The Waveform Description Language

Edward D. Willink

Thales Research Limited

02/01/02 09:59

The benefits of Software Defined Radios (SDR) are identified in many chapters of this book. SDR will potentially permit new protocols, applications and air interface waveforms to be flexibly and even dynamically deployed across a variety of implementation platforms and products, sourced from multiple vendors. It will bring the benefits of open standards that have been enjoyed in the PC world to the arena of wireless communications. However the basic hardware and software technologies, addressed elsewhere in this book, need to be supported by design tools and technologies that enable the simple, flexible, rapid and cost-effective development, deployment and reconfiguration of SDR implementations.

These issues are being addressed at many levels, with early efforts having focussed particularly on hardware and applications. In this chapter, the issue of design techniques and tools to specify and implement an air interface waveform is addressed, so that the specification can be re-deployed rapidly without incurring major costs or incompatibilities. The approach that has been identified to address this issue is the concept of the Waveform Description Language. This chapter provides an introduction to this concept, describes its rationale and origins, describes proof-of-concept experience within the framework of the FM3TR[1] programme and describes the status and detailed techniques associated with the emerging WDL, as at the mid-2001 timeframe[2]. Interrelationships with other languages such as SDL, UML and XML are also briefly discussed.

1 The Specification Problem

The traditional process for conversion of a specification into an implementation involves significant manual intervention because programming occurs at the implementation (or solution) level. The Waveform Description Language exploits compilation technology advances to support programming at the specification (or problem) level, with a highly automated translation to an appropriate implementation. This change of perspective provides a more readable specification, avoids transformation errors between specification and implementation, and eliminates the premature coupling to a particular implementation approach that causes so many portability problems and so WDL provides for genuine re-use.

System specification presents considerable challenges that are usually resolved by fairly large specification documents. The authors of these documents may seek to minimise the size by attempting to provide concise specifications, but this tends to cut out useful editorial descriptions and may leave some aspects of the system underspecified. Alternatively, they may avoid the hazards of conciseness by providing extra explanations and occasionally repeating parts of specifications to reduce the need for cross-referencing. Unfortunately, repetitions may introduce contradictions between the specification and the clarifying descriptions. A concise specification therefore tends to be incomplete, and a verbose specification contradictory. Many specifications are both. In either case, independent attempts to implement the specification may resolve ambiguities in different ways with the result that nominally compliant equipments are incompatible.

---

[1] A description of the four-nation FM3TR defence SDR programme may be found in the chapter by Bonser in the companion volume, “Software Defined Radio: Origins, Drivers and International Perspectives”, Ed W Tuttlebee, pub Wiley, 2002, ISBN **xxxx.**

[2] This chapter describes the work performed at Racal Research (now Thales Research) under Phase 1 of the UK PDR programme between December 1999 and April 2000 and incorporates some further insights. Phase 2 was awarded in March 2001 to a rival consortium led by BAE Systems. The author has no knowledge about how the preliminary concepts will be carried forward. WDL as here described thus represents the ideas resulting from Phase 1; future development of WDL may be significantly different.

Mathematics can be used to resolve these problems, and indeed even informal specifications often contain some mathematics, but sometimes the meaning of an equation relies on intuition and contradicts a clarifying explanation. Specifications can be written using formal methods with the result that they are complete and unambiguous. Unfortunately, it is substantially harder to produce specifications using formal methods and few customers, managers or programmers are able to understand them. The major problems of obtaining a specification that satisfies the customer and successfully realising that specification are not resolved in practice. Formal methods are therefore restricted to safety critical domains where the need for provable correctness outweighs the difficulties and costs.

The Waveform Description Language provides an approach to specifying systems that falls between the textual and mathematical extremes. The required behaviour of a system is specified by hierarchical decomposition using well established graphical techniques involving state and block diagrams. The leaf behaviour is provided by re-usable library entities or custom entities specified using a constraint language. The specification style is familiar and as a result, specifications are readable. The decomposition and leaf semantics are defined by the principles of Synchronous Reactive and Data Flow computing and consequently, the specification is mathematically rigorous. The specification is executable ensuring completeness and lack of ambiguity. Analysis, validation and instrumentation are possible.

Hardware designers have always been constrained by physical reality to use modular decomposition. Software has greater complexity, more freedom, a variety of perspectives, and a ‘software crisis’. Many of the methodologies proposed to alleviate the crisis, have succumbed as increased encapsulation has been provided by technology advances; first function, then value and then type encapsulation have led to Object Orientation. WDL continues this trend by providing encapsulation of behaviour, with the result that software is specified in the same way as hardware.

2 WDL Overview

The Waveform Description Language (WDL) [13] combines principles from a number of increasingly mature research areas to produce a practical specification language. The language gives a well-defined meaning to practices that are widespread in the implementation domain but applies them to the specification domain. The result is natural and easy to use for everyday system designers, but has the underlying rigour needed for a useful specification that is both unambiguous and deterministic.

2.1 Decomposition

WDL uses hierarchical principles to decompose a specification into smaller and smaller sub-specifications until the specification problem becomes tractable. The hierarchical decomposition uses well established block and state diagram styles.

Block diagrams have long been used, with a variety of intuitive semantics, whenever engineers abstract from details such as support structures or bias networks to consider just the basic principles of a system. The WDL form of a block diagram is called a (message) flow diagram to stress the important role that message flows play in establishing rigorous semantics.

The use of state diagrams can be traced back at least to the early 1950s and the initial work on finite automata leading to Mealy [8] and Moore [9] machines. The graphical style has evolved, particularly with the introduction of hierarchy by Harel [4], and its use in OMT [11] to that known as a statechart in UML [12]. The minor differences between a WDL statechart and a UML statechart are described in Section 2.4.3.

Statecharts support decomposition into alternate behaviours, whereas message flow diagrams support decomposition into composite behaviours. Interleaving statechart and message flow diagrams therefore provides for an arbitrary decomposition.

2.2 Communication

The block diagram approach leads to a very strong form of encapsulation. Each block (rectangle) encapsulates its internal behaviour. External interaction occurs only along interconnecting arcs and through the directional ports that the arcs connect.

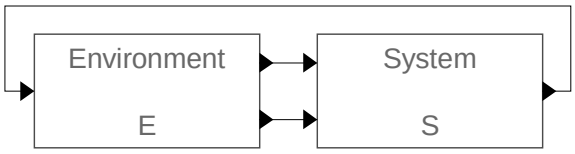

Figure 1 System Specification Problem

The overall system specification problem is immediately partitioned into the three sub-problems of Figure 1:

- specify σ, the behaviour σ of system entity S
- specify η, the behaviour of the external environment entity E
- specify ζ(), the interactions between S and E

The latter two sub-problems are often rather neglected.

Each entity is self-sufficient: responsible for its own scheduling, maintaining its own internal state and interacting solely through its ports, along the designated communication paths. Therefore decomposing the system (or the environment) just introduces more self-sufficient

entities and more designated communication paths.

An example decomposition converts the original problem into the new problem shown in Figure 2, where the ends of external connections on the internal diagram are shown by named port symbols whose connections continue at the same-named ports on the outer interface.

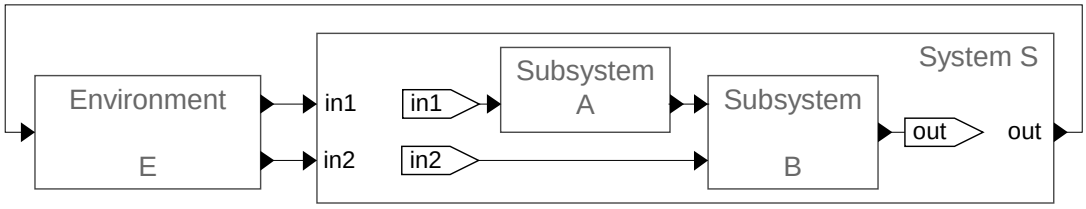

Figure 2 Decomposed Specification Problem

The decomposed problem is:

- specify σ, the behaviour of system S
  - specify α, the behaviour of subsystem A
  - specify β, the behaviour of subsystem B
  - specify ξ(), the interactions between A and B
- specify η, the behaviour of the external environment E
- specify ζ(), the interactions between S and E

such that the composed behaviour is the same as the composite:

$\zeta(\xi(\alpha,\beta),\eta) \equiv \zeta(\sigma,\eta)$.

It is clear that the ζ and ξ (and further functions resulting from further decomposition) are concerned with the interconnection and communication between the (sub)systems. Accurate specification of these operators is difficult when an informal approach is used, and daunting when elaborated mathematically. WDL uses rigorous composition rules parameterised by data type and flow type annotations to support the graphical definition.

Each communication operator expresses the total behaviour of one or more independent interconnection paths, each of whose properties are defined in WDL.

How? WDL does not mandate how an implementation resolves the specification; WDL just defines the observable behaviour of all valid implementations.

Where? The graphics defines the direction and end points of each connection.

What? A data type annotation defines the information content(s) of each communication path. See Section 5.1.

When and Why? A flow type annotation defines a scheduling protocol for each path. The WDL semantics define the way in which different paths interact. See Section 5.2.

WDL semantics are based on the Esterel synchrony hypothesis [1] that no two events occur at the same time and that all event processing is instantaneous. The defined behaviour is therefore deterministic, and as noted in [1], deterministic systems are an order of magnitude easier to specify. However this defines an ideal behaviour that cannot be achieved by any practical implementation, and so a WDL specification must incorporate tolerances as annotations that bound the practical degradations of an implementation.

Since the specification defines an ideal behaviour, the tolerances are unidirectional; the

practical implementation is worse. This avoids complexities that arise from a 'gold standard' practical implementation, against which an arguably superior implementation might have to be considered inferior.

The above example decomposes a system into a subsystem. It is of course possible to decompose the environment into more manageable sub-environments. It is also possible to decompose a connection to insert a nominally transparent composition of entities such as a coder and decoder.

2.3 Influences

The concepts used by WDL are drawn from a variety of language domains as indicated in Figure 3. A few examples of each language domain are also identified.

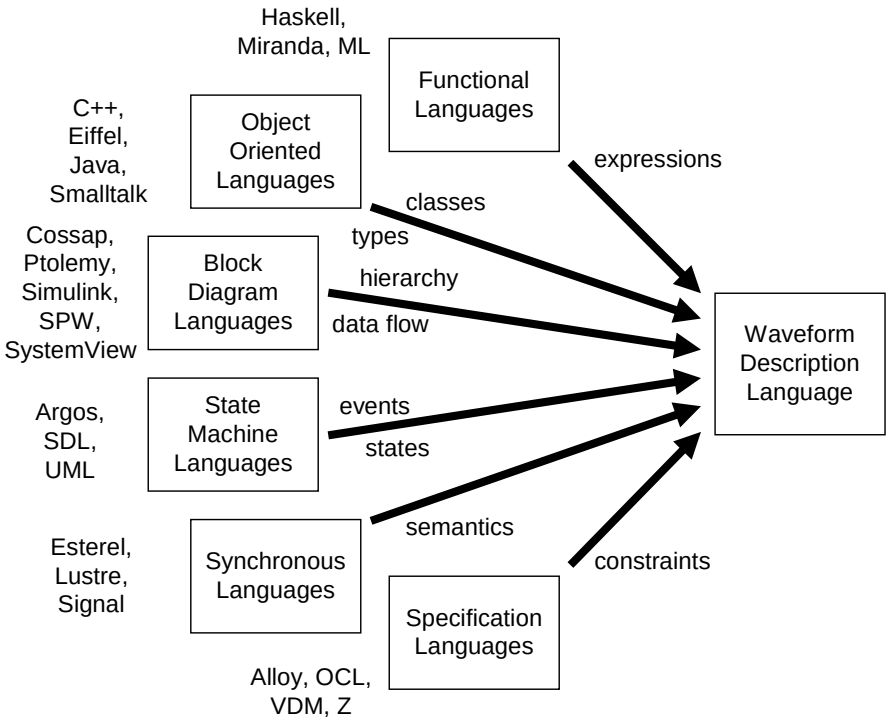

Figure 3 WDL Antecedents

The utility of WDL arises through the use of hierarchical block and state diagrams, supporting data-driven interaction as in tools such as Cossap, and event-driven interaction as in SDL [10]. Successful co-existence of these alternate forms of interaction relies on the analysis presented as *charts (starcharts) in [3]. Effective communication and flexible computation is supported by applying Object Oriented principles to declare data types.

WDL provides a deterministic specification through adoption of the synchrony hypothesis from synchronous languages such as Esterel [1].

Detailed mathematical specifications as embodied by Z or VDM are powerful but unapproachable, while the capabilities of OCL in UML [12] are limited. WDL therefore follows Alloy [5] and SDL in the search for a more acceptable mathematical presentation. The discipline of functional languages is followed to ensure referential transparency and consequently ease the code generation task. Further flexibility is provided by exploiting data parallel concepts to extend scalar activities to arrays.

A WDL specification is a hierarchical decomposition of behaviour only. Since there is just this one dimension for decomposition, there are no conflicts of perspective corresponding to those between class inheritance, static object relationships, dynamic message interactions

and module partitioning that arise when using UML at the implementation level.

Languages such as SDL [10] and Ptolemy [2] already embody many of the WDL concepts. SDL comprises specification constraints, types and hierarchical block diagrams as well as state machines. Research by the Ptolemy group into using data flow for DSP within a more general computing context has used Synchronous Reactive principles for a block diagram tool that supports arbitrary hierarchical use of state machines and data flow, with user-defined types.

WDL incorporates concepts available in other approaches, but provides sufficient breadth to cover the full range of specification problems, and powerful parameterisation facilities to support genuinely re-usable specification libraries. The concept of a flow type is fundamental to supporting analogue specification, computational data flow and event-driven protocols. The concept of refinement is fundamental to evolving the specification into, rather than rewriting as, the implementation during the development lifecycle.

In comparison to SDL, with which WDL has many philosophical similarities, the addition of support for arbitrary hierarchy, data flow and multiple inputs leads to a major change of perspective. All nodes on all WDL diagrams are behavioural entities. SDL diagrams are confused by the use of a rich graphical notation to provide support for low level computational data flow.

In comparison with a simple block diagram tool such as Cossap, the addition of event handling and state machines resolves the higher level specification issues concerned with alternate behaviours. The addition of data types supports specification of low level intent and high level message complexity.

In comparison with the more substantive Ptolemy, WDL provides the required abstraction to support use as a specification rather than an implementation tool [15].

In comparison with Esterel, WDL provides the same fundamental semantics, but provides the missing algorithmic capabilities to accompany the scheduling framework.

The presentation in this chapter uses graphics for the hierarchical diagrams and text for details. The underlying representation uses an XML dialect for both graphical and textual elements. The graphical aspects can therefore also be edited using extensions of the textual dialects. (Any form of editing can of course be performed using XML tools.)

2.4 Hierarchical Diagrams

A large WDL specification for a system entity is progressively decomposed into the smaller more manageable sub-specifications of sub-system entities using one of two forms of hierarchy diagram.

2.4.1 Interfaces

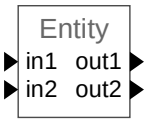

Figure 4 Example Entity Interface

At each level of decomposition, the external behaviour of an entity is defined by its interface, which comprises a number of ports, each of which has a name, direction (input or output) a data type and flow type.

The data and flow type may be explicitly defined. However, leaving data and flow types undefined or only partially defined provides for re-use, since WDL defines how types can be

deduced along interconnections and across entities. A re-usable entity may therefore adapt to use the types appropriate to the re-using context.

WDL does not impose detailed graphical stylistic rules, and so the diagrams in this chapter are examples rather than definitions of a graphical style. The rather bland icon of Figure 4 uses just names. It can be replaced by a more meaningful icon, as in Figure 5, in order to make a flow diagram more understandable.

2.4.2 Message Flows

A message flow diagram defines the internal behaviour of an entity by composition. The nodes of the diagram define partial behaviours, all of which are exhibited concurrently. The arcs express the communication between entities.

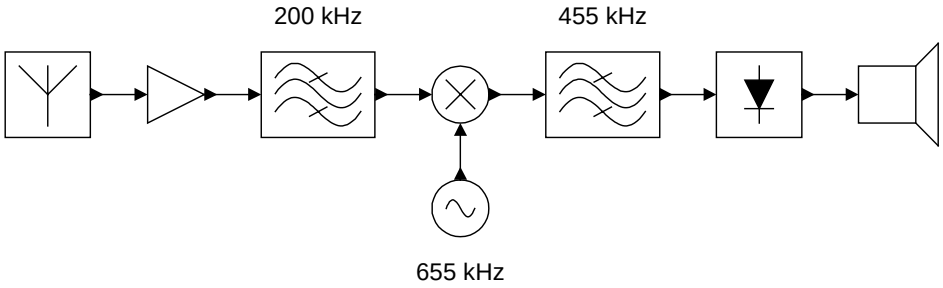

Figure 5 Example Flow Diagram

Figure 5 provides a simple example of a fixed-tuned long wave receiver. The diagram is equally intelligible to valve, transistor, ASIC or DSP practitioners, since it decomposes the system into a number of potentially parallel behavioural entities with designated directional communication paths between them, without imposing any particular implementation approach.

Communication between entities occurs through the interchange of messages subject to an appropriate flow type (or scheduling protocol), which in the example requires continuous one-way signals (or an emulation thereof).

Additional annotation is required to convert the above diagram from an informal intuitive interpretation to a specification with formal semantics. The WDL flow diagram is therefore annotated to define `signal` as the flow type of each communication path, together with constraints on the maximum latency and an appropriate 'fidelity' over a specified frequency range. More detail than just a centre frequency is also needed to pin down the filters.

An implementation using valves will indeed use continuous signals, whereas a DSP implementation will sample some of the signals at a sufficiently high rate to simulate the required continuity. Practical implementations will presumably distribute the initial amplification over a variety of isolation and gain stages.

Figure 6 Use of test harness

A simple fidelity specification such as within 1% can be directly specified in WDL. A rather more challenging specification involving a signal to integrated spurious energy ratio, or a coloured phase noise characteristic cannot; it is impossible for WDL to have every test measurement criterion built-in.

Difficult fidelity specifications are therefore resolved as in Figure 6 by specifying the behaviour of a test harness and asserting that the test harness output satisfies some criterion.

```
constraint: assert.in < 100`ms;
```

A library of re-usable test harnesses will include the specifications for a variety of distortion analysers. A test harness with a necessarily true output is easily recognised as a redundant part of the specification and can be omitted from an implementation, although it may be retained by a simulation. Ensuring that the test harness output is always true is not generally provable, although there are many practical cases where a proof tool may be successful. Where proof fails, a tool can at least enumerate all the unproven specifications so that engineering judgement and directed testing may be applied.

2.4.3 Statecharts

A statechart defines the internal behaviour of an entity as a Finite State Machine. The outer nodes of the diagram define the alternative behaviours, exactly one of which is exhibited. The outer arcs of the diagram express the transitions between behaviours.

A WDL statechart follows the conventions of a UML statechart; the internal behaviour of a state is defined using an application-specific language, which is a nested message flow diagram in the case of WDL. The concurrent state machine concepts of UML are therefore unnecessary in WDL, since a message flow diagram expresses conjunction of behaviour more clearly.

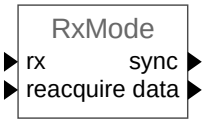

Figure 7 Example Statechart Interface

The interface of a receiver mode control entity `RxMode` is shown in Figure 7. It is defined in exactly the same way as for a message flow diagram, and so does not reveal that a statechart is used for the internal decomposition shown in Figure 8.

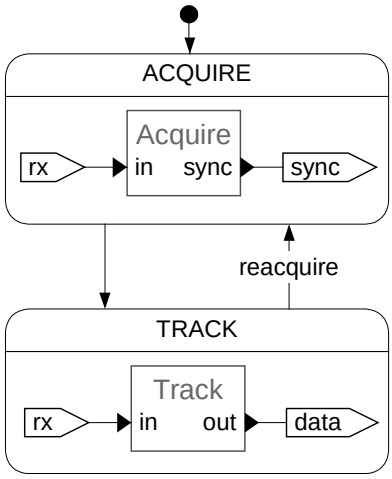

Figure 8 Example Statechart

The control entity may be in either `ACQUIRE` or `TRACK` mode. It starts in `ACQUIRE` mode making a transition to `TRACK` mode when the `Acquire` activity terminates. It then remains in `TRACK` mode until an instruction to `reacquire` is detected.

3 FM3TR Example

An indication of the style of a WDL specification may be obtained by considering a simple example. The following example shows part of the respecification of the Future Multi-band Multi-waveform Modular Tactical Radio (FM3TR) using WDL in place of a more traditional verbal specification. The full decomposition may be found in [14].

The FM3TR waveform is the result of a four power initiative to provide an unclassified waveform with some similarities to Saturn. It provides 16 kbit/s voice or 9.6 kbit/s data communication at between 250 and 2000 hops/second in 25 kHz channels from 30 to 400 MHz.

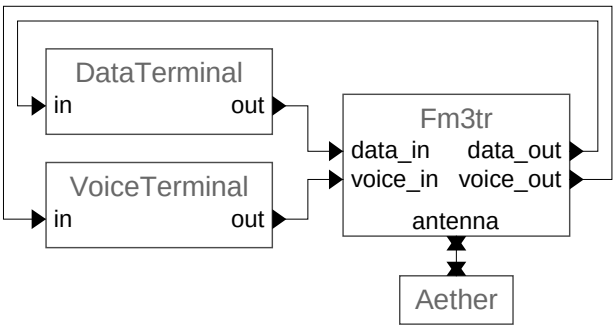

Figure 9 FM3TR Interface

Figure 9 shows the interface of the radio together with a decomposition of its operating environment into a data terminal, voice terminal and the aether. (The connection to the aether provides a rare instance of a fully bi-directional connection.)

3.1 Protocol Layers

The FM3TR specification defines a layered behaviour corresponding to the physical, data-link and network layers of the OSI model. In order to better accommodate the characteristics of a radio channel, the specification defines a physical (`Phl`), medium-access (`Mac`), data link control (`Dlc`) and network (`Nwk`) layers.

These layers and the communication between them are shown in Figure 10 together with a Human Computer Interface (`Hci`) to stub out all the unspecified control interfaces.

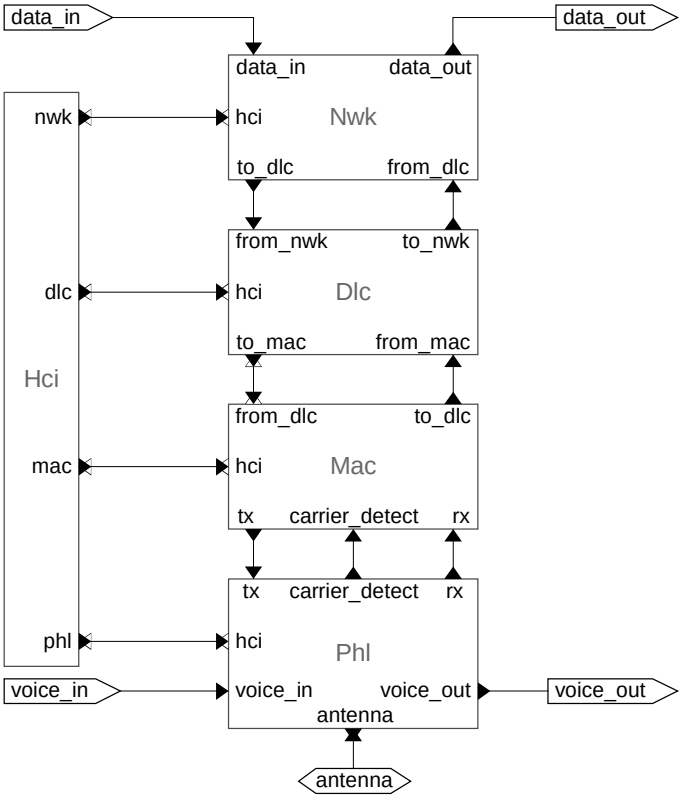

Figure 10 FM3TR Layers

Voice communication involves only the physical layer and so the communication path is from `voice_in` to `antenna`, and from `antenna` to `voice_out`. Data transmission uses all layers with `data_in` flowing down to `antenna` and then back up to `data_out`.

The extra `carrier_detect` signal supports the pCSMA (Persistent Carrier Sense Multiple Access) collision avoidance algorithm in the `Mac` layer.

A master-slave connection is necessary for the `Hci` connections to support dispatch of a command message and an associated response. A master-slave connection is also required between `Dlc` and `Mac` to ensure that the `Dlc` does not choose the next message while a previous message awaits transmission.

3.2 Physical Layer Modules

The major mode changes in the physical layer between transmit and receive are resolved by specification of alternate behaviours using a state machine in the next section. Before that machine can be defined, the specification must also identify all the physical layer behaviours that are not affected by the state changes. The outer perspective of the physical layer, shown in Figure 11, therefore uses a message flow diagram to express the conjunction of behaviours that consist largely of signal conditioning for the main state machine.

The `Ptt` entity debounces the push-to-talk input, and the `Rx` and `Tx` entities generate increments for some statistics counters in the `Hci`. The `Hci` makes these available for external interrogation as well as controlling attack and decay time constants for carrier detection in the `Cd` entity, and maintaining a set of `config`uration parameters for the `Fsm`.

The `TranSec` sources cryptographic information.

The `Radio` provides the conversion between the `antenna` signal and `rf_in`/`rf_out` signals centred on an `rf_freq`. There are many alternate implementations of this functionality and so the `Radio` entity is not decomposed further in order to avoid a

specification bias to a particular approach. The abstract capabilities of a radio are therefore expressed directly as a leaf specification.

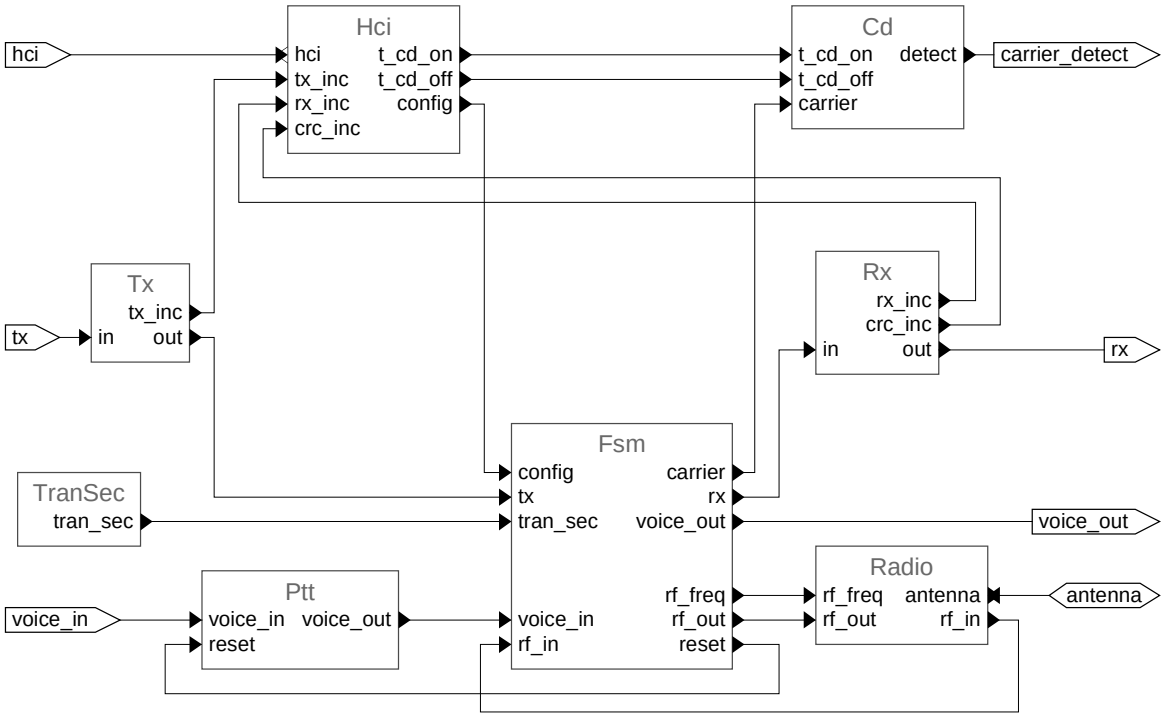

Figure 11 Physical Layer Components

3.3 Physical Layer Finite State Machine

The main physical layer FSM is responsible for selecting the major behaviour; whether to receive or transmit, and in the case of transmit, whether to transmit voice or data. The state machine shown in Figure 12 expresses this alternation of behaviours.

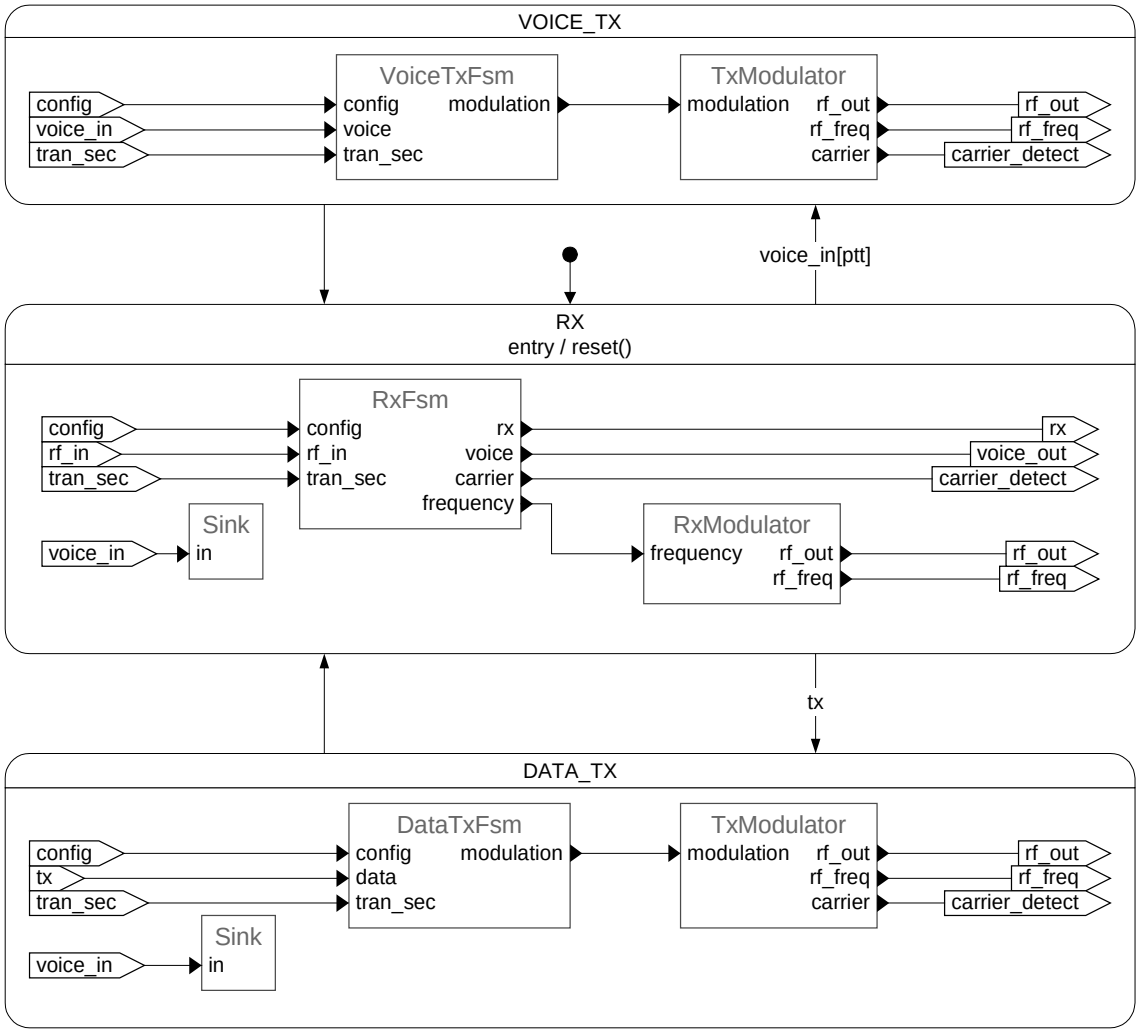

Figure 12 Main FM3TR Physical Layer State Machine

The physical layer starts in, and is normally in, the receive (`RX`) state, but switches to one of two alternate transmit states (`VOICE_TX` or `DATA_TX`) when a `ptt` (Push-To-Talk) activation occurs on the `voice_input`, or a `tx` message is available. The simplicity of the state machine clearly shows that the behaviour is half-duplex and that exit from transmission occurs on completion of the transmission; there is no pre-emption.

The behaviour within each of the three states is defined by a simple flow diagram. The two transmit states share a common `TxModulator` behaviour but require distinct `VoiceTxFsm` and `DataTxFsm` state machines to sequence the various synchronisation and information hops. In either case the relevant `TxFsm` supervises the integration of `tx` or `voice_in` data with the `tran_sec` keystream in accordance with the prevailing `configuration` parameters to produce a `modulation` control signal that is then modulated to produce an `rf_out` signal at an `rf_freq`.

The behaviour of the three states is exclusive, and well-defined at transitions, so an implementation may share the same hardware resources for `RxModulator` and `TxModulator`.

With the exception of the `Sink` entity, all other entities require further decomposition. `Sink` is a standard library utility that discards its inputs. The usage in `RX` and `DATA_TX` states requires that `voice_input` be discarded. The lack of a `Sink` on `tx`, requires that `tx` messages are buffered until they can be transmitted. The voice input is therefore only active while connected, whereas all data packets are transmitted.

3.4 Voice and Data Finite State Machines

The continued decomposition requires more space than can be justified here. It may be found in [14]. The decomposition so far has decomposed the behaviour first into the static layers and then the dynamic state that handles a complete message. Decomposition proceeds by breaking a message transmission into its pre-amble then body then post-amble phases. Each of those phases comprises a conversion between source data and source frames that are encoded, interleaved and resequenced as hops. This example continues with the conversion of the composite information packet defining the behaviour of a hop into a continuous time waveform.

3.5 Hop Modulator

Encoding and interleaving of the information for each hop are subject to two timing constraints; they cannot be started before source information is available and must be completed prior to the start of the hop. Causality imposes the early bound, a real-time specification applied to the hop in Figure 13 imposes the late bound.

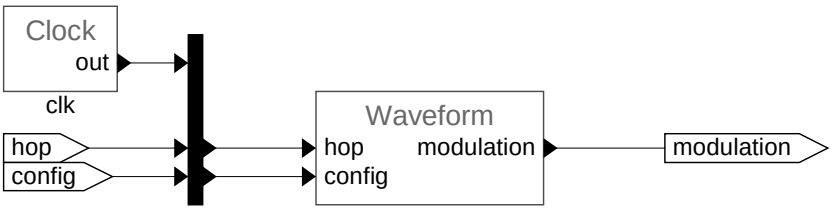

Figure 13 Hop Modulator

Precise timing information is imposed for each hop by synchronising the dynamic `hop` content with the static `configuration`. The thick bar is a library entity that performs the synchronisation. The clock source is constrained to operate at the hop rate by the annotation:

```
constraint: clk.period = config.hop_period;
```

3.6 Hop Waveform

A frequency hopper normally changes frequency between hops, and in order to avoid undue spectral splatter must modulate its amplitude as shown in Figure 14. The amplitude is reduced while the frequency synthesisers retune, and is adjusted smoothly to full amplitude before and after information transmission.

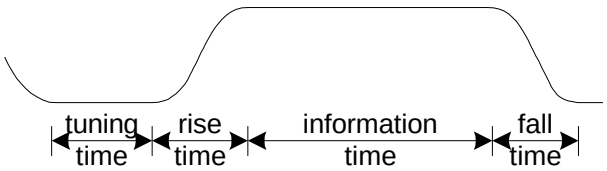

Figure 14 Hop Amplitude

The requirement for these four phases is easily specified using a state machine that sequences the four distinct operational behaviours within the hop. Each state has a behaviour that combines the relatively static `configuration` parameters with the dynamic `hop` information to produce a `modulation` control signal for a modulator.

Transition between three of the states occurs after fixed time delays. The remaining transition from the `INFORMATION` state occurs when the `InfoModulator` exits after processing all the information bits.

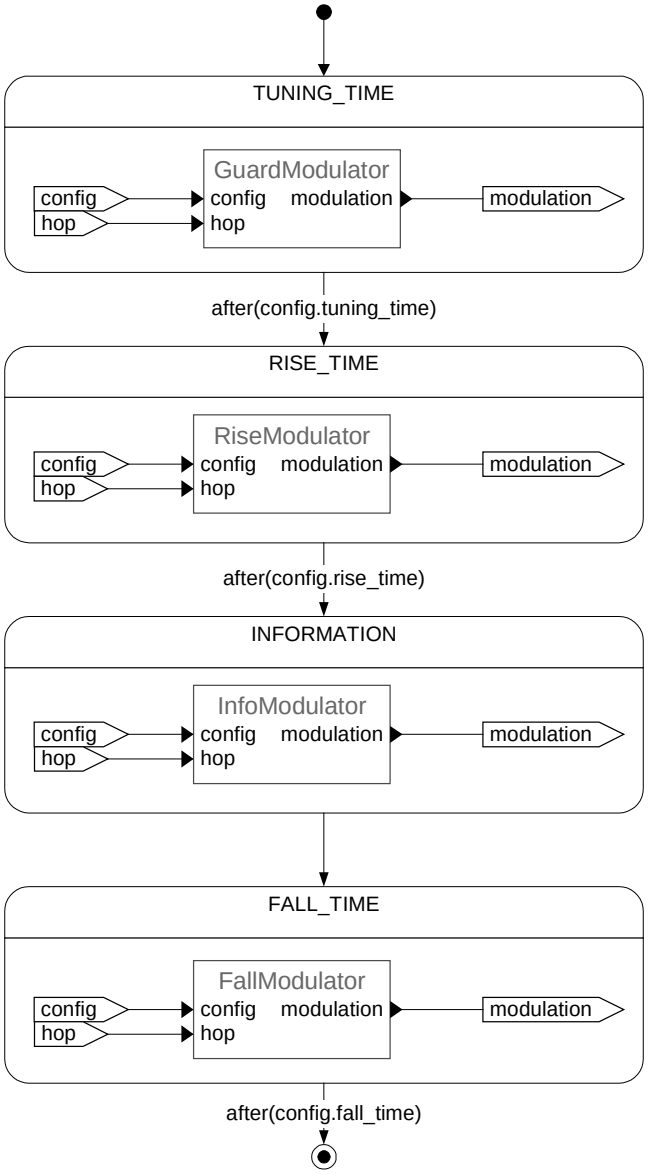

Figure 15 Hop Modulator

3.7 Rise Modulator

The rise-time behaviour is elaborated in Figure 16 and is sufficiently simple to be defined entirely by re-usable library entities.

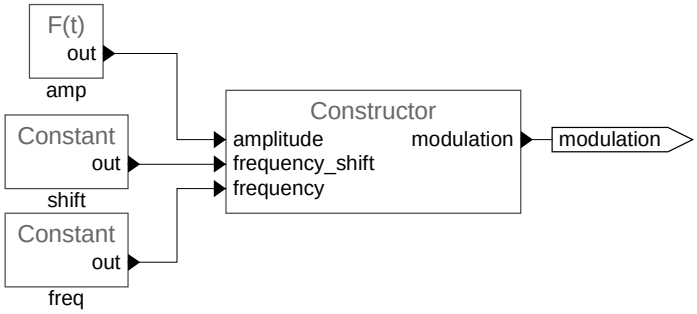

Figure 16 Rise Time Modulator

The subsequent `TxModulator` (Figure 12) is controlled by a modulation signal which

defines its amplitude and frequency. A `Constructor` is therefore used to construct the multi-field signal from its component `amplitude` and `frequency` parts. And, in order to simplify refinement of the specification to a non-zero-IF implementation, the actual frequency is composed from two parts, `frequency_shift` typically corresponding to a fixed carrier frequency, that is added to the `frequency`.

The frequency does not change during the hop and so a pair of `Constant` entities can be used to define the two part frequency, subject to the constraint that the net frequency be the required frequency:

```
constraint: shift.out + freq.out = hop.frequency;
```

and with a zero IF suggestion to define a default behaviour for a reference model simulation.

```
constraint: shift.out = range
                        {
                            value 0`Hz;
                        };
```

Zero-IF is only a suggestion, so an implementation may refine to choose an offset IF:

```
constraint: shift.out = 1.4`MHz;
```

The time-varying envelope for the rise-time is provided by the `F(t)` entity whose output is a programmable function of the time since the entity was created. The entity forms part of the `RiseModulator` entity which is created on entry to the parent `RISE_TIME` state. The time therefore starts at zero when the rise-time starts. Version 1.0 of the FM3TR specification imposes no spectral splatter requirements, so the respecification in WDL just imposes the constraint that the amplitude lie within a range bounded by a `minimum` value of 0 and a `maximum` value of 1. The suggested `value` corresponds to a raised cosine but is not mandatory.

```
constraint: amp.out = range
                      {
                          minimum 0;
                          value 0.5 * (1 - cos(pi*t/config.rise_time));
                          maximum 1;
                      };
```

3.8 Summary

The example extracts from the FM3TR respecification represent just part of the fuller decomposition [14]. In order to give a realistic perspective of a practical specification, relatively few of the details have been glossed over. The missing detail is concerned with re-usable library functionality and textual annotation; the graphics is complete.

Decomposition of a specification is restricted to the behavioural dimension and therefore corresponds to a hardware decomposition. This exposes the maximum possible degree of parallelism, since each entity could be implemented on a separate processor. This hardware perspective results in a specification that is readable and has good modularity. It is relatively easy to determine where a particular sub-specification will be found.

By way of contrast, a UML decomposition is confused by the many different implementation concerns: processor deployment, static class hierarchy and collaboration, dynamic object interaction and state change.

This is not to say that UML should be discarded. UML use cases are necessary to help clarify the customer requirements. WDL then provides a sound specification, that may eventually support direct code generation, possibly aided by UML deployment diagrams to provide a system designer with control over the transformation. In the absence of WDL translators, the WDL specification provides a much sounder basis from which to produce the requisite UML diagrams for a manual transformation.

4 Refinement to an implementation

A WDL specification is implementation neutral and so cannot be converted directly to a particular vendor's favoured architecture, since the target environment is not part of the specification. However, with the addition of sufficient further constraining specifications the conversion becomes possible.

A WDL specification may be refined into a program which is sufficiently constrained to support automated code generation in the favoured implementation style. WDL programming therefore consists of augmenting the specification with constraints rather than re-expressing that specification in code.

Refinement is not just a vendor activity. A specification may be written in relatively abstract form to exploit concepts such as modulators, interleavers and coders. This abstract specification may then be refined into the concrete specification by imposing specific behaviours. Further refinements may be made by the specification sponsor to demonstrate how non-mandatory parts of the specification, such as an acquisition algorithm could be performed. These additional refinements may support simulation of a non-real-time reference model, enabling any vendor to start exploring the specification within minutes of receipt.

4.1 Traditional Development Process

An engineering development starts when some user identifies the requirements for some product and ends when no user has any further use for the product. The initial and sustained satisfaction of those requirements is resolved by an engineering life cycle, such as that shown in very generalised form in Figure 17, with an activities workflow on the left and documents on the right.

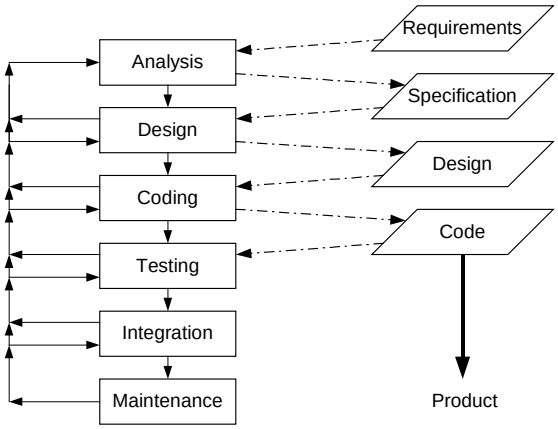

Figure 17 Traditional Development Process

The degree of activity iteration is a feature of a particular methodology, such as a strict waterfall (no iteration) or rapid prototyping (many iterations). Whatever the methodology, the requirements should be reflected by specifications that are satisfied by designs and implemented in source code. Requirements, specifications, designs and code are independent and so requirements tracking tools are needed to ensure that requirements are satisfied.

4.2 Refinement Process

The WDL concepts of an executable reference model, progressive refinement and specification domain programming support an apparently more complicated but really rather

simpler development process shown in Figure 18.

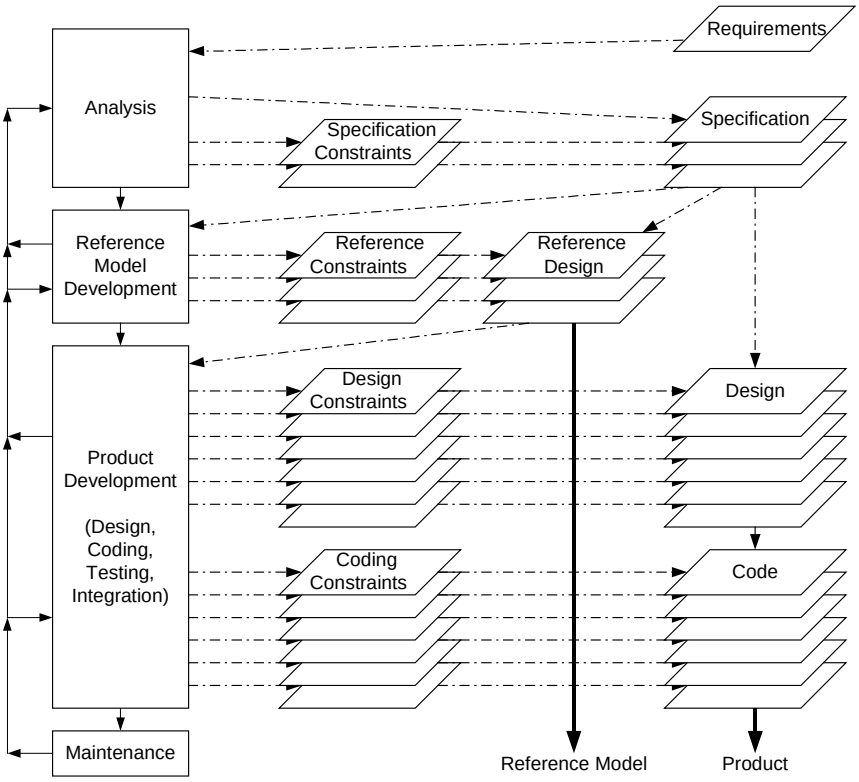

Figure 18 WDL Development using Refinement

The analysis activity is essentially unchanged, differing only through the optional exploitation of refinement to express the concrete specification as an abstract specification to which a number of layers of specification constraints are applied. The specification is then independently refined by two distinct development activities (each involving design, coding, testing and integration).

The first development is performed by the specification sponsor to demonstrate the completeness of the specification through the production of a reference model. Relatively minor refinements should be required to provide specific example solutions where only abstract behaviour had been defined. For instance, the sub-specification of a sorting algorithm cannot sensibly be implemented automatically; it would need refining to a specific algorithm such as a bubble-sort.

The second development is performed by each product vendor in order to tailor the specification to produce a superior product on a competitive hardware platform. System designers will impose constraints that resolve design decisions such as:

- architecture selection
  - select intermediate and sampling frequencies
  - select target hardware
  - partition analogue and digital parts
- system performance
  - allocate implementation (loss) budgets to subsystems
  - determine signal precisions/operating levels
- resolution of abstract and non-mandatory specifications
  - choose specific filter designs

    - select a smart acquisition strategy
- hardware and software partitioning
    - exploit pre-existing hardware devices or software libraries
    - allocate entities to processors/processes/chips

The design and coding constraints are shown separately to ease comparison of Figure 18 with Figure 17, but in reality there may be a continuum of sub-activities that impose further constraints as each implementation perspective is accommodated. This is in accord with the separation of concerns advocated by Aspect Oriented Programming [6]; an aspect such as power consumption, may contribute its constraints. An appropriate tool may then assist a designer in determining what constraints to impose.

Design decisions are often made with what experience suggests to be an adequate safety margin. There is little opportunity to revise the decisions, since the actual margin is not discovered until too late in the development. WDL provides an opportunity to achieve much more automation, so that tools may assist in resolving each of the decisions. A first pass at the entire system design activity could be completed in a few weeks, and iterated in minutes. This should allow system designers to assess the merits of alternative approaches.

The design and code comprise the specification and its refinements, rather than its re-expressions. This eliminates the development discontinuity and simplifies requirements tracing. Only the discontinuity between user requirements and specification remains. This is perhaps unavoidable since bridging this boundary involves a compromise between the often excessive and misconceived desires of a customer and the limited but defined capabilities of a product.

Incorporation of the specification as part of the final program ensures that the implementation is free from the transcription errors of an independent implementation, and more able to adjust rapidly to changes in the specification, or its refinements. Porting the specification to another platform may require substantial reworking of some of the refining constraints, but since the specification ‘code’ is preserved, the port should be substantially easier than with more conventional approaches.

Direct transformation of an appropriately constrained specification into code requires a WDL translator in addition to conventional compilers, but eliminates the need for most low level coding since the code already exists in the specification. The need for much of the conventional Coding and Testing activities is therefore removed.

A WDL specification is a top-down behavioural decomposition[3], and so the specification is always integrated. There is no need for the significant integration phase late in the development program. The Design, Coding, Testing and Integration activities are therefore merged into a single Development activity, in which increasingly detailed constraints are added to the specification until the automated translation produces the required result.

Mixed mode development presents considerable challenges using a conventional development lifecycle because the design is partitioned into distinct analogue and digital sections, which are then developed independently. In the absence of high quality mixed mode simulation tools, it is not until the integration phase that misunderstandings are discovered. The WDL lifecycle with early integration avoids this problem. A tested and specified system is partitioned by constraints into analogue and digital parts, and transformed into the product. This transformation could be fully automated, although full synthesis of analogue designs must remain a research challenge for a few years.

[3] A WDL library is a bottom-up composition of re-usable behaviours.

4.3 Automation

The WDL development process just outlined demonstrates the benefits that arise from use of the specification as the programming medium. It represents a major change in the approach to programming and requires a WDL translator to synthesise an implementation as well as a compiler to optimise it.

One day, a perfect WDL translator might need just the specification and a description of the target hardware. A more realistic translator will always need manual intervention to constrain (allocate) particular entities to suitable processors, or system design assistance to constrain (configure) a particular signal flow to be sampled with 12 bit precision at 1 MHz.

Even this more realistic translator cannot be developed instantly, so a pragmatic approach is required whereby refinements that could be determined automatically are resolved with manual assistance. The translator may then respond to very broad constraining hints corresponding to “use a shared memory buffer” for a communication path, or “use a ABC1234 ASIC” for the equaliser. Each hint would be backed up by manually implemented functionality wrapped up in a WDL interface.

In the short term, manually determined constraints can be used to overcome limitations in the WDL translation tool suite. As tools improve, many constraints can then be resolved automatically, reducing the need for constraints to those that impose major system design decisions or adjust the specification to exploit pre-existing hardware or software.

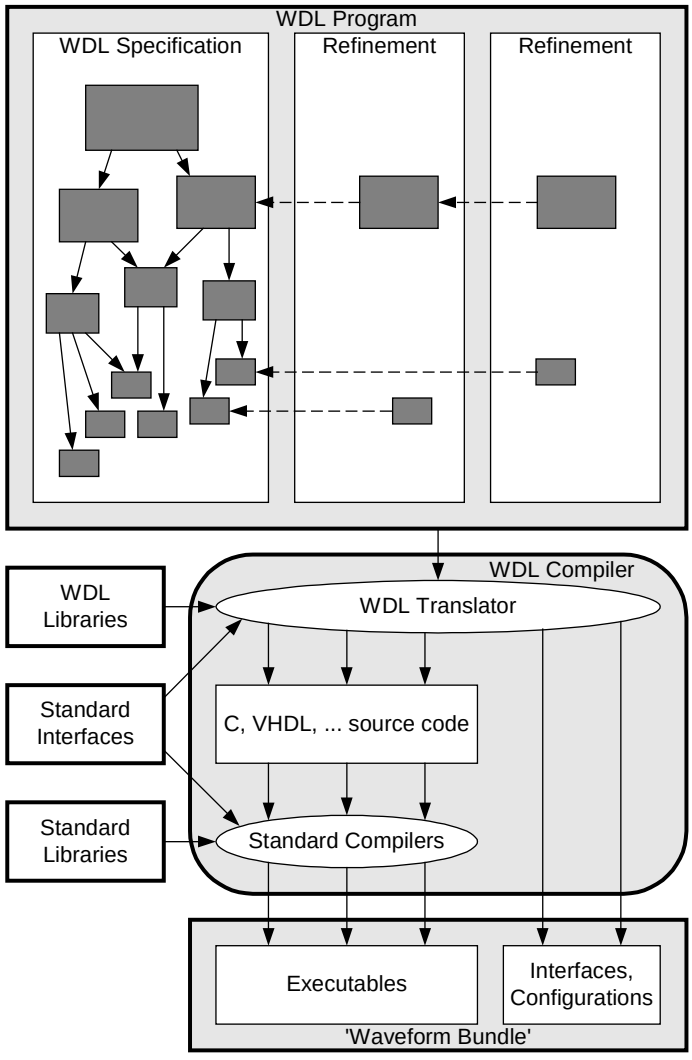

Figure 19 WDL Activities

Some constraints will always be necessary, but not all constraints are necessarily valid. A constraint that narrows the permitted behaviour is safe, but a constraint that rewrites part of the specification to exploit pre-existing functionality may not be. Tools can be developed to prove that some revisions remain compliant, and to generate test suites to verify those than cannot be proved.

The transformation of a WDL specification is shown in Figure 19. The specification is created as a behavioural decomposition down to leaf entities, which may be resolved from libraries of pre-existing specifications, or by user definition. A number of stages of refinement may be applied to the specification to constrain it sufficiently to satisfy the particular target environment. The constrained specification is a WDL Program, which may be compiled to give the appropriate combination of executable and configuration files for loading on the target platform. The WDL Compiler comprises a translator to convert the WDL to formats for which compilation can be completed by standard compilation tools.

The translator resolves leaf specifications from libraries, and interprets externally defined interfaces. This avoids the need to re-express an existing CORBA IDL interface in WDL. The subsequent compilation can make use of libraries suited to that compiler. This enables existing code bodies or ASICs to be exploited by defining a wrapper that creates a WDL interface for a non-WDL implementation.

The diagram shows the minimum tools required for transformation. Additional tools can automate generation of refinements, prove compliance of refinements and test the unprovable.

4.4 The Reference Model

Development of complicated systems is often accompanied by a simulation so that important aspects of the system behaviour can be explored before too great a commitment is made to an inappropriate implementation approach. These simulations add considerably to the design time and are costly to develop, however skimping on simulation can be counter-productive.

Using the reference model development approach of Figure 18, and with the aid of good automated code generation, the vendor may be able to re-use large parts of the reference model coding. The cost and time of the multiple exploratory or validating simulations of each vendor may then be transferred to the single reference model development by the specification sponsor. This immediately reduces the production time and cost, since a simulation is not only available but also accurate.

A WDL specification with an accompanying reference model refinement provides a complete executable behaviour, and so a producer may find that the specification supplies a high proportion of the production code. Eventually good quality code synthesis and optimisation will be available, and the producer will find that little functionality merits manual optimisation. Even with poor quality code generation, large parts of the system run at modest speed and so the automatically generated will still be of use.

It appears that the specification sponsor incurs significant additional costs through the provision of the reference model. However, these costs are offset:

It is unlikely that a specification will be approved without some form of demonstration, which can therefore form the basis for the reference model. Once effective WDL translators become available, it may well be appropriate to use WDL to develop that demonstrator. Conversion of the demonstrator to a reference model is then a matter of challenging all the unnecessary constraints.

Approval of a traditional specification involves substantial committee effort to endeavour to understand the specification and resolve the inevitable ambiguities and contradictions. The

use of WDL and an executable reference model should ensure that the standardisation committee can focus on the important functional issues.

When the specification sponsor is also the purchaser of compliant products, the additional cost of producing a better quality specification should be recouped many times over in shorter development times, increased compliance, compatibility and consequently reduced product costs.

4.5 Target Environments

The target environment does not form part of a WDL specification, so a WDL specification is portable to any target able to support the specification. Configuration of the specification to suit a particular target environment occurs through introduction of constraints during the refinement process. Partitioning constraints allocate sections of the specification to appropriate hardware or processes. The target-specific run-time support library provides the requisite scheduling primitives, either as Operating System calls, or as dedicated drivers.

A WDL specification may therefore be mapped to use an Operating System when that is appropriate or to take total control of the processor when that is permissible.

An Operating Environment such as the Joint Tactical Radio System (JTRS) OE may be exploited by constraining parts of the specification to become particular JTRS components. Code generation of these components then uses the requisite CORBA messaging and produces profiles to support application loading. WDL is therefore used to provide the portable specification of JTRS components, enabling rapid regeneration of these components on alternative platforms.

And, since a JTRS component structure is imposed on a WDL specification during refinement, the same specification can be re-used in lightweight environments where the costs of a JTRS infrastructure may be too great.

The target environment need not be an implementation environment; the reference model is supported by using Java as a simulation environment. Alternate simulation environments such as High Level Architecture (HLA) can also be supported; an HLA federate can be produced directly from the specification.

5 WDL Details

The motivation for use of block diagrams was given in Section 2, and a demonstration of their utility was provided in Section 3. In this section a little more detail is provided on how the traditional intuitive semantics of a hierarchical diagram such as Figure 20 are made rigorous, and on how state machines and blocks diagrams are treated uniformly.

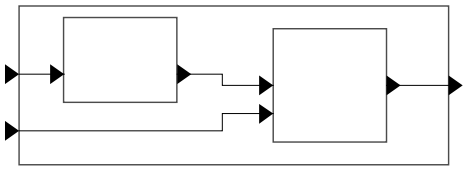

Figure 20 Composition

The communication between the entities must be definable in a way that permits the composite behaviour to be predicted. WDL uses a type system. A data type defines the information, a flow type the scheduling, and port directions the overall direction. The type system imposes many validity constraints on the specification:

- a sample value cannot be used where an Ethernet packet is expected
- a continuous signal cannot be used (directly) as a triggering event
- two outputs cannot be connected together

Manual annotation of every arc and/or port with its types would be a very onerous and restricting activity. Omitted types can be deduced. Firstly on the basis that a definition of any port or arc is sufficient to define all ports and arcs constituting a connection. More powerfully, entities may impose consistency constraints between inputs and outputs, enabling definition of just a few selected ports or arcs to ripple through the design. These consistency constraints enable re-usable library entities to adapt automatically to the types of the surrounding design. The example in Section 5.4.1 shows a subtractor that works for any subtractable data type and any consistent interaction of scheduling requirements.

5.1 Type Abstractions

Advances in type theory underpin the recent successes of Object Oriented and Functional Programming. It is therefore rather unfortunate that many of the more abstract implementation tools lack anything but the most rudimentary type system. A comprehensive type system is required to express real systems.

A typical communication protocol involves a number of alternative formats for a message packet. Each format has a number of fields, each of which has an appropriate numeric interpretation. Each packet must exhibit a precise bit layout to ensure interoperability. In C terminology, the protocol is a union of structures of built-in types or bit fields.

The WDL data type system specifies abstract requirements, where it is permissible for the compilation system to choose an optimum type, and overlays concrete requirements when there is a need to satisfy an externally imposed bit truth.

The abstract type system comprises primitive types, tailored types, enumerations, arrays, records and discriminated unions.

The built-in primitive types are of unlimited range and resolution, and so more practical tailored types can be specified by imposing constraints such as at least 8 bit precision. Rather than specifying the 8 bit precision, which may be suitable for today's microcontroller, WDL permits the more abstract intent to be captured by specifying a type with at least 40 dB dynamic range. Since most systems operate as well, if not better, with excess precision, a

compilation system is free to choose any type available on the target platform that satisfies the constraints. If none is available, then one must be synthesised.

A discriminated union has an internal mechanism to distinguish the active sub-type. This ensures that a discriminated union is a safe component of a type system.

Bit truth can be specified directly by imposing constraints, but this is not always appropriate, since re-use of a protocol such as X25 may require a different bit-truth for the new environment. It is therefore possible to provide an abstract specification of the concept of a Connection Request message and later impose the bit-truth to ensure that the implementation of the message uses the mandatory bit patterns over a particular communication path.

Consistent declaration of communication protocols using a type system introduces rigour to informal tables of bit patterns, and supports automatic generation of marshalling code for transmission, and type discovery and unmarshalling code for reception; the demodulated bit stream can be reliably interpreted as a data type without a 'cast'.

Many numbers in specifications have an associated dimension: 10 MHz, 1 ms, 5 nm. When these dimensions are interpreted manually, unfortunate consequences arise if a Centigrade specification is misread as Fahrenheit. WDL supports user-defined dimensions to eliminate this informality: `10`MHz`, `1`ms`, `5`nm`.

5.2 Scheduling Abstractions

The data type system ensures that the information content can be adequately defined. The much simpler flow type system resolves the much harder problem of specifying the scheduling: when and why information is transferred. It is the failure to define the scheduling that leaves many existing approaches informal. The need to define the scheduling and the need to allow user-selection from alternate policies will be motivated by elaborating a simple example.

Let us consider a hierarchical entity that implements a simple Boolean operation, known to mathematicians as the (material) implication operator. a $\rightarrow$ b. Engineers may be more comfortable with its equivalent expression: `!`a `&` b. The behaviour may be defined hierarchically as a composition of two other entities:

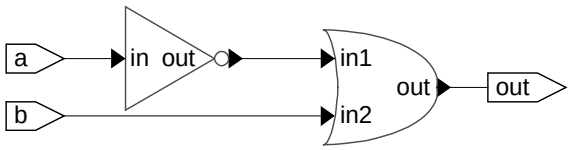

Figure 21 Implication Operator Decomposition

Its meaning seems obvious, however, brief consideration of the corresponding UML collaboration diagram will show the importance of establishing satisfactory semantics.

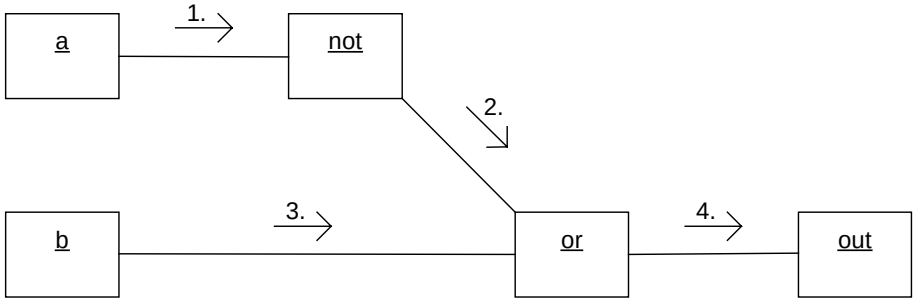

Figure 22 UML Collaboration Diagram

UML has no concept of a hierarchical port and so it is necessary to represent the external nodes as objects with informal semantics. The two messages converging on the `or` entity are not associated with the distinct input interfaces, so this aspect of the behaviour must be externally annotated. The rationale behind the sequential numbering of the messages must also be externally annotated. The UML diagram therefore fails to establish any semantics between arcs and nodes, or between arcs and arcs.

In a direct hardware realisation of the operator, separate gates (or transistors) could be provided for each of the internal nodes with the result that the output would directly reflect the state of the inputs subject to a small propagation delay. However in a software realisation, some form of sampled execution is necessary, perhaps by evaluation at regular intervals, or in response to a changed input value, or a potentially changed input value, or a potentially changed pair of input values, or a requirement for an output value.

Lee [3] has popularised the concept of a model of computation to define the composite scheduling behaviour of a block diagram. There are many, possibly infinitely many, alternate models of computation, each suited to a particular application.

The Continuous Time (CT) model of computation is well suited to dedicated hardware components, in which each arc corresponds to a physical interconnection, that may have a continuously varying voltage and current.

Detailed simulation of digital systems is more efficiently handled by a Discrete Time (DT) model of computation. Each arc again corresponds to a physical interconnection, but the simulator need only model the behaviour within the limited bounds imposed by the propagation delays; simulation of quiescent behaviour is unnecessary.

Event driven systems, such as protocol stacks, are better represented by a Discrete Event (DE) model of computation. Each arc now represents an event, and the computer need only execute in response to events; no execution is required between events.

Digital Signal Processing involves regular calculations under predictable conditions. These are well modelled by a Data Flow (DF) model of computation. The conditions are often sufficiently predictable for a much more efficient Synchronous Data Flow (SDF) model to be used [7].

Each model of computation is appropriate for a particular form of computation, and there are specialised languages and simulators optimised to particular domains. Unfortunately, real systems involve a mixture of domains, leading to interest in mixed mode simulators, since attempting to use a simulator with an inappropriate model of computation requires considerable ingenuity and achieves limited benefit.

The original version of Ptolemy (now called Ptolemy Classic) pioneered simulation with a mixture of models of computation (domains). Lack of uniformity for state machines provided one of the motivations for the Ptolemy II rewrite; domains can be freely intermixed and definitions of leaf entities can be re-used under a variety of different models of computation.

The meaning of a hierarchical diagram is therefore defined by its model of computation. In Ptolemy, a single model is defined by associating a chosen domain with the diagram. However this prevents the implementation using an alternate model, and requires unnecessary conversions when routing signals. WDL takes a more appropriate view for a specification by choosing a model of computation for each path, rather than each diagram. This is specified by a flow type and so a WDL block diagram is called a flow diagram.

A flow type expresses the way in which a flow expects to synchronise with other flows, and so enables the WDL translator to select an appropriate model of computation for the entities using the flow. Four flow types appear to be adequate: `event`, `token`, `signal`, `value`.

An `event` flow expresses an independent communication that must be handled immediately

and before all others. Events therefore observe the exclusion principles of the synchrony hypothesis [1], which prohibit two events occurring at once.

A `token` flow expresses a collaborative communication that occurs as soon as all collaborators are ready. Tokens observe the principles of data flow [7]. Token flow is therefore a secondary model of computation that defines the behaviour during the processing of an event.

A `value` flow expresses an asynchronous or non-communication. Values provide the ability to pass asynchronous configuration and break the one to one sender/receiver discipline associated with event or token flows.

A `signal` flow specifies potentially continuous communication, that may be realised by either analogue or digital signal processing in a practical implementation.

The connectivity of each flow is defined graphically by an arc connected to a number of ports. Connections normally comply with a broadcast policy with one output connected to a number of inputs. However, in rare circumstances, a more arbitrary bus policy is required between a number of bi-directional ports. A limited form of synchronisation may be obtained through the use of a master and slave ports in place of an output and inputs; this enables the master to associate replies with a query.

5.3 Unified Scheduling Model

The behaviour of an entity is defined by the way its state variables and its outputs respond to its inputs. The apparently diverse behaviour of Finite State Machines and arithmetic entities is accommodated by a unified scheduling model depicted in Figure 23 (using an informal notation).

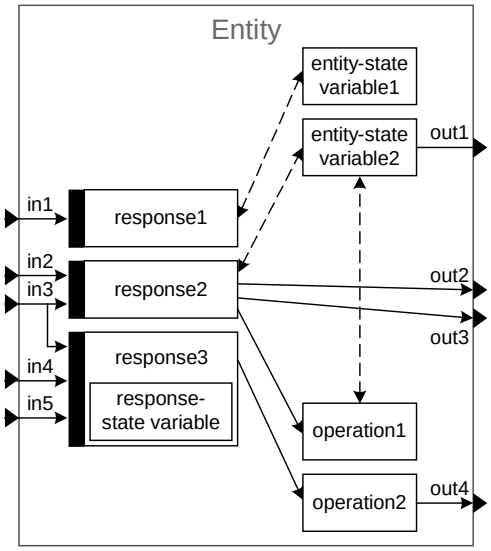

Figure 23 Anatomy of an entity

The overall behaviour of an entity is the result of one or more responses to one or more stimuli. The model shows that a response is activated (solid line) by the co-occurrence of its inputs (stimuli). Thus `response2` is activated when `in2` and `in3` co-occur.

The extended form of rendezvous that we call co-occurrence is denoted by the thick synchronisation bar at the left of each response. Co-occurrence may be simply viewed as an OR condition on `event` flows or an AND condition on `token` flows. The more complicated rules applying to `value`, `signal` and mixtures of flows are defined in [13].

Once activated, a response may access or update (dashed line) a state variable, or generate

an output message. Operations (subroutines) may be invoked to share functionality between responses. State variables may be connected directly to outputs to make their values visible but not mutable.

External integrity (encapsulation) is maintained by restricting interaction to the typed messages flowing through the ports. Internal integrity is ensured by prohibiting concurrent activation of two responses that can access the same entity state variable. Unnecessary entity synchronisation is avoided by permitting state variables to be encapsulated within a response.

With this model, a FSM comprises multiple responses, one for each possible state machine action; the FSM state is maintained as an entity state variable, whose value enables the appropriate subset of possible responses.

A simple arithmetic activity fits within this model as a response such as `response2` that generates outputs in response to inputs. An example is provided in the Section 5.4.1, with a more interesting user-defined hybrid behaviour in Section 5.4.2.

Encapsulating the scheduling of each entity internally satisfies the need for hierarchical composability, but if implemented naively would require a separate thread for each mathematical operation. A practical WDL compiler must therefore analyse adjacent flows and entity responses to identify regions that may be scheduled together, and select appropriate scheduling strategies that realise the context switches for state machines and irregular flows, using the capabilities of the target processor or operating system.

The overall behaviour is defined by the synchrony hypothesis; no two events may occur at the same time. Strict observation of this principle resolves many of the more detailed semantic issues in WDL, identify checks that a WDL translator should apply to a specification, and limit the synthesis alternatives for generation of compliant code.

The strong encapsulation provided by a WDL entity may be contrasted with the limited capabilities of an Object Oriented class. A class encapsulates its state, and provides a number of methods to operate on and with its state. Each method is activated from some calling context. The interaction between two methods is therefore dependent on the external disciplines observed by alternate calling contexts. There is no mechanism to resolve the concurrent activation from two alternate contexts. Interaction with a WDL entity must occur through its ports, and the synchronisation between different ports is defined. A WDL entity therefore encapsulates synchronisation and consequently scheduling as well as state.

5.4 Leaf Specifications

Once hierarchical decomposition ends, leaf entities are encountered. These must have a defined behaviour. Some can be provided by a support library, with built-in functionality and consequently may use an extra-lingual form of specification. However, it should be possible to create additional user-defined entities. This technique can then be used to define most of the built-in entities as well so that extra-lingual specifications are only required for a very small number of fundamental entities.

Existing tools tend to use rather clumsy approaches restricted to a particular implementation approach in a particular language. This is not suitable for a specification, and so WDL must provide an implementation neutral way of specifying leaf behaviour using a constraint language, for which space does not permit more than a couple of very simple examples.

5.4.1 Simple Arithmetic Entity

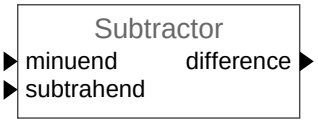

Figure 24 Subtractor Interface

The behaviour of a subtractor, whose interface is shown in Figure 24 is defined by:

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

After the pre-amble defining the context and interface ports, the response construct defines a response to be activated when `minuend` and `subtrahend` co-occur. The specification of this response requires a message at the `difference` output to have the value that is the difference between the values at the inputs. The mention of an input port accesses the value of the input port at the co-occurrence. The message is sent by construction, thus depending on the language you compare with, `difference(...)` denotes a call of, send to, or construction at the `difference` output.

This specification is highly polymorphic. It applies directly to all types for which subtraction is defined, and element-wise over arrays of subtractable types. It applies to all combinations of input flows for which a co-occurrence is meaningful. Since WDL is a specification language, it applies to all target implementation languages for which a WDL code generator is available.

5.4.2 Simple Entity with State

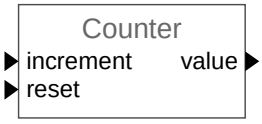

Figure 25 Counter Interface

A custom entity, such as a resettable counter, may be specified within the unified framework of Figure 23. Two responses are required: one, for the `reset` input zeroes the state variable, the other, for the `increment` input, increments the counter.

```
entity Counter
{
    in reset : void;
    in increment : void;
    out:value count = internal_counter_state;
    attribute internal_counter_state = 0 : count.type;
    response reset
    {
        specification
        {
            internal_counter_state := 0;
        };
    };
    response increment
    {
        specification
        {
            internal_counter_state := internal_counter_state + 1;
        };
    };
};
```

6 A Practical WDL Support Environment

The Waveform Description Language like any other computer language is of greater utility when associated with a suitable support environment.

The most basic level of support requires a combination of schematic and text editors to maintain a WDL specification. Some semantic validation of the integrity of the specification would clearly be beneficial.

Incorporation of an executable reference model is one of the major innovations of WDL, and so a more realistic support environment should support refinement of that reference model and simulation in Java. A more comprehensive environment would also support refinement and code generation in the wide variety of languages suitable for practical product implementations. Schedule visualisation and partitioning tools will almost certainly be needed.

If these facilities are provided in an open framework, further tools can be added to support increasing degrees of automation in the refinement and code generation process. The potential formality of WDL can be exploited by advanced tools to give stronger verification and in some cases proof that a particular implementation satisfies the original specification. Where a proof is not possible, or awaits enhanced symbolic proof technology, test vectors can be generated to challenge the candidate implementation. These test vectors may be extracted from simulation runs and so represent typical modes of operation, or may be generated by analysis of the sub-specification, and so stress boundary condition behaviour. There are no doubt many other tools that can be developed to exploit the solid specification and so assist in improving the quality of a product.

There are a number of packages that support code generation and simulation from block diagram languages, and these could be adapted to support WDL. However most packages are proprietary, and so the enhancement option is only open to that vendor. The Ptolemy II framework, developed by the University of California at Berkeley (UCB), is written in Java, is freely available and supports a Java simulation using a greater variety of models of computation than other packages. Ptolemy II supports the interleaved statechart and message flow concepts and their associated semantics. Extending Ptolemy II to support WDL is therefore a relatively modest undertaking. Indeed the flexibility of Ptolemy allows WDL to be added as an additional abstract model of computation (domain) that translates itself to a mix of the available concrete domains.

Support for WDL using Ptolemy II requires addition of some more abstract concepts at the front end of Ptolemy. Simulation of a Java reference model is already provided by Ptolemy II. Effective code generation is an ongoing activity at UCB and the University of Maryland. Development of WDL should therefore support and exploit these programmes.

7 Conclusions

As will by now be apparent to the reader, WDL brings together the good characteristics of many niche languages to create a candidate for an industry standard specification language for SDR (and indeed other applications).

Use of WDL to replace existing informal textual specifications offers the potential for specifications to improve through

- removal of ambiguities
- removal of contradictions
- addition of an executable reference model

- increased intelligibility
- increased reusability

Once tool sets are available to handle the extra stages of compilation needed to convert a specification into an implementation, products will benefit from

- reduced development times
- reduced development costs
- specification portability
- increased reliability
- increased flexibility
- greater opportunities for re-use
- greater scope for optimisation

Once tool sets are available with standard interfaces, the quality of tools may begin to improve rapidly, and libraries of reusable entities may be effectively exploited.

All of which provides the requisite flexibility to handle future requirements that involve supporting ever increasing numbers of modes of operation on both general and special purpose hardware platforms. These motives are particularly strong in the radio field, which is why much of the initial consideration has been concerned with Programmable Digital Radios or Software Definable Radios. There is however nothing uniquely special that is not applicable to another domain that involves continuous operation reacting to external stimuli.

8 Acknowledgements

The author would like to thank Thales Research for permission to publish this work, and the UK Defence Evaluation and Research Agency for funding under contract CU009-0000002745.

9 References

[1] G. Berry and G. Gonthier: *The Esterel Synchronous Programming Language: Design, Semantics, Implementation*. Science of Computer Programming 19(2): 87-152, 1992. ftp://ftp-sop.inria.fr/meije/esterel/papers/BerryGonthierSCP.pdf.

[2] J.T. Buck, S. Ha, E.A. Lee and D.G. Messerschmitt, *Ptolemy: A Framework for Simulating Heterogeneous Systems*, International Journal of Computer Simulation, vol. 4, April 1994. http://ptolemy.eecs.berkeley.edu/publications/papers/94/JEurSim/JEurSim.pdf.

[3] 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, 18(6), June 1999. http://ptolemy.eecs.berkeley.edu/publications/papers/99/starcharts/starcharts.pdf.

[4] D. Harel, *StateCharts: A Visual Formalism for Complex Systems*, Science of Computer Programming, vol. 8, 231-274, 1987.

[5] D. Jackson, *Alloy: A Lightweight Object Modelling Notation*, July 2000. http://sdg.lcs.mit.edu/~dnj/pubs/alloy-journal.pdf.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J-M Loingtier and J. Irwin. *Aspect-Oriented Programming*. Proceedings of the 11th European Conference on Object-Oriented Programming(ECOOPÕ97), LNCS 1241, Springer-Verlag, June 1997 http://www.parc.xerox.com/spl/groups/eca/pubs/papers/KIczales-ECOOP97/for-web.pdf

[7] E.A. Lee and D.G. Messerschmitt, *Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing*, IEEE Transactions on Computers, 36(2), February 1987.

[8] G.H. Mealy, *A Method for Synthesizing Sequential Circuits*, Bell System Technical Journal, vol. 34, no. 5, 1045-1080, September 1955.

[9] E.F. Moore, *Gedanken Experiments on Sequential Machines*, In Automata Studies, Princeton University Press, 1956.

[10] A. Olsen, O. Faergemand, R. Reed, J.R.W. Smith, B. Møller-Pedersen, *Systems Engineering Using Sdl-92*, North Holland, 1994.

[11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[12] J. Rumbaugh, I. Jacobson and G. Booch. *The Unified Modelling Language Reference Manual*. Addison Wesley, 1999.

[13] E.D. Willink, *Waveform Description Language*, Thales Research Report, P6957-11-014, April 2000. http://www.computing.surrey.ac.uk/Personal/pg/E.Willink/wdl/documents/Language.pdf.

[14] E.D. Willink, *FM3TR Decomposition*, Thales Research Report, P6957-11-05, April 2000. http://www.computing.surrey.ac.uk/Personal/pg/E.Willink/wdl/documents/Fm3tr.pdf.

[15] E.D. Willink, *Waveform Description Language : Moving from Implementation to Specification in Ptolemy II*, Ptolemy Mini-Conference, March 2000. http://ptolemy.eecs.berkeley.edu/conferences/01/src/miniconf/27edwillink.pdf http://www.computing.surrey.ac.uk/Personal/pg/E.Willink/wdl/documents/PtolemyMiniConference.ppt.