

The shape of the shape of a datum is always a vector of its rank

```
attribute datum : AnyType[**];
```

The following constraint upon the built-n attributes is therefore satisfied.

```
constraint: datum._shape._shape = [ datum._rank ];
```

A multi-dimensional array may be indexed by a vector of the indexes

```
attribute vector : natural[datum._rank];
datum[vector]
```

rather than

```
//  datum[vector[0]][vector[1]]...[vector[N-1]]    -- C-like
```

```
// datum(vector(N), ..., vector(1))                    -- Fortran-like
```

In order to treat a scalar consistently as a zero dimensional array, the indexing operation for a scalar by the empty vector is defined to return the value of the scalar.

A raster scan of an array is a traversal of all valid indexes counting leftmost index fastest. Thus a 2 by 2 matrix is raster scanned as the index sequence `[0,0]`, `[1,0]`, `[0,1]`, `[1,1]`.

The definition of a valid index might be interpreted (or implemented) as a nested iteration over all index values less than their corresponding size. This is incorrect, since it overlooks the empty vector, which is a valid index for a scalar. An iteration over all indexes with no individual index greater than the array bounds gives the required behaviour.

The utility of expressing valid index vectors is recognised by introducing the infix `in` operator for use in constraint and specification expressions.

```
constraint: for all i in datum { out[i] = in[i]; };
```

is a specification shorthand and significant convenience for expressing that `i` is a vector of all valid multi-dimensional index vectors for `datum`, which is the empty vector if `datum` is scalar. Without the `in` operator it is necessary to write:

```
constraint: for all i : natural[datum.rank]
    where ({ for all k : natural
                where (k < datum.rank)
                    { x[k] < datum.shape[k]; }; })
        { out[i] = in[i]; };
```

5.4.2 Multi-port arrays

Many primitive computational entities can sensibly be generalised to handle multiple inputs. For instance, an adder is conventionally considered to be a 2-input entity, but is well defined for any natural number of inputs. The input ports of an adder may therefore be declared as a multiport array.

```
generic Shape : shape;
generic Type : type;
in[Shape] inputs : Type;
```

or making use of auto-generics:

```
in[**] inputs;
```

The `[]` on the `in` port keyword denotes a multiport array of `in` ports.

> A multiport array might be expected to be a vector, however a matrix of inputs could be appropriate for some image processing or beam forming applications. Medical or particle physics applications might even use a box of inputs. There is therefore no constraint upon the shape of a multiport array.

All elements of a multiport array share the same flow policy, and observe the normal constraints upon multiple flows. Token flows must rendezvous and so token flow is synchronised for each element. Event flows must not rendezvous and so concurrent events are resolved sequentially in raster scan order. Value flows are unconstrained.

A more complex type cannot be constructed from a multi-port array.

5.4.3 Temporal arrays

Although many DSP algorithms process a block of data, the sequential nature of most communication channels requires blocks to be sequenced in time. This sequencing may be restricted to a few serial to parallel converters, but time sequencing cannot be ignored completely. Treating the time sequence as an array reduces the need for special

notations to handle time sequences.

The input and output ports of a serial to parallel converter may be declared to comprise a temporal array of elements of some arbitrary shape and type as the serial input, and a conventional array of the same shape and type as the parallel output:

```
generic Shape : shape;
generic Type : type;
in input[Shape] : Type;
out output : Type[Shape];
```

or making use of auto-generics and built-in attributes:

```
in input[output._shape];
out output : input._type[**];
```

The `[]` on the (input) port name denotes a temporal array.

The `[]` on the (output) type name denotes a conventional array.

(A `[]` on a port direction keyword denotes a multi-port array)

A temporal array exists solely within the confines of the defining entity. External to the entity, the array elements are presented without any framing information in raster-scanned order. Type checking ensures that the use of `Type` is consistent for each element. There is no checking to ensure that consistent framing is applied. If consistent framing is required, the entire frame should be passed as a conventional rather than temporal array.

Time is one-dimensional, and so a temporal array might be expected to be a vector. However a temporal array exists to reduce a data set of arbitrary information dimensionality to zero dimensions of information and one dimension of time, and so there is no reason why matrices or boxes should not also be used; the transmission protocols for television pictures involve a two-dimensional temporal array.

Temporal arrays are only valid for signal and token flows. Conventional arrays of events violate the no rendezvous constraint. Temporal arrays of value flows would impose a scheduling policy on a policy-less flow.

The only more complex type that can be constructed from a temporal array is a multi-port array of temporal arrays.

5.4.4 Continuous arrays

Specification of signal processing functionality should avoid imposing either a discrete or continuous implementation, and so operations involving time (or space) sequences should be specified in a neutral fashion. Array operations are therefore generalised so that a continuous time (or space) record can be treated uniformly as an array indexed either by discrete or continuous time.

Some unifying concept such as _size or _duration or _period is needed to support conversion between continuous and discrete perspectives, and to make integration and differentiation consistent.

Continuous time representations naturally run from a start time for a given duration. The discrete equivalent comprises equally spaced samples throughout the same time interval, with the first sample at the start of the interval. Each sample is an instantaneous measure of amplitude. Care must be exercised in deriving subsequent discrete results for which the effect of a fractional sample may be significant.

> Alternative definitions of sample timing may seem attractive when considering higher order statistics such as energy, for which it might seem that a sample should represent the root

mean square energy surrounding it. However these definitions lack consistency. In the case of an energy measure, the problem lies in the difficulty of realising either a squaring or integral operation in discrete time. A squaring operation requires a doubling of the Nyquist rate and an integral requires the introduction of a half sample time shift.

5.4.5 Array precedence

With three distinct forms of array, a potential ambiguity arises when attempting to index a multi-port array of temporal arrays of conventional arrays. The ambiguity is resolved by a precedence definition.

The outermost (rightmost) index indexes the multi-port (if any).

The (next) outermost index indexes the temporal array (if any)

The remaining indexes index conventional arrays.

5.4.6 Flow arrays

Events cannot occur concurrently. An array of events therefore denotes a grouping of events. The array event carries a single event from the element source to the element destination.

Tokens must occur concurrently. An array of tokens is therefore the same as the equivalent token comprising the array of token values. A rendezvous is necessarily performed to create the composite array.

Continuous signal, values and constants are continuous. An array of continuous flows is the same as a single flow comprising the equivalent composite array. This is the same as for a token, since a continuous rendezvous is degenerate.

Is this distinct behaviour too pragmatic and potentially irregular and therefore anomalous for polymorphic flow modes?

5.4.7 Construction

The lack of assignment requires changed arrays to be constructed. This is not always as convenient as it might be. Is there something to be learned from List Processing and Functional languages here?

5.4.8 Strings

Strings support ASCII text operations. IDL requires string lengths to be specified. WDL should allow strings to be self-describing and so offer a simpler specification concept, that leaves an implementation to choose an appropriate realisation. Strings are not particularly important to low level radio applications, and so little thought has been given to the details.

IDL needs strings because IDL lacks an object context for types. Perhaps a string is just a one dimensional array of character, but should character be built-in? If not, then is

```
type Character : integer { minimum -128; maximum 127; };
type CharacterString : Character[*];
```

sufficient? and is there any point in the shorthand

```
string CharacterString : character;
```

It is certainly not clear that wstring is needed.

Perhaps the string shorthand just makes some built-in functions such as uppercase available that would clearly be inappropriate for a vector.

5.5 Product Types

5.5.1 Records

A WDL record defines a class with base-classes, attributes, operations, generics and constraints.

> A record is not actually called a class, since class is used by MoML to indicate simulation classes, and class has a much stronger and better defined semantics in languages such as C++.

In comparison with C++, a WDL record

- lacks private/protected/public distinctions. These are perceived to be an implementation consideration.
- lacks static to give a clear class/instance distinction, the class context can be provided as a generic value.
- lacks constructors and destructors.
- lacks conversion operators.
- has a much more general treatment of generics
- has constraints

```
record MyInterface : FirstBase, SecondBase
{
    generic ElementType : type;
    attribute x : int;
    operation f(in i : int, inout j : int, out k : int) : long;
};
```

A WDL record may be extended by respecification. Additional distinct base classes are appended. Additional distinct members are appended. Additional non-distinct members provide additional constraints on those members.

A record is what you might expect it to be, which leaves rather too much to be resolved, in particular a policy for construction and destruction.

5.5.2 Bundles or Buses

There are often many flows that exhibit similar connectivity in high level message flow specification diagrams, which can easily become cluttered when each flow is shown individually. A bundle (or bus) allows an ad hoc grouping for graphical convenience, without imposing any constraints upon their mutual behaviour. This contrasts with a `record` whose contents flow as part of the record.

The only operations that can be performed upon bundles are bundling, connection and unbundling. Since a bundle has no direction, there is no distinction between bundling and unbundling; a `Bundler` associates input and output ports with a bundle port.

A bundle is declared by:

```
bundle BundleName
{
    event reset : void;
    value valid = false : boolean;
    signal audio : Audio;
};
```

Bundles are normally connected to bidirectional ports with the `bundle` ‘flow’ type and the corresponding bundle type name.

```
inout:bundle inputBundle : BundleName;
```

A connection to an **in** or **out** port asserts the checkable constraint that elements of the bundle are used only as inputs or outputs respectively.

Elements of bundles are accessed using the dot operator in the same way as members of records.

```
inputBundle.reset
```

A bundle is just a notational convenience that is easily flattened. The only complexity lies in discovery of direction at each connection point. The `Bundler` at the ends of the bundle has explicit direction specifications that may be propagated along the flattened bundle.

5.5.3 Discriminated Unions

A discriminated union exhibits the behaviour of exactly one of a number of alternative types selected by a discriminator value, whose representation is unspecified.

There is no direct counterpart to the ill-behaved C `union`.

```
union Discriminated : DiscriminatorType
{
    case CaseX : int;
    case CaseY1, CaseY2 : float;
    default : bool;
};
```

The `DiscriminatorType` may be any integral or enumeration type. The default case may only be omitted when the list of cases is exhaustive.

A discriminated union may be used as any other type, save for the constraint that any value must have an unambiguous type. This may be enforced by using the indexed case as a constructor.

```
attribute example = Discriminated[CaseX](3) : Discriminated;
```

discriminator

The discriminator is a hidden attribute that may be accessed as the reserved `_discriminator` member.

```
example._discriminator
```

member selection

A particular member of a discriminated union may only be accessed after using the discriminate operator to create distinct contexts for relevant cases.

```
discriminate (example)   // example is Discriminated here
{
    case CaseX:
    {
        // example is int here
    };
    case CaseY1:
    {
        // example is float here
    };
    default:
    {
        // example is Discriminated here
    };
};
```

Within each named case the discriminated record is refined to support use with the selected type. Within the default case, the type remains unrefined.

The type of a particular member can be similarly accessed, without the risk of an exception.

```
type CaseY1Type : Discriminated.CaseY1;
```

The discrimination of a nested discriminated union may be resolved using square brackets to enclose the nested selection.

```
type D;
union U : DT
{
    case C1 : D;
    case C2 : D;
};
record A
{
    attribute a1 : U;
    attribute a2 : E;
};
attribute a = A(U[C1](0), E(0)) : A;
attribute u = U[C1](0) : U;
```

The nested member of a may be accessed as

```
a[a1.C1]
```

which yields a value of the type

```
A[a1.C1]
```

which is equivalent to the declaration

```
record some_name
{
    attribute a1 : D;
    attribute a2 : E;
};
```

It would be convenient to just use `a[C1]` to resolve the discrimination, but this leads to an ambiguity for `a.a1[C]`, which is the same as `a.a1.C` unless `[]` changes the meaning of preceding dot operators. `a[a1.C]` indexes with respect to the object whose external identity is preserved.

5.6 Bit-true types

Bit-true representations are necessary to comply with standard message change formats and is supported by

- tailored simple types (Section 5.2.1)
- explicit enumerator values (Section 5.2.2)
- bit fields (Section 5.6.1)
- overlays (Section 5.6.2)
- layouts (Section 5.6.3)

5.6.1 Bit-fields

Bit-fields are a detailed implementation issue, and so supported only within layouts and overlays. Bit-fields cannot be used for normal program attributes.

A bit-field is specified using a data type with an explicit bit precision and an optional shift

to introduce padding bits with respect to the previous declaration (in a layout) or the base type (in an overlay).

```
type nibble : natural { bits 4; };
attribute ls_nibble : nibble;
attribute ms_bit = 0 : boolean[1] << 3;
```

The notation `boolean[x]` is a shorthand for an anonymous natural type tailored to x bits.

Bit fields may be initialized.

5.6.2 Overlays

An overlay defines a group of bit-true declarations each of which share the same starting bit position. This may be used to access bits. When initializers are provided, overlays can be used as part of the conversion from abstract WDL types to bit-true external representations, and can provide the constraints needed to distinguish between alternate types in a bit or byte array.

```
type Octet : natural { bits 8; };
overlay BitsAndNibbles : Octet
{
    attribute ls_bit : boolean[1];                  // Bit 0
    attribute mid_nibble : boolean[4] << 2;         // Bit 2,3,4,5
    attribute ms_bit : boolean[1] << 7;             // Bit 7
};
```

The overlay type `BitsAndNibbles` is defined to be a more detailed view of the `Octet` base type. The resultant type is therefore an overlay of the newly defined fields and all fields inherited from all ‘base’ types. Three distinct bit fields are identified named by size and bit offset. The size or offset are constrained so that all bits lie within the size of the base type which must be a scalar or one dimensional array of a bit-true type.

Further declarations may fill in gaps or add value constraints.

```
overlay BitsAndNibbles : Octet
{
    attribute bit1 : boolean[1] << 1;
    attribute bit6 = 0 : boolean[1] << 6;
};
```

Members of an overlay may overlap provided no two initialization conflicts. The following is therefore legal:

```
overlay BitsAndNibbles : Octet
{
    attribute ms_nibble = 8 : boolean[4] << 4;
};
```

but the following conflicts:

```
overlay BitsAndNibbles : Octet
{
    attribute ms_bits = 3 : boolean[2] << 6;
};
```

> Since overlays are intended to enforce external requirements, all sizes positions and initializers should be constant expressions; it should not be necessary to deduce values that satisfy constraints.

5.6.3 Layouts

A layout type defines a record with a bit true layout and the mapping of an abstract data type to that layout. The attributes within a layout are positioned bit sequentially, starting at bit 0.

```
type Octet : natural { bits 8; };
type Word : natural { bits 16; };
layout BigEndianWord = Word : Octet[2]
{
    attribute big = Word >> 8 : Octet;
    attribute little = Word & 0xFF : Octet;
};
```

The types `Octet` and `Word` are defined to support the numeric ranges of a byte and a word. Then `BigEndianWord` is defined as a layout of `Word` using an array of 2 `Octets` as its bit true representation. The layout of `BigEndianWord` is defined as a sequence of two `Octets`, the first of which derives its value from the top 8 bits of the `Word`, and the second from the bottom 8 bits.

The layout can be data dependent making use of a discriminator to select alternate layouts:

```
layout DataDependentLayout = AbstractType : Octet[2]
{
    discriminate (DataDependentLayout)
    {
        case Layout1:
        {
            ...
        };
        case Layout2:
        {
            ...
        };
        default:
        {
            ...
        };
    };
};
```

Bit-true encoding

A layout type may be used to create a bit-true representation of an abstract WDL representation.

```
attribute aWord : Word;
let bigEndianWord = BigEndianWord(aWord);
```

The attributes of a bit-true layout can be manipulated in the same way as the attributes of a record. Alternatively the bit true representation can be used in a context where its underlying bit-true representation (`Octet[2]`) is appropriate.

Bit-true decoding

Layout types may be used in conjunction with a `layin` construct to verify the consistency of an instance of the underlying type with its potential layout.

```
type Ascii : natural { bits 7; };
type Ebcdic : natural { bits 7; };
layout AsciiByte = Ascii : Octet
{
    attribute data = Ascii : boolean[7];
    attribute flag = 0 : boolean[1];
};
layout EbcdicByte = Ebcdic : Octet
{
    attribute data = Ebcdic : boolean[7];
```

```
        attribute flag = 1 : boolean[1];
    };
```

Two 7 bit types are defined for ASCII or EBCDIC character encoding. Two 8 bit `Octet` layouts are defined using the eighth bit to flag ASCII or EBCDIC usage. The following example entity dispatches an incoming byte to either an ASCII or EBCDIC output.

```
entity CharacterDispatcher
{
    in aByte : Octet;
    out asciiByte : Ascii;
    out ebcdicByte : Ebcdic;
    response aByte
    {
        specification
        {
            layin(aByte)
            {
                case AsciiByte:
                {
                    asciiByte(aByte.data);
                };
                case EbcdicByte:
                {
                    ebcdicByte(aByte.data);
                };
            };
        };
    };
};
```

The layin construct performs a compatible type search to promote the `Octet` to either an `AsciiByte` or an `EbcdicByte`, with action code responding accordingly.

The layin construct moves the definition of an untidy bit test to a type declaration, and provides type safety for the conversion of a bit-true `Octet` to an `Ascii` or `Ebcdic` datum.

The case coverage must be exhaustive. A case is only selected if the data is compatible.

> In order to avoid exhaustive checks of inner-field validity while doing outer field discrimination it may be necessary to define intermediate layout types that retain the raw bit representation for fields that are not to be validated.

5.7 Exceptions

IDL defines a full policy for declaring and handling exceptions.

The WDL syntax provides an IDL-like mechanism for declaring exceptions, but using a WDL-consistent syntax.

There is no definition of the semantics of WDL exceptions.

This is a significant TBD in WDL at present. The CFP for an exception handling workshop at ECOOP'2000 suggests that there is a lot wrong with existing systems. Maybe they'll have some solutions too.

Exceptions may be raised automatically when

- arithmetic overflows for a type tailored with `overflow raise`.
- an event rendezvous encounters a token deficiency

5.8 Anatomy

A type has the following attributes, which may be accessed directly for a type or indirectly with respect to the `_type` attribute of any datum.

The meta-level names are all prefixed by _ in the following discussion. Is this really necessary? If the _ is omitted, as may be the case in examples elsewhere in this and other documents, is there a potential clash between user and reserved names?

`_Object`

`_type : _Type`

`_Type : _Object`

`_flow : _FlowType`

The flow type.

`_is_modulo : boolean`

`_nyquist : _Frequency`

The suggested Nyquist sampling rate.

`_port_shape : natural[*]`

The multi-port array shape, a vector of extents.

`_rank : natural`

The array rank (0 for scalar).

```
x._rank = x._shape._shape[0]
```

`_shape : natural[*]`

The array shape, a vector of extents.

`_time_shape : natural[*]`

The temporal array shape, a vector of extents.

`_type :`

?.

`_FlowType : _Type`

`_is_constant : boolean`

`_is_continuous : boolean`

`_is_discrete : boolean`

`_is_event : boolean`

`_is_present : boolean`

Identifies whether a source (token) is available for a rendezvous. This provides the underlying language support for polling and scheduling.

_is_signal : boolean

_is_token : boolean

_is_value : boolean

_BehaviourType : _Type

_is_boolean : boolean

_is_complex : boolean

_is_integer : boolean

_is_natural : boolean

_is_real : boolean

_Entity

_ports

Vector of all ports.

_ins

Vector of all input ports.

_outs

Vector of all output ports.

_inouts

Vector of all input/output ports.

etc

etc

6 Constructs

6.1 Operators

6.1.1 Assignment

There is no assignment operator.

The `:=` operator specifies end of context state update.

The = operator in an expression is an equality test.

The = operator in a `constraint` is an equality assertion.

The = in a `let` or defining a default is a syntactic separator.

6.1.2 Arithmetic

The arithmetic operators are similar to C and Java.

Unary: `+, -, ~, !`

Binary: `+, -, &, |, ^, *, /, %, <<, >>`

>> yields the same result as division by an appropriate power of 2.

Each defines a type preserving operation resolved by

```
operation add();
```

etc of the type.

`/` returns a type at least as general as `real`. Use `idiv` for integer divide.

Define rounding, overflows and %.

6.1.3 Short-circuit operators

The `&&, ||` operators perform short-circuit evaluation as in C or Java, so that right hand terms rendered obsolete by left-hand evaluations are not considered. The purpose of this is to suppress use of invalid terms, not to suppress side effects; a side-effect cannot occur as part of an expression; there are no increment/decrement operators or assignments.

6.1.4 Relational

`=, !=, <, >, <=, >=`

Each specifies a `boolean` result.

With respect to C or Java, == is spelled =.

6.1.5 Precedence

Operator precedence is C-like:

Parentheses may be used to enforce a distinct evaluation order.

6.2 Expressions

6.2.1 call, construct, send

The conventional function call syntax

```
name ( comma-separated-arguments )
```

has three meanings in WDL.

If the name is a function name, the construct is a function call. If the types of the function arguments are all the same shaped array of the argument types that the function expects, then the function is invoked element wise for each argument to return a result of the same shape as the arguments.

Is there an ambiguity if functions are polymorphic in parameter rank/dimensionality?

If the name is a port name, the construct is a message send, typically to an output port, but potentially of a return flow to an input port.

If the name is a type name, the construct is a type constructor, for which the arguments align to the declared attributes.

WDL does not support user-defined constructors. Is this a problem?

Overload resolution rules.

6.2.2 if

An `if` always has braces, so there is no dangling `else` ambiguity.

```
if (cond) then {};
if (cond) then {} else {};
if (cond1) then {} else if (cond2) then {} else {};
```

Data parallel if.

`if` is not a reserved word, so the `then` is required to avoid ambiguity with a call.

6.2.3 switch

There is no drop-through, and each case needs braces. Multiple cases may be grouped:

```
switch (cond)
{
    case A,B,C : {};
    case D : {};
    default : {};
};
```

Data parallel switch.

6.2.4 discriminate

Discriminate is the same as switch, except that the argument is a discriminated union, and within each case, the type of the argument is flattened to comprise only the discriminated case.

A similar form of discrimination is used to configure declaration rather than execution alternatives in layout and overlay types.

Example.

Data parallel discriminate - if possible.

6.2.5 layin

A lay-in supports controlled type discovery, converting from a relatively ill-defined type to one of many more precisely defined type. The bit pattern of the argument is successively analysed against each the data type of each case, to locate the first case for which the argument is a valid instance.

```
attribute data : octet[*];
layin(data)
{
    case MessageType1: { ... };
    case MessageType2: { ... };
    case MessageType3: { ... };
    default: { ... };
};
```

The source is a self describing array of octets, which is expected to be one of three possible message types, each of which should be a layout type that defines a bit-true representation.

> If the bit-true layouts can be easily distinguished by examining just a few bits or bytes, the implementation of layin can be efficient. If not, a costly search may be necessary.

The layin ensures that a case body is only executed when `data` satisfies the associated type. The data type of `data` is therefore the more precise type within the case body. No conversion of data type occurs for the default case.

6.2.6 seq

A sequential ordering can be imposed on expressions by the use of a seq construct.

> This does not prevent a compiler performing parallel evaluation if it can be shown to give the same externally observable behaviour.

6.2.7 par

A `par` expression encompasses multiple expressions that may be evaluated in parallel. A par expression has no net value for use by its invoker.

```
par
{
    expression1;
    expression2;
};
```

The concurrent generation of events is illegal and detectable.

6.2.8 let

A `let` statement caches an expression in a temporary variable, to facilitate multiple usage of the expression without repeated specification.

Use of `let` does not imply evaluation of the expression. Thus let may be used to structure constraints without inhibiting the bidirectional process of deduction.

Is this consistent or just potentially confusing?

6.2.9 collect

A collect expression performs an iteration to collect objects that satisfy a selection criteria, primarily for use in constraint specifications.

collect i **in** set **where** (*constraint*) { *body*; }

performs a data parallel iteration over all possible values of i within set, for which the constraint is satisfied and then places the value of body into the returned vector, which is ordered in accordance with a raster scan of set.

6.2.10 reduce

A reduction applies an operator along one or more array dimensions. Thus

```
reduce ElementType.xor for all i in datum { datum[i]; }
```

returns the exclusive or appropriate to `ElementType` over all elements of datum. This syntax should have an easier way of defining which array dimensions are preserved and which are reduced.

6.3 Constructs

6.3.1 Attributes

A data field in a type, which may be a record, discriminated union, layout, or overlay type is declared by the attribute `construct`.. A state variable in an entity or in the response of an entity is similarly declared.

```
attribute name = initial_value : type;
```

The initial value is required for a state variable, but may be omitted for data types.

6.3.2 Operations

A operation of a type, which may be a record, discriminated union, layout, or overlay type is declared by the `operation` construct.. A operation of an entity is similarly declared.

```
operation name(in arg_name = default_arg : arg_type) : type
    { body };
```

The operation arguments are specified as a comma-separated list of direction, name, optional default value and type tuples. The return type (which may be `void`) is followed by an optional specification of the behaviour.

Operations do not have side effects.

What does this mean for entities? an entity operation may only be invoked from within the entity, and so update of entity state is permitted.

What does this mean for types? a type operation may only be invoked for the type, and so update of type attributes is permitted. But what about child state? child state must be private to the parent type to allow update; shared child state would inconsistently prohibit update or non-deterministically permit update.

6.4 Constraints

A constraint declaration defines an expression that must always be true in any valid implementation. Violation of a constraint should be detected by a compilation system.

> In practice, some constraints may be impossible or impractical to check automatically. Other constraints may require special analysis tools. WDL provides a mechanism for all constraints to be specified in a machine-readable form. It is not expected that all constraints will be checked. A compiler may diagnose those constraints it cannot check, and possibly provide some selective suppression to avoid repeated diagnosis of uncheckable constraints such as case colour.

Some constraints are imposed automatically by WDL syntax

- resolution of generics participates in type, shape and flow checking
- causality imposes lower bounds on calculation delays
- latency constraints impose bounds on timing uncertainties
- inertia constraints impose upper bounds on calculation delays
- runaway constraints impose upper bounds on premature calculation
- rendezvous and token flow require consistent scheduling

Explicit constraints may

- impose requirements on a compliant implementation
- support type, shape and flow deduction
- define behaviour
- expose limitations of a candidate implementation

Non-functional specifications must generally be specified by constraints:

```
constraint MINIMUM_CONCURRENT_CHANNELS = 3;
```

Shape deduction can be assisted by:

```
constraint OutputType.rank = InputType.rank;
```

The behaviour of a sort routine can be defined as a constraint on the properties that the output data set exhibits with respect to the input, without specifying how the mapping is implemented.

An automated code generator may select between optimized implementations with restricted functionality if each announces its limitations:

```
constraint FFT_SIZE = 256;
```

OCL has been incorporated as part of the UML specification to add formality to the otherwise ad hoc textual annotations. OCL has a very traditional type system but adopts a very OO approach to its operations. WDL constraints have many similarities, but use the more general WDL type system. WDL constraints are OO but use a functional approach for quantifiers. Neither approach seems entirely satisfactory: A dot operator tacked onto a braced expression looks bad in WDL. The use of arrow to operate on rather than through, and dot to operate through rather than on an OCL collection is surprising.

6.4.1 Constraint Expressions

Many constraints can be expressed by simple comparisons. More complicated constraints require the use of the existential and universal quantifiers ∃ and ∀, which are not easily represented in ASCII and not particularly familiar to all programmers. Wdl therefore uses the textual forms: `for all`, `for no`, `for some` and `for one`. `for one` is a variant of `for some` asserting that there is only one solution. This assertion is to be checked before exploitation.

We may therefore specify a sort by:

```
entity sort
{
    generic DataType : type;
    generic Shape : shape = natural[1];
    in inputs : DataType[Shape];
    out outputs : DataType[Shape];
    // No output value is greater than its predecessor.
    constraint in_order : for no j in Shape where (j < Shape[0]-1)
    {
```

```
        outputs[j] > outputs[j+1];
    };
    // Output is a permutation of the input.
    constraint is_permutation : for all j in Shape
    {
        let matchingInputs = collect i in Shape
            where (!(inputs[i] > outputs[j])
                 & !(outputs[j] > inputs[i]))
                { i; };
        let matchingOutputs = collect i in Shape
            where (!(outputs[i] > outputs[j])
                 & !(outputs[j] > outputs[i]))
                { i; };
        matchingInputs.shape = matchingOutputs.shape;
    };
};
```

The negation in the `is_permutation` constraint avoids a constraint violation for repeated values, and the need to distinguish between equality with respect to the sort criterion and equality with respect to object identity.

The generic constraint that input and output arrays have the same shape ensures that there are the same number of inputs and outputs, which when combined with the matched frequency of occurrence ensures that each input appears in the output and vice-versa and consequently the output is a permutation of the input.

The stronger constraint for a stable sort can be established by removing the pair of `.shape` suffixes to require the `matchingInputs` and `matchingOutputs` collections to be identical rather than same sized.

7 Hierarchy

A WDL specification may be composed from a number of (sub-)specifications. Each specification defines an entity, that may interact with other entities through the messages passed through the ports of the entity.

The term entity is used rather than component (or class or object) to avoid confusion with the CORBA component model.

WDL re-uses the UML statechart concepts almost without change. It is worth clarifying the distinction between WDL message flow diagrams and other forms of UML diagram to explain what is different about WDL.

The UML building block is an object (or class) which has a number of attributes and operations. The degree of encapsulation of the resulting state depends on design practices. Synchronisation between (active) objects relies on concurrency annotations. Communication occurs through invocation of an operation with a multiplicity of secondary arguments.

The WDL building block is an entity, whose attributes and operations are totally encapsulated. All entities are inherently concurrent; the properties of token flow must be exploited by a compiler in order to reduce the concurrency to a practical level. External interaction occurs through responses, each of which performs a rendezvous of some subset of its input ports before executing its update atomically. Communication may therefore involve multiple connections.

UML has no ports, concurrent flows or token flows and rather accidental synchronisation. WDL supports ports, strict rendezvous and message flows. Message flows may be continuous or discrete. Discrete flows may be event based for sequential determinacy, or token-based to expose parallelism.

WDL statecharts, message flow diagrams and leaf specifications are sufficient to specify a system. Additional UML diagrams may be useful to improve the clarity of exposition.

7.1 Interfaces

The externally visible interface of an entity is defined by

- (auto-) generics
- ports
- types
- constraints

The internally visible interface may be augmented by

- attributes
- operations
- response specifications
- response implementations

The internally visible interface is visible within the entity and within derived entities.

7.1.1 Generics

WDL supports specification using polymorphic behaviour with respect to

- entity
- flow

- type
- shape
- value
- enumerator

A particular polymorphic behaviour may be specified using a generic. An explicit specification in, or a deduction from, the instantiation context may resolve the actual behaviour.

```
generic DataType : type;          // Generic type declaration
generic ArrayShape : shape;       // Generic shape declaration
generic Size : value;             // Generic value declaration
```

A default value (such as 256) and a minimum behavioural requirement (such as `natural`) may be specified:

```
generic Size = 256 : value = natural;
```

Scheduling policies and message flow policies are inherently generic and so there is no need for an explicit specification.

7.1.2 Ports

All interaction between entities occurs as messages flow through their ports.

Ports must be specified to have

- a name
- a direction: `in`, `out`, `inout`

Ports may be specified to have

- a flow type: unspecified, `event`, `signal`, `token`, `value` or ,`bundle`
- a (message) type (and shape)
- a default value

Type

Omitted flow or message type information are resolved by auto-generics. Thus the port declaration

```
in in;
```

is equivalent to

```
generic InFlow : flow;
generic InType : type;
in: InFlow in : InType;
```

where *`InFlow`* and *`InType`* denote automatically generated unique names, which may be referenced using the `_flow` and `_type` attributes of the port. *`InType`* is generic with respect to both element data type and array shape.

Thus an entity with matching input and output behaviour can be declared overtly as:

```
generic FlowType : flow;
generic DataType : type;
in:FlowType in : DataType;
out:FlowType out : DataType;
```

or opaquely as:

```
in in;
out out;
constraint: in._flow = out._flow;
```

```
constraint: in._type = out._type;
```

Return types

The pair of types associated with an event and its return may be specified using a pair of types.

```
in:event port : in_type : return_type;
```

or by using a flow type

```
type FlowType : in_type : return_type;
in:event port : FlowType;
```

Default value

A default value may be specified for a (non-`event`) input to allow the input to be left unconnected or unconfigured and to specify the value for use at that input. Providing a reset value that defaults to 0 could configure the behaviour of a reset input.

```
in:value reset_value = 0;
```

An expression may be specified for a `value` output in order to export some function of the internal state and `value` inputs.

```
attribute internal_state : StateVariable;
out:value state = internal_state : StateVariable;
```

It is an error if propagation of `value` inputs to `value` outputs establishes a loop.

> In C++ terms, this is an inline function. Whenever a consumer requires the output value, the expression is evaluated. Strategies for efficient evaluation of complicated expressions are left to be resolved by implementations. The events at which values change can be identified enabling an eager or lazy policy to be implemented.

Internal Ports

A port may declared internal to support communication within an entity. This typically occurs within statecharts where an internal event may be generated by the action of one concurrent state machine and responded to by another. Alternatively a state machine may need access to a result computed by a nested message flow.

7.1.3 Usage of generics

Conventional DSP programming makes extensive use of built-in types. This occurs through severe limitations in the available programming tools and consequently the need to use highly implementation specific data types.

In a typical DSP block diagram, knowledge of the data type on one arc enables the data type of connected entities and consequently of further arcs to be deduced. Conventional programming systems do not support this deduction and so mandate explicit specification of all types hampering any attempt to change types.

WDL generics support this process of deduction, and most of the primitive entities defined for the library are generic. The port connections of a `Subtractor` can be specified as

```
generic DataType : type;
in in1 : DataType;
in in2 : DataType;
out out : DataType;
```

introducing `DataType` as a generic type (and shape) to be shared by the inputs and output. Once the type used by one has been resolved, the others are automatically constrained to correspond.

Shape and type may be independently generic as is the case for a `Serializer`:

```
generic DataType : type;
generic Shape : shape;
in:token in : DataType[Shape];
out:token out[Shape] : DataType;
```

This supports bidirectional resolution of the `DataType`, but only forward resolution of the `Shape` (backward resolution is not possible since the shape of the generated temporal array is not externally visible). The actual resolution may occur by deduction, or by explicit specification of the implementation:

```
entity serializer1 : Serializer
{
    constraint : DataType = natural;
    constraint : Shape = [ 2, 2 ];
};
```

A few appropriately placed explicit specifications can be used to resolve generics throughout a very large design.

Generics may be specified for types or values; `shape` is just a short form for a vector of `natural[*]`.

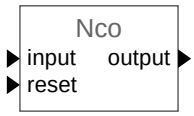
The constraints implied automatically by the shared use of a generic name can resolve many programming problems. More complicated constraints must make use of constraint statements.

7.2 Message Flows

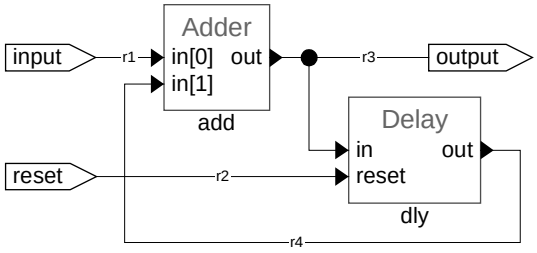
A message flow diagram provides a level of hierarchical decomposition in which the external perspective as an entity is decomposed through interconnection of a number of internal entities, each of which may a hierarchical message flow, statechart or leaf implementation.

> The graphical style used in this document is not intended to be definitive. It is merely the result of a rapid configuration of a flexible but ultimately inappropriate drawing package. A more suitable editor would be configured with symbols that clearly denote functionality.

A numerically controlled oscillator might have the external interface



and internal decomposition



A message flow diagram may have the following properties

- multiple input ports
- multiple output ports
- multiple bidirectional ports (very rare)
- internal state

asynchronism between some or all ports

Internal state precludes presentation as pure functions.

The existence of multiple non-concurrent outputs precludes a simple presentation as a functional transformation of inputs to an output, although a generalisation to a set of outputs could be conceived.

The asynchronism between ports precludes presentation as a single function.

The functional part of the decomposition is represented in Wdl as

```
entity nco
{
    in input;
    in reset;
    out output;
    entity add : Adder;
    entity dly : Delay;
    relation r1 { link add.in[0]; link input; };
    relation r2 { link reset; link dly.reset; };
    relation r3 { link add.out; link output; link dly.in; };
    relation r4 { link dly.out; link add.in[1]; };
};
```

The omitted graphical layout attributes may be maintained through use of `location` and `rendition` constructs on the `link`, `in`, `out`, `entity` or `relation` constructs.

The decomposition makes extensive use of auto-generics; the flow and data type declarations are missing from all ports. Resolution of generics during compilation will propagate the constraints from the internal entities so that the reset input is defined by the limited capability of the `Delay` input. The `Adder` constrains input and output to a compatible type that can be deduced from a connection to either.

An additional declaration is necessary to ensure that the output wraps around rather than ramps indefinitely.

```
constraint output._type._is_modulo;
```

7.3 Statecharts

State machines provide an alternate form of hierarchical decomposition of a specification. WDL combines the arbitrary hierarchy semantics of [Girault97] with the statechart notation of UML [Rumbaugh99].

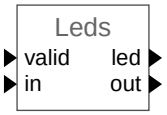
Some of the more obscure UML facilities such as history states and deferred events do not appear to be necessary. These are therefore left unspecified pending an adequate justification.

A single extension is made to support the drawing of a message flow diagram within a state to define the behaviour of that state.

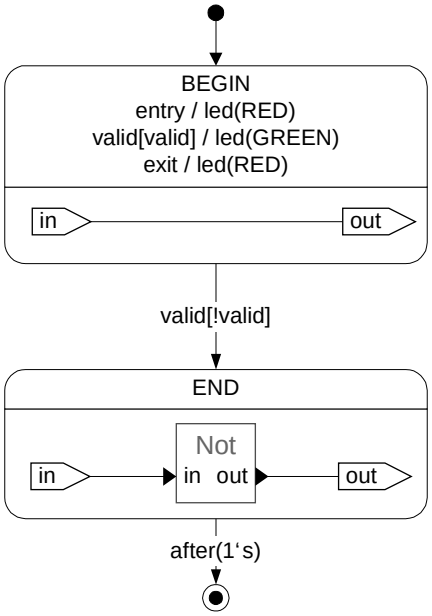
The extension supports specification of the `do` action by a message flow, and provides a tidy solution to the problem of associating outer and inner names. The complexity of the nested message flows is a matter of style. Excessive detail can lead to diagram clutter, but overzealous insistence on the use of separate message flows can lead to an unnecessary number of diagrams and a loss of clarity.

It is very unfortunate that this nesting results in the two different hierarchical semantics coexisting on the same diagram, with the result that a line may indicate either a message flow or a state transition depending on its context. The practice arose completely by accident after recognising that, since the style was in regular use "on the back of an envelope", it might as well be permissible.

The external interface of a state machine is indistinguishable from the external interface of a hierarchical message flow or leaf implementation, so the interface:



could encapsulate the state chart:



which may be represented together with missing declarations in Wdl as:

```
enumeration States { BEGIN, END };
enumeration Colors { RED, GREEN };
statechart Example : States
{
    generic SignalType : type;
    in:event valid : boolean;
    out:event led : Colors;
    in:token in : SignalType;
    out:token out : SignalType;
    initial state
    {
        goto BEGIN;
    };
    state BEGIN
    {
        transition entry
        {
            led(RED);
        };
        internal transition valid guard(!valid)
        {
            led(GREEN);
        };
        transition valid guard(valid)
        {
            goto END;
        };
        transition exit
```

```
            {
                led(RED);
            };
            relation r1 { link in; link out; };
        };
        state END
        {
            after(1's)
            {
                goto exit;
            };
            entity not : Not;
            relation r1 { link in; link not.in; };
            relation r2 { link not.out; link out; };
        };
    };
```

The graphical context is omitted from the above Wdl but may be maintained through use of `location` and `rendition` constructs on the `link`, `in`, `out`, `transition`, `state`, `entity` or `relation` constructs.

7.3.1 Semantics

UML defines a relatively clear semantics for state transitions in response to events. WDL event semantics is consistent with this, and is elaborated to define completion of token computation before exiting the old state and after entry to the new state.

While processing an event as much token flow or signal flow computation as possible is completed. In particular, the specified processing order while processing a state transition is

- complete token and signal computation in the old state
- perform state exit actions
- change state and perform transition actions
- perform state entry actions
- complete token and signal computation in the new state

A statechart fits into the WDL entity model through the treatment of input event ports as statechart events. Output event ports can be activated by action routines. Value input ports can be exploited by transition actions and guards. Token inputs and outputs cannot be used directly by a statechart, but may be connected through to the specification of one or more states.

7.3.2 States

Each `state` defines the behaviour for a particular state.

In order to support equivalence to a graphical representation, a number of degenerate state-like contexts defined by UML are treated as states in the textual representation:

Initial State

The initial state may be designated by an `initial` qualifier prefixing the `state` keyword. The state need not be named and should comprise only default transitions.

Final State

The final state may be designated by a `final` qualifier prefixing the `state` keyword. The state should not have any transitions.

Junction states

The transient junctions states defined by UML may be defined by preceding the `state` keyword by `junction`, and introducing a subsequent name that is not part of the enumeration of machine states. This policy correctly reflects the UML semantics that a state machine is never in a junction state, since the junction state is just a modularization aid.

History states

History states may be identified by a `history` or `deep history` qualifier.

It is not clear that history states are needed in WDL.

Fork and Join states

The UML fork and join synchronizations may be captured by `states` with `fork` or `join` qualifiers. They should comprise only default transitions.

It is not clear that fork and join states are needed in WDL.

Branch 'states'

It is convenient to represent branch nodes as transient states similar to junction states, since this provides a suitable context for capturing graphical properties.

7.3.3 Transitions

Each transition whether as part of the overall state or a sub-state is represented by a function, named by the event and parameterised by the message type. The function is followed by an optional `if` tree of guard conditions and consequent actions. The above example shows just simple guards.

Multiple declarations of the same operation name may occur for different states, but must have the same parameter list.

Transitions always involve a single argument corresponding to the message type, or no argument in the case of a `void` message type.

Internal transitions

An internal transition (one which stays in the same state) may be indicated by an `internal` qualifier to the transition keyword. There should be no `goto` action.

Default events

The default transition (one without any associated event) may be specified by omitting the event:

```
transition { goto target1; };
```

Entry events

The action for the entry event may be specified as the transition for the reserved `entry` event.

```
transition entry(parameter) { actions; };
```

If an entry transition has no parameter, entering transitions may be of any type.

If an entry transition has a parameter, all entering transitions must be compatible with the parameter type.

Exit events

The action for the exit event may be specified as the transition for the reserved `exit` event.

```
transition exit() { actions; };
```

After events

The after event may be specified using the reserved event name `after` and supplying a number with dimensions of time.

```
transition after(time) { goto target1; };
```

When events

There is no support in WDL for when/change events.

There is no fundamental reason why they should not be. When a suitably motivating example is provided, the lack of support can be reviewed.

Branches

The more complex form of UML branch may be represented by:

```
transition event()
{
    if (guard1()) { do action1(); goto target1; };
    else if (guard2()) { do action2(); goto target2; };
    else { do action3(); goto target3; };
};
```

Actions

The action may comprise executable expressions and at most one `goto`. A `goto` the `entry` state signifies an exit from the state machine, execution of the action and then re-entry at the `entry` state. A `goto` the `exit` state signifies execution of the action and exit from the state machine.

Deferred events

Deferred events can be resolved by user application code implementing `defer()` as an action routine. If this is sufficiently common, it could be implemented as part of a run-time support library or even as a recognised language idiom. For the time being deferred events are application problems.

Deferred events can be modeled by token flows.

Call events and signals

UML draws a distinction between (synchronous) signals, which are objects that carry (some of) their own parameters internally, and (asynchronous) call-events that must be separately parameterised. Both forms have the same graphical representation, although the signal event tends to be named as the object, so that an event without parameters actually has an implicit `this` parameter.

WDL interaction is by event or token flow, each of which carries its parameters as part of a message object. These are similar to UML signals. WDL supports asynchronous interaction with a nested message flow, but does not allow external access to the internal operations.

Hierarchy

The UML `do()` event is not supported. Nested message flow diagrams are used instead.

7.3.4 Queries

Can statecharts have inheritance?

What does it really mean for a statechart to exit? How does a message get sent to the parent? Maybe it's the responsibility of the user exit code? Maybe a reserved done message is generated, in which case we need a scoped naming to enable a parent to tell which of many child state machines exited?

Are the UML semantics of occluding context for event handling sensible/honoured?

Is this notation extensible to all UML statechart syntax?

- synch states

7.4 Leaf behaviour

7.4.1 Specification

The behaviour of each response may be specified using procedural statements or constraints. The procedural statements can avoid imposing sequential constraints by using data parallelism or statement parallelism. For many simple leaves there is no difference between a procedural statement and a constraint.

For instance, a combinatorial entity such as a `Subtractor` may be specified as

```
entity Subtractor
{
    in minuend;
    in subtrahend;
    out difference;
    specification
    {
        response minuend subtrahend
        {
            difference(minuend - subtrahend);
        };
    };
};
```

or

```
entity Subtractor
{
    in minuend;
    in subtrahend;
    out difference;
    constraint: difference = minuend - subtrahend;
};
```

The response specifies a rendezvous between `minuend` and `subtrahend` inputs, supporting subsequent access to the rendezvoused variables by name. Sending a message to the `difference` output propagates the result. The specification is polymorphic with respect to

- almost any mix of flow types
  - a pair of event inputs is an illegal concurrency violation

  - a mix of value and signal inputs is illegal, unless the value satisfies the Nyquist criteria of the signal.
- data element type
  - any type for which subtraction is defined may be used
- array shape
  - operation is performed element-wise for same shaped arrays

How is the polymorphic output flow type constrained? By the a priori combination of input flow types followed by a safe a posteriori conversion.

What is the name of the subtraction operator?

An entity with state such as a Counter may be specified as:

```
entity Counter
{
    generic CounterType : type;
    in:event increment;
    in:event reset;
    out:value value = _value : CounterType;
    attribute _value = 0 : CounterType;
    specification
    {
        response reset
        {
            value := 0;
        };
        response increment
        {
            value := value + 1;
        };
    };
};
```

The counter state is maintained by the (internal) attribute `_value`, and exported as the `value` output. Each of the `reset` and `increment` responses specifies an event rendezvous. A token rendezvous is illegal since there are multiple state-updating responses.

A state machine is a combination of leaf behaviour and hierarchical message flow behaviour. Each combination of state and event input has an associated response, but only the responses for the prevailing state are enabled. When an event occurs, the relevant response is activated and the state variable updated (at the end of the event). Update of the state variable may trigger further computation of value flows, but cannot activate token or event flows.

The response rendezvous computation model applies to all flows.

7.4.2 Implementation

7.5 Derived Behaviour

The behaviour of an entity may be refined by derivation.

```
entity DerivedEntity : BaseEntity { ... };
```

What does this really mean?

- can extra ports be added?
  - if so what does it mean to substitute such a derived entity

- can attributes/constraints/responses/specifications/implementations be added/extended/replaced/removed?
- should there be a C++ style level of protected access?
- can there be multiple inheritance?

Presumably generics should be characterised by the need for a compliant type (GJ style) rather than any old type (C++ style).

8 Facilities

8.1 Remodelling

WDL supports both specification and implementation through progressive refinement from an abstract specification to a detailed implementation from which code may be generated.

Some stages of the refinement may be automated, but it is very unlikely that all system design skills can ever be eliminated completely.

Code generation should be fully automatic, provided the refinement has introduced sufficient detail. The degree of refinement required will depend on the quality of the available code generator tools and extent of suitable implementation libraries.

A WDL specification comprises one or more files to be processed and a specific ordering for that processing.

A refined WDL specification comprises the same (unedited) files processed in the same order, followed by further files. The additional files may contain

- extra decomposition or detail
- recomposition
- optimized implementations

Extra information restricts the implementation alternatives so that after sufficient refinement an automated code generator is able to implement one. Extra information is supplied using unchanged WDL syntax, although the `refine` keyword prefix may be used to make the programming intent clearer and error diagnosis more helpful.

```
constant MAX_USERS : natural;    // Declares constant but no value

constant MAX_USERS = 10;         // Associates value with constant
refine constant MAX_USERS = 10;  // More explicit intention
```

Recomposition can be used to discard part of a specification, possibly to exploit a pre-existing component. Existing parts of the specification are eliminated by using `remove link`, `remove entity` or `remove relation`. Thereafter additional components may be connected using unchanged Wdl syntax.

A WDL specification provided by a sponsor may include a Java reference implementation. It is unlikely that an appropriate implementation will be provided for the target hardware, and it is unlikely that the Java implementation will be suitable for production usage. Optimized implementations may be added and the required implementations for each entity selected.

An extensive set of rules are required to define how multiple declarations are to be validated.

8.2 Bit-truth

There are two aspects to bit-truth. Bit-true computation and bit-true representation. Bit-true computation must occur throughout an implementation and so must form part of the internal as well as external specification. Bit-true representation is only necessary at external interfaces, and need not pervade the implementation, although an implementation may choose to use an externally valid representation internally to avoid translation costs.

Computation

Bit-true computation is enforced by the specification of the overflow and underflow corrections that are to be made after each idealised mathematical operation is performed. Specification of tailored types supports this requirement.

Representation

Bit-true data structures are supported through alternate forms of type declaration. Tailored types support bit true values, whose position can be offset in a bit-field. Layouts support multiple fields, and overlays support alternate perspectives. Initialization values and expressions for fields provide for automatic conversion of a type with unspecified layout to a specific bit-true layout, and in conjunction with the layin operator supports type discovery for the reverse conversion.

9 WDL Grammar

WDL may be maintained using three distinct but equivalent representations

- the Wdl textual language - like C or Ada or IDL
- The WdML (Wdl mark-up) language - an XML [XML98] dialect
- graphical language - like UML, or DSP block diagrams

The textual form is favoured in this language definition, since the textual form is most accessible to human readers. The graphical form is more appropriate for structuring large specifications and so only a few examples appear in this definition. The XML form is expected to be the basis for interchange and long-term storage in a 'database', since the use of XML enables a variety of language-neutral tools to be used.

Equivalence

Equivalence between textual and mark-up forms is assisted by the use of a regular syntactic structure for constructs in the textual language.

```
construct:     keyword-phrase arguments ;
    |          keyword-phrase arguments { child-constructs } ;
```

A *keyword-phrase* comprises an optional sequence of secondary non-reserved words terminated by a primary non-reserved word. The syntax of the arguments is determined by the keyword-phrase. *child-construct*s repeat the same syntax, with the keyword-phrase restricting the *child-construct*s to a subset of all possible constructs. Thus in:

```
internal transition test { ... };
```

`internal` is a secondary non-reserved word. `transition` is a primary non-reserved word. `internal transition` is the keyword-phrase. `test` is the syntax dependent argument.

It is anticipated that this should translate to an XML construct using the primary non-reserved word as the XML tag.

```
<!ELEMENT primary-keyword (child-constructs)*>
<!ATTLIST primary-keyword arguments secondary-keywords>

<transition internal name="test">
    <child-constructs>
<\transition>
```

> If a standard XML style sheet [XSL97] cannot perform this, then a significant revision may be necessary.

Name and Type order

There are two prevalent syntaxes for declaring entities:

`C, C++, Java, (CORBA) IDL:`

```
    TypeName object_name ;
```

Algol, Pascal, Miranda, ... (the precise introducer and punctuation is language-dependent)

```
    introducer object_name punctuation TypeName ;
```

The former approach is the result of what many regard as a failed experiment, whereby C declared variables in the style of their usage, creating many ambiguities as a result. In

particular, the mix of prefix (*) and suffix([]) type constructors causes considerable difficulties for machine parsers and human readers.

The latter approach is technically far superior.

Maximising commonality with standard approaches, suggests that the former approach is desirable. When tamed by the more restrictive context of IDL, the more unpleasant problems that appear in C++ can be avoided.

IDL is a language neutral interface specification with standard bindings to the major languages. It is therefore a good starting point for defining a more substantial specification language, and IDL is maintained as a subset of WDL, then short term implementations can be realised by restricting designs to the subset.

It is possible to adopt the IDL convention, but it does not map trivially to XML, and so the Algol approach has been adopted for WDL.

9.1 Lexical Analysis

The lexing requirements of the Wdl grammar are relatively simple and easily implemented using lex.

9.1.1 Reserved Words

The regular structure of the Wdl grammar avoids the need for most reserved words. However a few constructs can look like expressions of types and so the following words are partially reserved. (They cannot be used as operation names).

```
"discriminate"                { LEX_STATIC_TOKEN(DISCRIMINATE); }
"if"                          { LEX_STATIC_TOKEN(IF); }
"layin"                       { LEX_STATIC_TOKEN(LAYIN); }
"switch"                      { LEX_STATIC_TOKEN(SWITCH); }
```

9.1.2 Non-Reserved Words

```
"after"                       { LEX_STATIC_TOKEN(AFTER); }
"all"                         { LEX_STATIC_TOKEN(ALL); }
"attribute"                   { LEX_STATIC_TOKEN(ATTRIBUTE); }
"bits"                        { LEX_STATIC_TOKEN(BITS); }
"branch"                      { LEX_STATIC_TOKEN(BRANCH); }
"bundle"                      { LEX_STATIC_TOKEN(BUNDLE); }
"case"                        { LEX_STATIC_TOKEN(CASE); }
"class"                       { LEX_STATIC_TOKEN(CLASS); }
"clip"                        { LEX_STATIC_TOKEN(CLIP); }
"collect"                     { LEX_STATIC_TOKEN(COLLECT); }
"confidence"                  { LEX_STATIC_TOKEN(CONFIDENCE); }
"configure"                   { LEX_STATIC_TOKEN(CONFIGURE); }
"constant"                    { LEX_STATIC_TOKEN(CONSTANT); }
"constraint"                  { LEX_STATIC_TOKEN(CONSTRAINT); }
"convergent"                  { LEX_STATIC_TOKEN(CONVERGENT); }
"deep"                        { LEX_STATIC_TOKEN(DEEP); }
"default"                     { LEX_STATIC_TOKEN(DEFAULT); }
"dimension"                   { LEX_STATIC_TOKEN(DIMENSION); }
"director"                    { LEX_STATIC_TOKEN(DIRECTOR); }
"distribution"                { LEX_STATIC_TOKEN(DISTRIBUTION); }
"do"                          { LEX_STATIC_TOKEN(DO); }
"doc"                         { LEX_STATIC_TOKEN(DOC); }
"else"                        { LEX_STATIC_TOKEN(ELSE); }
"entity"                      { LEX_STATIC_TOKEN(ENTITY); }
"enumeration"                 { LEX_STATIC_TOKEN(ENUMERATION); }
"epsilon"                     { LEX_STATIC_TOKEN(EPSILON); }
"event"                       { LEX_STATIC_TOKEN(EVENT); }
"exception"                   { LEX_STATIC_TOKEN(EXCEPTION); }
"final"                       { LEX_STATIC_TOKEN(FINAL); }
"for"                         { LEX_STATIC_TOKEN(FOR); }
"fork"                        { LEX_STATIC_TOKEN(FORK); }
"generic"                     { LEX_STATIC_TOKEN(GENERIC); }
"goto"                        { LEX_STATIC_TOKEN(GOTO); }
"guard"                       { LEX_STATIC_TOKEN(GUARD); }
"history"                     { LEX_STATIC_TOKEN(HISTORY); }
"huge"                        { LEX_STATIC_TOKEN(HUGE); }
"implementation"              { LEX_STATIC_TOKEN(IMPLEMENTATION); }
```

```
"import"                { LEX_STATIC_TOKEN(IMPORT); }
"in"                    { LEX_STATIC_TOKEN(IN); }
"infinity"              { LEX_STATIC_TOKEN(INFINITY); }
"initial"               { LEX_STATIC_TOKEN(INIT_IAL); }
"inout"                 { LEX_STATIC_TOKEN(INOUT); }
"interface"             { LEX_STATIC_TOKEN(INTERFACE); }
"internal"              { LEX_STATIC_TOKEN(INTERNAL); }
"join"                  { LEX_STATIC_TOKEN(JOIN); }
"junction"              { LEX_STATIC_TOKEN(JUNCTION); }
"language"              { LEX_STATIC_TOKEN(LANGUAGE); }
"larger"                { LEX_STATIC_TOKEN(LARGER); }
"layout"                { LEX_STATIC_TOKEN(LAYOUT); }
"let"                   { LEX_STATIC_TOKEN(LET); }
"link"                  { LEX_STATIC_TOKEN(LINK); }
"location"              { LEX_STATIC_TOKEN(LOCATION); }
"maximum"               { LEX_STATIC_TOKEN(MAXIMUM); }
"minimum"               { LEX_STATIC_TOKEN(MINIMUM); }
"model"                 { LEX_STATIC_TOKEN(MODEL); }
"nan"                   { LEX_STATIC_TOKEN(NAN); }
"nearest"               { LEX_STATIC_TOKEN(NEAREST); }
"no"                    { LEX_STATIC_TOKEN(NO); }
"none"                  { LEX_STATIC_TOKEN(NONE); }
"one"                   { LEX_STATIC_TOKEN(ONE); }
"oneway"                { LEX_STATIC_TOKEN(ONEWAY); }
"operation"             { LEX_STATIC_TOKEN(OPERATION); }
"optimize"              { LEX_STATIC_TOKEN(OPTIMIZE); }
"out"                   { LEX_STATIC_TOKEN(OUT); }
"overflow"              { LEX_STATIC_TOKEN(OVERFLOW); }
"overlay"               { LEX_STATIC_TOKEN(OVERLAY); }
"par"                   { LEX_STATIC_TOKEN(PAR); }
"path"                  { LEX_STATIC_TOKEN(PATH); }
"property"              { LEX_STATIC_TOKEN(PROPERTY); }
"raise"                 { LEX_STATIC_TOKEN(RAISE); }
"raises"                { LEX_STATIC_TOKEN(RAISES); }
"range"                 { LEX_STATIC_TOKEN(RANGE); }
"readonly"              { LEX_STATIC_TOKEN(READONLY); }
"record"                { LEX_STATIC_TOKEN(RECORD); }
"reduce"                { LEX_STATIC_TOKEN(REDUCE); }
"refine"                { LEX_STATIC_TOKEN(REFINE); }
"relation"              { LEX_STATIC_TOKEN(RELATION); }
"remove"                { LEX_STATIC_TOKEN(REMOVE); }
"rendition"             { LEX_STATIC_TOKEN(RENDITION); }
"replace"               { LEX_STATIC_TOKEN(REPLACE); }
"response"              { LEX_STATIC_TOKEN(RESPONSE); }
"rms"                   { LEX_STATIC_TOKEN(RMS); }
"round"                 { LEX_STATIC_TOKEN(ROUND); }
"seq"                   { LEX_STATIC_TOKEN(SEQ); }
"sigma"                 { LEX_STATIC_TOKEN(SIGMA); }
"signal"                { LEX_STATIC_TOKEN(SIGNAL); }
"smaller"               { LEX_STATIC_TOKEN(SMALLER); }
"some"                  { LEX_STATIC_TOKEN(SOME); }
"specification"         { LEX_STATIC_TOKEN(SPECIFICATION); }
"state"                 { LEX_STATIC_TOKEN(STATE); }
"statechart"            { LEX_STATIC_TOKEN(STATECHART); }
"string"                { LEX_STATIC_TOKEN(STRING); }
"then"                  { LEX_STATIC_TOKEN(THEN); }
"tiny"                  { LEX_STATIC_TOKEN(TINY); }
"token"                 { LEX_STATIC_TOKEN(TOKEN); }
"transition"            { LEX_STATIC_TOKEN(TRANSITION); }
"type"                  { LEX_STATIC_TOKEN(TYPE); }
"union"                 { LEX_STATIC_TOKEN(UNION); }
"value"                 { LEX_STATIC_TOKEN(VALUE); }
"vertex"                { LEX_STATIC_TOKEN(VERTEX); }
"where"                 { LEX_STATIC_TOKEN(WHERE); }
"within"                { LEX_STATIC_TOKEN(WITHIN); }
"wrap"                  { LEX_STATIC_TOKEN(WRAP); }
"wstring"               { LEX_STATIC_TOKEN(WSTRING); }
"zero"                  { LEX_STATIC_TOKEN(ZERO); }
```

9.1.3 Special character sequences

```
".."                    { LEX_STATIC_TOKEN(DOT_DOT); }
":="                    { LEX_STATIC_TOKEN(COLON_EQ); }
"<<"                    { LEX_STATIC_TOKEN(SHL); }
">>"                    { LEX_STATIC_TOKEN(SHR); }
"!="                    { LEX_STATIC_TOKEN(NE); }
"<="                    { LEX_STATIC_TOKEN(LE); }
">="                    { LEX_STATIC_TOKEN(GE); }
"&&"                    { LEX_STATIC_TOKEN(AND_AND); }
"||"                    { LEX_STATIC_TOKEN(OR_OR); }
```

9.1.4 Numbers

The C preprocessor rule for recognition of a possible number is used.

```
digit                           [0-9]
pp_number                       (\.?{digit}({digit}|{non_digit}|[eE][-+]|\.)*)

{pp_number}                     { LEX_NUMBER_TOKEN(yytext, yyleng); }
```

9.1.5 Identifiers

The C preprocessor rules for recognition of an identifier are used, complete with all the universal number complexity.

```
digit                           [0-9]
hex                             [0-9A-Fa-f]
letter                          [A-Z_a-z]
simple_escape_sequence          (\\\'|\\\"|\\\?|\\\\|\\a|\\b|\\f|\\n|\\r|\\t|\\v)
octal_escape_sequence           (\\[0-7]|\\[0-7][0-7]|\\[0-7][0-7][0-7])
hexadecimal_escape_sequence     (\\x{hex}+)
escape_sequence                 ({simple_escape_sequence}
                               | {octal_escape_sequence}
                               | {hexadecimal_escape_sequence})
universal_character_name        (\\u{hex}{hex}{hex}{hex}
                               | \\U{hex}{hex}{hex}{hex}{hex}{hex}{hex}{hex}{hex})
non_digit                       ({letter}|{universal_character_name})
identifier                      ({non_digit}({non_digit}|{digit})*)

{identifier}                    { LEX_IDENTIFIER_TOKEN(yytext, yyleng); }
```

9.1.6 TextLiteral

In order to embed code from other languages we need to be able to accept a large block of text without interpretation, and minimise the risk and difficulty that accrues from unfortunate character sequences in the embedded text. XML achieves this using a CDATA construct that starts with <![CDATA[and ends with]]>. This is fine for a mark-up language but fairly unacceptable for manual editing. I suggest recognition of {{{ as a starter and }}} as a terminator, with the option of {@{ as starter and }@} as terminator for embedded text that might contain }}}. In the foregoing @ represents any of a range of TBD punctuation characters. Such a sequence is recognised as a single lexical token and passed to the grammar as a TextLiteral.

9.2 Syntactic Analysis

The Wdl grammar is presented in its yacc [Levine92] form, since this is the form in which it is currently maintained and used to syntax check some of the Wdl in associated documents. The grammar has no conflicts and reserves only four words with respect to operation names. It may eventually be transliterated into the BNF style used in CORBA IDL specifications.

The yacc style involves a left-hand side production followed by a colon followed by the terms of the right-hand side rule. A bar separates multiple right-hand side rules.

```
production:       rule1
    |             rule2
```

A lower case name such as `specification` denotes a non-terminal.

An upper case name such as `STATE` denotes a terminal, which is usually the lower-case text string such as `state`. The `INITIAL` keyword is spelled `INIT_IAL` to avoid a `#define` clash.

A single quoted character such as `';'` is the equivalent terminal character.

A mixed case name such as `Identifier` or `IntegerLiteral` denotes a parametric

terminal.

A few productions are commented out with //. These are variously

- IDL concepts that have not been/do not need to be incorporated
- experimental syntax

```
grammar:                definitions                                     {}
definitions:            definition
    |                   definitions definition
definition:             constant ';'
    |                   entity ';'
    |                   exception ';'
    |                   interface ';'
    |                   model ';'
    |                   statechart ';'
    |                   type ';'
    |                   error ';'
```

9.2.1 Names

```
edit_id:                REFINE | REMOVE | REPLACE
most_ids:               Identifier
    |                   edit_id
    |                   AFTER | ALL | ATTRIBUTE | BITS | BRANCH | BUNDLE
    |                   CASE | CLASS | CLIP | COLLECT | CONFIDENCE | CONFIGURE
    |                   CONSTANT | CONSTRAINT | CONVERGENT | DEEP
    |                   DEFAULT | DIMENSION | DIRECTOR | DISTRIBUTION | DO | DOC
    |                   ELSE | ENTITY | ENUMERATION | EPSILON | EVENT | EXCEPTION
    |                   EXIT | FINAL | FOR | FORK
    |                   GENERIC | GOTO | GUARD | HISTORY | HUGE
    |                   IMPLEMENTATION | IMPORT | IN | INFINITY | INIT_IAL | INOUT
    |                   INTERFACE | INTERNAL | JOIN | JUNCTION
    |                   LANGUAGE | LARGER | LAYOUT | LET | LINK | LOCATION
    |                   MAXIMUM | MINIMUM | MODEL | NAN | NEAREST | NO | NONE
    |                   ONE | ONEWAY | OPERATION | OPTIMIZE | OUT | OVERFLOW |
OVERLAY
    |                   PAR | PATH | PROPERTY
    |                   RAISE | RAISES | RANGE | READONLY | RECORD
    |                   REDUCE | RELATION | RENDITION | RESPONSE | RMS | ROUND
    |                   SEQ | SIGMA | SIGNAL | SMALLER | SOME | SPECIFICATION
    |                   STATE | STATECHART | THEN | TINY | TOKEN | TRANSITION |
TYPE
    |                   UNION | VALUE | VERTEX
    |                   WHERE | WITHIN | WRAP | ZERO
    |                   IF            // IF(x) looks like subroutine call
//  |                   STRING | WSTRING
awkward_ids:            DISCRIMINATE // DISCRIMINATE(x) looks like subroutine call
    |                   LAYIN         // LAYIN(x) looks like subroutine call
    |                   SWITCH        // SWITCH(x) looks like subroutine call
identifier:             most_ids
    |                   awkward_ids
identifiers:            identifier
    |                   identifiers identifier

scoped_name:            most_ids
    |                   DOT_DOT identifier
    |                   scoped_name '.' identifier
    |                   awkward_ids '.' identifier
scoped_name_list:       scoped_name
    |                   scoped_name_list ',' scoped_name

base_name:              scoped_name
class_name:             scoped_name
dimension_name:         scoped_name
director_name:          scoped_name
location_name:          StringLiteral
source_name:            scoped_name
type_name:              scoped_name
```

9.2.2 Numbers

```
literal:                IntegerLiteral
    |                   StringLiteral
    |                   WideStringLiteral
    |                   CharacterLiteral
    |                   WideCharacterLiteral
    |                   FloatingPtLiteral
```

```
number:                 scoped_name
    |                   literal
    |                   '(' expression ')'
    |                   '(' expression ')' '.' scoped_name
dimensioned_number:     number
    |                   number ''' dimension_name
distributed_number:     dimensioned_number
    |                   RANGE '{' distributed_attrs '}'
distributed_attrs:       /* epsilon */
    |                   distributed_attrs distributed_attr
distributed_attr:       VALUE dimensioned_number ';'
    |                   CONFIDENCE expression ';'
    |                   DISTRIBUTION expression ';'
    |                   doc ';'
    |                   let ';'
    |                   MAXIMUM expression ';'
    |                   MINIMUM expression ';'
    |                   OPTIMIZE expression ';'
    |                   RMS expression ';'
    |                   SIGMA expression ';'
    |                   WITHIN expression ';'
    |                   error ';'
```

9.2.3 Expressions

```
primary_expr:           distributed_number
    |                   '[' expression_list ']'
    |                   primary_expr '(' expressions ')'
    |                   primary_expr '[' expression ']'
    |                   primary_expr '(' expressions ')' '.' scoped_name
    |                   primary_expr '[' expression ']' '.' scoped_name
    |                   collect_expr
    |                   collect_expr '.' scoped_name
    |                   reduce_expr
    |                   PAR '{' statements '}'
    |                   SEQ '{' statements '}'
    |                   DISCRIMINATE '(' expression ')' '{' switch_cases '}'
    |                   SWITCH '(' expression ')' '{' switch_cases '}'
secondary_expr:         primary_expr
    |                   if_exprs
    |                   if_exprs ELSE '{' statements '}'
//if_expr:                IF '(' expression ')' '{' statements '}'
if_expr:                primary_expr '(' expressions ')' THEN '{' statements '}'
if_exprs:               if_expr
    |                   if_exprs ELSE if_expr
primary_expr2:          secondary_expr
    |                   awkward_ids
unary_expr:             primary_expr2
    |                   '-' primary_expr2
    |                   '+' primary_expr2
    |                   '~' primary_expr2
    |                   '!' primary_expr2
mult_expr:              unary_expr
    |                   mult_expr '*' unary_expr
    |                   mult_expr '/' unary_expr
    |                   mult_expr '%' unary_expr
add_expr:               mult_expr
    |                   add_expr '+' mult_expr
    |                   add_expr '-' mult_expr
shift_expr:             add_expr
    |                   shift_expr SHR add_expr
    |                   shift_expr SHL add_expr
rel_expr:               shift_expr
    |                   rel_expr '<' shift_expr
    |                   rel_expr '>' shift_expr
    |                   rel_expr LE shift_expr
    |                   rel_expr GE shift_expr
eq_expr:                rel_expr
    |                   eq_expr '=' rel_expr
    |                   eq_expr NE rel_expr
and_expr:               eq_expr
    |                   and_expr '&' eq_expr
xor_expr:               and_expr
    |                   xor_expr '^' and_expr
or_expr:                xor_expr
    |                   or_expr '|' xor_expr
log_and_expr:           or_expr
    |                   log_and_expr AND_AND or_expr
log_or_expr:            log_and_expr
    |                   log_or_expr OR_OR log_and_expr
```

```
const_exp:              log_or_expr
expression:             const_exp
expressions:            /* epsilon */
    |                   expression_list
expression_list:        expression
    |                   expression_list ',' expression

where_expr:             expression
    |                   for_constraint
    |                   '{' constraint_exprs '}'
constraint_exprs:       /* epsilon */
    |                   constraint_exprs constraint_expr
constraint_expr:        where_expr ';'
    |                   let ';'
    |                   error ';'
for_key:                FOR ALL
    |                   FOR NO
    |                   FOR ONE
    |                   FOR SOME
for_constraint:         for_key identifier for_type for_where
                            '{' constraint_exprs '}'
for_type:               ':' declared_type
    |                   IN declared_type
for_where:              /* empty */
    |                   WHERE '(' where_expr ')'
collect_expr:           COLLECT identifier for_type for_where
                            '{' constraint_exprs '}'
reduce_expr:            REDUCE scoped_name '{' expression ';' '}'
    |                   REDUCE scoped_name FOR ALL identifier for_type for_where
                            '{' expression ';' '}'
```

9.2.4 Statements

```
statements:             /* epsilon */
    |                   statements statement
statement:              doc ';'
//  |                   EXIT ';'
    |                   expression ';'
    |                   expression COLON_EQ expression ';'
    |                   layin_statement
    |                   let ';'
    |                   operation ';'
    |                   quantified_stmt ';'
    |                   response ';'
    |                   type ';'
    |                   sub_statement
//    |                   PAR '{' statements '}'
//    |                   SEQ '{' statements '}'
//    |                   DISCRIMINATE expression '{' switch_cases '}' ';'
//    |                   SWITCH expression '{' switch_cases '}' ';'
    |                   error ';'
sub_statement:          ';'
    |                   '{' statements '}' ';'
//    |                   IF '(' expression ')' '{' statements '}' ';'
//    |                   IF '(' expression ')' '{' statements '}' ELSE
sub_statement
quantified_stmt:        for_key identifier for_type for_where '{' statements '}'
switch_cases:           switch_case
    |                   switch_cases switch_case
switch_case:            case_label ':' '{' statements '}' ';'

layin_statement:        LAYIN '(' expression ')' '{' layin_statement_cases '}' ';'
layin_statement_cases:  /* epsilon */
    |                   layin_statement_cases layin_statement_case
layin_statement_case:   case_label ':' '{' specification_attrs '}' ';'
```

9.2.5 Types

```
type:                   TYPE identifier ':' type_type
    |                   TYPE identifier ':' declared_type ':' declared_type
    |                   TYPE identifier ':' type_name '{' tailored_attrs '}'
    |                   TYPE identifier '{' tailored_attrs '}'
    |                   bundle_type
    |                   enum_type
    |                   layout_type
    |                   overlay_type
    |                   record_type
    |                   union_type
    |                   BUNDLE identifier
    |                   ENUMERATION identifier
```

```
    |                   LAYOUT identifier
    |                   OVERLAY identifier
    |                   RECORD identifier
    |                   TYPE identifier
    |                   UNION identifier
integral_type:          type_name
    |                   enum_type
simple_type:            type_name
    |                   string_type
    |                   wide_string_type
indexed_type:           simple_type array_shape
    |                   indexed_type array_shape
    |                   indexed_type '.' identifier
undimensioned_type:     simple_type
    |                   indexed_type
declared_type:          undimensioned_type
    |                   declared_type '*' undimensioned_type
    |                   declared_type '/' undimensioned_type
type_type:              declared_type
    |                   record_type
    |                   union_type
    |                   enum_type

array_shape:            '[' ']'
    |                   '[' '*' '*' ']'
    |                   '[' array_extents ']'
array_extents:          array_extent
    |                   array_extents ',' array_extent
array_extent:           '*'
    |                   expression

bundle_type:            BUNDLE identifier '{' bundle_members '}'
bundle_members:         /* epsilon */
    |                   bundle_members bundle_member ';'
bundle_member:          bundle_key port_decl
    |                   bundle_key port_decl '{' bundle_attrs '}'
bundle_key:             BUNDLE
    |                   CONSTANT
    |                   EVENT
    |                   EVENT array_shape
    |                   SIGNAL
    |                   TOKEN
    |                   VALUE
bundle_attrs:           /* epsilon */
    |                   bundle_attrs bundle_attr
bundle_attr:            configure ';'
    |                   doc ';'
    |                   property ';'
    |                   error ';'

enum_type:              ENUMERATION identifier '{' enumerators '}'
enumerators:            enumerator
    |                   enumerators ',' enumerator
enumerator:             identifier
    |                   identifier '=' const_exp

exception:              EXCEPTION identifier '{' record_members '}'

layout_type:            LAYOUT identifier '=' expression ':' declared_type
                            '{' layout_members '}'
layout_members:         /* epsilon */
    |                   layout_members layout_member
layout_member:          ATTRIBUTE identifier '=' expression ':' declared_type ';'
    |                   ATTRIBUTE identifier '=' expression ':' declared_type
                            SHL expression ';'
    |                   constraint ';'
    |                   DISCRIMINATE '(' expression ')' '{' layout_cases '}' ';'
    |                   error ';'
layout_cases:           layout_case
    |                   layout_cases layout_case
layout_case:            case_label ':' layout_member
    |                   case_label ':' '{' layout_members '}' ';'

overlay_type:           OVERLAY identifier record_bases '{' overlay_members '}'
overlay_members:        /* epsilon */
    |                   overlay_members overlay_member
overlay_member:         ATTRIBUTE identifier':' declared_type ';'
    |                   ATTRIBUTE identifier':' declared_type
                            SHL expression ';'
    |                   ATTRIBUTE identifier '=' expression ':' declared_type ';'
```

```
    |                   ATTRIBUTE identifier '=' expression ':' declared_type
                            SHL expression ';'
    |                   constraint ';'
    |                   DISCRIMINATE '(' expression ')' '{' layout_cases '}' ';'
    |                   error ';'

record_type:            RECORD identifier record_bases '{' record_members '}'
record_bases:           /* epsilon */
    |                   ':' scoped_name_list
record_members:         /* epsilon */
    |                   record_members record_member
record_member:          attribute ';'
    |                   attribute.noinit ';'
    |                   constant ';'
    |                   constraint ';'
    |                   generic ';'
    |                   operation ';'
    |                   type ';'

string_type:            STRING
//    |                   STRING '<' positive_int_const '>'

tailored_attrs:         /* epsilon */
    |                   tailored_attrs tailored_attr
tailored_attr:          BITS const_exp ';'
    |                   EPSILON const_exp ';'
    |                   HUGE const_exp ';'
    |                   '+' INFINITY ';'
    |                   '-' INFINITY ';'
    |                   MAXIMUM const_exp ';'
    |                   MINIMUM const_exp ';'
    |                   NAN ';'
    |                   OVERFLOW tailored_overflow_attr ';'
    |                   ROUND tailored_round_attr ';'
    |                   TINY const_exp ';'
    |                   operation ';'
    |                   error ';'
tailored_overflow_attr: CLIP | INFINITY | NONE | RAISE | WRAP
tailored_round_attr:    CONVERGENT | LARGER | NEAREST | SMALLER | ZERO

union_type:             UNION identifier ':' integral_type '{' union_body '}'
union_body:             union_case
    |                   union_body union_case
union_case:             case_label ':' declared_type ';'
    |                   case_label ':' declared_type ':' declared_type ';'
case_label:             CASE expression_list
    |                   DEFAULT

wide_string_type:       WSTRING
//    |                   WSTRING '<' positive_int_const '>'
```

9.2.6 Statecharts

```
statechart:             STATECHART identifier ':' integral_type
                            '{' statechart_attrs '}'
statechart_attrs:       /* epsilon */
    |                   statechart_attrs statechart_attr
statechart_attr:        attribute ';'
    |                   constant ';'
    |                   doc ';'
    |                   generic ';'
    |                   let ';'
    |                   location ';'
    |                   operation ';'
    |                   port ';'
    |                   rendition ';'
    |                   state ';'
    |                   statechart ';'
    |                   type ';'
    |                   error ';'

state:                  state_key '{' state_attrs '}'
    |                   state_key identifier '{' state_attrs '}'
state_key:              STATE
    |                   BRANCH STATE
    |                   DEEP HISTORY STATE
    |                   FINAL STATE
    |                   FORK STATE
    |                   HISTORY STATE
    |                   INIT_IAL STATE
```

```
    |                   JOIN STATE
    |                   JUNCTION STATE
state_name:             scoped_name
state_attrs:            /* epsilon */
    |                   state_attrs state_attr
state_attr:             attribute ';'
    |                   constant ';'
    |                   doc ';'
    |                   entity ';'
    |                   GOTO state_name ';'
    |                   let ';'
    |                   location ';'
    |                   operation ';'
    |                   relation ';'
    |                   rendition ';'
    |                   statechart ';'
    |                   transition ';'
    |                   type ';'
    |                   error ';'

transition:             transition_key identifier parameter_dcls actions_clause
    |                   transition_key identifier actions_clause
    |                   transition_key actions_clause
    |                   AFTER '(' expression ')' actions_clause
    |                   INTERNAL AFTER '(' expression ')' actions_clause
transition_key:         TRANSITION
    |                   INTERNAL TRANSITION
actions_clause:         guarded_actions
    |                   '{' actions '}'
guarded_actions:        guarded_action
    |                   guarded_actions ELSE guarded_action
guarded_action:         GUARD '(' expression ')' '{' actions '}'
actions:                /* epsilon */
    |                   actions action
action:                 doc ';'
    |                   entity ';'
    |                   expression ';'
    |                   GOTO state_name ';'
    |                   let ';'
    |                   location ';'
    |                   path ';'
    |                   relation ';'
    |                   error ';'
```

9.2.7 Constructs

```
attribute.noinit:       attribute_key identifier':' declared_type
    |                   attribute.noinit '{' attribute_attrs '}'
attribute:              attribute_key identifier '=' expression ':' declared_type
    |                   attribute '{' attribute_attrs '}'
attribute_key:          ATTRIBUTE
    |                   READONLY ATTRIBUTE
attribute_attrs:        /* epsilon */
    |                   attribute_attrs attribute_attr
attribute_attr:         constraint ';'
    |                   doc ';'
    |                   error ';'

class:                  CLASS class_name
    |                   class '{' model_attrs '}'

configure:              CONFIGURE StringLiteral
    |                   CONFIGURE StringLiteral StringLiteral

constant:               constant_head1
    |                   constant_head1 '{' constant_attrs '}'
    |                   constant_head2
    |                   constant_head2 ':' '{' constant_attrs '}'
constant_head1:         constant_key identifier
    |                   constant_key identifier ':' declared_type
    |                   constant_head2 ':' declared_type
constant_head2:         constant_key identifier '=' const_exp
constant_key:           CONSTANT
    |                   edit_id CONSTANT
constant_attrs:         /* epsilon */
    |                   constant_attrs constant_attr
constant_attr:          configure ';'
    |                   doc ';'
    |                   location ';'
    |                   property ';'
```

```
    |                   rendition ';'
    |                   error ';'

// ':' necessary to avoid expression ambiguity for constraint(x)
constraint:             constraint_key ':' where_expr
    |                   constraint_key identifier ':' where_expr
constraint_key:         CONSTRAINT
    |                   edit_id CONSTRAINT

dimension:              DIMENSION identifier
    |                   DIMENSION identifier ':' scoped_name
    |                   dimension '{' dimension_multipliers '}'
dimension_multipliers:  /* epsilon */
    |                   dimension_multipliers dimension_multiplier
dimension_multiplier:   expression ';'      // scalar'unit^exponent
    |                   error ';'

director:               DIRECTOR director_name
    |                   director '{' director_attrs '}'
director_attrs:         /* epsilon */
    |                   director_attrs director_attr
director_attr:          class ';'
    |                   configure ';'
    |                   property ';'
    |                   error ';'

doc:                    DOC StringLiteral

entity:                 entity_key identifier
    |                   entity_key identifier ':' declared_type
    |                   entity '{' entity_attrs '}'
entity_key:             ENTITY
    |                   edit_id ENTITY
entity_attrs:           /* epsilon */
    |                   entity_attrs entity_attr
entity_attr:            model_attr
    |                   attribute ';'
    |                   implementation ';'
    |                   operation ';'
    |                   port ';'
    |                   response ';'
    |                   for_key identifier for_type for_where
                            '{' entity_attrs '}' ';'

generic:                GENERIC generic_id ':' generic_type
    |                   generic '{' generic_attrs '}'
generic_id:             identifier
    |                   identifier '=' const_exp
generic_type:           declared_type
    |                   identifier '=' declared_type
//    |                   ENTITY '=' declared_type
//    |                   FLOW '=' declared_type
//    |                   SHAPE '=' declared_type
//    |                   TYPE '=' declared_type
//    |                   VALUE '=' declared_type
generic_attrs:          /* epsilon */
    |                   generic_attrs generic_attr
generic_attr:           configure ';'
    |                   doc ';'
    |                   location ';'
    |                   property ';'
    |                   rendition ';'
    |                   error ';'

implementation:         IMPLEMENTATION identifier
    |                   IMPLEMENTATION identifier ':' language_name
    |                   implementation '{' java_tokens '}'
language_name:          StringLiteral
java_tokens:            /* epsilon */
    |                   java_tokens java_token
java_token:             identifier
    |                   literal
    |                   AND_AND | COLON_EQ | DOT_DOT | OR_OR
    |                   GE | LE | NE | SHL | SHR
    |                   '*' | '&' | ',' | '(' | ')' | ';'
    |                   '[' | ']' | ':' | '?' | '.' | '/'
    |                   '+' | '-' | '%' | '^' | '|' | '~' | '!' | '=' | '<' | '>'
    |                   '\'' | '\"' | '\\'
    |                   '{' java_tokens '}'
```

```
import:                 IMPORT source_name
    |                   IMPORT source_name ':' base_name

interface:              interface_id
    |                   interface_id '{' interface_attrs '}'
    |                   interface_id inheritance_spec '{' interface_attrs '}'
interface_id:           INTERFACE identifier
interface_attrs:        /* epsilon */
    |                   interface_attrs interface_attr
interface_attr:         attribute ';'
    |                   constant ';'
    |                   exception ';'
    |                   let ';'
    |                   operation ';'
    |                   type ';'
    |                   error ';'
inheritance_spec:       ':' scoped_name
    |                   inheritance_spec ',' scoped_name

let:                    LET identifier '=' expression
    |                   LET identifier '=' expression ':' declared_type

link:                   link_key link_port_name
    |                   link '{' link_attrs '}'
link_key:               LINK
    |                   edit_id LINK
link_port_name:         scoped_name
    |                   scoped_name array_shape
link_attrs:             /* epsilon */
    |                   link_attrs link_attr
link_attr:              vertex ';'
    |                   error ';'

location:               LOCATION location_name

model:                  MODEL identifier
    |                   model '{' model_attrs '}'
model_attrs:            /* epsilon */
    |                   model_attrs model_attr
model_attr:             class ';'
    |                   configure ';'
    |                   constant ';'
    |                   constraint ';'
    |                   dimension ';'
    |                   director ';'
    |                   doc ';'
    |                   entity ';'
    |                   generic ';'
    |                   import ';'
    |                   let ';'
    |                   property ';'
    |                   relation ';'
    |                   rendition ';'
    |                   statechart ';'
    |                   type ';'
    |                   error ';'

operation:              operation_key identifier operation_signature
                            raises_expr.opt
    |                   operation_key identifier operation_signature
                            raises_expr.opt '{' operation_attrs '}'
operation_key:          OPERATION
    |                   ONEWAY OPERATION
operation_signature:    '(' parameter_dcls ')' ':' declared_type
    |                   '(' ')' ':' declared_type
parameter_dcls:         parameter_dcl
    |                   parameter_dcls ',' parameter_dcl
parameter_dcl:          parameter_dir identifier ':' declared_type
parameter_dir:          IN | OUT | INOUT
raises_expr.opt:        /* epsilon */
    |                   raises_expr
raises_expr:            RAISES '(' scoped_name_list ')'
operation_attrs:        /* epsilon */
    |                   operation_attrs operation_attr
operation_attr:         constraint ';'
    |                   implementation ';'
    |                   specification ';'
    |                   statement

path:                   PATH '{' path_attrs '}'
```

```
path_attrs:             /* epsilon */
    |                   path_attrs path_attr
path_attr:              doc ';'
    |                   location ';'
    |                   error ';'

port:                   port_key port_flow port_decl
    |                   port_key port_flow port_decl '{' port_attrs '}'
port_key:               port_dir
    |                   INTERNAL
port_flow:              /* empty */
    |                   ':' scoped_name
port_dir:               parameter_dir
    |                   parameter_dir array_shape
port_id:                scoped_name
    |                   scoped_name array_shape
port_init:              port_id
    |                   port_id '=' expression
port_decl:              port_id
    |                   port_init ':'                           // Ugh!
    |                   port_init ':' declared_type
    |                   port_init ':' declared_type ':' declared_type
port_attrs:             /* epsilon */
    |                   port_attrs port_attr
port_attr:              configure ';'
    |                   doc ';'
    |                   location ';'
    |                   property ';'
    |                   rendition ';'
    |                   error ';'

property:               PROPERTY identifier
    |                   PROPERTY identifier '=' expression
    |                   PROPERTY identifier '{' property_attrs '}'
property_attrs:         /* epsilon */
    |                   property_attrs property_attr
property_attr:          class ';'
    |                   configure ';'
    |                   doc ';'
    |                   property ';'
    |                   error ';'

relation:               relation_key identifier
    |                   relation '{' relation_attrs '}'
relation_key:           RELATION
    |                   edit_id RELATION
relation_attrs:         /* epsilon */
    |                   relation_attrs relation_attr
relation_attr:          class ';'
    |                   link ';'
    |                   property ';'
    |                   vertex ';'

rendition:              RENDITION ':' class_name
    |                   rendition '{' rendition_attrs '}'
rendition_attrs:        /* epsilon */
    |                   rendition_attrs rendition_attr
rendition_attr:         class ';'
    |                   configure ';'
    |                   location ';'
    |                   property ';'
    |                   error ';'

response:               RESPONSE identifiers '{' response_attrs '}'
response_attrs:         /* epsilon */
    |                   response_attrs response_attr
response_attr:          attribute ';'
    |                   constraint ';'
    |                   implementation ';'
    |                   specification ';'
    |                   statement

specification:          SPECIFICATION '{' specification_attrs '}'
    |                   SPECIFICATION identifier '{' specification_attrs '}'
specification_attrs:    /* epsilon */
    |                   specification_attrs specification_attr
specification_attr:     attribute ';'
    |                   constraint ';'
    |                   statement
```

```
vertex:                 VERTEX identifier
    |                   vertex '{' vertex_attrs '}'
vertex_attrs:           /* epsilon */
    |                   vertex_attrs vertex_attr
vertex_attr:            location ';'
    |                   path ';'
    |                   property ';'
    |                   error ';'
```

9.3 Compilation

A vendor should refine an abstract WDL specification into an implementation rather than rewrite it, since this should improve compliance and turn-around time. The extra stages involved in the conversion from abstract WDL to compilable source for target architectures are outlined in the following sections.

9.3.1 Semantic Analysis

The WDL must be checked for consistency. Names must be defined before use. Repeated specifications must be consistent. Flows must be consistent, the same types at each end.

Rendezvous must be legal

- at most one source event
- event input cannot occur when token input missing

Flows must be legal

- no use of concurrent events
- no generation of concurrent events
  - event sources have precedence
- no merge of non-events
- token outputs must be connected (at least in some state)

Unusual flows can be diagnosed

- unconnected event inputs
- intermittent token flows (unconnected in some states)

At the end of this activity the WDL specifies a deterministic behaviour.

9.3.2 Deduction

WDL generics and constraints avoid the need for explicit parameterisation of library blocks, but require the fixed point solution of a number of interacting constant, type, shape, flow, enumerator and entity constraints to be identified. Conflicts detected during this process correspond to contradictions in the specification. Specification ambiguities lead to a failure to deduce the full value of some generics.

This process should be fully automated, and could beneficially be available during specification entry so that contradictions, ambiguities and resolutions can be viewed within the entry context. The entry system may need to provide assistance so that the reason for a contradiction or ambiguity can be easily and relevantly located.

At the end of this activity every part of the specification is complete and consistent.

9.3.3 Revision and Partitioning

The specification must be refined to suit the implementation capabilities of a particular

vendor, and to partition the implementation over appropriate hardware resources.

It may be possible to do this automatically, but for most realistic systems, it is likely that the skills of a system designer will be required. Partitioning of many systems is often very easy to specify, so a simple assistant that is able to present estimates of the resource requirements of different parts of a specification would be useful.

An automated assistant might be able to

- report cumulative amplitude distortions
- report cumulative timing properties
- report cumulative resource requirements
- identify relevant hand-optimized implementations

When the target platform is virtual, as is the case for a JTRS object, the boundary of the WDL for each object must be automatically elaborated to incorporate the requisite message and structural interfaces so that the subsequent product complies with the protocols of the virtual platform. The WDL then describes the behaviour of JTRS objects, and the bit-true representations of CORBA compliant messaging between them.

At the end of this iterative activity, the WDL has been refined to

- identify practical algorithms for abstract specifications
  - ripple/bubble/quick sort
  - receive synchronisation strategy
  - Butterworth/Bessel/... filter
- exploit vendor libraries
- apportion each part of the specification to appropriate hardware
- apportion degradations to particular parts of the specification
  - scheduling deadlines
  - quantisation limits
  - sample rates
- impose appropriate bit-truths on interface messages

9.3.4 Type Selection

WDL specification types identify the requirements that each type must satisfy and avoid premature assumptions that may favour a 16 bit implementation. The decision about precisions is deferred at least until the target hardware is known, so that the implementation type can be chosen to suit the capabilities of the target environment. If the total compilation system can support it, the decisions should be further deferred until registers are assigned at the assembler level, since this may allow more effective code to be generated on processors with a mixture of working lengths. In practice, the abstract WDL is to be translated into a concrete language such as C or VHDL, which does not support such flexibility; concrete types must be selected slightly prematurely.

9.3.5 Scheduling and Code Generation

Flows are analysed to identify the boundaries of regions within which efficient scheduling strategies can be employed and the requisite scheduling code is generated around the executable. There are significant complexities in this area.

Scheduling strategies for token flows become increasingly efficient as the token flows

become more disciplined. Most DSP systems have large regions of extremely uniform token flow that satisfy the Synchronous Data Flow paradigm [Lee87], which can often be scheduled without scheduling overhead. The techniques for data flow scheduling have been pioneered in Ptolemy (and Gabriel before it) and must be exploited if a WDL specification is to be translated into an efficient implementation. The availability of token flows and rigorous rendezvous semantics supports the analysis necessary to discover that the very large number of potentially concurrent processes and infinite FIFOs are amenable to trivial implementation.

At this point it is also appropriate to select

- eager or lazy evaluation strategies for published expressions of state variables
- resolve message layouts to avoid redundant calculations

9.3.6 Code synthesis

Code generation from WDL requires WDL statements and data types to be represented in a language suitable for compilation or assembly on the target platform. Some WDL statements correspond quite closely to equivalent constructs in more conventional languages, others particularly those involving arrays require significant code synthesis.

Naive generation of code for array operations will produce local loops for each library entity, with the result that a sequence of array operations incurs excessive looping overhead. Analysis of a cascade of array operations will often find sequences with no cross-interaction allowing a single loop to be established for the cascade eliminating loop overhead and reducing temporary buffer sizes.

9.3.7 Code generation

The most direct approach to code generation requires an appropriate code template for each combination of primitive library entity and implementation language, and a suitable template resolution mechanism to enable the code to adapt to its context. This approach fails to achieve any useful sharing of effort between alternate implementation targets, and has very limited capability to do custom optimizations.

Ptolemy Classic pursued an indirect approach to code generation, whereby a program was generated to run on the host, which then generated the required code. Custom optimizations and reaction to context could be performed while the host program executed. This provided a level of meta-programming, but did not significantly impact on the ease of implementation of multiple targets.

Ptolemy II does not yet support code generation, but offers extreme forms of polymorphism with respect to scheduling, and some flexibility with respect to type.

The two approaches could be combined so that the host code generation run made requests to data types to generate code for specific operators. Implementation of another target then involves generating code templates for each data type and operator, rather than for each library entity.

9.3.7.1 Entity-Oriented Code Generation

Automated code generation for DSP systems has traditionally pursued an entity-oriented approach, in which a code template is provided for each leaf entity. This requires code to be provided for supported target language for each new entity and tends to tie the entity to a very restrictive (singleton) set of usable types.

A fully implemented library of E entities, operating on D data types supported on T targets requires E*D*T entity implementations with this approach. The cost of adding an

additional entity, data type or target is large.

This approach is supported by WDL, primarily to support highly optimized custom blocks. The type-oriented approach is intended for general purpose and default behaviour.

```
entity X
{
};
```

9.3.7.2 Type-Oriented Code Generation

Using type-oriented code generation, code fragments are associated with the operations of data types, and so the fragments are reusable by all entities that use the same combinations of operators and types. It is not necessary to supply implementations for new leaf entities, since the single definition of the entity in terms of WDL operators enables the code generator to select the approach type-specific fragments. The type polymorphism of entity implementations is preserved.

A fully implemented library of E entities, operating on D data types supported on T targets requires E + D*T entity implementations with this approach. The cost of adding an additional entity is small, and the cost of adding a data type or target is more manageable.

This approach may appear to generate less efficient code, since there are no entity level optimizations. However, since the generated code merely expresses the data manipulations and dependencies, a good compiler should have little trouble in optimising. WDL provides an opportunity to communicate programming intent much more clearly to the compiler without the redundant implementation clutter of inappropriate languages. Unfortunately DSP compilers are of relatively poor quality at present. Hopefully this will change as DSPs become more RISC-like, compiler technology is exploited, and customer pressure is applied.

9.3.8 Compilation Paths

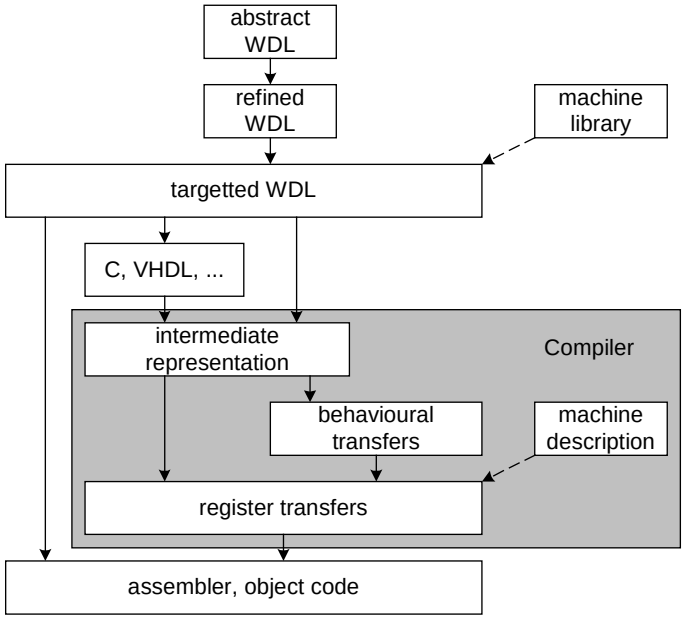


Figure 10 Compilation paths

There are a number of different compilation paths that may be pursued using WDL, some of which require less effort in the short term, while others offer greater optimization

opportunities in the longer term.

Direct to assembler.

It is possible to generate assembler code directly from WDL, however this will require assembler templates for each combination of mathematical operation and type, and templates for schedulers. Register allocators must also be provided.

Implementation of this approach for any particular target is likely to require substantial effort. Implementation for a further target is not significantly easier, since a distinct machine library has to be provided for each target.

High level language

Conversion of WDL to a high level language is somewhat simpler, because the compiler takes responsibility for register allocation and provides support for a number of relevant data types.

Implementation of this approach for a first language will require significant effort. Subsequent languages should be much easier, since language-specific machine libraries will have much in common.

Intermediate Representation

Most compilers convert the source code into an intermediate representation which is suitable for performing optimizations. This is then converted to a register transfer form which can be scheduled on the target processor.

Conversion of WDL to a high level language is a rather retrograde step, particularly if the compiler has vector optimizations. It is more appropriate to generate the intermediate representation directly.

This approach is only possible once compilers offer access to the intermediate representation, or if an existing compiler is opened up to support more direct WDL compilation.

Only a single machine library is necessary, since machine specifics are handled by the compilers machine description.

Behavioural Transfers

WDL specifies the minimum required behaviour of each data type, and in order to work with conventional compilers must make premature decisions as to the actual precisions to be used.

A compiler that supports a more general behavioural description of each register transfer could defer precision decisions until the available register set was known.

Machine Description

Retargetable compilers make use of a machine description to target the required architecture. Optimizations based on use of the machine description will be far superior to those that could sensibly be provided by the direct to assembler approach.

9.4 GUI support

An appropriately configured GUI environment should support

- schematic entry of message flow and statechart diagrams
- creation and configuration of entity icons

- context sensitive forms for constraint, type, and operations
- entry of refinements as change with respect to reference
- feedback of errors, deductions and costs
- control of the compilation process

10 References

[Aho86] A.V. Aho, R. Sethi and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Bhattacharyya96] Shuvra S.Bhattacharyya, Praveen K. Murthy and Edward A. Lee. *Software Synthesis from Dataflow Graphs.* Kluwer Academic Publishers, 1996.

[Booch99] Grady Booch, James Rumbaugh and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[CORBA98] OMG, *IDL from CORBA V2.2*, February 1998

http://www.omg.org/pub/docs/forml//98-03-01.idl

[Davis99] John Davis II, Mudit Goel, Christopher Hylands, Bart Kienhuis, Edward A. Lee, , Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, John Reekie, Neil Smyth, Jeff Tsay and Yuhong Xiong, *Overview of the Ptolemy Project.* ERL Technical Report UCB/ERL No. M99/37 University of California, Berkeley, CA, USA 94720, July 1999.

http://ptolemy.eecs.berkeley.edu/publication/papers/99/overview/overview.pdf

[Girault97] A. Girault, B. Lee, and E. A. Lee, Hierarchical Finite State Machines with Multiple Concurrency Models, IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems, Vol. 18, No. 6, June 1999 (revised from Memorandum UCB/ERL M97/57, Electronics Research Laboratory, University of California, Berkeley, CA 94720, August 1997).

http://ptolemy.eecs.berkeley.edu/publications/papers/99/starcharts/starcharts.pdf

[Lee87] Edward Ashford Lee and David G. Messerschmitt. *Static scheduling of synchronous data flow programs for digital signal processing*. IEEE Transactions on Computers, vol. 36, no. 1, 24-35, January 1987.

[Lee00] Edward A. Lee and Steve Neuendorffer, *MoML - A Modeling Markup Language in XML* Version 0.4, Technical Memorandum UCB/ERL M00/12 Dept. EECS University of California Berkeley, CA 94720, USA March 14, 2000.

http://ptolemy.eecs.berkeley.edu/publication/papers/00/moml_erl_memo.pdf

[Levine92] John R. Levine, Tony Mason and Doug Brown. *lex & yacc*. Second Edition, O'Reilly and Associates, Inc, 1992.

[Rumbaugh99] James Rumbaugh, Ivar Jacobson and Grady Booch. *The Unified Modeling Language Reference Guide*. Addison Wesley, 1999.

[UML99] OMG. *Unified Modeling Language Specification Version 1.3* June, 1999

http://www.omg.org/cgi-bin/doc?ad/99-06-08.pdf (Includes OCL).

[XML98] W3C, *Extensible Markup Language (XML) 1.0*, 10 February 1998.

http://www.w3.org/TR/1998/REC-xml-19980210.pdf

[XSL97] W3C, *A Proposal for XSL*, 27 August 1997

http://www.oasis-open.org/cover/note-xsl-970910.html

Index