Project: Formal Meta-Modelling for Tool Development
Document Reference: CS-11-01

CASE Technical Report: Automated Transformations from VHDL to CSP

James Sharp
Helen Treharne
Department of Computing,
University of Surrey, UK

March 2, 2011

Table of Contents

1 Introduction

This work is being carried out as part of a collaborative PhD with AWE plc.

1.1 Motivation

Manufacturing hardware can lead to costly recalls and potentially dangerous device failures when the hardware design used for a system contains a flaw. In this report we focus on using the Very-High-Speed Integrated Circuit (VHISC); VHISC Hardware Description Language (VHDL) [32] to describe and simulate hardware specifications.

The ability to verify the behaviour of a hardware design prior to its implementation has the potential to highlight problems in the design and help to ensure that the hardware system behaves in a manner which meets the system requirements and furthermore does not produce any unwanted behaviour.

Formal methods can be thought of as providing rigour to the design of a system. Model checking, one of the approaches in formal analysis, explores

the entire state space of finite state space systems from all possible paths. By using a formal language such as Communicating Sequential Processes (CSP), [11], it is possible to verify a system such that we can ensure that certain properties hold and check for unwanted behaviour. By translating VHDL descriptions into CSP it is possible to verify hardware designs prior to their implementation.

1.2 Purpose

The purpose of this document is to provide detail on the transformation framework created to support a transformation from VHDL to CSP using meta-modelling tools. The transformation framework was built as a tool during the 6 month placement at AWE from August 2010 to February 2011. This document provides background knowledge of the source and target languages used in the transformation framework, VHDL and CSP, as well as those languages used in transforming VHDL to CSP, EMFText [10, 9] and the Epsilon Transformation Language (ETL) [17]. This document also discusses in the mapping from VHDL to CSP and the transformation framework required to support an automated transformation from VHDL to CSP.

1.3 Scope

This document assumes that the reader has a basic understanding of both the source language, VHDL, whilst a brief overview is provided it is not the intention to go into the details of the language, the same applies for the target language, CSP. The document also assumes that the reader has a general understanding of concept of meta-modelling, and will go into depth in the use of both Concrete Syntax Mapping and Model Transformation.

1.4 Overview

In Section 2 an overview of the source and target languages is given, following this, Section 3 and Section 4 go on to describe the meta modelling transformation languages which will be used throughout the rest of the report. Section 5 discusses the VHDL to CSP || B mapping on which the work detailed in this document is based, and Section 6 proceeds to detail the mapping rules from VHDL to CSP which the model transformation will perform. Section 7 provides an overview of the transformation framework which has been created and details the 3 distinct aspects of the transformation from VHDL to CSP. The Concrete Syntax Mappings for VHDL and CSP are then discussed in Section 8 before Section 9 then goes on to describe

the ETL transformation which has been developed for the model-to-model transformation between VHDL and CSP. This document is then concluded in Section 10 where the work is reviewed.

2 Source and Target Languages

In this section we give a brief introduction to the formal analysis language, Communicating Sequential Processes (CSP). We then go on to give an overview of the hardware description language, Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). This overview of the source language (VHDL) and target langauge (CSP) is intended to introduce the reader to the languages being used within the meta model transformation discussed later in this report.

2.1 Formal Analysis using CSP

Communicating Sequential Processes (CSP)[11, 22, 23] is a process algebra for defining the control flow of systems which interact with their environment by communication. In this section we present the CSP operators used in this report.

$$P ::= \mathit{Stop} \mid \mathit{Skip} \mid a \rightarrow P \mid P \,\Box\, Q \mid P \sqcap Q \mid \Box_{a \in A}\ a \rightarrow P$$

$$\mid a?x!y \rightarrow P(x) \mid \mathit{if}\ a\ \mathit{then}\ P_1\ \mathit{else}\ P_2$$

$$\mid P \setminus A \mid P \triangle Q \mid P \underset{A\,B}{\|} Q \mid P \underset{A}{\|} Q \mid P \,|||\, Q$$

Stop describes a termination operator which ensures that a process ceases to perform internal or external communications and events. *Skip* describes the successful termination of a process, it is in effect *Stop* followed by a $\surd$. The process, $a \rightarrow P$, is prepared to engage in the event a and immediately after will behave as process P.

The choice process, $P \,\Box\, Q$, provides an external choice between process P and process Q, if the environment is willing to perform process P but not Q then the choice is resolved in favour of P. Conversely, $P \sqcap Q$ is an internal choice between P and Q.

The operation $\Box_{a \in A}\ a \rightarrow P$ presents an external choice of event a, where a is drawn from a set A, and will immediately after behave as process P. Channels are able to receive and transmit values; channel a is said to accept an input x and is able to transmit a value y. The operation $a?x!y \rightarrow P(x)$ will transmit value y (a previously known value) and accept a value x, the subsequent behaviour of P may depend on x.

The *if then else* operator (*if a then* P_1 *else* P_2) is a conditional branching operator. The choice of branch is determined by the boolean expression a, if the expression evaluates to true then the branch P_1 will be performed, otherwise P_2 will be performed. From [22], "The guard construct $a \,\&\, P1$ is used as a convenient shorthand for *if a then* P_1 *else Stop*".

We have covered the CSP sequential operators so far, the operators in the third line of the grammar given are the CSP compositional operators, also thought of as the high level operators.

$P \setminus A$ denotes a process P with the events in set A hidden. These hidden events whilst still being performed will no longer be visible and therefore synchronisations cannot occur on these channels, and likewise these events will not be displayed in traces. We will explain the trace semantic model below.

$P \triangle Q$ denotes an interrupt to an event, where the interrupt allows a different process Q to take control regardless of the current position in the process P; it does not necessarily mean however, that the process Q will be entered.

CSP provides an operation for creating synchronised parallel combinations, in particular a binary parallel combination $P \underset{A\,B}{\|} Q$ where the events in the set $A \cap B$ must synchronise between P and Q; this form of parallel composition is know as alphabetised parallel. If $A \cap B = \varnothing$ then processes P and Q are said to be disjoint and will therefore not synchronise on any events; this is equivalent to the behaviour exhibited by interleaving P and Q, $P \,|||\, Q$.

There exists another operation for defining a parallel composition, interface parallel. Interface parallel, defined as $P \underset{A}{\|} Q$, synchronises the processes P and Q on the event set A. If there exists an event in the alphabet A which occurs in P but not in Q then due to the definition of a shared interface parallel, the parallel composition will deadlock.

2.2 VHDL

As digital systems have become more complex, the requirement for machine readable hardware description languages which are also human readable has become crucial. As a result Very-High-Speed Integrated Circuit (VHSIC) Hardware Design Language (VHDL) [30] [32] was introduced, and since it's introduction in the 1980's it has become an IEEE standard for Application Specific Integrated Circuit (ASIC) and Field Programmable Gate Arrays (FPGAs) design and development.

VHDL describes a piece of hardware in terms of the high and low (or '1' and '0') signal inputs and outputs, and provides a simple, yet expressive language for defining the processing of the input signals required to give the correct output signals. Within the scope of our research, we will only be interested in a subset of IEEE standard synthesizeable VHDL [12].

VHDL descriptions have two aspects, the *entity* and the *architecture*. The *entity* defines the *ports* of the hardware system and the type of each *port*, be it of type *bit* or *bit_vector*. Furthermore each *port* has an associated direction in the *entity* description: *in*, *out* or *inout* (*inout* is outside of scope of our current work). In summary, the *entity* of a VHDL description defines the inputs and outputs of a hardware component and their associated types; an *entity* can be thought of as the (Java) Interface for a VHDL description.

A VHDL description also provides the behaviour of the hardware component in the form of an *architecture*, whilst the VHDL standard allows for multiple *architectures* within one VHDL description, we will only focus on there being one *architecture* to a VHDL description. An *architecture* may define *signals* (internal *ports*), for holding information and for providing communication within the behavioural description. These *signals* are defined in the same way as an $entity's$ *port*, however a *signal* may also take on a user defined data type. For instance a user defined data type *STATE* can contain for different possible values A, B, C and D, a user defined data type is declared within an *architecture*.

An *architecture* my contain multiple *processes*, each *process* is executed when one or more of the values change in the $process's$ *sensitivity list* change. The *sensitivity list* defines the *ports* and *signals* to which a VHDL *process* should react, and so a *process* must monitor these *ports* and *signals* for changes.

Processes may use several known programming functions to aid in the behavioural description of the hardware component. We focus on the use of *if* statements, *case* statements and *when* statements in our current work. There also exist the ability to perform other program functions such as loops, but this is out of scope.

Signals and *ports* can be assigned a new value based on the outcome of a boolean check performed by and *if* or *when* statement or arbitrarily based on a branching made by a *case* statement.

One special input *port* that is commonly found in VHDL descriptions is known as the *clock*. The *clock*, as the name suggests, provides a constant alternating signal which moves between high and low (0 and 1) and is used to signify the timing of a hardware device.

The use of boolean operations (*OR*, *AND* and *NOT*) are available for use within the VHDL language as well as the more complex hardware operators *XOR*, *NOR* and *NAND*. There is also the provision of some VHDL specific functions to aid in the behavioural description of a hardware component. One such function which aids in *signal* change detection is the function *rising_edge*(). This determines if the change in a *signal* or *port* was from a low edge to a high edge and will respond with a suitable boolean result (true or false).

A simple VHDL hardware description for a three state monitor which outputs the completion as well as the current activity status of a sequence of inputs is shown in Figure 1 and the VHDL I/O diagram for this example is visible in Figure 2.

```
entity and2 is
    port( in1, in2 : in bit;
          result : out bit);

architecture rtl of and2 is
begin
    result <= in1 and in2;
end rtl
```

Figure 1: A simple VHDL hardware description

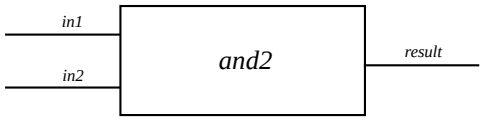


Figure 2: A simple VHDL hardware I/O diagram

3 Meta Modelling

In this section we discuss the approach and languages required for transforming one language into another. We first introduce model transformation, in particular we describe the different aspects to the model transformation language, the Epsilon Transformation Language (ETL). We then discuss textual modelling; the translation of a source Domain Specific Language (DSL) into a model representation of that DSL and some of the tools available for performing this translation.

3.1 Background

Model Driven Engineering (MDE) is a well used design method which focuses on the use of visual representations of a system and a high level of abstraction to aid in its development. In order to meet the required development timescales new methods have been developed to aid in the transition between design and implementation. Once such method is know as meta modelling, or model to model transformations.

Model to model transformations takes a source model which meets the criteria set out in a meta model and with the use of a set of rules derived in a transformation language, transforms the model into a target language. Whilst this concept can be applied within a Rapid Application Development scheme it also introduces the ability to produce automated translations from one language to another.

In this section we first discuss the two main methodologies of model transformation, relational and graph-transformation approaches, as well as highlight some of the meta modelling languages available. We then review some of the work which looks at the validation of model transformations.

3.2 Meta Model Transformation Languages

There are many different languages and techniques available for use in the meta- modelling realm. There are relational approaches such as the Object modelling Group's (OMG) Query/View/Transformation (QVT) and there are graph transforms which adopt an operational approach. In [18] these two approach are discussed and whilst there is no clear suggestion that one approach is better than the other, the strengths and merits of each approach are highlighted.

Languages such as GReAT [3] [20], Triple Graph Grammars [24] and USE [6] use graph transformations to provide clear rule specifications which can be easier for user implementation. However, languages such as MOMENT2-GT [4] [4] and Query/View/Transformation [21] adopt a relational approach, providing a cleaner method for bidirectional transformations. There are other languages, such as the one discussed in Section 3.3.2 which is a hybrid of these two transformation approaches.

In [31] Varro *et. al* reviews many of these different modelling languages and tries to assess them on speed, flexibility and additional features which they may offer, for instance Triple Graph Grammars provide bi-directionality of model transformations (from model A to B and B to A).

Whilst many of the model transformation languages have chosen to perform transformations from UML State Machines or Activity diagrams to CSP; the aim has not been for formal analysis of the models. Rather the transformation to CSP has been chosen due to CSP's clean, small grammar which results in simple transformations which are suitable for demonstrating tool designs.

3.3 Epsilon

The Extensible Platform of Integrated Languages for mOdel maNagment (Epsilon) Suite [16] comprises a series of task specific languages. Each of these languages is derived from a core language, the Epsilon Object Language (EOL), which is discussed in Section 3.3.1.

There are several task specific languages within the Epsilon suite, however we will focus on the Epsilons Suite's transformation language: the Epsilon Transformation Language (ETL) which is discussed in Section 3.3.2.

Other key languages within the Epsilon Suite are the Epsilon Generation Language (EGL) which enables the generation of text (or source code) from a model, and the Epsilon Verification Language (EVL) which is used to provide verification of transformations, inter-model element matching and model to text generation.

The other languages available within Epsilon, that we do not discuss here, include: the Epsilon Comparison Language, the Epsilon Merging Language, the Epsilon Wizard Language and Flock; more information on these can be found in [15].

3.3.1 Epsilon Object Language

The Epsilon Object Language (EOL)[15] [28] forms the base syntax with which all other languages within the Epsilon suite are derived. EOL has been designed with the idea of managing models at a high level of abstraction whilst providing significant expressive power.

The syntax of EOL, according to the creators, can be thought of as a Javascript and OCL like language[16]; however, EOL does allow the user to apply elements of Java as well. The format of an EOL operation is similar to that of a Java operation, declaring input parameters, a return value and the operation name:

operation $\langle$*return value*$\rangle$ *operation name*() : $\langle$*parameters*$\rangle${}

The basic Java-like programming functions are available within EOL such as: *if* and *switch* statements as well as *for* and *while* loops. Additionally the $\rightarrow$ function has been lifted from OCL in order to allow quick referencing and access to set information within a meta model. Likewise the necessity to distinguish between a model and a model element has meant the the ! from ATL has been implemented so that the model element A of model Ma can be referenced as $Ma!A$. Furthermore EOL enables the use of pre and post conditions for operations.

Finally some other useful functions which have been included within EOL to aid in meta model traversal and manipulation include a *depth* function, enabling the ability to navigate through a model to child elements. Also a *transaction* statement which provides the ability to roll back to a start state if a transaction must be aborted. EOL also equips the user with the ability to cache operation results which can be highly useful during large meta model translations. Further details on EOL can be found in the Epsilon Book [15].

3.3.2 Epsilon Transformation Language

As previously mentioned in Section 3.2, there are imperative, declarative and hybrid transformation languages; the Epsilon Transformation Language (ETL)[17] [15] [28] is a hybrid language. ETL provides a task-specific rule execution scheme but also allows for the imperative features within EOL to be used for transformation rules.

An ETL file consists of a *module* which contains transformation *rules*, each *rule* must have a unique name (with respect to the *module*) and operates on one *source* element that can be transformed into many *target* elements.

A transformation *rule* is defined as one of three *keyword rules*: *abstract*, *lazy* and *primary*. A rule can also extend another, using the *extends* attribute. Transformation *rules* may also contain *guards*, which are defined in EOL, that restrict the *rules* applicability to the *source* model elements.

The structure of each transformation *rule* requires a keyword to aid in its definition, as well as a *transform* keyword which must list the *source* keyword and the *to* keyword that declares one or more *targets*. The *extends* keyword can then be used if necessary to define a comma-separated list of *rules* to which the current *rule* extends, enabling further detail to be given to pre-defined *rules*, or *abstract rules*. (This is similar to the extends functionality in Java.) If required a *guard* can then be used which defines an EOL block, and finally the body of the rule is specified as a sequence of EOL statements. The structure of a rule is illustrated in Figure 3.

• $(primary)$

rule $\langle name \rangle$
    transform $\langle source\ parameter \rangle : \langle source\ parameter\ type \rangle$
    to $(\langle target\ parameter \rangle : \langle target\ parameter\ type \rangle, ...,$
        $\langle target\ parameter_n \rangle : \langle target\ parameter_n\ type \rangle)$

    extends $(\langle rule\ name \rangle, ..., \langle rule\ name_n \rangle)\{$

        (**guard**($\{EOL\ statements\}$)

          EOL transformation statements

    }

Figure 3: ETL transformation *rule* structure

ETL offers the ability to implement *pre* and *post* conditions which again are defined using the EOL syntax. These *pre* and *post* conditions are fired before and after *rule* execution respectively; these are written simply within the ETL *module* as described in Figure 4.

(**pre**| **post**)$name\{$
    EOL statements
}

Figure 4: ETL *pre* | *post* conditional block

The ETL syntax allows for the use of an *import* function which inherits all *rules* and *pre/post* conditions from the imported module. As with Java, the local *module* is able to perform overriding on any/all aspects of the imported *module* as required.

The transformation rules within an ETL *module* fire in a specific order. Firstly all *pre* conditions specified within the *module* are fired, as well as those within any imported *modules*. Then all the *rules* which are not *abstract* or *lazy* are fired, assuming they have applicable elements in the *source* model and that the *guards* are met. *Lazy rules* must be called from within other *rules*, and *abstract rules* are the basis for *rules* which use the *extends* clause; both of these *rule* types are not fired directly. Finally all *post* conditional blocks are fired.

ETL also offers the ability to define *operations*, which are imperative blocks of EOL that can be written to aid in the model transformation process. These *operations* can take multiple inputs as well as provide return values, furthermore they are able to create new instances of model elements. This provides the freedom to choose between imperative *operations* and the more declarative *lazy rules* when defining a transformation *module*.

As previously mentioned EOL is the basis for the Epsilon suite, and as such different aspects of the Epsilon suite can build upon this. A key addition to EOL within ETL, that enhances its transformational powers, is the *equivalent*() operator. The *equivalent*() or *equivalents*() operators automatically resolves source elements to their transformed counterparts in the target model; *equivalent*() returning a single element and *equivalents*() returning a bag.

4 Textual Modelling

Although we can perform model to model transformations using tools such as ETL or ATL, we require methods for converting from text-to-model and model-to-text to enhance the application of model transformations. There are a group of tools which perform textual modelling - a mapping between the concrete syntax of a Domain Specific Language (DSL) and an associated model representation. Textual modelling tools make use of textual recognition languages such as ANother Textual Recognition Language (ANTLR) [5] to detect patterns in the textual representation of a DSL and then uses concrete syntax rules to map the text to a meta model.

Using the combination of concrete syntax rules, text recognition languages and an appropriate meta model, textual modelling tools are able to provide

a development environment for creating seamless text-to-model and model-to-text transformations. There are several different solutions available which come under the heading of textual modelling tools which are used for describing DSLs, e.g. Xtext [8], EMFText [10, 9], Textual Editing Framework (TEF) [1] and Textual Concrete Syntax (TCS) [13, 14]. Each of these languages provides an environment and a methodology for capturing keywords and describing the structure of a DSL to enable mapping a DSLs source text to a model.

Within this section we look at two textual modelling tools, Textual Concrete Syntax from the AMMA suite, and EMFText, a tool developed by TU-Dresden.

4.1 Textual Concrete Syntax

Textual Concrete Syntax (TCS) [13] [14] is part of the AMMA framework and uses the KM3 (meta-meta model) language as its meta modelling reference language when generating a model from the source text.

TCS defines the structure of a DSL within a KM3 meta model, this structure is used to define where, within a model representation of a DSL description, elements can be placed. The capture of the DSL source text is handled by the TCS file, and so it is necessary to define how the DSL should be interpreted with respect to the meta model by defining templates for each class within the meta model.

A template describes the layout of the DSL with respect to a particular aspect of the DSLs syntax, the templates map segments of the DSLs syntax to classes in the meta model. This is carried out by specifying the ordering of the occurrence and frequency of class features, as well as defining the DSLs reserved words and their relative position within a template.

Square brackets can be used to emphasise a block of DSL syntax with a particular functionality within a template, ensuring the correct grouping of features and keywords.

DSLs contain reserved words which are used to define how the language should be interpreted at any given point. For instance, an if statement is recognised because it uses the reserved words if and then and optionally else. These reserved words are coined as keywords within the textual modelling realm. In TCS the keywords are placed within double quotation marks to highlight them as reserved words and are used to distinguish the DSL syntax from the variable and behaviour definitions and declarations being described within the DSL.

Quite often within a DSL, separators are used, such as commas or semi-colon. Separators can vary based on the location and context in which they are used. As a result TCS provides the ability to add arguments to a class feature such as the separator argument to aid in this more complex DSL syntax detection. Furthermore it is not uncommon in a DSL to reference a previously mentioned/created variable or function. To allow for this DSL requirement, TCS uses the option addToContext within the initial template declarations so that those objects which are generated can then be referenced by other objects in the model.

To then reference an already created object in TCS, the optional argument refersTo may be used when defining how a reference is identified within the DSL, the refersTo argument must specify what attribute to match on for the referenced class. If no refersTo argument is declared, then TCS assumes that the feature is a contained reference.

Many meta models contain boolean attributes within classes and their value is often based on the discovery of a keyword. Within TCS the attribute name is written followed by a ? to imply that the value will be set to true or false depending on the subsequent keywords or references defined within the template. The use of a ? is similar to that of a predicate and so naturally gives rise to a branching within a TCS template. The else clause, or alternative branch, is given as the TCS syntax which follows after a colon, ':'.

To fully define a DSL there is also the concept of adding additional lexers within TCS. Lexers are sets of characters, symbols or strings which can be used to distinguish between different types or operator. Lexers are defined in a separate section of the TCS file and are written using the ANTLR syntax as opposed to TCS's natural syntax.

The lexers are used in conjunction with TCS primitiveTemplate definitions, these primitiveTemplates specify how a particular lexer should be parsed. Whilst many primitiveTemplates simply pass the detected value in its entirety to be stored within a class feature, the primitiveTemplate for the Integer lexer however also used TCS functions to cast the recognised token to being of type Integer so that it can be stored correctly within the model.

Finally the isDefined() function can be used to aid in matching on primitiveTemplate types and is used in conjunction with the branching operators ? and : which we have previously discussed.

4.2 EMFText

As with TCS the development of a concrete syntax in EMFText is split into two parts, the structure of the DSL is contained within the meta model and the textual syntax mapping is carried out by creating concrete syntax rules; EMFText also uses ANTLR as its language recognition tool.

The EMFText Concrete Syntax language used for defining how the DSL is constructed is very similar to that of TCS. There is a concept of a rule for each class within the associated DSL meta model, and the ability to define and detect reserved words (keywords).

Keywords in EMFText are written inside of double quotation marks, "x", and as an optional functionality of EMFText, these keywords can be added to the TOKENSTYLES section of the CS definition to enable colour and font styling to be applied to these keywords.

Contained references to other classes are represented by specifying the reference name within a rule along with the required multiplicity. For uncontained references, square brackets are added to the end of the reference name. Note that unlike TCS EMFText does not require the additional detail of a refersTo statement; EMFText will match the value in the DSL to whatever value within the referenced class is available.

Attributes are assigned their values in EMFText by declaring the attribute name along with the addition of square brackets in a rule. This is the same as a reference to another class and it is left to EMFText to determine the feature type for the value assignment.

The use of multiplicities are key when defining a rule to match a DSL's syntax. The multiplicity symbols, ?, + and * are used to represent 0..1, 1..* and 0..*. Their use within EMFText does not, however, have to be related purely to a class feature. The multiplicities can be assigned to a selection of features and keywords, as required, and are applied to anything contained within parenthesis. Alternatively the multiplicity can be applied purely to a reference.

Separators in EMFText are defined in a more natural manner than TCS. As EMFText is able to define a multiplicity around a selection of features and/or keywords the DSL separators are simply placed within the parenthesis along side a feature.

Whilst separators are defined within multiplicities, sometimes the DSL requires that there only be separators between the same features. For instance when separating value with commas, we do not need a comma if there is only one element in a list, and we do not require a comma after the last

element of a list. Therefore the multiplicities employed do not always need to match that of the original class feature definition. We can write a list which has the multiplicity 1..* as the combination of a singular reference to the feature, followed by the combination of the comma keyword and the feature reference within parenthesis with the * multiplicity applied.

EMFText defines rules, and as such ordering takes precedence, this is the same for the branching operator in EMFText, |. The branching operator can be used in conjunction with several other EMFText functions to create more complex concrete syntax rules.

Finally, TOKEN definitions may also be created in EMFText to enhance the details of a DSL. TOKEN definitions in EMFText can be used to restrict what can be assigned as values to an attribute, and likewise can aid in the language recognition. TOKEN names are placed within the square brackets of a feature to define where they should be used.

EMFText provides a Zoo alongside the tool which contains numerous concrete syntax mappings for languages varying from Java and C# to formal languages such as B. However, some of these concrete syntax mappings are incomplete or only cover the aspects of the language which were relevant to the creator at the time of writing. The EMFText Zoo currently contains approximately 60 different mappings, although some of these are variations or particular implementations of the same language.

5 Formal Verification of VHDL

In this section we first discuss the framework proposed by Evans [7] for translating VHDL to CSP || B. We then go on to formalise translation rules from VHDL to CSP based on Evans translation from VHDL to CSP || B which form the basis for our meta model transformation, discussed in Section 9.

5.1 The VHDL to CSP || B Framework

The VHDL language is based around the concept of signal manipulation within different hardware elements. Because of this it is not suitable to think of signal changes as atomic, but rather there is a delay between a signal changing and a point at which the signal has once again stabilised (a delta delay). A delta delay in VHDL is a minute amount of time in which it takes the signal to stabilise between components.

The way this is modelled in B, is that two explicit values are stored. One version to hold the current signal values and one version to hold the new signal value. Thus if a signal is changed which is required by another signal assignment, then the original value can be used as seen in Figure 5.

$$\mathbf{a} \Leftarrow \mathbf{c};$$
$$\mathbf{b} \Leftarrow \mathbf{a};$$

Figure 5: Updating signals in VHDL

Processes are triggered when changes in their sensitivity list list are identified. To show this, upon completion of their execution the B variables are updated accordingly. The synchronisation and firing of B operations is handled by CSP controllers, in accordance with the CSP || B methodology

This method allows for a replication of the VHDL behaviour to be modelled in the formal verification language CSP || B. The CSP || B architecture we have described is presented in Figure 6.

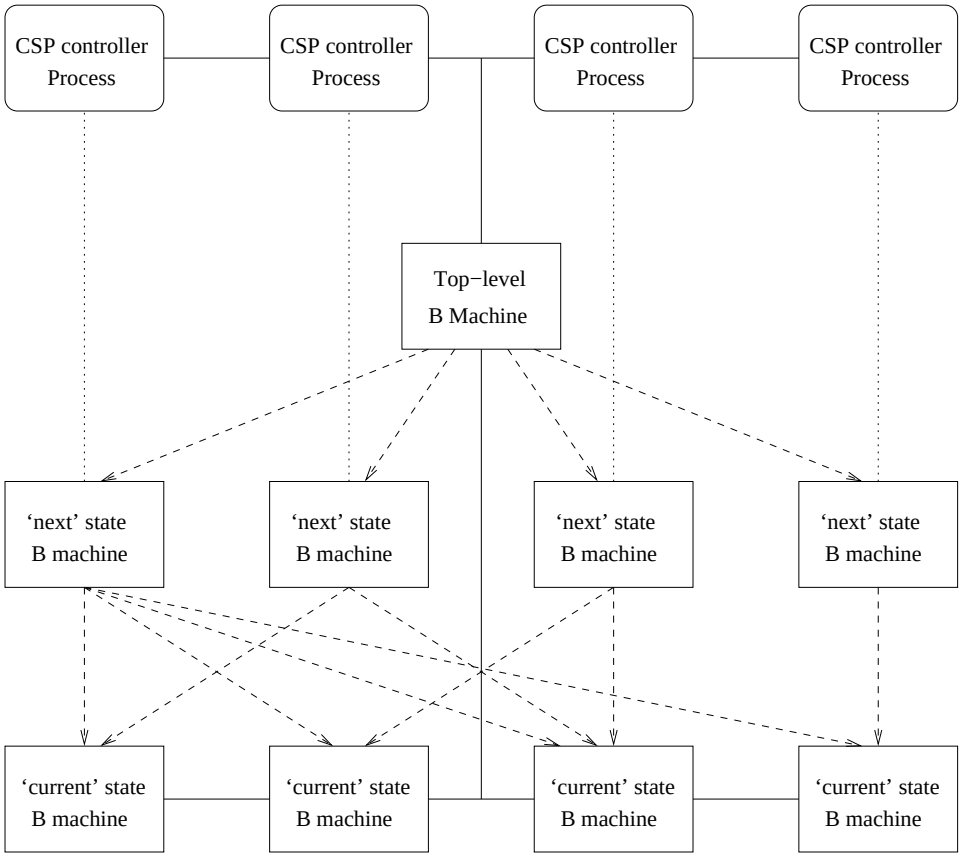


Figure 6: CSP || B architecture for VHDL

5.2 Lifting the B to CSP

In order to utilise the tool FDR to perform automated model checking of VHDL a methodology has been suggested by Evans [7] which removes the need for the B-method language [2] to be used in the architecture described in Section 5.1. Lifting the B machines to the CSP level removes the restriction to the tool ProB [19] and enables the use of CSP defined safety specifications to be run against the transformed VHDL design in FDR.

When the B machines were present the CSP was responsible purely for communicating the data to the relevant B machines, it must now also perform the logical assignments and store the values. By creating processes in CSP which mimic the behaviour of the processes in the B we can lift the computational aspect of the CSP || B system into CSP.

However, it is important to also represent the values of a working set of signals/ports as well as the new signal/port values generated between signal stabilisations in order to fully replicate the framework proposed in [7]. To do this each process is split into two parts, a computation process and an update process.

The computation process can be written in a CSP *let within* statement which performs the logical assignments to internal variables. These computation processes must also accept a set of parameters to represent the inputs used within the B machines (the sensitivity lists in the VHDL). Once any changes to signals have been performed, the update process must then be called.

The update process will communicate over the global channel, *dd*, all changes made with respect to its associated computation process. If a change is detected in any of the signals/ports to which the update process listens, then control and the new values will be passed to the computation process. If no relevant changes are detected, then the update process will allow the *dd* event to fire repeatedly until such time as a relevant signal changes which will require the computation process to fire once more.

6 VHDL to CSP Translation Rules

Our translation rules defined within this section are derived from the framework described in Section 5. In this section we discuss the translation rules for the two aspects of any VHDL description: *entities* (Section 6.1) and *architectures* (Section 6.2). We then proceed to go into further detail with respect to the translation rules for processes (Section 6.3) and the associated VHDL computational functions.

6.1 VHDL Entities

A VHDL entity contains the definitions of *input* and *output ports*, as identified in Section 2.2. We present the mapping rules for these three different elements of VHDL entity descriptions.

6.1.1 Data Types

As the VHDL language being used in the translation to CSP is a subset of synthesizable VHDL, our translation rules only deals with two data types: *bit* and *bit_vector*. The VHDL *bit* data type is mapped to CSP as a CSP data type:

$$\textbf{datatype}\, BIT \;=\; \{0, 1\}$$

Ther proposed mapping for the VHDL data type *bit_vector* varies from that proposed by Evans in [7], instead of mapping a *bit_vector* to an integer number with a maximum value, we have taken the approach of mapping a *bit_vector* to unique channel 'parameters' of CSP data type *BIT*.

For instance, a *bit_vector* of length 4 would be represented as four consecutive channel 'parameters' of CSP data type *BIT*. The CSP translation of a *bit_vector* of length 4 is written in CSP as the channel values:

$$BIT.BIT.BIT.BIT$$

6.1.2 Input Ports

$$port\,(\mathbf{A}, \mathbf{B} : \textbf{in bit};) \quad \Rightarrow \quad \begin{array}{l} A\_BIT \;=\; BIT \\ B\_BIT \;=\; BIT \\ \textbf{channel}\, dd \;:\; A\_BIT.B\_BIT \end{array}$$

Figure 7: VHDL Input Port translation to CSP

Section 5.2 discussed the use of a global channel, *dd*. Each input port of a VHDL entity is represented as a value that will be communicated along the channel *dd*. The type of those values will then be the same as the types of the input ports. This translation rule is given in Figure 7.

The CSP types are simple conversions of the VHDL type *bit*, but will be prefixed by the names of the input ports. So for example the input port A of type *bit* is converted into the first value of type A_BIT to be communicated along channel dd, this is similar for input port B.

As we can see, the input ports within Figure 7, A and B, are no longer given equivalent names within the CSP translation, instead the data types A_BIT and B_BIT are added to dd. Both A_BIT and B_BIT are simply the composition of the port name and it's data type of type BIT.

6.1.3 Output Ports

$$\begin{array}{rcl} \textit{port}\,(& & \textbf{channel}\,C\ :\ BIT \\ \quad \textbf{C}, \textbf{D}\ :\ \textbf{out bit}); & \Rightarrow & \textbf{channel}\,D\ :\ BIT \end{array}$$

Figure 8: VHDL Output Port translation to CSP

Whilst VHDL input ports are all represented as values communicated on dd, the output ports are defined using a different approach. All output ports are given individual CSP channels with their associated type given as the CSP channel value, we illustrate this method of channel definition in Figure 8.

6.2 VHDL Architectures

VHDL architectures can define both user defined data types, and signals to aid in the implementation of the entity to which an architecture relates. The user defined data types can only be utilised by signals. Signals may also be defined as the standard data types already mentioned in Section 6.1. All CSP signals defined must be added to the dd channel.

6.2.1 User Defined Data Types

$$\begin{array}{l} \textit{architecture asm of entity is} \\ \textbf{type stype is}\ (\textbf{start}, \textbf{middle}, \textbf{finish}); \quad \Rightarrow \quad STYPE = \{start, middle, finish\} \end{array}$$

Figure 9: VHDL User Defined Data Type translation to CSP

Similarly to the translation of data types, VHDL user defined data types can be translated to CSP data types for use on the global channel dd. Figure 9 illustrates the translation from VHDL to CSP.

6.2.2 VHDL Signals

architecture asm of entity is		$present\_STYPE = STYPE$
type stype is (*start*, *middle*, *finish*);		$next\_STYPE = STYPE$
signal present, **next** : **stype**;	$\Rightarrow$	**channel** dd : $present\_STYPE.next\_STYPE$

Figure 10: VHDL Signal translation to CSP

As with VHDL input ports, signals must also be represented on the global channel. There is no difference to the way in which a signal is represented on *dd*, except that the data type used to represent a signal may be a user defined data type, not just of type *BIT*, we demonstrate this translation in Figure 10.

For example, if we had input ports *A* and *B* as well as signals *present* and *next*, then *dd* would be defined within the CSP model as:

channel dd : $A\_BIT.B\_BIT.present\_STYPE.next\_STYPE$

6.3 VHDL Processes

6.3.1 Typical Process

A VHDL process is described as using 2 CSP processes, P and P' as shown in Figure 11. The process P is a construction of the logical assignments within the associated VHDL process, wrapping all translated CSP logical assignments inside a *let within* statement. P also contains a set of parameters which map to the sensitivity list of the VHDL process.

In order to pass the changes made to signals and ports within the CSP process P, it is necessary to share this information across the globally defined channel *dd*. To provide this functionality a secondary CSP process is defined, P', which takes the updated signal/port information from P and broadcasts the values over the global channel upon each synchronisation.

Furthermore the process P' not only updates the global channel, but also listens for changes in the sensitivity list of P. When changes are detected the process P is then called with the updated sensitivity list values. P' must also take in the previous values pertaining to the sensitivity list of P in order to perform a comparison between the old and new values in order to detect any changes. If there are no changes then the process must not

block *dd* from occurring, so it must be differed. If there is an update to an output port within a VHDL process the behaviour must be captured in the CSP process as the firing of a channel with the name of the output port, carrying the associated value.

The translation rule for a VHDL process to two CSP processes are illustrated in Figure 11.

$$
\begin{array}{l}
\textbf{P} : \textbf{process}(\textbf{A})\ \textbf{is} \\
\textbf{begin} \\
\quad D \Leftarrow A; \\
\textbf{end process P}
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
\textbf{P}(\textbf{v\_A}) = \\
\quad \textbf{let} \\
\quad\quad trans(D \Leftarrow A) \\
\quad \textbf{within} \\
\quad\quad D!nv\_D \rightarrow P'(v\_A,\ nv\_D)
\end{array}
\qquad
\begin{array}{l}
P'(v\_A,\ nv\_D) = \\
\quad dd?nv\_A!nv\_D \multimap \\
\quad if(nv\_A \neq v\_A)then \\
\quad\quad P(nv\_A) \\
\quad else \\
\quad\quad P'(v\_A,\ nv\_D)
\end{array}
$$

Figure 11: VHDL Process translation to CSP

6.3.2 Simple Process

A VHDL process can be declared in another form, a single line definition. A single line VHDL process is not given a unique name and it only updates one port or signal, it therefore takes the name of the port/signal it is updating. Furthermore it does not have a sensitivity list clearly defined, instead the sensitivity list is implicitly derived from the arguments which the right hand side of the process uses when formulating the value to assign to its associated port/signal.

As with the previous CSP mapping for a VHDL process, it is necessary to create two CSP processes for each VHDL process, one to perform the algorithmic assignments and one to determine when the sensitivity list has been triggered. Again this is captured with a process P and P', using the global update channel *dd*.

We represent the mapping from a VHDL single line process to two CSP processes in Figure 12.

The VHDL in Figure 12 shows that the value of port A is assigned to the value of port D, and is triggered each time the value of port A changes. This mapping is represented in CSP by monitoring the change of the value nv_A on the channel *dd* and comparing it with the value previously detected on the channel *dd*. If the value of nv_A changes then the process P_D is entered which assigns the latest value of port A (nv_A) to the port D (nv_D).

$$
\begin{array}{lll}
 & \mathbf{P\_D}(v\_A) = & \mathbf{P'\_D}(v\_A,\ nv\_D) = \\
D \Leftarrow 'A' & \quad \mathbf{let} & \quad dd?nv\_A!nv\_D \rightarrow \\
\Rightarrow & \quad\quad nv\_D = v\_A & \quad if(nv\_A \neq v\_A) then \\
 & \quad \mathbf{within} & \quad\quad P\_D(nv\_A) \\
 & \quad\quad D!nv\_B \rightarrow P\_D'(v\_A,\ nv\_D) & \quad else \\
 & & \quad\quad P'(v\_A,\ nv\_D)
\end{array}
$$

Figure 12: VHDL Simple Process translation to CSP

As this process also updates an output port, D, this must be represented as the firing of event D after the within statement before then going back to a state of monitoring.

6.3.3 Signal Assignments

$$D \Leftarrow A; \quad \Rightarrow \quad nv\_D = v\_A$$

Figure 13: VHDL assignment translation to CSP

Signal assignment translation to CSP is written in a similar vein to the VHDL, we assign a value to a port or signal using the assignment operator $' ='$. The value assigned to the port/signal can be from another port/signal or match the data type assigned to the particular port/signal. We clarify this assignment translation in Figure 13. Note the change in naming for the ports/signals in the CSP. We represent old values with the prefix '$v\_$' and new values with the prefix '$nv\_$', this is to ensure that we are able to distinguish between the two.

6.3.4 IF Statements

$$
\begin{array}{lll}
present \Leftarrow if(A = B)\ then & & nv\_present = if(A == B) then \\
\quad next; & & \quad v\_next \\
else & \Rightarrow & else \\
\quad start; & & \quad start
\end{array}
$$

Figure 14: VHDL *if* statement translation to CSP (style 1)

The VHDL IF statement can be mapped to CSP in several ways, the first being a mapping from VHDL to the Haskell syntax within CSP. The second

a mapping to a CSP Guarded Choice statement. The two mapping styles of the VHDL *if* statement to CSP are described in Figure 14 and Figure 15. Note the negation required in the CSP Guarded Choice mapping for the else statement, this is required as the CSP Guarded Choice is not an exclusive *if* statement and therefore this must be specified explicitly.

$$\begin{array}{l} present \Leftarrow if(A = B)\ then \\ \quad next; \\ else \qquad\qquad \Rightarrow \\ \quad start; \end{array}$$

$$\begin{array}{l} (v\_A == v\_B)\&\ let\ nv\_present\_state = v\_next\ within\ P' \\ \Box\ (v\_A! = v\_B)\&\ let\ nv\_present\_state = start\ within\ P' \end{array}$$

Figure 15: VHDL *if* statement translation to CSP (style 2)

6.3.5 CASE Statements

$$\begin{array}{lcl} & & P(start) = \\ & & let \\ & & \quad nv\_C = 0 \\ & & within \\ & & \quad C!nv\_C \rightarrow P'(start,\ nv\_C) \\ P : process(present)\ is & & \\ \quad case\ present\ is & & P(middle) = \\ \quad\quad when\ start \Rightarrow & & let \\ \quad\quad\quad C \Leftarrow' 0'; & & \quad nv\_C = 1 \\ \quad\quad when\ middle \Rightarrow & \Rightarrow & within \\ \quad\quad\quad C \Leftarrow' 1'; & & \quad C!nv\_C \rightarrow P'(middle,\ nv\_C) \\ \quad\quad when\ finish \Rightarrow & & \\ \quad\quad\quad D \Leftarrow' 1'; & & P(finish) = \\ & & let \\ & & \quad nv\_D = 1 \\ & & within \\ & & \quad D!nv\_D \rightarrow P'(finish,\ nv\_D) \end{array}$$

Figure 16: VHDL *case* statement translation to CSP (style 1)

Mapping the VHDL *case* statement to CSP is not a direct translation. As CSP does not have a *case* statement within its grammar, a different method must be used, we map the VHDL *case* statement to a method in CSP known as pattern matching. In the CSP definition of the process P, we map the VHDL sensitivity list to the CSP process parameters. As a VHDL *case* statement must validate its values against elements of the associated data type, we can write specific cases of the CSP process P that relate the

different VHDL *when* statements of a VHDL *case* statement to an associated pattern matched CSP process, as shown in Figure 16.

Alternatively a similar mapping to that shown for the IF Statement in Figure 15 may be used to map a VHDL Case statement to a CSP Guarded Choice. This is illustrated in Figure 17.

$$
\begin{array}{l}
P : process(present)\ is \\
\quad case\ present\ is \\
\quad\quad when\ start \Rightarrow \\
\quad\quad\quad C \Leftarrow' 0'; \\
\quad\quad when\ middle \Rightarrow \\
\quad\quad\quad C \Leftarrow' 1'; \\
\quad\quad when\ finish\ \Rightarrow \\
\quad\quad\quad D \Leftarrow' 1';
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
(v\_present == start) \& \ let\ nv\_C = 0\ within\ P' \\
\Box\ (v\_present == middle) \& \ let\ nv\_C = 1\ within\ P' \\
\Box\ (v\_present == finish) \& \ let\ nv\_D = 1\ within\ P'
\end{array}
$$

Figure 17: VHDL *case* statement translation to CSP (style 2)

6.3.6 WHEN statement

The VHDL *when* statement, not to be confused with the *when* syntax used in the VHDL *case* statements, is used for simple port and signal assignments. The *when* statement can be thought of as having the same functionality as a VHDL *if* statement, but with a different syntax structure. The structure comprises of an assignment, then a boolean test, and then an *else* statement which provides the option of a different assignment. When translating this VHDL syntax into CSP we map it directly to the CSP *if* statements as this provides the appropriate behavioural match.

We illustrate this mapping and the syntax of the *when* statement in Figure 18.

$$
present \Leftarrow start\ when\ (A =' 1')\ else\ finish \quad \Rightarrow \quad
\begin{array}{l}
nv\_present\ =\ if\ (v\_A == 1)\ then \\
\qquad start \\
\quad else \\
\qquad finish
\end{array}
$$

Figure 18: VHDL *when* statement translation to CSP

A point worth noting is that a *when* statement may be used in conjunction with a simple process definition, such that the simple process definition is given a branching option as opposed to a standard port or signal assignment.

7 CSP to VHDL Meta Model Transformation

Creating a framework for transforming VHDL descriptions into CSP scripts can be carried out using some of meta modelling techniques discussed in Section 3. There are three aspects to transforming one DSL into another. First we must capture the source language as a model, to do this we must create a meta model of the source DSL and provide suitable concrete syntax rules to carry to capture the different model objects from the text. It is necessary to define a meta model for the target DSL as well, and again concrete syntax rules must be defined to allow generation of the DSL syntax from the model objects. Once there exists a meta model for both the source and target languages, then a set of transformation rules may be developed to complete this transformation. Figure 19 provides a visual representation of these three individual stages necessary for converting one DSL into another, in our case VHDL into CSP.

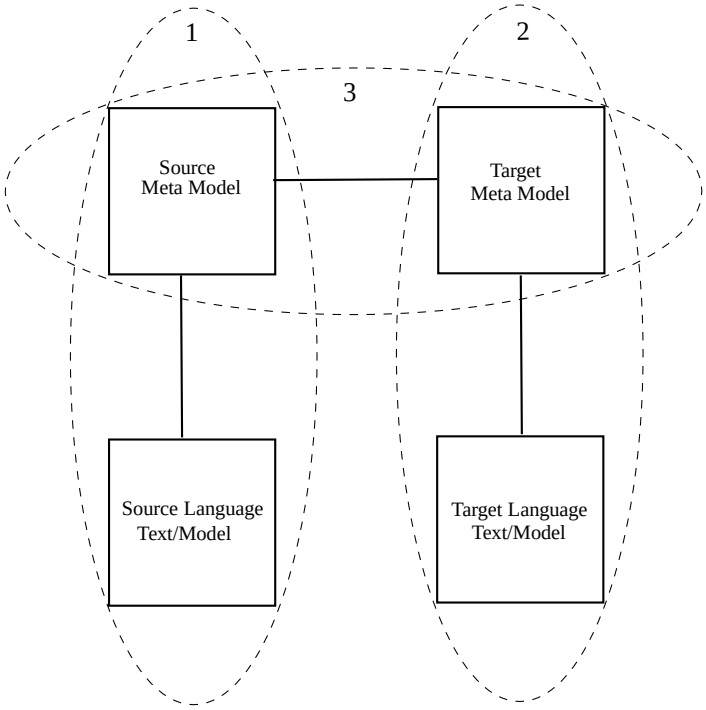


Figure 19: A Model Transformation Framework

We have numbered the three separate stages within Figure 19. Stages 1 and 2 will both be developed using EMFText which can perform both model-to-text and text-to-model transformations. Stage 3, the model transformation from VHDL to CSP, will be written using the Epsilon Transformation Language. Stages 1 and 2 will develop both the meta models and concrete syntax

rules for both VHDL and CSP, the meta models developed in these stages will then be used when defining and performing the ETL transformation between the two languages.

We discuss the implementation of Stages 1 and 2 in Section 8 and Stage 3 in Section 9.

8 Implementing Textual Mappings for VHDL and CSP

This section describes the work which we have undertaken in enabling automatic textual mappings for VHDL and CSP. We first present a concrete syntax mapping for VHDL which includes a VHDL meta-model as well as a concrete syntax definition for VHDL. We then proceed describe a concrete syntax mapping for CSP which includes a CSP meta-model, loosely based on the SystemB CSP meta-model [27] and a concrete syntax definition for CSP.

When first developing the textual mapping for this transformation from VHDL to CSP we began to developing the textual mapping in TCS, however due to the limitations of the language we found that we could not fully capture the VHDL source text. As a result, we moved to using the CSM tool, EMFText.

8.1 VHDL Concrete Syntax Mapping

The first stage in creating an automatic transformation between languages is developing a suitable tool for capturing a VHDL model from a VHDL textual description, this aspect of the transformation process is illustrated in Figure 20.

Within this subsection, we first discuss the highlights of the VHDL meta model that we have created and then continue to discuss the accompanying rules which are used to provide the mapping between the textual representation and a model representation. We also give some details on the post processor that was necessary to fully define the subset of the VHDL language addressed in this report.

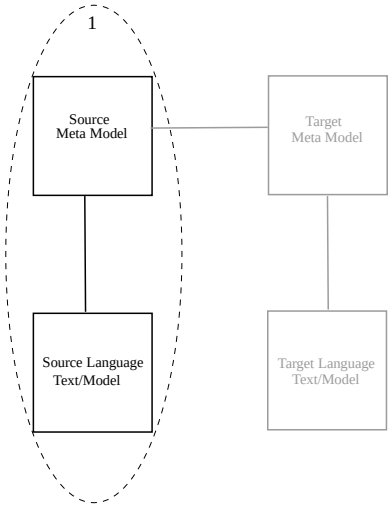


Figure 20: Development of Stage 1 - VHDL Model-to-Text

8.1.1 VHDL Meta Model

In creating a textual model for parsing a DSL and mapping it to a suitable meta model, a different approach to meta model development was undertaken. Previously, the development of a meta model would be driven purely by the designers understanding of the language and what they required the language to do.

However, when developing a meta model as part of a concrete syntax mapping, it becomes apparent that it is not always possible to directly map the syntax of a DSL to a meta model which has been developed purely from expert knowledge of the DSL in question. When defining a concrete syntax rule which maps directly to a class within the meta model the flexibility of the language is forced into the meta model by the concrete syntax rule. We have found that it has been necessary on several occasions to add additional classes or even restructure particular parts of the the meta model. This is not to say that the original meta model was incorrect in its definition of the DSL but that the structure of the DSL syntax could not be fully captured in that particular meta model definition.

In our development of the VHDL meta model, as seen in Figure 24, we discovered that certain classes must be added to ensure that the meta model flowed correctly. Also the use of super classes means that a model is more flexible and thus capable of mapping a wider variety of source DSL source code to a model.

Previously we have only developed meta models for model-to-model and model-to-text generation. However the emphasis of a structurally correct meta model is much higher when performing text-to-model generations. Not only does generalisation need to play a heavy part in the meta model definition but also the inclusion of extra classes which play no other function than to allow flexibility in the assignment and creation of other classes from the DSL source code. For instance, see the structure of the Predicates and Predicate classes in the VHDL meta model Figure 24.

Whilst it is important to start the design of a meta model based purely on the developers understanding of the DSL, it is felt that the ability to mould the meta model alongside the mapping of the concrete syntax for a DSL means that many iterations are seamlessly performed and minor corrections made without excess effort. The changes made when mapping the DSL to a model means that when a model-to-model transformation is required there is already a framework in place for generating the text from the transformed model.

The meta model itself has a root class in the VHDLDescription which contains an Entity and an Architecture. As we can see from the meta model in Figure 24, the Entity definition references the Interface class, a super class to enable referencing both ports and signals.

The use of generalisation within the meta model is key to ensuring that the DSL's textual representation can be represented as a model which conforms to the meta model. Likewise we use generalisation to provide a method for referencing between DefinedTypeItems and Interfaces so that a distinct value or a stored value can be used within a VHDL Assignment.

Our use of contained references within the VHDL meta model is a mirror of the VHDLs own structure, with an Entity containing a Port and an Architecture containing Signals and Process definitions.

8.1.2 VHDL Concrete Syntax

As discussed in Section 4.2, the text-to-model transformation requires not only a meta model to define the structure of the DSL, but also concrete syntax rules associated with each class in the meta model. Whilst some of these concrete syntax rules are trivial, such as defining the rule for a VHDL Simple Process which references a contained Process Expression:

```
SimpleProcess ::= processAssignment;
```

However other rules must be able to allow more freedom in what can follow, for instance a Port may be defined as not only having an attribute name, references to direction and type, but also potentially be a Vector and must therefore specify the vector length as required. This is potential for a Vector is described in the rule by using the 0..1, ?, multiplicity identifier for EMFText. Furthermore the rule must also define the keywords, ":","(" and ")", associated with the definition of a Port:

```
Port ::= name[] ":" hasDirection[]
   isOfType[] ("(" vectorDetails ")")?;
```

There are two different ways of defining a reference in EMFText, dependant on whether the reference is a contained reference or not, as previously highlighted in Section 4.2. The Port rule defines a reference to a direction with the syntax *hasDirection*[] and a contained reference to vector details using *vectorDetails*.

When looking at the definition of superclasses which contain multiple different classes as contained references, multiplicities are used heavily. For instance, the Architecture class for VHDL must not only have a name and end name attribute (as it is a specialisation of the class AfterNamedElement), but it also references an Entity class and then may contain multiple DefinedTypes, Signals and must contain at least one Process. The corresponding EMFText concrete syntax rule which defines this behaviour, and includes the necessary VHDL keywords can be written for the VHDL meta model, Figure 24, as:

```
Architecture ::= "architecture" name[]
   "of" implementsEntity[] "is"
   (definesUserType ";")*
   (definesSignal ";")*
   "begin"
   (definesProcess ";")+
   "end" "architecture" afterName[] ";"
   ;
```

The Architecture rule, specifies not only the multiplicities of the User Define Type and Signal as being 0..* but also that each User Defined Type or Signal should be separated by a ";", by using parenthesis to contain not only the contained reference but also the keyword, the multiplicity can be applied to the contents of the parenthesis.

We have previously mentioned another aspect of EMFText, see Section 4.2, TOKENS. TOKENS in EMFText are used to define specialised symbols which are associated with attributes, for instance, the assignment value in

VHDL may be the addition or negation of two or more signal or port values. Thus to ensure that the value of a conjunction of two or more port and signal values a TOKEN may be specified, such that the only acceptable values to be assigned to the logicalOperator attribute of a SimpleExpression class is:

```
DEFINE LOGIC_SYMBOL $'and'|'or'|'not'|'nand'|'nor'|'xor'|'&'$ ;
```

This TOKEN may then be used in the concrete syntax rule for the Simple-Expression class by placing the TOKEN type in between the square brackets of the attribute assignment:

```
SimpleExpression ::= simpleExpressionLHS (logicalOperator[LOGIC_SYMBOL]
   nestedSimpleExpression)? ;
```

Within the VHDL concrete syntax mapping we have also used the EMFText branching operator, —, to distinguish between different VHDL syntax. For instance a VHDL vector may be defined as *x to y* or *y downto x* such that y is the larger value. Whilst we cannot assert directly within the rule that y is larger than x, we can ensure that the assignment of the upper and lower bounds of the Vector class attributes are correct by using the branching operator:

```
Vector ::= (lower[INTEGER] "to" upper[INTEGER])
   | (upper[INTEGER] "downto" lower[INTEGER]) ;
```

As may be seen the Vector concrete syntax rule as well as distinguishing the ordering of the upper and lower bounds based on the keywords *downto* and *to* also uses the EMFText predefined TOKEN *INTEGER* to ensure that the attribute value is numeric and cannot contain symbols or other non-numeric values.

Whilst developing the textual mapping for VHDL within EMFText several issues were encountered, issues arose in trying to detect *std_ulogic* values within the VHDL DSL. Whilst a TOKEN had been specified for *std_ulogic* the Java packages generated by EMFText for the VHDL DSL did not allow for the detection of both the *std_ulogic* TOKEN and the standard TEXT TOKEN automatically. Instead a choice branch was required to enable the detection of both these value types freely:

```
SimpleValue ::= usesElement[STD_ULOGIC] | usesElement[TEXT];
```

This use of branching ensures that whilst the TEXT TOKEN is available, the STD_ULOGIC TOKEN is tried first and matched where appropriate. The next issue faced pertained to the assignment of Boolean values to object features based on text recognition within the DSL syntax. Whilst we had defined TOKENS for the phrase rising_edge and event, it was not possible to assign a Boolean value to the associated attribute upon their detection. To resolve this issue the automatically generated Java classes for the specific TOKENS needed to be modified. Initially the value detected in the DSL was simply passed to the Boolean attribute by the Java method:

```
public void resolve(String lexem,
 org.eclipse.emf.ecore.EStructuralFeature feature,
 org.emftext.language.vhdl.resource
  .vhdl.IVhdlTokenResolveResult result) {
   defaultTokenResolver.resolve(lexem, feature, result);
}
```

However, as we wanted to assign a Boolean value on the occurrence of these values, we modified this Java method to assign a true or false value based on the string:

```
public void resolve(java.lang.String lexem,
 org.eclipse.emf.ecore.EStructuralFeature feature,
 org.emftext.language.vhdl.resource
  .vhdl.IVhdlTokenResolveResult result) {
   if ("Rising_edge".equals(lexem)||"rising_edge".equals(lexem)) {
       result.setResolvedToken(true);
   }else if ("".equals(lexem)){
      $result.setResolvedToken(false);
   }
}
```

By altering the Java class method in the analyser package we were able to detect the TOKEN values and assign the correct value to the associated attribute accordingly. Also it must be noted that an escape character was required for the apostrophe in the EVENT TOKEN, however as would be expected only one escape character was required (
event).

The final issue we faced worth highlighting was the restriction that elements could not be created arbitrarily for each DSL file loaded. After consulting with the EMFText developers a solution was identified which involved the writing of a PostProcessor; a key feature that had been added to provide more flexibility and functionality to EMFText. This enabled the inclusion of more complex language recognition, model manipulation and element generation than previously available from just the CS rules.

PostProcessor The PostProcessor must be developed as another package, which is itself associated with EMFTexts specific DSL packages. It uses the Java classes which define individual elements as well as the core Java class eFACTORY. The eFACTORY Java class is automatically generated by EMFText for the creation, deletion and manipulation of Ecore elements, from classes to feature values.

With the aid of the EMFText developers [25] a plug-in feature was developed which was able to generate the standard VHDL data types, as well as the directions for port communication. The definition of the VHDL meta model was updated to allow for an extra aspect to be added to a VHDLDescription, a Package.

The package contained Directions as well as DefinedTypes allowing us to implement a PostProcessor for automatically generating the required data types. We include the Java code for the main method which was added to the post processor package for the VHDL DSL. Further information on defining PostProcessors can be found in Section 4.2 of the EMFText Guide [26].

An instance of each class must be generated and assigned the relevant values before being added as a reference (contained) to another object. The method begins by discovering the root object and then proceeds to generate the objects required to define a suitable solution for our previously mentioned problem. Please note that we have only provided a fragment of the entire method required for the VHDL DSL due to its repetitive nature.

```
public void process(VhdlResource resource) {
   EObject root = resource.getContents().get(0);
   VHDLDescription description = (VHDLDescription) root;
   Package packageStandard = vhdlFactory.eINSTANCE.createPackage();
   packageStandard.setName("STANDARD");
   description.setContainsPackage(packageStandard);

   Direction directionIn = vhdlFactory.eINSTANCE.createDirection();
   Direction directionOut = vhdlFactory.eINSTANCE.createDirection();
   directionIn.setName("in");
   directionOut.setName("out");");
   packageStandard.getDefinesDirection().add(directionIn);
   packageStandard.getDefinesDirection().add(directionOut);

   DefinedType typeBit = vhdlFactory.eINSTANCE.createDefinedType();
   typeBit.setName("bit");
   packageStandard.getDefinesType().add(typeBit);

   TypeItem typeItemB0 = vhdlFactory.eINSTANCE.createTypeItem();
   TypeItem typeItemB1 = vhdlFactory.eINSTANCE.createTypeItem();
   typeItemB0.setName("'0'");
   typeItemB1.setName("'1'");
   typeBit.getContainsItem().add(typeItemB0);
   typeBit.getContainsItem().add(typeItemB1);
}
```

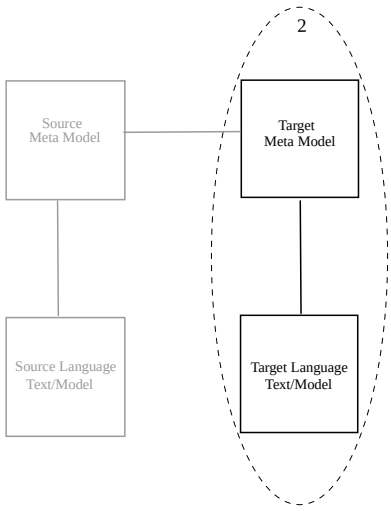


Figure 21: Development of Stage 2 - CSP Text-to-Model

This stage in defining the VHDL to CSP transformation framework has resulted in the definition of a VHDL meta model and accompanying concrete syntax rules which are able to capture the style and language of our source DSL, VHDL. We have created a total of 34 concrete syntax rules which map directly to every concrete class defined within the VHDL meta model we have defined, as well as define a post processor to generate the arbitrary objects associated with the VHDL STANDARD package.

8.2 CSP Concrete Syntax Mapping

Whilst in Section 8.1 we specified how we were able to use textual modelling to map a DSL concrete syntax to a model conforming to a meta model, this same approach can be used for generating text from a model, as previously mentioned in Section 4.2. The automated transformation from VHDL to CSP requires that an automatically generated CSP model representation of the VHDL source text be translated from model to text.

It is therefore reasonable to use EMFText to produce a method for generating a CSP model into a CSP source text file. By using EMFText to develop this required tool it is possible to use the same development technique for defining a suitable CSP meta model.

The definition of concrete syntax rules does not differ from the style discussed for our VHDL text-to-model translation in Section 8.1 but additional EMFText syntax must be added. This additional syntax does not provide

any further description for the mapping between DSL text and a model representation; it only adds spacing to any generated text, that is, white space and new lines. These additional markers add the detail required to pretty print the output text to ensure that it will be parse-able by the DSL specific tools as well as make the text more human readable.

8.2.1 CSP Meta Model

We initially planned to make minor modifications to the SystemB CSP meta model [29]. However, upon close inspection it was discovered that the SystemB CSP meta model would not be able to support the required CSP output from the VHDL to CSP translation discussed later in this document. Our primary reasoning for this decision related to the lack of functionality to add more than one CSP function to each process, as well as the inability to reference parameter values and static values for both process and event parameters inexplicitly. Thus, it was necessary to strip back the meta model to its core components and remap the classes to enhance the use of generalisation. Not only did redefining the meta model provide the flexibility required for a VHDL to CSP transformation, but it also ensured that the meta model structure developed would adhere to the CSP syntax.

The SystemB CSP meta model did not use super-classes effectively to enable the flexibility required for CSP text-to-model mapping, a method discovered when implementing VHDL in EMFText. An example of the benefits within our CSP textual mapping is that by lifting the different Types in CSP to an abstract class, referencing process parameters and event data values can be carried out without the need for complex conditional branches within the concrete syntax.

As with the VHDL concrete syntax mapping, we developed the new CSP meta model shown in Figure 25 alongside the definition of concrete syntax rules, thus ensuring that the meta model would conform to our required output.

When defining the if statement predicates in CSP we found that we were able to directly lift not only the meta model definitions from our VHDL meta model but also the concrete syntax rules as well. The ability to implement this code reuse across different DSLs came from the modular way in which meta models are defined and the similarities between the DSL in terms of syntax structure.

The inclusion of the Haskell classes to the meta model followed the same method of development, they were built up slowly in conjunction with the concrete syntax rules and again it was possible to apply code reuse with

respect to the predicate classes. When reviewing the SystemB CSP meta model, [29], and our newly defined CSP meta model, Figure 25, there is a noticable difference in meta model styles and the enhanced use of generalisations in the newly defined CSP meta model.

8.2.2 CSP Concrete Syntax

The method for defining the concrete syntax rules for our CSP mapping is similar to the VHDL concrete syntax mapping Section 8.1.2. Like the VHDL textual mapping, we did discover some restrictions and issues when defining the concrete syntax rules.

The first issue we encountered that proved problematic was the inability to construct a suitable rule for parallel composition. We wanted to define a parallel process as, being constructed of ProcessExpressions, which meant that these could either be defined as references to previously created processes, or in-line CSP processes. However it was not possible to enter this rule using EMFText, instead of the parser assigning a parallel process as an object of type Parallel, it instead assigned the parallel process as an object type SequentialExpression, and then failed to parse the rest of the DSL syntax correctly. Whilst this issue would not be an immediate issue as we would be performing model-to-text generation, the implementation, once it had generated the text, would instantly rescan the text and flag this issue once again.

It was therefore decided that parenthesis should be included into the parallel composition rule to enable the ordering in which the composition of multiple processes are made. These parentheses provided a suitable distinguishing feature for the underlying ANTLR grammar to identify between parallel compositions and sequential expressions.

When defining the rule for the let within class, one of our requirements was to enable the definition of Haskell statements within a let within as well as a standard process definition. As the CSP language is suitably flexible to allow this sort of behaviour there was again a need to distinguish between two different rules which both started in the same way. (name =)

It was decided that a compromise should be made with respect to the flexibility of the CSP language. As a result we decided that CSP processes must always begin $P\_$, at the same time as implementing this restriction we also allowed for an apostrophe to be used in the name of a process as well.

Coding this up in EMFText required the specific definition of a PROCESS token:

```
DEFINE PROCESS $'P_'$ + $('A'..'Z' | 'a'..'z' |'0'..'9' | '_' | '-' | '\'')+ $;
```

The PROCESS TOKEN can then be used within the CSP Process concrete syntax rule as:

```
Process ::= name[PROCESS] ("("definesParameters")")? "=" processStatement;
```

By ensuring that all process names began with a 'P_' it was possible to distinguish between Haskell statements and CSP process definitions.

Our implementation within the let within rule contains one other restriction. The rule states that if both Haskell and Process definitions are defined within a let within statement, then the Haskell statements must all be placed prior to any process definitions:

```
LetWithin ::=
   "let"
   (letHaskellStatements)*
   (letProcesses)*
   "within"
   withinExpression
   ;
```

By introducing a generalised representation of for the HaskellStatement and Process classes, this issue could be avoided, we illustrate this change in Figure 22. We have not adapted our concrete syntax mapping to include this feature at present as it can add ambiguities to the text recognition and we feel that the ordering we are enforcing is not cause for concern in this particular circumstance.

DBLP:conf/ictac/TurnerTSE08

The final issue concerned the definition of the CHOICE token, which contained both the ASCII representation for internal and external choice. The use of the CHOICE token was to set a Boolean value to true if external choice ($\Box$) was written in the CSP syntax and false if the internal choice ($\sqcap$) was detected.

Upon defining the CHOICE token we were faced with an error message declaring that the ASCII for internal choice was an empty string. This error message was reported to the EMFText developers and it was found to be a bug (0001493) in EMFText. The SVN version of EMFText, contained a fix for this which is to be included in the next release of EMFText (1.3.1).

This stage in defining the VHDL to CSP transformation framework has resulted in the definition of a CSP meta model and accompanying concrete

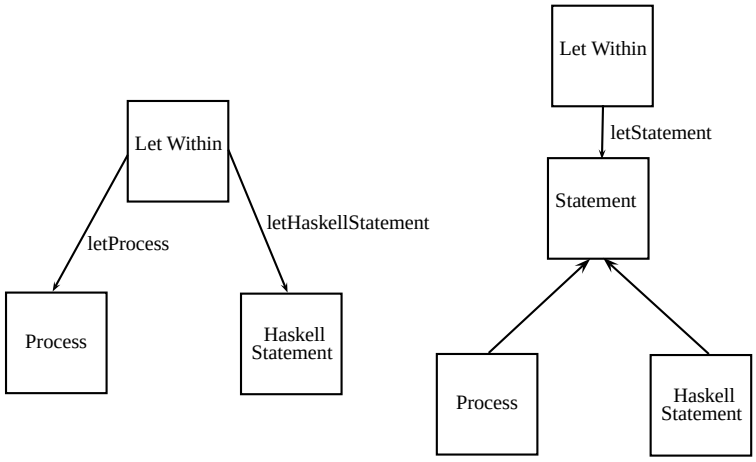


Figure 22: Generalisation for the Let Within CSP References

syntax rules which are able to capture the style and language of our source DSL, CSP. We have created a total of 47 concrete syntax rules which map directly to every concrete class defined within the CSP meta model we have defined.

Whilst it is not a requirement we could also introduce two post processors for this DSL concrete syntax mapping. The first of these post processors would ensure that the length of the parameter and assignment lists are equal, thus ensuring that the CSP syntax generated for moving to a process was syntactically correct. The other rule would be to ensure that references to processes were bound within the Let Within clauses in CSP, i.e., that the a standard process could not reference (move to) a process defined within a let within clause.

9 Transforming VHDL to CSP

In this section we discuss the final aspect of the translation from VHDL to CSP. We describe the methodology and highlight some of the ETL code generated for mapping between a VHDL source model and a CSP target models.This aspect of the transformation is shown in Figure 23. Throughout this section we discuss the different rules and operations developed to perform the transformation from VHDL to CSP using ETL.

Throughout this section we provide illustrations to depict how each rule maps the VHDL to the CSP throughout this section.

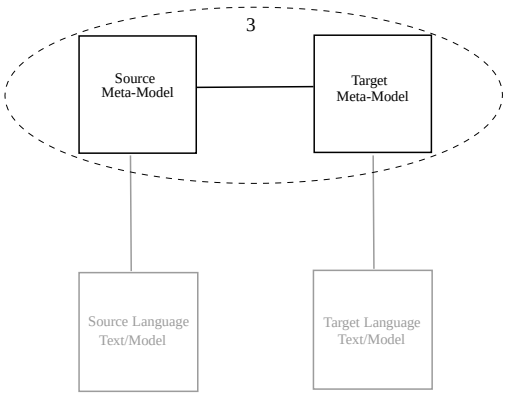


Figure 23: Development of Stage 3 - VHDL to CSP Model Transformation

The Epsilon Transformation Language provides several different attributes which can be applied to rules; the first of these attributes which we use is the @primary attribute. This attribute ensures that the rule with which it associates has a preferential ordering when it comes to the firing of the rules, and so will be fired before any rules without this attribute.

9.1 ETL Primary Controlling Rule

The primary rule, Description2Script is the initial rule which generates a CSP script from a VHDL Description; that is the top level elements of both the CSP and VHDL meta models. From within this rule we can populate the generated CSP script with data types, output channels and processes as well as create a reference to the already generated dd channel to contain this class.

```
@primary
rule Description2Script
 transform v : vhdl!VHDLDescription
 to c : csp!CSPScript {
   c.containsTypes.addAll(v.containsPackage.definesType.
    equivalent());
   c.containsTypes.addAll(v.containsArchitecture.
    definesUserType.equivalent());
   c.containsChannels.addAll(v.containsEntity.definesPort.
     select(port : vhdl!Port|port.hasDirection.name.
      toLowerCase() == "out").equivalent());
   c.containsProcesses.addAll(v.containsArchitecture.
     definesProcess.equivalent());
   c.containsChannels.add(ddChannel);
}
```

One of the extended functions to EOL placed within ETL is the equivalent() function. As previously discussed in Section 3.3, the equivalent() function

is used to create the references and containments for other transformed objects. In calling the equivalent() function, all rules associated with the source object to which it is applied. Furthermore the equivalent() function refines the source objects on which the rule is fired to those associated to the source reference to which the equivalent() function has been applied; that is the rule is applied on a subset of all objects of a particular class, and the subset are those objects which are contained by the reference on which the equivalent() function was used.

As can be seen in our primary rule, Description2Script, we use the equivalent function to associate all the various aspects our CSP model as contained objects of the CSPScript object.

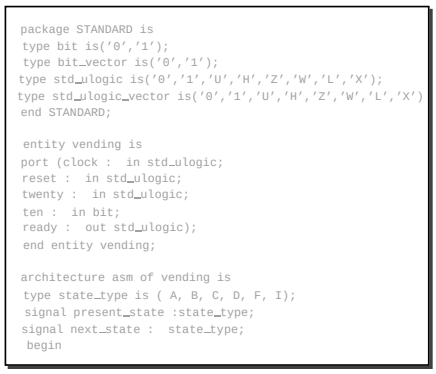
As we discussed in Section 3.3, rules with the attribute @lazy do not fire unless they have been called; in ETL the use of the equivalent() function fires the lazy rules associated with the object on which the equivalent function was applied.

The first lazy rule called by Description2Script is the lazy rule which defines a transformation VHDL DefinedType objects, this is determined by following the references containsPackage and then definesType from the initial VHDL Description (as given by the line v.containsPackage.definesType). Notice that the reference to which we attaching the resultant transformed CSP objects are assigned to the containsType reference (c.containsTypes), ETL does not ensure that the generated objects returned by the equivalent() function are in fact of type CSP Type, if there exists a rule which transforms VHDL DefinedType objects into some other CSP class, then these would also be returned causing an error to arise. In the case of the statement v.containsPackage.definesType.equivalent(), there are in fact two lazy rules fired in response, DefinedType2DataType and DefinedType2NameType. All returned objects generated by these rules are then assigned to the CSP containsType contained reference for the CSPScript object.

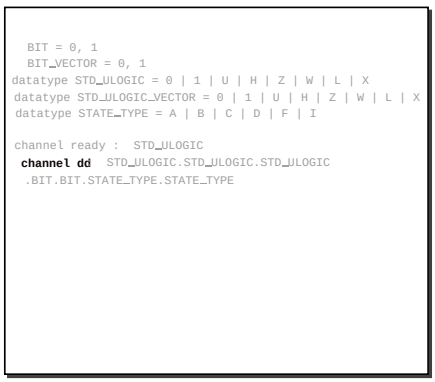
Likewise the statement v.containsArchitecture.definesUserType.equivalent() fires two lazy rules, these lazy rules happen to be the same DefinedType2NameType and DefinedType2DataType. This functionality is available because both the VHDL Package and VHDL Architecture classes in the meta model both contain objects of class type DefinedType. Since the rules are written in isolation of the rest of the meta model it is possible to re-use these rules to perform transformations on the same class type objects which are associated with a different aspect of the meta model. Only the objects which are contained by the v.containsarchitecture.definesUserType containment reference will be used when firing these two lazy rules.

9.2 Transforming VHDL Entities

9.2.1 Generating the dd Channel



```
package STANDARD is
 type bit is('0','1');
 type bit_vector is('0','1');
type std_ulogic is('0','1','U','H','Z','W','L','X');
type std_ulogic_vector is('0','1','U','H','Z','W','L','X')
 end STANDARD;

 entity vending is
 port (clock :  in std_ulogic;
 reset :  in std_ulogic;
 twenty :  in std_ulogic;
 ten :  in bit;
 ready :  out std_ulogic);
 end entity vending;

 architecture asm of vending is
 type state_type is ( A, B, C, D, F, I);
  signal present_state :state_type;
 signal next_state :  state_type;
  begin
```



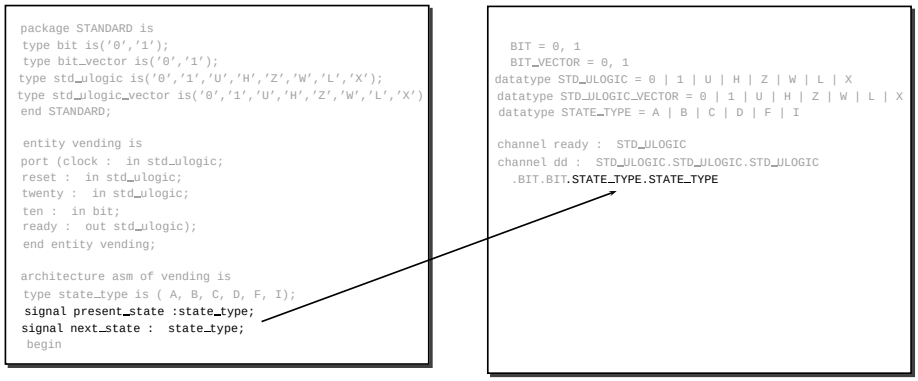
```
  BIT = 0, 1
  BIT_VECTOR = 0, 1
datatype STD_ULOGIC = 0 | 1 | U | H | Z | W | L | X
datatype STD_ULOGIC_VECTOR = 0 | 1 | U | H | Z | W | L | X
 datatype STATE_TYPE = A | B | C | D | F | I

channel ready :  STD_ULOGIC
 channel dd  STD_ULOGIC.STD_ULOGIC.STD_ULOGIC
  .BIT.BIT.STATE_TYPE.STATE_TYPE
```

As the translation from VHDL to CSP requires the generation of the dd channel, we are not able to produce a direct mapping from the VHDL ports and signals. Instead we create the dd channel within a pre block which is fired before any rules are fired. Within this pre block we not only create a CSP channel with name dd, but we also must create a CSP parameter list which is then contained within the dd channel, this will then allow us to add parameters to the channel later in the transformation. We also generate a variable, ddChannelValues, which can be used to store information regarding the ordering of the parameters on the dd channel; this will be crucial later in the transformation to ensure that parameter values are placed in the correct position within dd events.

```
pre {
    var ddChannelValues := new OrderedSet;
    var ddChannel = new csp!Channel;
    ddChannel.name := "dd";
    var ddParameterList = new csp!ChannelParameterList;
    ddChannel.definesParameters := ddParameterList;
}
```

In order to populate this arbitrarily generated dd channel, we must create two rules, one for adding all signals in the VHDL as parameters on the CSP dd channel, and one for adding all input ports in the VHDL as parameters on the CSP dd channel.

9.2.2 VHDL Signals to CSP Parameters



The first rule we define is the Signals2ddChannel, this rule is a standard rule and thus is fired in no particular order with respect to the other rules within the transformation; this is why the creation of the dd channel itself has been placed within a pre block to ensure that it is available when the rules fire. The rule transforms VHDL Signal objects into CSP TypeRef objects. The TypeRef object is a container object for a reference to a CSP Type. The TypeRef class was introduced to the CSP model as the non unique flag for Ecore OrderedSet objects is non-functional. As a result, it is not possible to reference an object more than once in any Ecore OrderedSet.

```
rule Signal2ddChannel
 transform signal : vhdl!Signal
 to typeRef : csp!TypeRef {
    typeRef.containsType := GetDataType(signal);
    ddChannelValues.add(signal.name);
    ddParameterList.parameterTypes.add(typeRef);
}
```

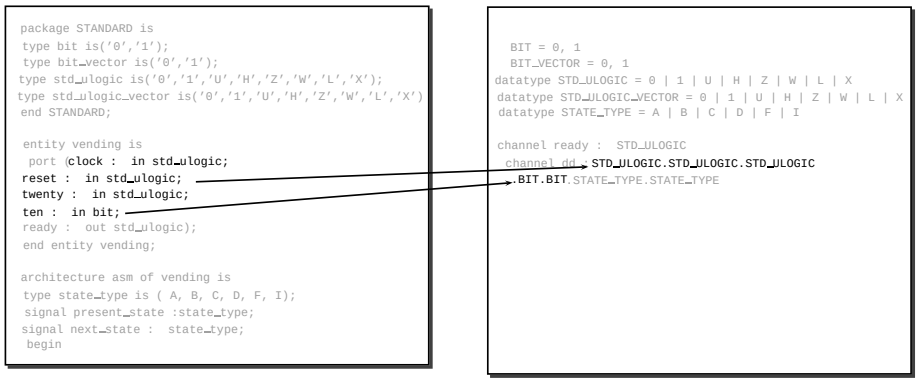
The TypeRef object is generated and the operation GetDataType is called to return a reference to the CSP DataType which is associated with the VHDL Signal data type. Finally the typeRef is added to the parameter list for the dd channel and the name of the signal is added to the ddChannelValues variable to ensure that we can determine where within the dd channel parameters the newly added signal is positioned.

We have defined the operation GetDataType to aid in the generation of target elements. The Epsilon language allows us to define the abstract VHDL class Interface as the input parameter for this operation; by doing so the operation can be used for all specialisations of the Interface class (Signal and Port). The operation then returns a CSP Type, again an abstract class, enabling the return of both DataType and NameType objects.

```
operation GetDataType(interface : vhdl!Interface) : csp!Type {
    var interfaceType := interface.isOfType.name.toUpperCase();
    var dataType := csp!Type.select(type|type.name=interfaceType)
     .first();
    return dataType;
}
```

This operation does not generate any new target elements; instead it takes the Interface Type name, and converts it to upper case. The operation then proceeds to use the Epsilon Object Language to match this name to a CSP Type with the same name, and then return the reference to the CSP Type, be it a NameType or DataType, to the caller of the operation.

9.2.3 VHDL Input Ports to CSP Parameters



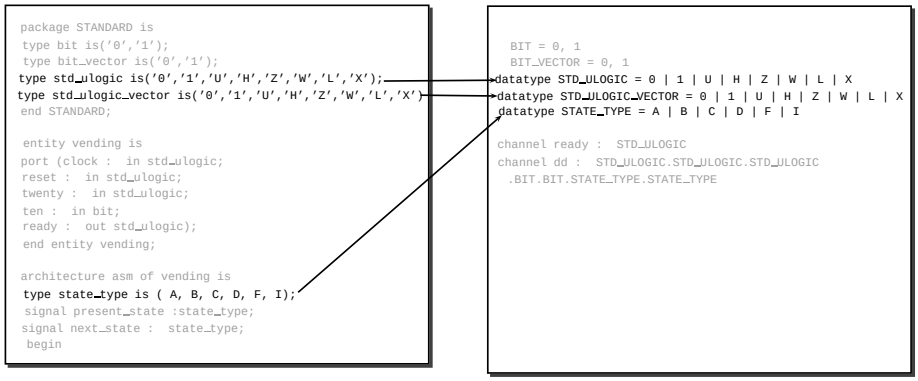
The second rule we have defined for populating the dd channel, InputPort2ddChannel, is very similar to the Signal2ddChannel rule, except that we are transforming a VHDL Port into a CSP TypeRef. As with the Signal2ddChannel we again call the GetDataType operation, this time passing a VHDL Port (the port currently being transformed by the rule) instead of a VHDL Signal. And again we add the generated CSP TypeRef to the dd Channel parameter list as well as the Port name to the ddChannelValues variable.

```
rule InputPort2ddChannel
 transform port : vhdl!Port
 to typeRef : csp!TypeRef {
  guard : port.hasDirection.name.toLowerCase() == "in"
    typeRef.containsType := GetDataType(port);
    ddChannelValues.add(port.name);
    ddParameterList.parameterTypes.add(typeRef);
}
```

However, this rule contains a guard to restrict the Port objects on which it operates. We have defined a restriction to ensure that only ports with direction in are selected. By adding this guard we can ensure that correct VHDL ports are chosen for the transformation.

Both the DefineType2DataType and DefinedType2NameType rules perform the same functionality and contain the same statements, except for two key aspects.

9.2.4 VHDL Defined Types to CSP Data Types



First the DefinedType2DataType has the target object as the CSP class DataType whilst the DefinedType2NameType has the target object as the CSP class NameType. This does not change the assignments made within the rules differ as both DataType and NameType have the same features within the CSP meta model. The other key difference between the two rules is the guard which is applied, whilst one is the negation of the other, this guard is what distinguishes between source definedType objects and determines whether they will be CSP NameType or CSP DataType objects when transformed.

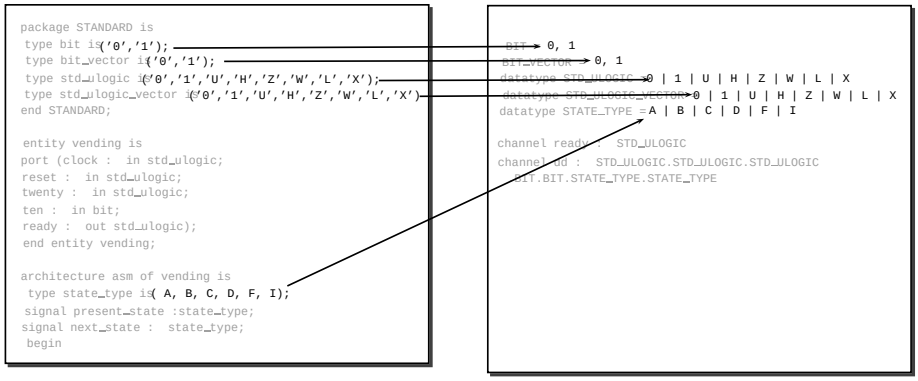
```
@lazy
rule DefinedType2DataType
 transform definedType : vhdl!DefinedType
 to dataType : csp!DataType {
  guard : definedType.containsItem.exists
   (typeItem : vhdl!TypeItem | typeItem.name <>"'1'" and
   typeItem.name<>"'0'")
    dataType.name := definedType.name.toUpperCase();
    dataType.definesDataTypeItems := new csp!DataTypeList;
    dataType.definesDataTypeItems.containsDataTypeItem.addAll(
      definedType.containsItem.equivalent());
}
```

9.2.5 VHDL Defined Types to CSP Name Types


```
@lazy
rule DefinedType2NameType
 transform definedType : vhdl!DefinedType
 to nameType : csp!NameType {
  guard : not (definedType.containsItem.exists
   (typeItem : vhdl!TypeItem | typeItem.name <>"'1'" and
    typeItem.name<>"'0'"))
     nameType.name := definedType.name.toUpperCase();
     nameType.definesNameTypeItems := new csp!NameTypeList;
     nameType.definesNameTypeItems.containsDataTypeItem.addAll(
       definedType.containsItem.equivalent());
 }
```

Both the DefinedType2DataType and DefinedType2NameType rules use the equivalent() function once more, in this case both rules are firing the TypeItem2DataTypeItem rule on their selected source objects.

9.2.6 VHDL Type Item to CSP Data Type Item



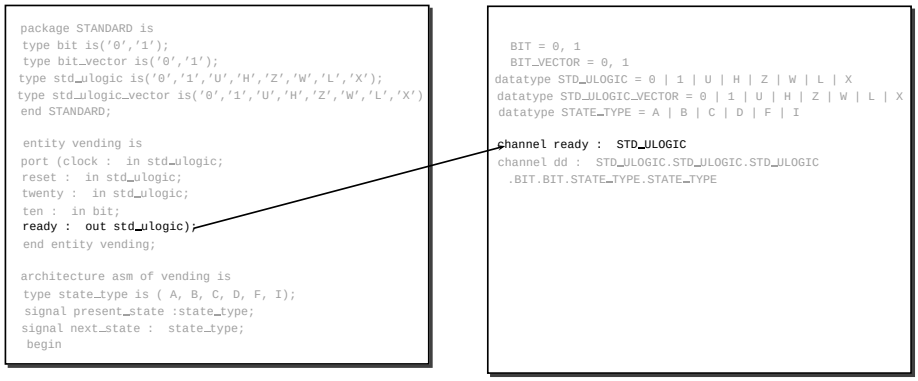
The TypeItem2DataTypeItem rule is not quite a direct mapping from a VHDL TypeItem to a CSP DataTypeItem. The CSP language is unable to handle single quotations around DataTypeItems, so an Operation is called for each transformation of a VHDL TypeItem into a CSP DataTypeItem to ensure that these single quotes are removed, DataTypeItemFormatter.

```
@lazy
rule TypeItem2DataTypeItem
 transform typeItem : vhdl!TypeItem
 to dataTypeItem : csp!DataTypeItem {
    dataTypeItem.name := DataTypeItemFormatter(typeItem.name);
}
```

The DataTypeItemFormatter operation takes in a String and returns a String, performing a processing on the input String to remove both the beginning and end characters from the input String in the event that single quotations are found as the first and last characters, otherwise the input String is returned unchanged. This operation could have been performed within the TypeItem2DataTypeItem rule, however as we have had to perform this modification of the string here, it is inevitable that we will also have to perform the same operation to references to the original VHDL TypeItem objects later on within the transformation.

```
operation DataTypeItemFormatter(vItemName : String) : String {
    var cleanedName = new String;
    if (vItemName.startsWith("'") and vItemName.endsWith("'")) {
       cleanedName := vItemName.substring(1,(vItemName.length()-1));
    } else {
       cleanedName := vItemName;
    }
    return cleanedName;
}
```

9.2.7 VHDL Output Ports to CSP Channels



Finally we must transform the Output Ports of the VHDL to CSP channels, as we discussed in Section 6, we must distinguish between input and output Ports as they are represented differently in our CSP translation. The OutputPort2Channel rule is again a lazy rule and relies on being fired by the Description2Script rule. And as with the InputPort2ddChannel rule, we use a guard to ensure that only the correct source objects are selected, this guard is the negation of the guard used in the InputPort2ddChannel rule and takes the subset of ports which have the direction out. Once the correct VHDL Ports have been selected, it is then necessary to generate a ChannelParameterList for each target object, this is carried out within the rule as there is no object type within the VHDL which provides a direct mapping.

```
@lazy
rule OutputPort2Channel
 transform port : vhdl!Port
 to channel : csp!Channel {
  guard : port.hasDirection.name.toLowerCase() == "out"
    channel.name = port.name;
    channel.definesParameters := new csp!ChannelParameterList;
    var typeRef = new csp!TypeRef;
    typeRef.containsType := GetDataType(port);
    channel.definesParameters.parameterTypes.add(typeRef);

}
```

After specifying the generation of a new ChannelParameterList and assigning it to the newly created CSP Channel, a new TypeRef must also be defined. In this scenario, we know that a VHDL Port can only have one DataType and so we use the previously mentioned operation GetDataType to return the reference to the equivalent CSP DataType. Once the TypeRef has been generated and the correct reference assigned this is added as a Parameter to the CSP Channel ChannelParametersList.

As can be seen in the rules we have discussed here, there is not always a direct mapping that can be defined in one rule between source object x and target object y. Within the transformation between VHDL and CSP it has been necessary to define two rules for the separation of the VHDL source objects DefinedTypes so that they can be generated as CSP target objects DataType or NameType accordingly. Furthermore these rules must be called twice to ensure that the source objects are taken not only from the VHDL Entity description but also the VHDL Architecture description. Likewise the separation of Port types requires two separate rules.

9.3 Transforming VHDL Architectures

9.3.1 VHDL Process to Two CSP Processes

In order to transform the VHDL processes to CSP processes we use ETLs ability to create two target elements from one source element, this enables us to generate both the P and P processes necessary in our translation. From this rule, multiple operations are called in order to generate the necessary CSP structure. Within these called operations we do still use the equivalent() function to transform the VHDL Assignments, IF and Case Statements, Predicates etc., using rules.

The first operations called (SensitivityList2ProcessParameterList and GenerateParametersFromProcessExpression) are used to generate the required CSP Process parameter lists from the VHDL Sensitivity lists, as well as any VHDL Predicate or Assignment functions which use signals or ports which are not included in the VHDL Sensitivity List. Note, that these operations are fired twice; once for the P process and once for the P process. This is followed with the RemoveDuplicatesFromParameterList operations, which as the name suggests cleans the parameter list for each process. This operation is also passed a Boolean value that signifies if the process parameter list belongs to a P process or a P process.

Finally two key individual processes are fired. The first, ConstructProcess, fires off a set of operations and rules to construct and populate the P process from the VHDL process behaviour. The second, ConstructPrime, constructs the P process and defines all the conditional arguments associated with triggering the calling of the P process.

```
@lazy
rule Process2Process
 transform vProc : vhdl!ComplexProcess
 to cProc : csp!Process,
  cProcPrime : csp!Process {

  cProc.name := "P_" + vProc.name;
cProcPrime.name := "P_" + vProc.name + "'";

cProc.definesParameters :=
  SensitivityList2ProcessParameterList(vProc.reactsTo);
cProcPrime.definesParameters :=
  SensitivityList2ProcessParameterList(vProc.reactsTo);

for(expression in vProc.processStatements){
cProc.definesParameters.containsTypeItem.addAll
(GenerateParametersFromProcessExpression(expression));
cProcPrime.definesParameters.containsTypeItem.addAll
(GenerateParametersFromProcessExpression(expression));
}
```

```
RemoveDuplicatesFromParameterList(cProc.definesParameters, false);
RemoveDuplicatesFromParameterList(cProcPrime.definesParameters, true);

cProc.processStatement := ConstructProcess(vProc);

GeneratePrimeRefs(vProc.reactsTo, cProc.processStatement,cProcPrime);

cProcPrime.processStatement :=
   ConstructPrime(cProc, cProcPrime, vProc.reactsTo);


 }
```

9.3.2 VHDL Process to CSP Process - Additional Operations

The ConstructProcess operations is used to transform the VHDL process behaviour into CSP and then manipulate the outputted CSP elements into the correct structure. To do this, the operation fires two rules, associated with generating CSP LetWithin statements from VHDL Assignments and CSP Conditional Choice statements from VHDL IF and Case Statements. Once all the behaviour for the VHDL process is captured, the Construct-ProcessExpression operation is fired to manipulate all the generated CSP process expressions into a one CSP process expression. This method for generating a the CSP Process is necessary as a VHDL process may contain multiple different assignments and Conditional Statements, however, a CSP Process may only contain one; thus it is necessary to reshape the output generated by firing the ETL rules using operations to obtain the required structure required for a CSP Process.

```
operation ConstructProcess(vProc : vhdl!ComplexProcess) :
  csp!ProcessExpression {
var expressions = new OrderedSet;
for(statement in vProc.processStatements)
switch (statement.type().name) {
case "ExpressionStatement" : expressions.add(statement.equivalent());
case "Assignment" : expressions.add(GenerateLetWithin(statement));
}
var cExpression := null;

cExpression := ConstructProcessExpression(expressions);
return cExpression;

}
```

9.3.3 VHDL Expression Statement to CSP Expression Wrapper

Within our transformation script we have several rules which simply generate required target elements, based on structure of the source elements in order to maintain the structure we require in our transformation. In the

case of the ExpressionStatement2ExpressionWrapper rule, we are simply adding parenthesis around any generate CSP Choice expressions in order to maintain the structure required in the CSP process.

```
@lazy
rule ExpressionStatement2ExpressionWrapper
 transform vStatement : vhdl!ExpressionStatement
 to cWrapper : csp!ExpressionWrapper {
  cWrapper.nestedExpression := vStatement.containsStatement.equivalent();
}
```

9.3.4 VHDL IF Statement to CSP Guarded Choice

IFStatement2GuardedChoie is a far more complex rule which generates an initial CSP Choice element and then, depending on the availability of VHDL elsif statements, generates more Choice elements, as nested elements of the initial Choice, as required.

Our IFStatement2GuardedChoice rule also uses the equivalent() function in order to use the transformation tools ability to generate the necessary target elements from the branch structure found within the VHDL IF Statements. By using the programmatic approach given by EOL we are able to define a while loop which creates the elements required to adhere to the CSP meta model, and generate the correct target elements.

A separate operation, CreateConditionalPredicate is used to generate the predicates used within the Choice Conditional Statement branches. It is necessary to block all but the wanted behaviour for each branch, and whilst this is normally given using the if and else clauses, the CSP conditional operator does not provide this functionality. As a result all the predicates used within the VHDL IF Statement must be stored and all but the newest predicate negated within the Conditional Statement Predicate for each recursion through the while loop. This provides the blocking needed to match the behaviour found in the VHDL in the CSP transformation.

Finally, we also generate the VHDL else statement, and if no arguments exist for the else branch of the VHDL IF Statement, then we ensure that there is a handle available which may be populated later with the call to the associated P process.

```
@lazy
rule IFStatement2GuardedChoice
 transform vIF : vhdl!IFStatement
 to cChoice : csp!Choice {
 var currentChoice := cChoice;
 var i := Integer;
```

```
 i := 0;
 var predicates := new OrderedSet;
 while(i < vIF.ifTrue.size()){
  var cWrapper := new csp!ExpressionWrapper;
  var cStatement := new csp!ExpressionStatement;
  var cConditional := new csp!ConditionalStatement;
  cStatement.nestedStatement := cConditional;
  cWrapper.nestedExpression := cStatement;
  predicates.add(vIF.ifPredicate.at(i).equivalent());
  cConditional.validatesUsing := CreateConditionalPredicate(predicates, false);
  cConditional.nestedExpression := ConstructNestedExpressions(vIF.ifTrue.at(i));
  currentChoice.isExternalChoice := true;
  currentChoice.nestedExpression.add(cWrapper);
  if(i<vIF.ifTrue.size()){
  var newChoice := new csp!Choice;
  currentChoice.nestedExpression.add(newChoice);
  currentChoice := newChoice;
  }
  i := i+1;
 }
 var cWrapper := new csp!ExpressionWrapper;
 if(vIF.ifFalse.isDefined()){
  var cStatement := new csp!ExpressionStatement;
  var cConditional := new csp!ConditionalStatement;
  cStatement.nestedStatement := cConditional;
  cWrapper.nestedExpression := cStatement;
  cConditional.validatesUsing := CreateConditionalPredicate(predicates, true);
  cConditional.nestedExpression := ConstructNestedExpressions(vIF.ifFalse);
 }else{
  var cStatement := new csp!ExpressionStatement;
  var cConditional := new csp!ConditionalStatement;
  cStatement.nestedStatement := cConditional;
  cWrapper.nestedExpression := cStatement;
  cConditional.validatesUsing := CreateConditionalPredicate(predicates, true);
  cConditional.nestedExpression := new csp!ExpressionWrapper;
 }
  currentChoice.nestedExpression.add(cWrapper);
  delete predicates;
}
```

9.3.5 VHDL Case Statement to CSP Guarded Choice

The CaseStatement2GuardedChoice rule takes a VHDL Case element and cycles through all it's When clauses to generate the correct CSP Guarded Choice mapping. Unlike the If statement rule, there is no need to provide the negation of the previous predicates for each guard of the CSP Guarded Choice, as the VHDL case statement uses an equivalent on one Port/Signal only, this can be translated directly for a CSP Guarded Choice.

The VHDL style for the predicate definition however does not map directly to CSP as the IF statement predicate does. Instead, an EOL operation is defined, ConstructPredicateForWhen, which generates the predicate, taking first the argument of the VHDL Case Statement and then provides the When statement predicate value. This operation then returns a suitable equivalence predicate.

```
@lazy
rule CaseStatement2GuardedChoice
 transform vCase : vhdl!CaseStatement
 to cChoice : csp!Choice {
 var currentChoice := cChoice;
 var i := Integer;
 i := 0;
for(when in vCase.whenClauses){
var cWrapper := new csp!ExpressionWrapper;
  var cStatement := new csp!ExpressionStatement;
  var cConditional := new csp!ConditionalStatement;
  cStatement.nestedStatement := cConditional;
  cWrapper.nestedExpression := cStatement;
  cConditional.validatesUsing :=
ConstructPredicateForWhen(vCase.referencesValue,when.validatingItem);
cConditional.nestedExpression :=
ConstructNestedExpressions(when.nestedStatement);
  currentChoice.isExternalChoice := true;
  currentChoice.nestedExpression.add(cWrapper);
  if(i<vCase.whenClauses.size()){
  var newChoice := new csp!Choice;
  currentChoice.nestedExpression.add(newChoice);
  currentChoice := newChoice;
  }
  i := i+1;
}
var cWrapper := new csp!ExpressionWrapper;
  currentChoice.nestedExpression.add(cWrapper);
}
```

Also the mapping for the behaviour contained within a When Statement is not a simple equivalent() rule call. Instead another EOL operation, ConstructNestedExpressions, must be defined to generate the correct structure required for the CSP Guarded Choice; this is the same operation called within the IFStatement2GuardedChoice rule.

9.3.6 Re-organising the generated CSP Processes

Once the rules have been fired which generate the CSP process expressions, they must be merged into one complete, correct CSP process expression. To do this it is necessary to distinguish between the choice expressions and the LetWithin expressions. Once we have separated these out we can merge them together independently before then bringing them together to create one complete CSP process.

The controlling operation which performs this task is ConstructProcessExpression. After splitting the two distinct process expression types, it then passes these off to the MergeContainedExpressions for all LetWithin CSP elements and MergeChoiceExpressions for all Choice CSP elements. Both these operations then return one single CSP process expression in which all expressions of that type have been merged.

With respect to the LetWithin elements, this is fairly trivial, and all arguments contained within the LetWithin statements are merged into one. However the MergeChoiceExpression operation must not only nest the conditional choice statements within one another, but must also then merge the associated LetWithin statements which they themselves contain. This requires several recursive calls as well as the ability to duplicate particular aspects of a choice expression, including all its nested elements.

Once these two operations have returned single CSP Process Expressions, then the LetWithin process expression must be duplicated and added to each nested LetWithin element inside the CSP merged Choice expression.

Only once all this has been completed is it possible to then define the Within argument for each CSP LetWithin statement inside this process. This operations is called from the Process2Process rule, after the Construction of the Process is complete. It is also called from the parent of the ConstructProcessExpression as it is necessary to pass a handle on the P process to the operation so that the references may be created.

```
operation ConstructProcessExpression(expressions : OrderedSet) : csp!ProcessExpression {
var containedExpressions := new OrderedSet;
var choiceExpressions := new OrderedSet;
for(expression in expressions)
switch(expression.type().name){
case "ExpressionWrapper" : choiceExpressions.add(expression);
case "ExpressionContainment" : containedExpressions.add(expression);
}
var containedExpression := null;
var choiceExpression := null;
if(containedExpressions.size()>0){
containedExpression := MergeContainedExpressions(containedExpressions);
delete containedExpressions;
}
if(choiceExpressions.size()>0){
choiceExpression := MergeChoiceExpressions(choiceExpressions);
delete choiceExpressions;
}
var cExpression := null;
if(choiceExpression<>null and containedExpression <> null){
if(choiceExpression.nestedExpression.isDefined()){
cExpression :=
InsertLetWithinIntoChoice(choiceExpression.nestedExpression,
containedExpression);
}else{
cExpression := containedExpression;
}
}else if(containedExpression<>null){
cExpression := containedExpression;
}else if(choiceExpression<> null){
cExpression := choiceExpression;
}
return cExpression;
}
```

```
operation MergeContainedExpressions(expressions : OrderedSet) : csp!ProcessExpression {
if(expressions.size()>0)
for(expression in expressions){
if(expression.eClass().name<>"ExpressionContainment"){
("The MergeContainedExpressions operation has been passed
the expression type : " + expression).println();
"Aborting this operation, all hope is lost!".println();
abort;
}
}
var letWithin := new csp!LetWithin;
for(expression in expressions)
if(expression.containedExpression.letStatements.size()>0){
letWithin.letStatements
.addAll(expression.containedExpression.letStatements);
delete expression;
}
var cExpression := new csp!ExpressionContainment;
cExpression.containedExpression := letWithin;
return cExpression;
}
```

9.3.7 VHDL Predicate Conjuctions to CSP Predicate Conjunctions

As we have shown, not all rules within this translation are simple, or trivial. However, the PredicateConjunction2PredicateConjunction rule, should by all accounts be a simple translation. However this is not so, as we not only have to deal with the change of operators for the Predicate Conjunctions, but also, we have to deal with the possibility of a VHDL Rising_edge function being used within the predicate. To handle this we call an operation from within the rule, which generates a nested PredicateConjunction (inside a PredicateWrapper) which performs a check against the signal or port used in the Rising_edge function to the value 1. This is only half of the functionality provided by the VHDL Rising_edge function, it also checks for a change in state. This aspect of the VHDL function is instead lifted to the P process where it can be checked, instead of stored in the P process.

```
@lazy
rule PredicateConjunction2PredicateConjunction
 transform vConjunction : vhdl!PredicateConjunction
 to cConjunction : csp!PredicateConjunction {
  for(predicate in vConjunction.nestedPredicate)
  if(predicate.type().name == "PredicateValue"){
  if(predicate.RisingEdge){
  var cWrapper := new csp!PredicateWrapper;
  cWrapper.nestedPredicate :=
PredicateRisingEdge2HaskellPredicate(predicate);
  cConjunction.nestedPredicate.add(cWrapper);
  }else {
  cConjunction.nestedPredicate.add(predicate.equivalent());
  }
  }else{
```

```
  cConjunction.nestedPredicate.add(predicate.equivalent());
  }

  switch(vConjunction.operator){
  case "and" : cConjunction.operator := "and";
  case "or" : cConjunction.operator := "or";
  case "not" : cConjunction.operator := "not";
  case "&" : cConjunction.operator := "and";
  case "=" : cConjunction.operator := "==";
  case "=<" : cConjunction.operator := "=<";
  case ">=" : cConjunction.operator := ">=";
  case "/=" : cConjunction.operator := "!=";
  case ">" : cConjunction.operator := ">";
  case "<" : cConjunction.operator := "<";
  }
 }
```

9.4 Generating CSP P' Processes

The CSP P' processes cannot be generated using rules as they are derived from the implicit behaviour of the VHDL Process Sensitivity Lists. As a result the generation of a CSP P' process must be carried out using EOL operations.

The P' process behaviour is generated using information on the associated P process, its own Process name and parameters as well as the VHDL sensitivity list to which it is associated.

The first part of the operation defines an event parameter list which is then associated with a dd event. This generation of an event parameter list requires the correct input output and wild-card flags to be applied depending on the process parameters. This generation is carried out by the EOL operation, ConstructEventParameterList.

```
operation ConstructPrime(cProc : csp!Process, cProcPrime : csp!Process,
vpSensitivityList : vhdl!SensitivityList) : csp!ProcessExpression {
var expression := new csp!Prefix;
var dd := new csp!Event;
//create dd event
dd.instanceOf := GetChannelRef("dd");
dd.definesParameters :=
ConstructEventParameterList(cProcPrime.definesParameters,
cProc.definesParameters);
//define lhs and rhs of prefix expression
expression.nestedExpression.add(dd);
//define rhs of prefix expression
expression.nestedExpression
.add(CreateSensitivityStatements(cProc,cProcPrime,
vpSensitivityList));
//return expression
return expression;
}
```

After creating a dd event with the correct flags for the event parameter list, the dd event is then added to a Prefix object. The second contained reference to the Prefix object is generated by the CreateSensitivityStatements EOL operation which generates the required guards and predicates from the sensitivity list and then calls the relevant CSP process (be it the P or P' process).

9.5 Generating CSP Parallel Process

Once all the CSP P and P' processes have been generated it is necessary to create a synchronisation between all the P processes to allow synchronisation to occur on the dd channel. This behaviour is once again generated by an EOL operation, CreateParallelComposition, as the behaviour is not explicitly given by the VHDL source description.

```
operation CreateParallelComposition() : csp!Process{
var cExpression;
if(csp!Process.allInstances().size()>2){
cExpression := new csp!ExpressionWrapper;
var currentParallel;
var first := Boolean;
first := true;
for(proc in csp!Process.allInstances()){
if(not proc.name.endsWith("'")){
var cProcRef := GenerateProcRef(proc);
if(not first){
var cParallel := new csp!InterfaceParallel;
cParallel.alphabet := new csp!ExplicitAlphabet;
var parameterList := new  csp!ProcessParameterList;
var parameter := new csp!Parameter;
parameter.name := "dd";
parameterList.containsTypeItem.add(parameter);
cParallel.alphabet.alphabetEventList :=  parameterList;
cParallel.nestedExpression.add(currentParallel);
cParallel.nestedExpression.add(cProcRef);
currentParallel := cParallel;
first := false;
}else{
currentParallel := cProcRef;
first := false;
}
}
}
cExpression.nestedExpression := currentParallel;
}else{
cExpression := GenerateProcRef(csp!Process.allInstances()
.select(proc|  not(proc.name.endsWith("'"))).first());
}
var paramList := GenerateParallelParameterList(csp!Process.allInstances());
var cProc := new csp!Process;
cProc.name := "P_" + vhdl!Entity.allInstances().first().name;
cProc.definesParameters := paramList;
cProc.processStatement := cExpression;
return cProc;
```

```
}
```

The CreateParallelComposition operation performs a select all on all the CSP processes defined, and from these only selects those which are CSP P processes. For each of the CSP P processes, a process reference is then made, including a process assignment list. For each process reference created, it is then put in a parallel composition with the other processes selected.

Once the parallel composition for all the processes has been made, the generation of a new process, named after the entity is created along with a parameter list which encompasses all the process parameters used within the system. The parallel composition of all the other processes are then assigned to this newly defined process, thus describing the entire system and providing a working CSP model of the VHDL description.

10 Conclusions

In this section we first highlight the current restrictions of the transformation tool before then providing a review of the transformation framework that has been developed, pointing out areas for expansion and further development.

10.1 Current Limitations

Whilst we have built a working tool which will translate VHDL into CSP using the mapping rules presented in Section 6, there are currently some limitations on what the tool is able to translate.

10.1.1 VHDL Libraries

At present the tool does not handle the VHDL "package" or "uses" keywords, and instead the only two Port types it is able to deal with are "BIT" and "STD_ULOGIC". These have been written into the tool within a post processor. However, this has been implemented as a current solution and the scope of the tools that have been used to develop this translation framework means that there should be no restrictions should further development be required to deal with this aspect of the VHDL language.

EMFText has the capacity, using post processors to access multiple files when reading in source text and generating the equivalent model. Whilst some additions may need to be made to the VHDL meta-model to deal with

all the information provided in the VHDL library files, the ETL translation rules should not require altering as these work with in translating the VHDL DataType model elements already; the extension would be to the VHDL CSM in recognising the new data types.

10.1.2 Multiple IF Statements

The translation framework, at present, is not able to handle multiple VHDL IF Statements inside one process. This issue has arisen due to the differences in VHDL and CSP: VHDL allowing multiple Process Expressions within one process and CSP requiring that there only be one Process Expression per process.

To counter this issue, several EOL operations have been written which detect whether there are more than one set of guarded CSP Choice statements. When multiple guarded Choice statements are detected for a single CSP process, the operations merge them into one single Process Expression; nesting all the guarded choice statements within one another and merging the assignments into a single let within for each branch.

Bug tracking in ETL appears to be very difficult, only allowing printing to the console as an option for highlighting issues etc. This bug currently lies in the EOL operation: MergeChoiceExpressions.

10.1.3 Simple Process

Due to the time restrictions the rules which would be associated with this transformation have yet to be written. Due to the implicit sensitivity list the transformation rule is not straightforward to implement correctly.

10.1.4 Vectors

Due to the time restrictions, the additional operations required for transforming VHDL Vectors into CSP Vectors, including the generation of parameters to the *dd* channel has not yet been undertaken. In order to add this functionality, operations would need to be written to define the relevant transformation to new CSP functions (for some of the vector manipulation in VHDL) as well as operations for generating CSP loops etc., as required.

10.2 Tool Evaluation

This tool has been used to generate four different VHDL descriptions into CSP at present. However, it is believed that there should be no issue in testing this tool with larger range of VHDL source files. As documented in Section 10.1, there are presently some restrictions associated with the transformation framework but as long as these aspects of the VHDL language are not used within the source VHDL description then there should be no reason that a syntactically correct CSP target script could not be generated.

The framework that has been developed is made of three distinct parts; a CSP CSM, a VHDL CSM and a model transformation script from VHDL to CSP. Both the CSP and VHDL CSMs may be used in isolation with each other, each forming a complete tool in their own right, which could be used in many different scenarios.

The CSP CSM contains all the CSP syntax that is commonly used and could be expanded to include those functions less commonly used. This was not carried out as part of this work as these extra functions fell outside of the mapping requirements of the VHDL to CSP mapping. However, the CSP CSM is a component of the framework and can be used outside of the VHDL to CSP transformation framework and can be considered an complete component for use alongside other transformations as required.

Meanwhile the VHDL CSM currently covers a small subset of VHDL and would be more versatile if extended to include a larger range of the VHDL syntax. This would enable the CSM to be used as a parser for a wider range of VHDL descriptions as well as providing a basis for an extension to the current VHDL to CSP model transformation. As with the CSP CSM, the VHDL CSM can be used in isolation to the VHDL to CSP transformation framework and can used along side other transformations as required.

With respect to the model transformation tool that has been written in ETL, there is a known issue with the current version which needs to be resolved to allow a larger range of VHDL descriptions to be transformed. Also the currently unwritten transformation rule for a VHDL simple process (one with an implicit sensitivity list) must also be added to extend the tools capabilities. However, if further functionality were to be added to the tool, the mappings between VHDL and CSP (as discussed in Section 6) would need to be developed further to capture other functions in VHDL and provide their equivalent in CSP.

The transformation framework that has been developed has provided a proof of concept for generating formal models of hardware based systems. To further this work, the restrictions of the tool must be addressed as well as the

current mappings from VHDL to CSP. In order to fully utilise this transformation framework more VHDL descriptions should be used for transformation, not only to test the tool, but also to provide larger CSP scripts which can then be used within the model checking tool FDR to determine the restrictions imposed by the model checker, not just the transformation tool.

References

[1] *Textual Editing Framework - Tutorial.*

[2] Jean-Raymond Abrial. *The B Book: Assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.

[3] Daniel Balasubramanian, Anantha Narayanan, Christopher P. van Buskirk, and Gabor Karsai. The graph rewriting and transformation language: Great. *ECEASST*, 1, 2006.

[4] Artur Boronat and Reiko Heckel. Sierpinski triangles in MOMENT2-GT. Technical report, University of Leeds, September 2007.

[5] Jean Bovet and Terence Parr. Antlrworks: an antlr grammar development environment. *Softw., Pract. Exper.*, 38(12):1305–1332, 2008.

[6] Duc-Hanh Dang and Martin Gogolla. Precise model-driven transformations based on graphs and metamodels. In *SEFM*, pages 307–316, 2009.

[7] Neil Evans. Integrating formal methods with informal digital hardware development. In *10th International Workshop on Automated Verification of Critical Systems*, 2010.

[8] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *SPLASH/OOPSLA Companion*, pages 307–309. ACM, 2010.

[9] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, chapter 9, pages 114–129. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2009.

[10] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, Christian Wende, and Marcel Böhme. Generating safe template languages. *GPCE '09:*

Proceedings of the eighth international conference on Generative programming and component engineering, Oct 2009.

[11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[12] IEEE. Ieee standard vhdl synthesis packages. Electronic, 1997.

[13] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. Tcs: a dsl for the specification of textual concrete syntaxes in model engineering. In Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, editors, *GPCE*, pages 249–254. ACM, 2006.

[14] Frdric Jouault and Jean Bzivin. On the specification of textual syntaxes for models. In *Eclipse Modeling Symposium*. Eclipse Summit, October 2006.

[15] Dimitrios Kolovos, Richard F. Paige, Louis M. Rose, and Fiona A.C. Polack. *Epsilon*. Sourceforge, January 2009.

[16] Dimitrios S. Kolovos. Epsilon website. Website, July 2010.

[17] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The epsilon transformation language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *ICMT*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2008.

[18] Jochen M. Küster, Shane Sendall, and Michael Wahler. Comparing two model transformation approaches. In *Proc. Workshop on OCL and Model Driven Engineering*, 2004.

[19] M. Leuschel and M. Butler. ProB: A Model Checker for B. In *FM 2003*, LNCS, pages 855–874.

[20] Anantha Narayanan and Gabor Karsai. Verifying model transformations by structural correspondence. *ECEASST*, 10, 2008.

[21] OMG. Meta object facility (mof) 2.0 query/view/transformation specification, April 2008.

[22] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[23] Steve Schneider. *Concurrent and Real-Time Systems: the CSP Approach*. Wiley, 1999.

[24] Andy Schürr. Specification of graph translators with triple graph grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *WG*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1994.

[25] EMFText Developers Mirko Seifert. Personal communication. E-mail, September 2010.

[26] Mirko Seifert. *EMFText User Guide*. Reuseware - TU Dresden, September 2010.

[27] Helen Treharne, Edward Turner, Richard F. Paige, and Dimitrios S. Kolovos. Automatic generation of integrated formal models corresponding to uml system models. In Manuel Oriol and Bertrand Meyer, editors, *TOOLS (47)*, volume 33 of *Lecture Notes in Business Information Processing*, pages 357–367. Springer, 2009.

[28] Edward Turner and Helen Treharne. Model Driven Development experiences using iUML and Epsilon. Technical Report SystemB-TR-08-v02, Department of Computing, University of Surrey, 2009.

[29] Edward Turner, Helen Treharne, Steve Schneider, and Neil Evans. Automatic generation of CSP || B skeletons from xUML. In John S. Fitzgerald, Anne Elisabeth Haxthausen, and Hüsnü Yenigün, editors, *ICTAC*, volume 5160 of *Lecture Notes in Computer Science*, pages 364–379. Springer, 2008.

[30] Frank Vahid and Roman Lysecky. *VHDL for Digital Design*. John Wiley & Sons, 2007.

[31] Dániel Varró, Márk Asztalos, Dénes Bisztray, Artur Boronat, Duc-Hanh Dang, Rubino Geiß, Joel Greenyer, Pieter Van Gorp, Ole Kniemeyer, Anantha Narayanan, Edgars Rencis, and Erhard Weinell. Transformation of UML Models to CSP: A case study for graph transformation tools. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *AGTIVE*, volume 5088 of *Lecture Notes in Computer Science*, pages 540–565. Springer, 2007.

[32] Mark Zwolinski. *Digital System Design With VHDL*. Prentice Hall, 1st edition, 2000.

Document Version Control		
Author	**Date**	**Notes**
James Sharp	17/01/11	Initial creation
James Sharp	18/02/11	Final draft for review
James Sharp	02/03/11	Reviewed and ready for submission

A Appendix

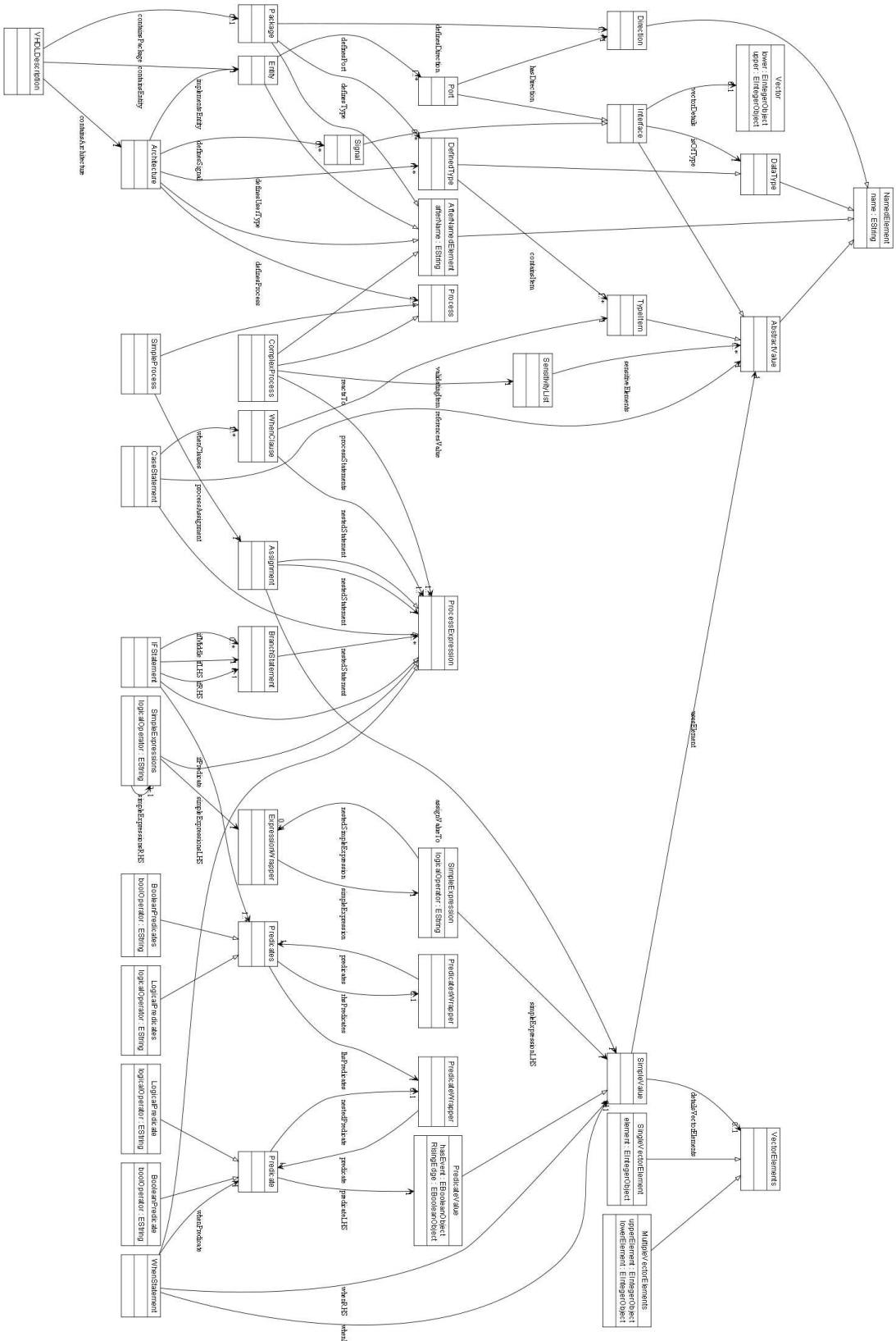


Figure 24: VHDL meta model

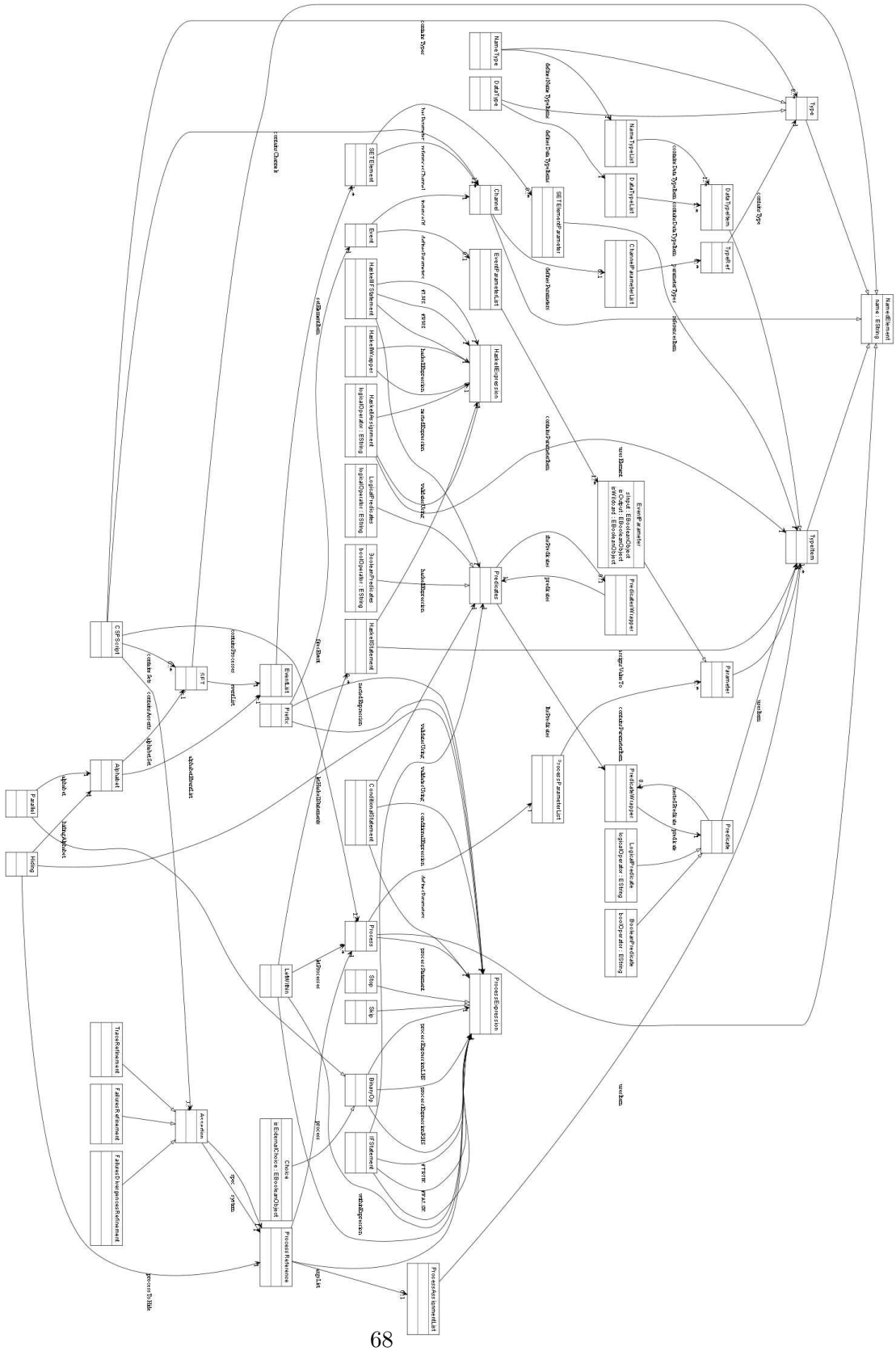


Figure 25: CSP meta model