THE WAVEFORM DESCRIPTION LANGUAGE : MOVING FROM IMPLEMENTATION TO SPECIFICATION

E.D. Willink,
Ed.Willink@rrl.co.uk

Thales Research Ltd, Reading, England

ABSTRACT

Many current research and development activities make significant contributions to the quality of some particular implementation approach. In this paper we describe the Waveform Description Language, in which the best characteristics of a variety of distinct programming approaches are exploited so that standard implementation domain practices can be applied in the specification domain. A single WDL specification may be refined to support semi-automated conversion to a variety of implementations. A WDL specification avoids the ambiguities and contradictions characteristic of many conventional specifications with an underlying formality that remains accessible and familiar to programmers.

THE SPECIFICATION PROBLEM

Technology advances hand in hand with competition between products that exploit that technology. This competition is only possible when the expected behaviour of the alternative products is understood. For sophisticated technology, this expectation is defined by system specifications.

System specification presents considerable challenges, that are usually resolved by fairly large specification documents. The size of these documents can be reduced by attempting to provide concise specifications, but this tends to cut out useful editorial descriptions and may leave some aspects of the system unspecified. These hazards of conciseness may be avoided by providing extra explanations and occasionally repeating parts of specifications to reduce the need for cross-referencing. Unfortunately, repetitions tend to introduce contradictions between the specifications and the clarifying descriptions. A concise specification is therefore 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.

These problems can be resolved, in principle, with the aid of mathematics. 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 (WDL) [1] 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 machines 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.

WDL OVERVIEW

WDL 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 programmers, but has the underlying rigour needed for a useful specification.

The graphical concepts of block diagram languages established by tools such as Ptolemy, COSSAP or SPW and of hierarchical state machines such as Argos, SDL or UML [2] are combined. The unifying principles of synchronous languages such as Esterel [3] underlie the ability to arbitrarily intermix state machines and block

diagrams within a decomposition hierarchy [4].

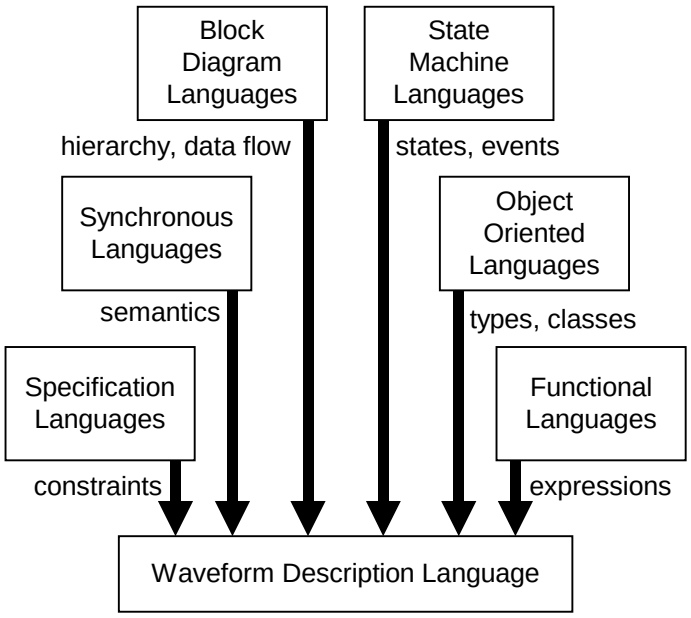

Figure 1 WDL Antecedents

Effective communication is supported by applying Object Oriented principles to declare data types. Definition of leaf behaviour borrows constraint concepts from specification languages such as Alloy [5] and imposes the discipline of functional languages to ensure referential transparency.

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 in UML at the implementation level.

EXAMPLE

An indication of the style of a WDL specification may be obtained by considering a simple example. The example shows the fourth (of ten) levels of decomposition of part of a Radio. The full decomposition may be found in [6].

Each level of decomposition specifies the behaviour of a single entity, which has an external interface shown graphically in Figure 2.

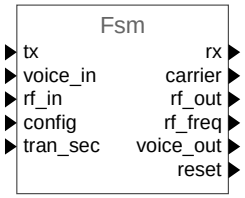

Figure 2 Example Entity Interface

The entity is concerned with data and voice transmission. Either a `tx` message or `voice_in` may activate `rf_out`. `rf_in` may activate an `rx` message or `voice_out`. The `rf_freq`uency is determined by a combination of the system `config`uration and `tran_sec` encryption data. `carrier` detect and `reset` interact with other protocol layers. The informal description above provides a context for this isolated example. The full specification defines the data and flow types for each input and output message flow.

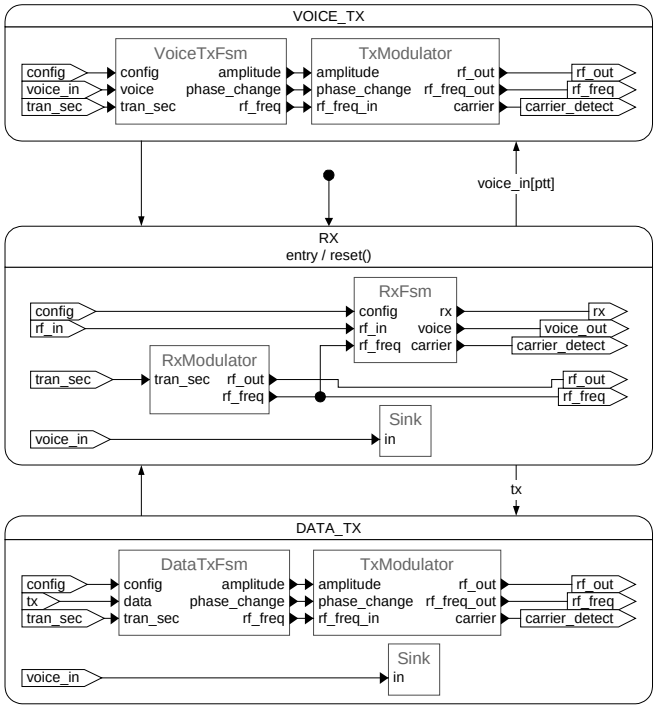

Figure 3 Example Entity Decomposition

Each level of hierarchical decomposition may use a message flow, a state machine, or a nested combination. A leaf entity may be directly specified in text. The example in Figure 3 shows a UML state chart with three states, each of which is defined by a nested message flow diagram. The state machine shows that the entity starts in the receive (`RX`) state and advances to either voice transmission (`VOICE_TX`) or data transmission (`DATA_TX`) in response to a `voice_in` [with push-to-talk asserted] or a `tx` trigger. It stays in the relevant transmit state until transmission completes and receive resumes. The system is therefore half-duplex with no pre-emption or premature termination.

The behaviour of voice and data transmission differs through the use of a distinct `VoiceTxFsm` and `DataTxFsm` state machine, but shares the common `TxModulator` functionality.

The transition between states is well defined. There is no requirement for concurrent operation of `RxModulator` and `TxModulator`.

With the exception of the `Sink` entity, all other entities require further decomposition. `Sink` is a standard library entity that discards its inputs. The usage requires that voice input during receive and data transmit be discarded. The lack of a `Sink` on `tx`, requires that `tx` messages are buffered until they can be transmitted.

SCHEDULING ABSTRACTIONS

Any programming methodology relies on an underlying model of computation [4]. Some methodologies support just a single model such as Synchronous Data Flow, Discrete Events or Discrete Time. This gives good performance in the intended domain but can cause problems when one model is chosen for the more challenging subsystems, although another is necessary for the overall system. The Ptolemy system [7] supports a mix of models of computation by partitioning the design into domains within which a single model applies. This provides the required flexibility, but the selection of a particular domain is an implementation decision.

WDL must provide a more abstract choice to suit the specification perspective. A separate flow type (model of computation) is specified, or deduced, for each message flow (communication path). Each entity is responsible for its own scheduling, and rendezvous-like principles determine when and how each entity should respond to its input flows. We use the term co-occurrence for this extended form of rendezvous.

Since scheduling is encapsulated within entities, entities decompose satisfactorily into entities that also encapsulate their scheduling. This contrasts with the most similar UML diagram; in a UML collaboration diagram the scheduling is an annotation to be observed by a scheduler external to the diagram. The UML diagram therefore fails to decompose into sub-diagrams.

Four scheduling flow types appear to be adequate: event, token, signal, value.

Events observe the exclusion principles of Synchronous Reactive computing [3], so no two events can occur at once.

Tokens observe the principles of data flow [8] and as much token flow activity as possible occurs in response to each event. Token flow is therefore a secondary model of computation within the primary Synchronous Reactive model.

Values provide the ability to pass asynchronous configuration and break the one to one sender/receiver discipline associated with event or token flows.

Signals specify potentially continuous time signals, that may be realised by either analogue or digital signal processing in a practical implementation.

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.

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.

WDL provides a comprehensive type system to express both the abstract requirements on a flow, where it is permissible for the compilation system to choose an optimum type, and the concrete requirements where there is a need to satisfy an externally imposed bit truth.

The abstract type system comprises primitive types, arrays, records and discriminated unions; a discriminated union has an internal sub-type identifier, which ensures that a discriminated union is a safe component of a type system. It is an almost essential part of a complete type system, enabling a message protocol to be modelled using a record for each possible message format, and a discriminated union for the type of all possible messages. A discriminated union provides an ‘OR’ type constructor to accompany the ‘AND’ provided by a record.

The built-in primitive types are of unlimited range and resolution, and so more practical 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. In either case, 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.

Bit truth can be specified directly by imposing constraints, but this is not always appropriate, since a protocol such as X25 may need to observe different bit-truths in different environments. 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 in the target application.

CONSTRAINTS

Leaf entities are specified using a constraint language:

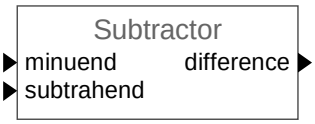

Figure 4 Leaf Entity Interface

The behaviour of a subtractor, whose interface is shown in Figure 4 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.

REFINEMENT TO AN IMPLEMENTATION

Some considerations in the conversion of a WDL specification to a practical implementation have already been alluded to. These considerations involve imposition of practical constraints or identification of effective scheduling strategies. They are normally resolved as skilled manual contributions to the final implementation.

A WDL specification is implementation neutral and so cannot be converted directly to a vendor's favoured (e.g. four processor) architecture. However, WDL is more than a specification language. WDL supports refinement through the addition of multiple layers of further constraints to push the specification in the required direction.

A perfect WDL translator might need just a description of the target hardware. A more realistic translator will need some system design assistance to constrain a particular signal flow to be sampled with 12 bit precision at 1MHz, or to allocate particular entities to suitable processors.

A relatively lightweight refinement should be provided as a non-mandatory part of any WDL specification. This should demonstrate a possible implementation for those parts of the specified system whose behaviour is not otherwise specified. The acquisition algorithm for a radio provides a typical example of behaviour that must be defined before a system can work, but which is omitted from conventional specifications. This lightweight refinement should support simulation and instrumentation of a reference model.

More intensive refinements can be provided by a vendor to constrain the specification to work on a particular processor configuration, with specific sampling rates and bit precisions. A refinement may introduce further or alternative decompositions, or even recompositions to adjust interfaces to exploit any pre-existing ASICs or Intellectual Property.

The usage of WDL is summarised in Figure 5. A WDL specification is created as a behavioural decomposition down to leaf entities, which can be user-specified if there is no appropriate entity available in an existing WDL library. One or more stages of refinement are applied to the WDL 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.

A refinement changes the compiled program, but does not change the specification. This approach therefore supports progressive development, starting with the lightweight reference model simulation that may be gradually refined to impose practical rate and precision constraints and then process/processor allocation constraints, until the program is fit for conversion to the target environment.

Since the specification is incorporated in the final

program, 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.

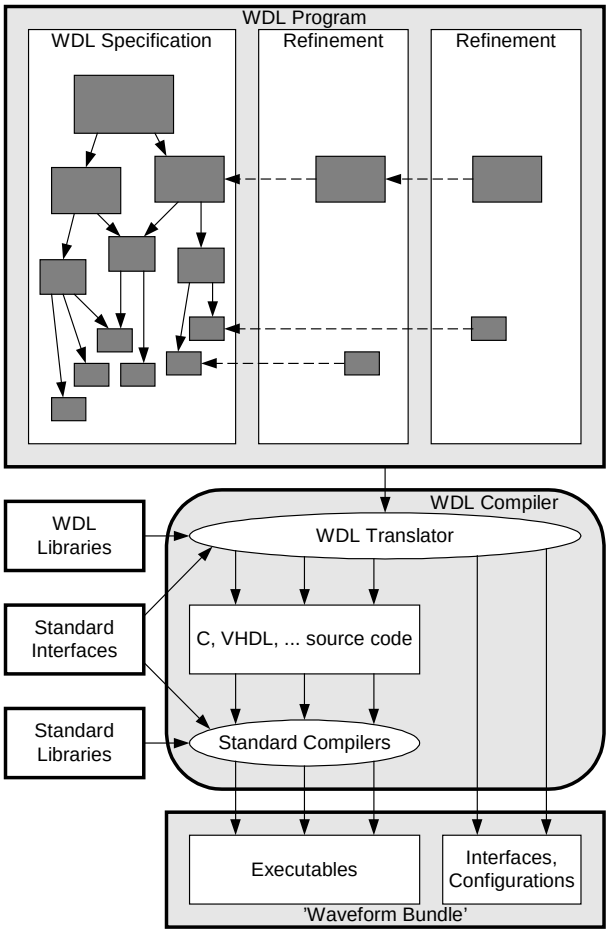

Figure 5 WDL Activities

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.

SUMMARY

WDL brings together the good characteristics of many niche languages to create a candidate for an industry standard specification language.

Once WDL is used to replace existing informal textual specifications, specifications will 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.

REFERENCES

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

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

[3] 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.

[4] 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

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

[6] E.D. Willink, *FM3TR Decomposition*, Thales Research Report, P6957-11-005, April 2000. http://www.ee.surrey.ac.uk/Personal/E.Willink/wdl/documents/Fm3tr.pdf.

[7] 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.

[8] 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.