6. Processing design description files

6.1 Overview

In accordance with requirements **R6** and **R7** and the integration principle (2), stated on page 40, we assume that design information is exchanged between encapsulated design tools and the design information management (DIM) service of an EDA framework by means of design files. In the EDA domain, hardware description languages (HDL) are used to encode design information and unambiguously define design semantics. Due to the many aspects of electronic design (domains, levels of detail) and varying design tool requirements, there are many HDLs in frequent use. Some languages cover many aspects of a design, others are specific to a single domain, level of detail or even design tool. Recent years, however, have seen a strong trend towards HDL standardization as only with standardized HDL syntax and semantics there can be seamless interoperability between design tools in an open, integrated design environment. Some design description languages and their preferred area of use are listed in Table 10.

language	mostly used for ...
VHDL	behavioural and structural descriptions on algorithm and functional block levels
Verilog	behavioural and structural descriptions on register transfer and gate levels
EDIF	netlists and graphical data for schematics and physical layout
GDS-II	graphical data for physical layout
C++	behavioural descriptions on algorithm level
Prolog	behavioural descriptions on architecture and algorithm levels

Table 10. *Some design description languages and their preferred area of use. Note that we have included two ordinary programming languages which are frequently used for behavioural descriptions on low levels of detail.*

This diversity of languages in design use implies that vendors of EDA frameworks make an arbitrary selection of the languages they want to support, mainly based on customer demands. As design methodologies are different from company to company and also evolve constantly, a framework vendor continuously has to support new design description languages. He may quickly be in a position in which supporting new languages* based on customer demands forms a major part of the overall effort put into framework development. There are three viable approaches to solve this problem:

1. Not to support design description languages at all. The JESSI Common Framework, for example, never looks into a design file but relies on the designer to chop his designs in pieces small enough to be managed individually by the DIM service of the framework.
2. To make an arbitrary selection of the languages to be supported. All design tools integrated with such a framework (e.g. Nelsis) have at least to provide converters from their native languages to the languages supported by the framework. If the supported language selection is not appropriate, and unsupported languages have to be used by design tools because a certain domain or level of detail cannot be represented, design files have to be manipulated manually by the designer to be digested by the framework.
3. To provide a framework service that allows a framework administrator or even design team members to add support for a desired design description language on the fly. Of course support for standard languages can already be provided by the framework vendor. Incorporating support for specific design description languages into an existing framework can even form the basis for a flourishing business niche.

Clearly, the third alternative is the most flexible approach. Without special support, however, the effort of writing the necessary language processors for complex languages like VHDL is a major engineering task and certainly out of the scope of a design team wanting to design chips.

---

*. “to support a design description language” here means to be able to manage structural information contained in design files by the DIM service and by framework tools like query interfaces and browsers.

Traditionally, writing language processors is accomplished by any of the following alternatives:

1. combination of the UNIX tools *sed/awk*, or dedicated report generation languages like *perl*[*]
2. *lex* generated lexical scanner
3. *yacc* generated parser
4. *yacc* generated parser enhanced with semantic actions to resolve references

Using (1) requires little coding effort and development of language processors can be incremental due to the interpreting nature of the tools applied; the resulting language processors, however, have only crude recognition capabilities and high probability of failure due to syntactic variations in the processed design files. Alternative (2) requires the compilation of generated "C" code; it provides more flexibility in the actions that can be triggered when recognizing a language construct. Language specification still is based on regular expressions and therefore too crude for most real design description languages. Alternatives (3) and (4) have detailed, syntactical language recognition capabilities. The coding effort is high because every language detail must be processed. The language processors thus created are highly dependent on fine-grained detail of the processed language. As design tools often rely on *dialects* of a standard language, this is an undesirable feature, even more so as the processing of every single detail of a design description is not necessary for successful tool encapsulation.

The choice of standard compiler construction tools like *yacc* and *lex* suggests that the processing of design files is very similar to the initial parsing phase of a compiler for a particular design description language. There are, however, significant differences:

- Lexical and syntactical analysis in a compiler try to digest every single detail of a design description to be able to derive an internal representation that matches the information content of the design description as thoroughly as possible. For the purpose of design tool encapsulation, only a small part of design files has to be analysed down to single lexems. Large parts can simply be skipped and regarded as design representation, uninterpreted as far as design information management is concerned.

[*]. Perl gains more and more acceptance as one of the primary scripting languages in EDA environments (cf. [CFI-EII 95])

- A large part of the lexical and syntactical analysis phases of a compiler is concerned with error detection and recovery. As this tedious work is performed by design tools already, we can rely on a design description to be syntactically correct for the purpose of tool encapsulation.
- Another great effort in the initial phase of a compiler is the construction and maintenance of a symbol table for semantic analysis. We are not interested in the exact data or control flow and can therefore greatly reduce the effort of symbol handling. The close cooperation of the new encapsulation service with a DIM framework service with versatile querying facilities renders manually constructing a symbol table obsolete.

On the other hand, there are also requirements in the realm of tool encapsulation that are not even an issue with ordinary compiler technology:

- All text that is considered design representation does not have to be interpreted in any way but must be saved complete for later perusal by design tools. Once an ordinary compiler has built an internal representation of the input read, it can safely forget about the exact textual representation, with the possible exception of line numbers for error messages.
- Global text that neither belongs to structural information nor is associated with single chunks of design representation must be preserved to be able to reconstruct complete design files later. In a compiler, this text is sometimes already resolved by a preprocessor and not even seen by the actual compiler or it serves as context information and is also translated to objects in an internal representation.

We conclude that an EDA framework has to offer a powerful and flexible service to incorporate support for specific design description languages. While this service certainly should provide the language processing power of conventional compiler construction tools, additional requirements have to be met. To understand the implications of these requirements, we now take a closer look at which kind of language processing is actually required to support a selected design description language in an EDA framework.

6.2 Requirements

6.2.1 Design file import

Design files can either be created manually by a designer or can be the result of a design tool execution. Whereas in the former case the designer would be free to arrange the files that describe a complete electronic system in any way that is required for design information management, design tools generally do not provide this flexibility. This implies that design files have to be analysed to extract the structural information necessary to perform design information management. Corresponding to the notions defined in our conceptual schema the following items have to be created:

- interface, implementation, and configuration *objects*
- *attribute* values for names, designer, modification dates
- *relationships* for compositional hierarchy, version derivation, and equivalence

In addition, textual design representation has to be associated with objects so that it can be retrieved again on export.

6.2.2 Design file export

When performing a design step by running a tool, design information must be exported from the DIM service to design files. This can be achieved by traversing the graph of objects managed by the DIM service according to our conceptual schema, starting from a selected root object, and retrieving the design representation that is associated with the objects visited.

The existence of component configurations in a design hierarchy complicates matters. In cases where a design description language is used that supports configurations (like VHDL), the configurations managed by the DIM service can be translated into appropriate configuration declarations in the language and emitted along with the selected interfaces and implementations. If, however, the design language does not have a notion of "configuration" or the notion supported by the language grossly deviates from the notion used by the DIM service, configurations have to be resolved and emitted as static bindings between compound implementations and their components.

6.2.3 Associating binary objects with structural information

The import and export mechanisms described above assume that design files have a well-defined structure and contain some kind of parseable and printable description of the design. Quite often, however, design tools produce and read an opaque, intermediate design description. Examples for this kind of files are results from VHDL analysers, simulation results, or schematics in undocumented, proprietary formats. When no specification of the lexical and syntactical structure of such opaque descriptions is available to the tool integrator, the files containing these descriptions can only be manipulated as a whole. If such files can be associated with a specific design object in a composition hierarchy they can be imported and exported together with it. If they cannot be associated with a specific design object, they have to be attached to the root object of the composition hierarchy and have to be imported and exported whenever a sub-module of this root object is imported or exported. We now introduce our approach to providing a framework service that fulfils these requirements.

6.3 Approach

We have developed a toolkit that can be offered as a new service by framework vendors to their customers or can be used by framework vendors to build design language support for standard languages into their products. We follow the principle of separating syntax specification from the actual creation of objects on import and retrieval of design information on export. The syntax specification of a given HDL is largely static and parsing can be greatly simplified when using standard compiler construction tools like yacc and lex. The actual interfacing to a DIM service is likely to evolve as requirements on design information management in a particular design situation mature. This part of language processing can therefore benefit from the accessibility gained from coding in an interpreted extension language. As suggested in Section 4.3, we use an extended Tcl for this purpose.

6.3.1 Syntax specification

As opposed to the situation in an ordinary compiler in which a program in some high-level programming language is translated into a program in a different language like assembler or machine language, input and output to and from the DIM service is in the same language.[*] As such, it is desirable to use the same syntax specification for both import and export.

As stated earlier, we can assume that only little information out of the whole information content of a design file is needed for successful design information management. We can furthermore assume design files to be syntactically correct. These two assumptions greatly simplify the amount of syntax that actually has to be specified exactly. On the other hand, partial analysis poses additional problems not normally encountered in syntax analysis. While large parts of a design file can be analysed according to a lexical and syntactical specification that generously ignores most of the tedious detail of a complete design description, other parts have to be analysed down to single lexems in order to retrieve all the information necessary for design information management. For example, the analysis of a VHDL file only has to reveal the names and extent of contained entity declarations and architecture bodies, but *all* the information specified in a configuration declaration has to be retrieved.

We solve this problem by allowing modular language specifications. The specification of a design description language may consist of several modules, each of which defines the lexical properties and syntax for a part of the whole language. There is always a main module that provides an entry point for syntax analysis and generation. With the recognition of certain constructs, another module may be invoked that takes over syntax analysis from where it was invoked. Once a complete program defined by the language of the subordinate module has been recognized, control is returned to the caller. As different modules can define completely different lexical properties, modular language specifications allow to analyse design files at varying granularities. The net result is a major reduction in the amount of language detail that has to be specified for design file analysis while we are still able to digest fine-grained detail where appropriate.

We use the same language specification to define both input and output. As such, the specification language needs to define both lexical properties and syntax in one place. In addition, grammar symbols may be tagged for ease of reference during import and export. The specification may be decorated with output formatting information.

---

[*]. We regard situations in which design information has to be converted from one language to another as the task of dedicated design tools.

6.3.2 Parse tree construction

Once analysed, the design information retrieved from design files must be organized in a structure that is easily traversed from extension language scripts to create the corresponding objects, attribute values, and relations in the DIM service. A parse tree can be created automatically during syntax analysis, and, combined with an extension language binding, is well suited as intermediate representation. We choose a generic data structure for the parse tree that is independent of the actual language syntax.* Parse trees are constructed from generic nodes that carry type information, user defined tags, and text that stems from analysed designed files.

We provide no operations to prune the automatically created parse trees, because there is little need for this extra complexity. With modular language specifications, parse trees proved to be small compared to those created from complete fine-grained syntax analysis. Parse trees directly reflect the extended BNF by which the syntax is specified. Tree nodes that stem from certain syntax rules carry the name of this rule as their type. Lists, possibly with separators like "," or ";", create dedicated list nodes. Text ranges that are parsed by nested specification modules result in sub-trees.

6.3.3 The creating of objects during parse tree traversal

The parse tree created automatically during syntax analysis has to be traversed to determine the objects, attribute values and relationships to be created in design information management. While all the data from the analysed design files is guaranteed to be in the parse tree, it is obscured by intermediate nodes that stem from the structure of the input grammar. We define traversal operations that access child nodes based on their type and optional tag values, ignoring intermediate tree nodes that do not carry valuable information. These traversal operations are bound to Tcl commands, so all tree processing can be done by extension language scripting. While visiting tree nodes, the programmer can store values found in variables and use their values to fill the attributes of objects in design information management once enough information is gathered.

*. Another popular approach for parse tree construction is the definition of special node types for individual grammar rules. While this seems attractive from a purist's point of view, it would require to define a subtype of the basic node type for each grammar rule. On the other hand, modules with well-defined interfaces rather than deep inheritance trees have proven more usable in practice [Udell 94].

We prefer the programmed approach to the more descriptive approach of statically linking language constructs to DIM object types because thus

- the relatively fixed language definition can be separated more cleanly from the mapping to DIM object types that is likely to evolve as information management requirements change;
- more flexibility can be offered to the integrator to gather values from anywhere in the parse tree. Otherwise, a complicated formalism would be necessary that would yet be likely to fail in unexpected situations.

6.3.4 Linking types to parse trees

Export from the DIM service to design files is an altogether different matter. Here we start from a well-structured object graph that already has all the information needed and is attached to objects in the DIM service so that it is easy to access. For every language specification module, we automatically generate a function *unparse* that takes the name of a syntax rule to be unparsed as parameter as well as a variable number of parameters that match variable components in a derivation of the named rule. DIM object types are statically associated with an invocation of the *unparse* function that determines the syntax rule according to which objects of this type are to be exported, as well as the association of attribute values to the rule's variables.

With this association in place, export of a DIM object into a valid design file is a matter of selecting the object and invoking its unparse method. The unparse method creates a parse tree with constant text in the right places and variables filled from the DIM database. Only a simple preorder traversal is needed to transform such a tree into a stream of text in the desired syntax.

Now that we have outlined our approach, we will present the technical realization of this new framework service in Sections 6.4 through 6.6, use VHDL as a case study in Section 6.7 and compare our approach with existing work in Section 6.8.

6.4 A specification language for HDL syntax

Our specification language has to specify the syntax of a design description language for both import and export. As such it allows to define both lexical properties and syntax in a single specification file. We have already described the basic language in Section 3.2, so here we only add the constructs that are specific to encapsulation support.

As a rule, the generated parser modules assume that they process syntactically correct design files. They therefore read over lexems that are not explicitly defined in the language specification. This behaviour can be exploited to simplify the language specification: Only those languages elements have to be defined that either carry valuable information for design information management or those that are necessary to recognize the syntactical structure of a design description. All other text will be ignored by syntax analysis but will nevertheless appear as text attributes in the parse tree to ensure that no information is lost for later export.

6.4.1 Nested parser modules

As specified in Section 3.2, the main part of a language specification consists of a list of grammar rules. A rule may be marked with the property *-syntax*. This property marks the right-hand side of a rule to be the prefix for a nested specification module. Once the prefix is recognized, it is pushed back into the input and a subordinate parser module with the same name as the rule's is used to specify the input that follows.

When the subordinate parser has recognized a valid text in the language that it defines, it returns the control back to the invoking parser.

rule rule {
  "rule" ID **opt** { "-syntax" } alternatives
}

6.4.2 Generating a parser from specifications

Figure 25 illustrates the process of generating a parser module from a language specification. A parser module provides the two functions *<module>parse* and *<module>unparse* as its public interface, and a wrapper function to create a parse tree from an extension language command.

There are two variants of the parse function:

```
Tree* parse (char* fileName);
Tree* parse (Tree* parent);
```

While the first variant is invoked directly from the extension language binding to create a parse tree of the named file, the second variant is used to pass control to a subordinate parsing module. The unparse function has the following signature:

```
Tree* unparse (char* rule, ...);
```

The first parameter always denotes the syntax rule to be unparsed. The following parameters supply name/value pairs for variable parts in a derivation of the named rule. We will explain the wrapper function to create parse trees when we describe the extension language binding to parser modules.

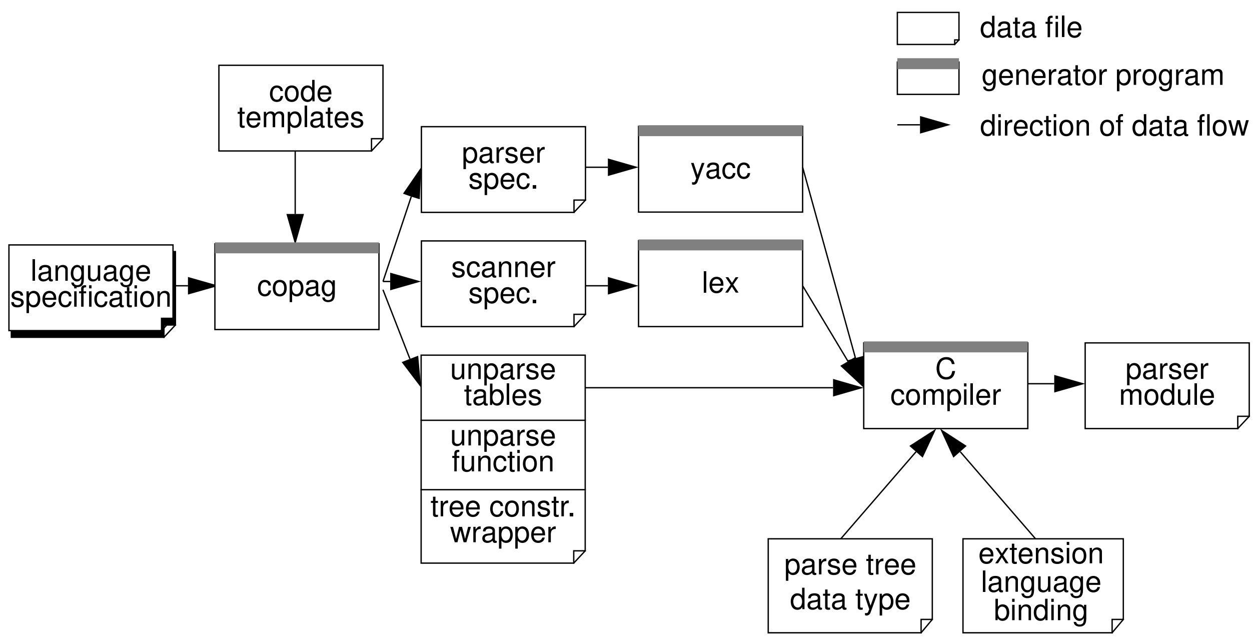

Figure 25. *How to generate a parser module from a syntax specification. The preprocessor* copag *reads one or more specification modules and translates them into a parser specification in* yacc *format, a scanner specification in* lex *format, unparse tables, an unparse function that uses these tables, and a wrapper function for the extension language binding of language specific parse tree constructors. The parse tree abstract data type itself is language independent and is simply linked into the created module together with a set of wrapper functions for extension language bindings.*

6.5 The generated parser and tree constructor

6.5.1 A conceptual schema for parse trees

We have chosen to construct parse trees from generic nodes that are independent of a particular language syntax (Figure 26). Each tree node is associated with a parse tree. A parse tree not only represents the root of a tree but carries information common to all the nodes in it. An important property of our parse trees is that the tree root is associated with all the input text recognized, including all text read over by the parser. This is necessary because regardless of the (in-) completeness by which the language specification defines the lexical properties and syntax of a design file, the whole file is considered syntactically correct and therefore contributes to a complete design description. All text is to be managed by the DIM service. To achieve this, we do not read a design file byte by byte with stream-i/o functions but use the virtual memory management facility offered by all modern operating systems to map the complete design file into virtual memory. The individual tree nodes do not store copies of the lexems they were associated with during parsing. They rather store the start and end offsets of their associated text regions in the mapped file text.

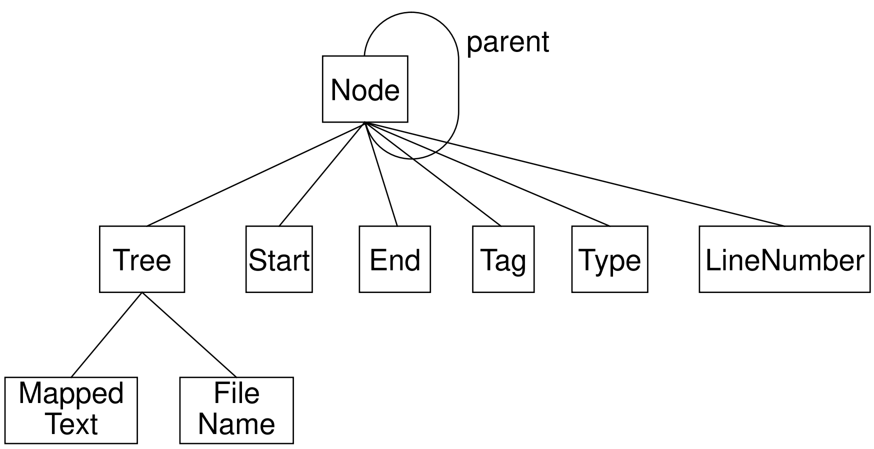

Figure 26. *Schema for parse trees. An analysed design file is mapped into virtual memory and is associated with the tree. Tree nodes only carry the start and end offsets into this mapped text.*

6.5.2 Creating a parse tree

The creation of parse tree nodes during parsing is straightforward. A leaf node is constructed when a lexem is recognized as token in the input. Its start and end values are determined by the respective start and end offsets in the input file. Intermediate tree nodes are created when a grammar rule is reduced. They receive the start value of their left-most child as start and the end value of their right-most child as end value. Thus all the text lying in between the left-most and the right-most children of a tree node is considered to belong to this node, even if there is no child that actually represents this text.

6.5.3 Nested parser modules

Grammar rules that have the *-syntax* property do not directly create a tree node when they are recognized, but delegate its construction to a subordinate parser module. For this purpose the parse procedure of the desired parser module is invoked with the current parse tree as parameter. When the subordinate parser has successfully constructed a parse tree for a region of input text, it returns control to the semantic action of the invoking parser module. There the tree nodes resulting from the right-hand side of the current grammar rule are discarded and replaced by the root of the parse tree created by the subordinate parser.

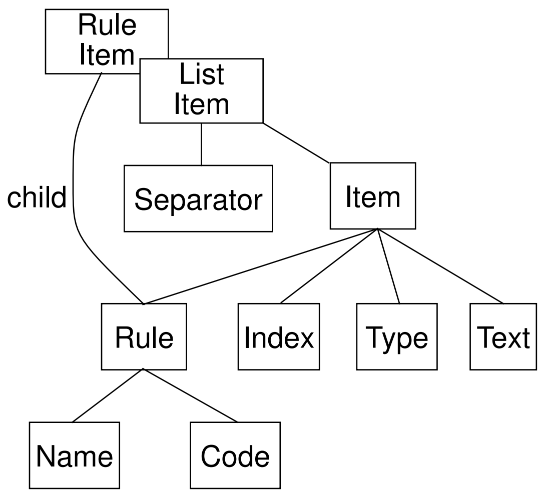

Figure 27. *Schema for the unparse tables. Rules carry their name and a numeric code. An item may either be a literal or optional (both stored as object of type* Item*), or it may be a reference to a subordinate rule. List items carry a separator if one was specified in the language specification.*

6.5.4 Unparsing

We have chosen to generate one *unparse* function per parser module that receives its knowledge about grammar items from two tables, *RuleTable* and *ItemTable* (Figure 27). These tables can be statically created during the run of the specification preprocessor *copag*. An alternative solution would have been to generate one function per grammar rule. We favour the table driven approach because it reduces code size and has less code duplication. The *unparse* function starts from a rule whose name is passed as parameter and creates a parse tree node for it. The function then iterates over the items in the rule. For each item, a child of this parse tree node is created by the following actions:

- If the item is a literal, the created child takes the literal text as value of its *text* attribute.
- For all other item types, the unparse function first checks if a named parameter has been given to the function that matches this item. A named parameter matches an item if it is yet unused and
  - if the item is tagged and the parameter name equals the tag, or
  - if the item is a *RuleItem* and the parameter name equals the name of the referenced rule.

  For maximum flexibility, a child node can be created from matching parameters in one of the following ways:
  - the parameter text is taken as the node text.
  - the parameter denotes a tree created elsewhere which is taken as the child.
  - the parameter denotes a list of DIM objects. A new tree node is created as child. This child receives the result of evaluating the unparse method on each object in the list as children.
- If no matching parameter could be found, two cases are distinguished:
  - the item denotes a list of one or more elements. In this case no child node can be created automatically. The *unparse* functions fails.
  - the item is of type *RuleItem*. The unparse function calls itself recursively, passing the rule and the named parameters along. The result of this recursive call is taken as the new child node.

The overall result of a successful invocation of *unparse* is a parse tree that is created according to the syntax of the rule passed to the function with all the variables filled by parameters passed to *unparse*. A simple preorder traversal writes this tree into a design file. Each node visited during this traversal is checked as to whether its *text* attribute is set. If so, the text is emitted to the file and traversal proceeds with the next sibling. If the *text* is not set, traversal proceeds with the children of the current node. We will present an example of unparsing in our case study of the VHDL language.

6.6 Extension language interface

6.6.1 Parsing design descriptions

Parse trees are represented by the abstract class *ParseTree* in the extension language.[*] The only method that *ParseTree* offers is the one to retrieve the tree root. For every parser module linked into the extension language engine there is a subclass of type *ParseTree*, named after the language specification from which this module was created. For example, our case study for the VHDL language comprises of two specifications, one for the overall structure of VHDL files, named *vhdl*, the other one for configuration declarations, named *config*. We therefore have two subclasses of *ParseTree* in this example, named *vhdl* and *config*. Each parser module contains a *parse* function that takes as parameter the name of a design file to be parsed and returns a parse tree after a successful parse. These *parse* functions are accessed through the constructors of the corresponding subclasses of *ParseTree*. For example, we create a parse tree from the VHDL file dp32.vhdl by issuing the Tcl command

```
vhdl t -file dp32.vhdl
```

This command creates a parse tree *t*. When an error is encountered during parsing, the constructor fails and no parse tree is created. In this case an error message is returned as the result of the command. The root node of this tree can be retrieved by issuing the method *root* to the parse tree:

```
set root [t root]
```

This command assigns the identifier of the tree root to the Tcl variable *root*. Nodes have three data members visible from the extension language: *type*, *tag*, and *text*. *Type* contains an enumeration value that encodes the rule from which this node was created. The domain of this enumeration is defined by the tree to which a particular

*. "Abstract class" means that the user cannot create objects of this class but only of subclasses.

node belongs. For example, the code 24 is assigned to nodes of type *architecture*, created by the *vhdl* parser. The code 24 corresponds to node type *block_configuration* for nodes created by the *config* parser. *Tag* contains the tag from a grammar symbol if any was defined. For example, the *config* parser assigns the tag *entityName* to the node created for the third identifier of *architecture_body* nodes. Finally, *text* contains the text region between the start and end offsets stored for a particular node. For the root node of a parse tree, this is always the complete input.

6.6.2 Tree traversal

Once a design file is successfully parsed and a parse tree is constructed, the information relevant to design information management needs to be extracted from the tree. For this purpose two methods are defined on tree nodes that allow flexible tree queries and traversal from within extension language scripts. As the tree is built during language parsing by automatically generated code, it contains much detail that is irrelevant to information extraction. The extension language statements take this into account by allowing to look at “interesting” nodes only. Two statements are defined:

```
rule traversal {
   node:IDENTIFIER “all” list { qualifier }
   “-var” var:IDENTIFIER code_block
 | node:IDENTIFIER “one” list { qualifier }
}
rule qualifier {
   “-type” type:IDENTIFIER
 | “-tag” tag:IDENTIFIER
}
```

The first variant executes a Tcl code block for every node, optionally considering only those nodes with selected type and tag in the tree rooted at *node*. The node currently looked at is available in the variable named *var* within the code block. The code block may contain *break* and *continue* statements to break out of the traversal completely or to continue with a sibling of the current node. The second variant traverses the tree rooted at *node* and breaks at the first node of type *node:IDENTIFIER* or fails if no such node exists.

The method on tree nodes that implements these commands realizes the following algorithm:

Algorithm 1. Node::traverse

in	set<tags>	tags to be matched against node tag
in	set<types>	types to be matched against node type
in	variable name	Tcl variable to which to assign the node identifier currently processed
in	action code	Tcl code to evaluate for a node with matching tag and type
in	one	*one* or *all* command
out	success/failure	return code resulting from evaluating the action code

```
if (node type and node tag match inputs) {
 set Tcl variables for current node type,
       current node tag, and current node text
 set Tcl variable named by parameter to current node identifier
 evaluate action code passed as input
 if (return code from action evaluation == OK) {
       if (one) {
             return BREAK;
       } else {
             return (return code from action evaluation);
       }
 }
 if (node type is list or node has children) {
       for each list element or child {
             traverse recursively,
             passing input tags and types, variable name,
             action code and one flag
```

```
            switch (return code from recursive invocation) {
            case OK or CONTINUE:
                break:
            case BREAK:
                return BREAK;
            case RETURN:
                if (node type is list) {
                    insert returned node identifier at current list position
                } else {
                    replace current child with returned node identifier
                }
                break;
            default:
                return (return code from recursive invocation);
            }
        }
    }
```

Node::traverse effectively performs a preorder tree traversal, checking each node visited for matching tag and type. If a matching node is encountered, some Tcl variables are set to establish the execution context for the action code, and the action code is evaluated. Depending on the return code of this evaluation, traversal either continues with the children or list elements of the matching node, or breaks off completely and continues with the next sibling of the matching node. By issuing appropriate *break*, *continue*, and *return* statements within the action code, and by recursively calling traversal commands from within the action code, the programmer is given fine control over the exact tree traversal. In the Section 6.7, we will see that this flexibility is in fact needed when we show in detail how to transform VHDL configuration declarations into DIM object graphs.

6.6.3 Unparsing

Unparse functions are defined globally in the scope of their parser module. As described above, *unparse* functions determine the parameters they actually consume during a traversal of the rule and item tables.

The following extension language command is defined:

```
rule unparse {
   module:IDENTIFIER "::unparse" "-rule" rule:IDENTIFIER
   list { "-" IDENTIFIER "{" parameter_value "}" }
}
rule parameter_value {
   "-text" LITERAL
  | "-code" tcl_code
  | "-node" node:IDENTIFIER
  | "-list" tcl_code
}
```

An *unparse* command is invoked for a specific rule. Tree node creation is controlled by a keyword that specifies a parameter to be either

- plain text,
- Tcl code to be evaluated to get some text as result,
- a tree node identifier, or
- a piece of Tcl code that - evaluated - results in a list of DIM object references. A list of tree nodes is yielded from this list by evaluating the *unparse* method for each of these objects.

For example, a block configuration in our VHDL language specification is defined as follows:

```
rule block_configuration {
 "for" implName:IDENTIFIER
      list { use_clause }
      items:repeat { configuration_item }
 "end" "for" ";"
}
```

A sequence of extension language commands that writes a block configuration to standard output would be

```
set tree [config::unparse -rule block_configuration
 -implName { -code $parent_DesignObject Name }
 -items { -list set ComponentConfigurations } ]
$tree export stdout
```

The *unparse* function defined in the *config* parse module is invoked. The implementation name is found by evaluating the method *Name* on the object stored in variable *parent_DesignObject*. Component configurations are filled in by evaluating the *unparse* methods on all the objects stored in the variable *ComponentConfigurations*. Note that no parameter has to be given for the list of *use_clauses* as this list may have zero or more elements.

6.7 Case study: VHDL

6.7.1 The language

VHDL is a mixed level design description language originally conceived as simulator input language. As such its semantics as defined in the VHDL language reference manual [VHDL 87] is defined in terms of simulation behaviour. As top-down design gains importance however, synthesizable subsets have been defined and implemented as input language for synthesis tools. VHDL supports design description in behavioural and structural domains.

VHDL provides a good case study of our specification language for the following reasons:

- VHDL is important. Almost all ECAD vendors support VHDL input to their design tools, be it high-level synthesis systems, mixed-level simulators from algorithmic down to gate-level, or even systems for hardware-software codesign [Heusinger 93].
- VHDL is syntactically complex. VHDL has a rather irregular, handcrafted syntax, tailored to the conception by designers, not by machines. This is why it is a good benchmark for our specification language.
- VHDL is semantically complex. VHDL inherits from ADA constructs of a modern, structured programming language like modularization, abstract data types, a rich type system, and polymorphism. It adds its own concepts to describe hardware specific data types, concurrency, module generation, and composition hierarchies.
- VHDL has built-in design management features. The language offers a concept of dynamic configurations, which is unusual for a programming language, but very useful during the design and test of electronic circuits on various levels of detail. The designer can first create alternative

implementations of a design entity and decide later which one to use at a certain place in her design.

In this section we will see how our specification language can be used to define those parts of VHDL that are relevant to design information management. It is important to note that we do not attempt to give an exact and complete syntax definition of VHDL. This effort would be rather pointless as complete VHDL grammars exist for parser generators like *yacc*. Our goal is rather to give a minimum specification of VHDL that is nevertheless large enough to digest a complete VHDL file and extract important structural information from it. The specification will accept a language that is larger then VHDL, that is, a file accepted by a language processor generated from our specification need not be a valid VHDL file. We assume that a proper syntactical and semantical analysis of design files will be done by a dedicated design tool and need not be the task of a framework service. This assumption radically simplifies our language processors and makes it feasible for a tool integrator to write processors for languages as complex as VHDL.

The primary design abstraction in VHDL is the design entity. An *entity declaration* defines the interface between a given design entity and the system in which it is used. An *architecture body* specifies the implementation of a design entity in terms of the relationships between its inputs and outputs. There may be multiple *architecture bodies* defined for a given *entity declaration*. The selection of a specific implementation for a design entity is handled by *configuration declarations*.

Concurrent statements are used in architecture bodies to define interconnected blocks and processes that jointly describe the overall behaviour or structure of a design. Concurrent statements come in various forms, notably *block statements* to group other concurrent statements and *process statements* which represent single independent sequential processes. Other concurrent statements exist for commonly occurring forms of processes as well as for representing structural decomposition and regular descriptions.

VHDL provides a rich type system encompassing scalar, composite, access, and file types much like ADA. *Subprograms* may be used to define algorithms. Object types provided are *constants*, *signals*, and *variables*. Objects may be associated with predefined or user-defined *attributes*. Object and subprogram definitions may be grouped into packages, realizing an abstract data type. *Entity*, *configuration* and *package declarations* as well as *architecture* and *package bodies* may be independently analysed and inserted into a *design library.*

6.7.2 Syntax specification

Our language specification for VHDL consists of two modules. The first module is concerned with the overall language structure. The granularity is coarse, leaving much detail unresolved. Nevertheless, this module describes enough of the language syntax to recognize the text regions that are to be associated with objects in design information management.

```
syntax vhdl {
  token comment -ignore     -pattern <“--” .* $>
  token ID                  -pattern <[_a-zA-Z0-9]+>
  token USE                 -pattern <“use” [^;]* “;”>
  token LIBRARY             -pattern <“library” [^;]* “;”>
  token END                 -pattern <“end” [^;]* “;”>

  rule design_file { repeat { toplevel } }
  rule toplevel { LIBRARY | USE | entity | architecture | package | config }
} // syntax vhdl
```

A design file contains at least one top-level design unit. In addition, there may be directives to include external libraries and selected objects from these libraries into the scope of the design file. Most of the detail specified in the top-level design units can safely be ignored for design information management. Their names are important, however.

```
rule entity { “entity” Name:ID <\n> guts <\n> END }
rule architecture { “architecture” Name:ID ID EntityName:ID guts END }
rule package { “package” guts END }
rule guts { list { item | ID } }
```

In the rule for entities we have included export directives (the two patterns “<\n>”) to show their use. The rule *guts* collectively describes all the internals of design units.

While we completely ignore special characters (remember that the generated scanner is instructed to read over all text for which it has no explicit pattern), certain keywords and identifiers have to be distinguished to find component declarations.

```
rule item {
   "procedure" guts END
  | "function" guts END
  | "units" guts END
  | "record" guts END
  | "block" guts END
  | "generate" guts END
  | "process" guts END
  | "case" guts END
  | "if" guts END
  | "loop" guts END
  | USE
  | component
}
rule component { "component" Name:ID guts END }
```

Although the granularity at which the top-level specification module is written is sufficient to locate design units and extract their names, it is too coarse to analyse configuration declarations. In fact, we need all the details from a configuration declaration, so the specification module for configurations is written at a very fine granularity. Configuration declarations are introduced with the keyword *configuration*, so a rule with this single keyword as right-hand-side provides a suitable anchor for invoking the config specification:

```
rule config -syntax { "configuration" }
```

Structural implementations can declare a component specification and create instances of components. A component thus declared can be thought of as a template for a design entity. The binding of an entity to this template is achieved through a configuration declaration. The declaration can also be used to specify actual generic constants for components and blocks. So the configuration declaration plays a pivotal role in organizing a design description in preparation for simulation or other processing.

The declarative part of a configuration declaration allows the configuration to use items from libraries and packages. The outermost block configuration in the configuration declaration defines the configuration for an architecture of the named entity.

```
syntax config {
  token comment -ignore     -pattern <"--" .* $>
  token IDENTIFIER          -pattern <[_a-zA-Z] [_a-zA-Z0-9]*>

  rule configuration_declaration {
    "configuration" configName:IDENTIFIER
    "of" interfaceName:IDENTIFIER "is"
        list { use_clause } block_configuration
    "end" opt { IDENTIFIER } ";"
  }
} // syntax config
```

Note that a nested specification module is completely self-contained. We have to specify a fresh set of comment conventions, tokens, keywords, and special characters for each module we define.

Within the block configuration for an architecture, the submodules of the architecture may be configured. These submodules include blocks and component instances. A block is configured with a nested block configuration. Where a sub-module is an instance of a component, a component configuration is used to bind an entity to the component instance. Note the extensive use of tags on rule items. We will see shortly how these are used in both import and export specifications.

```
rule block_configuration {
    "for" implName:IDENTIFIER
        list { use_clause }
        items:list { configuration_item }
    "end" "for" ";"
}
rule configuration_item {
    block_configuration
  | component_configuration
}
```

```
rule component_configuration {
   "for" component_specification
       binding:opt { "use" binding_indication ";" }
       block:opt { block_configuration }
   "end" "for" ";"
}
rule component_specification {
   instantiation_list ":" componentName:IDENTIFIER
}
rule instantiation_list {
   repeat -separator "," { instanceName:IDENTIFIER }
  | "others"
  | "all"
}
rule binding_indication {
   entity_aspect
       opt { "generic" "map" association_list }
       opt { "port" "map" association_list }
}
rule entity_aspect {
   "entity" "work." childInterfaceName:IDENTIFIER
   opt { "(" childImplName:IDENTIFIER ")" }
  | "configuration" "work." childConfigName:IDENTIFIER
  | "open"
}
rule use_clause { "use" ";" }
rule association_list { "(" ")" }
```

6.7.3 Import: Creating objects during parse tree traversal

In Section 6.7.2 we have given the language specification for VHDL configuration declarations. Now we want to use this specification as the basis for parsing a configuration declaration, such as the test bench configuration of Figure 14 on page 76, into the corresponding graph of objects according to the DIM conceptual schema. To achieve this, we write an extension language script that traverses a parse tree automatically created by the VHDL parser module, and creates the DIM objects via extension language commands bound to the DIM PI as soon as enough information is available to fill all attribute values of the created objects. Apart from the parse tree, no data structure is necessary.

We assume that the node identifier of the tree root has been assigned to variable *tree* by the following statement

```
set tree [[vhdl T -file dp32.vhdl] root]

$tree all -type configuration_declaration -var n1 {
        $n1 one -tag configName -type IDENTIFIER {
              set configName $text }
        $n1 one -tag interfaceName -type IDENTIFIER {
              set interfaceName $text }
        $n1 one -type block_configuration n2 {
              doBlockConfig \
                    $n2 $interfaceName $interfaceName $configName
        }
}
```

This top-level loop finds all nodes of type *configuration_declaration*, extracts the configuration name and the name of the associated interface from them, and processes the single block configuration contained in them by calling the procedure *doBlockConfig*. Configuration declarations always contain a single block configuration which configures an architecture body. The other two *one*-statements identify the individual identifiers found in a configuration declaration by checking their tags and exploiting the fact that the text of a matching node can be directly accessed through the text variable from within the action code. No node identifiers are required.

The procedure *doBlockConfig* has the following code:

Algorithm 2. doBlockConfig

in	n1	identifier of a block configuration
in	path	list of objects that represent the current nesting level
in	interfaceName	name of the interface of this block
in	configurationName	name of the outermost configuration declaration
out	configuration identifier	the identifier of the block configuration created

```
proc doBlockConfig {n1 path interfaceName configName} {
 set C [Configuration #auto -Name $configName]

 $n1 one -tag implName -type IDENTIFIER { set implName $text }

 $n1 all -type component_configuration -var n2 {
      $n2 one -tag componentName -type IDENTIFIER {
           set componentName $text }
      $n2 one -type entity_aspect -var n3 {
           $n3 one -tag childInterfaceName -type IDENTIFIER {
                set childInterfaceName $text
           }
           set childImplName “”
           catch { $n3 one -tag childImplName -type IDENTIFIER {
                     set childImplName .$text
           }}
      }
```

```
catch* { unset CC }
$n2 all -type block_configuration -var n3 {
      set nestedC [doBlockConfig
            $n3 $path.$implName
            $childInterfaceName $configName]
      if {$childImplName != ""} {
            # check whether the block configuration just created
            # is for childImpl
            $n3 one -tag implName -type IDENTIFIER {
                  set nestedImplName $text }
            if {"$childImplName" == ".$nestedImplName"} {
                  # create a component configuration
                  # that points to the block configuration
                  set CC [ComponentConfiguration #auto
                        -Name
                        "cc:$path.$implName.$componentName"
                        -Configuration $C
                        -Component
                        $interfaceName.$implName.$componentName
                        -Child $nestedC]
            }
      }
}
if [catch { set CC }] {
      # no nested block configuration has configured the childImpl
      # create a component configuration
      # that points to the childimpl directly
      set CC [ComponentConfiguration #auto
            -Name "cc:$path.$implName.$componentName"
            -Configuration $C
            -Component
            $interfaceName.$implName.$componentName
            -Child $childInterfaceName$childImplName]
}
```

*. *Catch* is a Tcl command that evaluates its parameter as usual but "catches" error conditions to avoid that the execution of the whole script is terminated.

```
        # don't search for component_configuration's
        # in nested block_configuration's
        continue
    }
    return $C
  }
```

Here, the parameters for the creation of a component configuration are simplified in that components are referenced by their path name instead of a DIM object identifier retrieved by a query operation. In brief, the purpose of procedure *doBlockConfig* is to resolve a single, possibly nested block configuration statement. It mainly consists of a big loop over all component configurations. The *continue* statement at the end of the loop prevents the traversal from implicitly diving into nested block configurations because these are to be handled explicitly. For each component configuration, a check is made whether a corresponding block configuration exists. If so, this block configuration is recursively resolved and used as child design object for the component configuration. If no such block configuration exists, the named implementation is used as child design object for the component configuration. The net result of letting this code operate on the parse tree created from a configuration in the DP32 processor is the object graph depicted in Figure 14 on page 76.

6.7.4 Export: Linking types to parse trees

To export an object graph managed by the DIM service, the type of each object to be exported has to have an *unparse* method that creates a parse tree for objects of this particular type. *Unparse* methods are implemented using the *unparse* functions generated for each parser module. Export simply involves the invocation of the unparse method on a selected DIM object and writing the resulting parse tree into a design file.

For a configuration, an enclosing VHDL *configuration_declaration* has to be created first (we assume the variable *C* to contain the DIM object identifier of the configuration to be exported):

```
[config::unparse -rule configuration_declaration
  -name { -code $C Name } -entityRef { -code [$C Interface] Name }
  -block_configuration { -node $C unparse } ] export $VHDLfile
```

Table 11 shows the unparse methods for the types *Configuration* and *Component-Configuration* in our conceptual schema. Every configuration visited is emitted as a nested block configuration in the VHDL file. Depending on whether the child design object of a component configuration is an implementation or a configuration, the respective VHDL binding indication is emitted.

DIM type	unparse method
Configuration	**return** [config::unparse **-rule** block_configuration -implName { **-code** [$this Implementation] Name } -items { **-list set** ComponentConfigurations }]

Component-Configuration	**switch** [$Child **info class**] {
	Implementation {
	return [config::unparse **-rule** component_configuration
	-component_specification { **-code**
	concat "all:" [$Component Name] }
	-binding { **-code**
	concat “use entity” [$Child Name] }]
	}
	Configuration {
	return [config::unparse **-rule** component_configuration
	-component_specification { **-code**
	concat “all:” [$Component Name] }
	-binding { **-code**
	concat “use entity”
	[[[$this Implementation] Interface] Name] }
	“(” [[$this Implementation] Name] “)”
	-block { **-node** $Child unparse }]
	}}

Table 11. *Implementations of unparse methods for configurations and component configurations. Only the alternatives for implementations and configurations are shown. The unparse method for component configurations that bind to an interface are not shown.*

6.8 Related work

In this section we will look into the work done by others related to the topics of language processing in the context of EDA frameworks. First, we review the support for design description languages in EDA frameworks together with the integration level (cf. Section 2.5) on which such a support is given. The JESSI Common Framework (JCF, [Kathöfer 92]) supports only a-priori tool encapsulation based on the copying of design files between a design data repository and the design tools. As no concern is given to the contents of design files, data integration in JCF is on carrier level. EDA Frameworks like Mentor Graphic's Falcon Framework [Mentor] provide format converters between their native design formats and standard or tool specific formats. However, there is no toolkit support for building new converters besides the standard UNIX tools.

The Nelsis CAD Framework [Dimes 93b] supports tool integration on a number of different levels, with a set of interface functions that allow to access design data at varying granularities:

granularity	interface functions
file-based	dmGetPathOfStream, dmMoveFileToStream, dmCloseStream
stream-based	dmOpenStream, (standard-i/o facilities), dmCloseStream
value-based	dmOpenStream, dmGetDesignData, dmPutDesignData, dmCloseStream

Table 12. *Integration granularities in the Nelsis CAD Framework*

While Nelsis, too, does not provide a toolkit to write language processors for arbitrary design description languages, it offers an extensible set of *format handlers* for common formats. A format handler is a library function that knows how to read and write design data in a specific format. The functions *dmGetDesignData* and *dmPutDesignData* take the format handler needed to process a particular design description language as a parameter. Format handlers provide a uniform interface to design data from a tool's point of view. Unfortunately, support to write new format handlers is restricted to scanf/printf-like functions. This is clearly insufficient for free-format languages like VHDL.

Regarding the problem of framework support for design description languages to find an answer to the question *"how to map between syntax and objects,"* some solutions are offered in the literature. Many "C++" class libraries have methods to dump an object graph to a file and to read such files back into memory. However, the format of such a dump is fixed, whereas the object-oriented database system OBST follows a more flexible approach. Its *Structurer and Flattener* (STF, [Pergande 93]) is a general tool to map between textual files and objects stored in an OBST database. STF translates a single syntax specification into a pair of programs, a structurer to read an object graph from a file, and a flattener to write an object graph to a file (the generation process is similar to the one depicted in Figure 25). The structurer contains a yacc-generated parser, the flattener is constructed from a set of recursive procedures. While the STF approach resembles ours, it can not be applied for the following reasons:

- STF assumes the OBST database system as back-end of the generated language processors. The mapping between (implicit) parse tree and an OBST object graph is compiled into the language processors.
- STF syntax specifications are restricted in that the syntax for a class must be described with a single syntax rule. While the provided EBNF constructs make up for this to a certain extent, in general it is complicated to map a given language specification to STF.
- STF generated parsers are expected to process every single detail from a file. This does not conform with our approach of only considering the bits in a design description that are important to DIM, ignoring most of the details. To provide more flexibility in choosing the parsing granularity, we provide modular syntax specifications.

Txl [Carmichael 92] and *ParsesraP* [Beaty 94] are two other systems that also provide a single specification language for both input and output. Both, however, operate as filters to translate between two different source representations. There is no documented programmatic access to the intermediate parse tree.

Graver describes the system *T-gen* to automatically generate string-to-object translators in the context of the Smalltalk programming environment [Graver 93]. T-gen supports a large class of grammars (LL(1), SLR(1), LALR(1), LR(1)). The generated translators are Smalltalk objects and either create two variants of parse trees, derivation trees or abstract syntax trees from their input. While derivation trees contain all intermediate non-terminals created during parsing, abstract syntax trees are pruned by parse-tree builder directives (PTBs) inserted in the syntax specification by the programmer. PTBs can be used to explicitly construct a parse tree node or to control the flow of information between tree nodes. For example, there is a PTB *liftRightChild* to help flattening the derivation tree. Consider a set of rules of the form*

```
ArgList    : Arg ArgList      { liftRightChild }
           | Arg              { ArgumentListNode };
Arg        : <argument>       { ArgumentNode };
```

The capitalized PTBs *ArgumentNode* and *ArgumentListNode* create tree nodes. The PTB *liftRightChild* prevents that a new tree node is created. Instead, the tree associated with the right-most symbol on the right-hand side is used as tree for the symbol on the left-hand side. The other right-hand side symbols are appended as children to this tree, thus effectively flattening the derivation tree to a list. Our system could benefit from the introduction of PTBs. However, because we skip large parts of an input file, the parse trees generated by our parsers are generally small. By means of our declarative operators *all* and *one* we create the illusion of a pruned tree that only contains relevant information for the programmer. This approach has the advantages that we can keep the syntax specification completely free of manually inserted action code. All “tree pruning” is done from extension language scripts.

Certainly, there are more sophisticated systems to process parse trees, yet all of them insist on processing files down to single lexems, as, after all, this is required for compiler construction. They also have limitations which make them unusable as general toolkits for building language processors to encapsulate design tools. *Txl* [Carmichael 92] relies on its own internal representation of parse trees and provides no documented hooks to extract information from such trees. *Puma* [Grosch 91] uses

*. We use *yacc's* notation here. Curly braces contain action code

Modula2 as implementation language and relies on the use of its associated compiler frontend. A viable alternative to our parse tree operators might have been the use of *Sorcerer* [Parr 94], but we prefer a solution using an interpreted language in which framework administrators or even designers can easily create new or modify existing mappings without the need for recompilation.