Tecton: a framework for specifying and verifying generic system components*

Deepak Kapur
The University at Albany
Albany, New York 12222

David R. Musser
Rensselaer Polytechnic Institute
Troy, New York 12180

July, 1992

Abstract

This paper presents the syntax and semantics of a small language for describing and using abstract concepts in formal software development and hardware design. The language provides *definition, abbreviation, extension*, and *lemma* constructs, which have general mathematical descriptive power, plus a computation-specific *realization* construct. The semantics, which is denotational, includes specification of the requirements ("legality conditions") that must be met when using each construct. The syntax and semantics are such that a corresponding proof theory requires only first order and inductive proof methods, rather than general higher order techniques as required in some frameworks. The language and some of the main proof issues are illustrated with an extended example of a behavioral and structural description of a carry-lookahead adder circuit, with the circuit realization given in terms of a generic parallel-prefix circuit.

*Partially supported by NSF Grant Number CCR-8906678. A preliminary version of this paper was presented at TPCD Conference 1992 (Theorem Provers in Circuit Design), University of Nijmegen, The Netherlands, June 22-24, 1992.

Contents

1 Introduction **1**

2 Concepts and concept descriptions **5**
2.1 Definitions 5
2.2 Concept instances 12
2.3 Inheritance from previous concept descriptions 14
2.4 Legality of concept instances 15
2.5 Abbreviations 16
2.6 Lemmas 17
2.7 Extensions 19
2.8 Realizations 19

3 Simple examples of concept descriptions **20**
3.1 Boolean and Natural concepts 20
3.2 Some algebraic concepts 22
3.3 Relations and ordering 24
3.4 Binary-trees 25

4 An extended example: carry-lookahead adder **26**
4.1 Basic circuit elements 26
4.2 Carry statuses 27
4.3 Prefix computation 29
4.4 Carry-lookahead adder 32

5 Concluding remarks **33**

1 Introduction

Tecton is a language and tool framework whose purpose is to foster structured development of computational systems (both hardware designs and software), using abstraction and specialization as the key structuring mechanisms. In Tecton abstraction and specialization are expressed using "concept descriptions." It is planned that concept descriptions will be supported by syntactic and semantic checking in an extension to the current Tecton Proof System [18]. In this paper we define the syntax and semantics for concept descriptions and develop a number of small examples and one extended example, a behavioral and structural description of a carry-lookahead adder circuit. A separate working paper [15] contains additional examples of concept descriptions, including an extended example of software specification (sorting algorithms based on a generic parallel-reduction operator).

The syntax and semantics presented here are based in part on a much earlier but incomplete description of a high-level specification language (with the same name and goals) [16, 17] and on other specification and programming languages with similar goals, including principally the OBJ series of languages [9] and the Larch Shared Language [6, 12]. The syntax and semantics of Tecton concept descriptions are considerably simpler than those of the OBJ languages, and in comparison with Larch the syntax is simpler and the semantics appears more suitable as a foundation for development of a proof theory, as it is given using set theoretic and algebraic notions rather than in terms of textual substitution.

One of the main goals of Tecton is to reduce the need for proof. If one proof can be done at an abstract level, and the results can be used in a variety of instances without repeating the proof, there will be a substantial savings of effort. There is something to be proved when an instance is created, namely that the name or concept replacements made in the instance imply the assumptions made in the abstract-level proof, but generally these proofs will be simpler than if abstractions were not used and all proofs were done directly about concrete cases.

In computer science probably the most common use of abstraction is the practice of abstracting away from implementation details; this is the focus of most work on data abstraction in software development and on structural abstraction in hardware design. The essential idea is to express "what" a computation is supposed to do, without getting into details of "how." But often we would also like to describe computations without specifying fully what they do; i.e., to abstract at the behavioral level also. As stated in [17],

> We need to be able to use abstraction and specialization of *concepts* for the purpose of *organization and understanding*, irrespective of the implementation question. The intention is to capture the general properties of an abstract notion without having to specify too much detail. When one wishes to describe a specialization of this abstract notion, only details particular to the specialization are described, while whatever is known about the abstract notion based on its general properties is also

carried to the specialization.

The benefits of behavioral abstraction and other kinds of abstraction, in software and hardware specification and verification are becoming more widely recognized, and languages and support tools are finding wider use. However, there are still formidable obstacles to fully realizing the potential benefits.

One obstacle may be the difficulty of understanding and working with specifications written in currently available languages, either because of the formidable mathematical prerequisites to understanding their semantics, or because the languages and supporting tools do not fully support some of the more useful kinds of constructions involving abstraction and specialization.

OBJ, for example, provides a comprehensive set of constructs for expressing abstractions and instantiating them; and it has a denotational semantics. However, there are some potential obstacles to its effective use. First, OBJ semantics is based on category theoretic notions of which, to our knowledge, there is no available intuitively appealing description. Thorough understanding of the semantics seems likely to remain inaccessible to all but a few dedicated proponents of the category theoretic approach. Second, the OBJ implementation has only a small part of the theorem proving support required to enforce the legality conditions implied by the semantics. Thus use of some constructs is informal; proof obligations corresponding to the legality conditions are not even generated by the OBJ implementation for examination by the user. Finally, the language has a large number of mechanisms that may not be easy to learn to use effectively or even correctly.

The Larch Shared Language also provides extensive support for expressing abstractions and instantiating them. Its semantics is defined in terms of textual substitutions that reduce layered specifications down to a flat set of formulas in first order logic, thereby avoiding the subtle issues of category theoretic semantics. Larch uses traditional explicit parameterization (as does OBJ), but also has a notion of substitution of names that does not require a specifier to have thought of all symbols which might usefully be treated as parameters. The Larch tool set contains a theorem prover designed specifically to handle many of the kinds of proof obligations associated with uses of the language. Again, however, there are obstacles to effective use of the language. First, in the Larch approach to semantics, understanding of the meaning of a construct depends on expansion into a flat collection of formulas in first order logic—a low level language—leaving behind the organizational structure that was supposed to be a major advantage over ordinary first order (or higher order) logic specifications. Second, having two approaches to parameterization and instantiation may be perplexing to the specification writer and confusing to the reader. Third, the language limits opportunities for abstraction and instantiation (in comparison with OBJ and with the Tecton language as described here).

The Higher Order Logic (HOL) language and theorem prover [7] supports use of abstraction in a fundamental way, since higher order functions and predicates are abstractions of concrete functions and predicates. Melham [21] illustrates how four different kinds of abstraction that are useful in specifying and verifying hardware can be expressed in higher order logic: *behavioral*, *structural*, *data* and *temporal* abstraction. However, higher order specifications can be difficult to understand (see [9] for one critique). While definitions of higher order functions can be used to set up generic specifications in HOL (see, for example, [14, Ch. 4.1]), this is a fairly low-level approach compared to the direct support found in Larch and OBJ for creating and combining behavioral abstractions.

The EHDM language and verification environment [25] provide higher order functions and predicates, but also have a parameterized module construct. EHDM's theorem prover contains decision procedures for arithmetic and other ground theories, permitting some proofs to be done with much less user interaction than in HOL. EHDM has been used in some very difficult formal verifications of fault-tolerance, e.g., [23]. It is, however, not as easy in EHDM as in Larch or OBJ to build up specifications as compositions of fundamental behavioral abstractions.

Another major obstacle to making the use of abstraction more common is the difficulty of constructing formal proofs. Abstraction does, we believe, help to reduce the overall effort required (and will more so as libraries of proved abstractions are expanded), but there will remain a substantial level of proof effort that is essential no matter how much abstraction is used. There must continue to be strong research efforts to develop theorem proving support for the above mentioned languages and others, including packaging it in a way that non-experts can use it productively. This is a goal for the Tecton Proof System [18], in which, for example, considerable attention is being devoted to development of a good human interface. [18] contains a discussion of the Tecton notion of proofs as forests of proof trees and how the system constructs and displays such proof trees in a graphical format. (See also [24] for a general discussion of improving interfaces to theorem provers.)

With regard to automation of theorem proving, one of the most successful approaches is the Boyer-Moore logic and theorem prover [2, 3]. Its well-developed heuristic approach to automation of induction proofs has achieved numerous impressive proofs, including many difficult examples of software and hardware verification. However, while many of these examples involve uses of abstraction, they are almost all instances of structural, data, or temporal abstraction. Behavioral abstraction can be done by axiomatizing uninterpreted function symbols, as has been done in HOL and EHDM, by using an AXIOM construct, but in the prover described in [2, 3] there is no specific mechanism to set up proof obligations showing the axioms are satisfied when the abstraction is later instantiated. However, a CONSTRAIN command has recently been added [1] which allows adding function symbols that are constrained by a set of axioms; the function symbols may be instantiated later, proof obligations then being generated to show that the axioms are satisfied. The Boyer-Moore logic does not allow quantifiers (universal quantification is implicit) and thus requires a witness to be supplied wherever one might ordinarily express a specification using existential quantification (or universal quantification over a limited scope).

The recently developed LUTINS logic and IMPS theorem prover [5] are oriented toward development of "little theories," supporting theory inclusions and theory interpretations, both of which aid in structuring the formalization of mathematical concepts. IMPS provides a greater degree of automation of proofs than the HOL, OBJ, Larch, or EHDM provers, though it does not have facilities comparable to the Boyer-Moore heuristics for inductive proof. Most applications of LUTINS and IMPS to date have been in the realm of algebra and analysis; it will be interesting to see their use on examples of behavioral abstractions of computations.

In this paper, we present Tecton language mechanisms that we hope will be seen as a step toward overcoming some of the obstacles to more widespread use of abstraction- and proof-based methods of system building. The main focus of the paper is on the semantics of concept descriptions and on illustrating the language with examples. We also briefly describe some of the issues of supporting the construction of proofs that are required whenever the semantics places legality conditions on concept descriptions.

Tecton concept descriptions have a denotational semantics, defined in terms of a few set theoretic and algebraic notions. A novel way of relating descriptions to sets of algebras is used, resulting in a relatively simple way of defining the semantics of combinations of concept descriptions in terms of set intersections. The syntax and semantics are such that a corresponding proof theory, though not spelled out in this paper, will require only first order and inductive proof methods, as in Larch and OBJ.

Syntactically, the language is relatively streamlined. First, there are no explicit parameters; instead instantiations are written using a replacement notation similar to that used in Larch (and to that recently advocated for object-oriented programming languages in [22]). Unlike Larch, there are some restrictions on which symbols may be substituted for, but these restrictions follow naturally from the denotational semantics. Second, there is no syntactic distinction made such as that between "objects" and "theories" in OBJ; in this respect, Tecton is closer to Larch and, we believe, to normal mathematical discourse. Third, there are fewer constructs than in either Larch or OBJ, yet some kinds of commonplace specification activity are more directly supported, such as expressing the realization (software implementation or hardware structural description) of a concept in terms of other concepts. Finally, the syntax is designed to strike a reasonable balance between conciseness and readability.

The following section presents the syntax and semantics, with a few small interspersed examples. We do not discuss proof theory except in a few places; we plan to present the proof theory in detail in a later paper. Sections 3 and 4 give more examples, including an extended example of hardware design description. The reader may find it useful to refer extensively to these examples while reading the next section.

2 Concepts and concept descriptions

In Tecton terminology, a "concept" is a set of many sorted algebras. We assume the existence of two disjoint sets of identifiers (strings), called *sorts* and *function identifiers*. An *indexed family of sets* is a family of pairs (s, S), where s is a sort and S is a set. An *indexed family of functions* is a family $\mathcal{F}$ of pairs (σ, f), where f is a function and σ is a *function description* consisting of a function identifier and an *arity*, which is a list of distinct sorts (called *domain sorts*) and another sort (called the *range sort*). Then a *(many-sorted) algebra* is

- an indexed family $\mathcal{S}$ of sets such that $\mathcal{S}$ is a mapping (i.e., if $(s, S_1) \in \mathcal{S}$ and $(s, S_2) \in \mathcal{S}$ then $S_1 = S_2$).
- an indexed family $\mathcal{F}$ of functions, such that $\mathcal{F}$ is a mapping and for each pair (σ, f) in $\mathcal{F}$, where σ has domain sorts $s_1, \ldots, s_n$ and range sort r, f is a function with domain $\mathcal{S}(s_1) \times \ldots \times \mathcal{S}(s_n)$ and range $\mathcal{S}(r)$.

Finally, we define a *concept* as a set of many-sorted algebras.

Concept descriptions may be given in Tecton by a series of statements. Statements take one of six forms:[1]

statement `::=` *definition* | *abbreviation* | *lemma*
| *extension* | *realization* | *remark*

The last kind of statement begins with `Remark:` but is otherwise free-form. Remarks are a place to make commentary on other statements. Each of the other forms of statement introduces or refers to a concept name, called the *subject* of the statement. A definition, concept abbreviation, or realization defines its subject: it associates the concept name with a meaning, which is a concept—i.e, a set of algebras. An extension or expression abbreviation adds information about new identifiers to a concept while preserving the meaning it gives to other identifiers, while lemmas assert that certain properties follow from the definition of and extensions to the concept.

2.1 Definitions

A definition is a concept description of the form:

[1]To give the context-free part of the syntax, we use BNF with the following extensions:

- clauses enclosed in { } are optional;
- *construct-list* means a list of *construct* separated by commas;
- *construct-list*`[;]` means a list of *construct* separated by semi-colons;
- *construct-list*`[ ]` means a list of *construct* separated by blanks.

BNF rules are in some cases augmented by context-sensitive syntax rules, stated in English.

definition `::=`
  `Definition:` *concept-identifier* { *refines-clause* } { `;` }
    { *uses-clause* } { `;` } { *introduces-clause* } { `;` }
    { *generates-clause* } { `;` } { *obeys-clause* } `.`

At least one of the refines, uses, introduces, generates, or obeys clauses must be present. A definition introduces a new description of a set of algebras if neither a refines clause nor a uses clause is present, or (the usual case) it derives a description from previously given descriptions.

The introduces clause introduces sorts and function descriptions.

introduces-clause `::=`
  `introduces` { *sort-or-subsort-list* } { `,` } { *function-declaration-list* }
sort-or-subsort `::=` *sort* `|` *subsort-declaration*
subsort-declaration `::=` *sort* `<` *sort*
sort `::=` *identifier*
function-declaration `::=`
  function-identifier { `(` *sort-list* `)` } `->` *sort* { `(private)` }
`|` *function-declaration-with-variables* `=` *expression* { `(private)` }
function-declaration-with-variables `::=`
  function-identifier { `(` *variable-and-sort-list* `)` }
variable-and-sort `::=` *variable-identifier* `:` *sort*
variable-identifier `::=` *identifier*

(Function-identifiers include symbols like `+`, `*`, etc. Such lexical rules, rules about operator precedence, and several context-sensitive syntax rules for expressions are not spelled out here.)

Examples:

```
Definition: Set
  uses Boolean;
  introduces domain,
     =(domain, domain) -> bool.

Definition: Binary-op
  refines Set;
  introduces +(domain, domain) -> domain.
```

Any function identifier declared with two arguments can be used in either prefix or infix notation; e.g., `+(x, y)` or `x + y`.

A function declaration includes a *function description*, as defined earlier. It may also include a *private* qualification, which means that the scope of the declaration is the current statement only. For other function declarations, and all sorts and subsorts, the scope is the current statement plus subsequent statements which contain an instance of the subject of the current statement.[2]

In a subsort declaration s_1 < s_2, the set indexed by s_1 is declared to be a subset of the set indexed by s_2.[3]

The meaning associated with a definition is a concept, a set of algebras. Actually, we associate two sets of algebras with a definition, called the *syntactic concept* and the *semantic concept* (when we use just the word "concept" we mean the semantic concept). If A is a concept name we write syn A for the syntactic concept of A and sem A for the semantic concept of A. We always have sem $A \subseteq$ syn A.

For the moment, we only consider definitions that have no refines or uses clause; these are called *simple concept definitions*.

The syntactic concept denoted by a simple concept definition D is defined as follows: an algebra with indexed sets $\mathcal{S}$ and indexed functions $\mathcal{F}$ belongs to the syntactic concept if and only if the sorts introduced in D belong to dom $\mathcal{S}$ (the domain of $\mathcal{S}$), the subsort declarations are satisfied (i.e., $\mathcal{S}(s_1) \subseteq \mathcal{S}(s_2)$ when s_1 < s_2), and the function descriptions introduced in D belong to dom $\mathcal{F}$.

Note: There may be other sorts in dom $\mathcal{S}$ besides those introduced in D, and, similarly there may be other function descriptions in dom $\mathcal{F}$ besides those introduced in D. (Either or both of dom $\mathcal{S}$ and dom $\mathcal{F}$ could in fact be infinite). This is a significant difference from the usual approach in which each algebra associated with a syntactic description (called a signature) has exactly the number of sets and functions named in the signature.

If A is the concept identifier of a simple concept definition with no obeys or generates clause, then sem A = syn A. An obeys clause or generates clause requires the sets and functions of a algebra to have certain properties. We consider obeys clauses first.

obeys-clause `::=` `obeys` *quantifiers* *property-list*
quantifiers `::=` `(` `for` { `some` } *variable-declaration-list* `[;]` `)`
variable-declaration `::=` *variable-identifier-list* `:` *sort*
property `::=` *expression*
expression `::=`
    variable-identifier { `:` *sort* }
  `|` *function-identifier* { `(` *expression-list* `)` }

[2]Such an instance, as will be seen in Section 2.2, may specify a renaming of the function and/or its sorts; it is the renamed description which is available in that statement.

[3]One use of subsorts is to allow specification of functions as total functions on the subsort which otherwise would be partial functions on the larger sort. (See [11], which also discusses other uses.)

| *expression* *function-identifier* *expression*
| *quantifiers* *expression*
| `if` *expression* `then` *expression* `else` *expression*
| *expression* `where` *variable-identifier* `=` *expression*
| `(` *expression* `)`

Example:

```
Definition: Associative
  introduces domain;
     +(domain, domain) -> domain;
  obeys (for x, y, z: domain)
    x + (y + z) = (x + y) + z.
```

In the quantifiers of an obeys clause, each sort must be one that is introduced in the current statement or in one of the concepts instantiated in the statement. No variable may appear more than once in the quantifiers. In $v_1, \ldots, v_n : s$, each variable v_i is said to be declared to be of sort s.

In each expression in the property list, each variable must be declared in the quantifiers, except for variables in `where` clauses; and each function identifier must be declared in the statement or in one of the concepts instantiated in the statement (these are said to be the *available* declarations). A function identifier may appear more than once in the statement and instantiated concepts, but not with identical arities. If it does appear with different arities, it is said to be *overloaded*. Using these declarations, the expression must be *sort-correct*, as defined below. If the expression is sort-correct using more than one set of available declarations, the expression, and the statement containing it, are considered illegal.

Let e be an expression in which no undeclared variables and no undeclared function identifiers appear. The sort of e is defined inductively as follows:

1. If e is a variable identifier, the sort of e is the declared sort of the variable.

2. If e is $v : s$, s must be a sort declared as a subsort of s', the sort of v. The sort of e is `bool`.

3. If e is f, where f is a function identifier, let $f \rightarrow r_1, \ldots, f \rightarrow r_k$ be the function descriptions declared for f. Then in subsequent sort-checking each of the sorts $r_1, \ldots, r_k$ is a candidate for the sort of e.

4. If e is $f(a_1, \ldots, a_n)$ or $a_1 f a_2$ (here $n = 2$), then for each n-argument function description $f(d_1, \ldots, d_n) \rightarrow r$ declared for f, the sort of a_i must be d_i or a subsort of d_i, for $i = 1, \ldots, n$, and the sort r is a candidate for the sort of e.

5. If e is `(for` *variable-declaration-list*`)` e' or `(for some` *variable-declaration-list*`)` e', the sort of e' must be `bool`, which is also the sort of e.

6. If e is `if` e_1 `then` e_2 `else` e_3, the sort of e_1 must be `bool` and e_2 and e_3 must have the same sort, which is also the sort of e.

7. If e is e_1 `where` $x = e_2$, then occurrences of x in e_1 are considered to have the sort of e_2, (regardless of whether x is declared or undeclared), and the sort of e is that of e_1.

8. If e is `(` e' `)`, the sort of e is the sort of e'.

If all of the requirements on subexpressions stated in the above rules are satisfied for a unique set of function descriptions, e is said to be sort-correct.

In the following example from Section 3.2, there are two available function descriptions for 1, namely $1 \rightarrow$ domain and $1 \rightarrow$ naturals, but in `x ^ 0 = 1`, only the former leads to sort-correctness for the whole expression.

Example:

```
Definition: Exponentiation
   uses Natural, Monoid [with * as +, 1 as 0];
   introduces ^(domain , naturals) -> domain;
   obeys (for x: domain; n: naturals)
     x ^ 0 = 1,
     x ^ (n + 1) = (x ^ n) * x.
```

Roughly speaking, the semantics of a concept description D with subject C is that an algebra $(\mathcal{S}, \mathcal{F})$ belongs to the concept $\mathrm{sem}\, C$ if and only if belongs to $\mathrm{syn}\, C$ and under the the sort and function description mappings $\mathcal{S}$ and $\mathcal{F}$, each property p in the obeys clause of D is a true statement. To make this precise, we must describe exactly how truth values are assigned to properties.

Let D be a concept description with subject C and let $A = (\mathcal{S}, \mathcal{F})$ be an algebra in $\mathrm{syn}\, C$. Define an $\mathcal{S}$ *valuation* ν to be a mapping of declared variables to values, in which the value $\nu(v)$ is from the set $\mathcal{S}(s)$ when the declared sort of v is s.

We assume `true` and `false` are declared as nullary function identifiers with range sort `bool`, which is assumed to be always assigned by $\mathcal{S}$ to a set containing exactly two distinct values, **true** and **false**. We also assume that for every sort s, the $=$ function identifier is declared as $= (s, s) \rightarrow \text{bool}$ and $\mathcal{F}(= (s, s) \rightarrow \text{bool})$ is equality in the set $\mathcal{S}(s)$.

For any expression e containing no undeclared variables or function identifiers, we define inductively the value that $A^{\nu} = (\mathcal{S}, \mathcal{F}, \nu)$ assigns to e:

1. If e is a variable identifier v, $A^\nu(e) = \nu(v)$.

2. If e is $v : s$, where s is a subsort of the declared sort of v, then $A^\nu(e) =$ **true** if $\nu(v) \in \mathcal{S}(s)$; **false**, otherwise.

3. If e is f, where f is any nullary function identifier other than `true` or `false`, let σ be the function description declared[4] for f. Then $A^\nu(e)$ is the nullary function (a constant) $\mathcal{F}(\sigma)$.

4. If e is $f(a_1, \ldots, a_n)$ or $a_1 f a_2$, let σ be the function description declared for f. Then $A^\nu(e) = \mathcal{F}(\sigma)(v_1, \ldots, v_n)$ where $v_i = A^\nu(a_i), i = 1, \ldots, n$.

5. If e is `(for` *variable-declaration-list*`)` e', then $A^\nu(e) =$ **true** if $A^{\nu'}(e') =$ **true** for every ν' with $\nu'(y) = \nu(y)$ for every variable y not in *variable-declaration-list*; **false**, otherwise.

6. If e is `(for some` *variable-declaration-list*`)` e', then $A^\nu(e) =$ **true** if $A^{\nu'}(e') =$ **true** for some ν' with $\nu'(y) = \nu(y)$ for every variable y not in *variable-declaration-list*; **false**, otherwise.

7. If e is `if` e_1 `then` e_2 `else` e_3, then $A^\nu(e) = A^\nu(e_2)$ if $A^\nu(e_1) =$ **true**, and $A^\nu(e) = A^\nu(e_3)$ otherwise.

8. If e is e_1 `where` $x = e_2$, then $A^\nu(e) = A^{\nu'}(e_1)$, where $\nu'(x) = A^\nu(e_2)$, and $\nu'(y) = \nu(y)$ for every variable y other than x.

9. If e is `(` e' `)`, then $A^\nu(e) = A^\nu(e')$.

Note that `true, false, bool, :` and `=` have a fixed interpretation, unlike other identifiers,

The quantifiers clause that appears at the beginning of the obeys clause is treated as though it were included in every property in the obeys clause.

For every subsort declaration, the obeys clause must include a statement of a property characterizing membership in a subsort, which can be done using `:` operator.

Example:

```
Definition: Natural
  introduces naturals,
     nonzero < naturals,
     0 -> naturals,
     succ(naturals) -> nonzero,
     ...
```

[4] In case of overloading, the function description used is the one selected by the sort-correctness algorithm.

```
obeys (for n: naturals)
  n: nonzero = not(n = 0),
  ...
```

We are now prepared to give the precise semantics of a simple concept description with subject C. An algebra A belongs to sem C if and only if:

1. it belongs to syn C;
2. for each property p in the obeys clause, $A^{\nu}(p) = \mathbf{true}$, for an arbitrary[5] $\mathcal{S}$ valuation ν; and
3. the `generates` clause, if any, is satisfied as defined next.

Roughly speaking, a `generates` clause

generates-clause `::=`
  `generates` *sort* { `freely` } `using` *function-descriptor-list*
function-descriptor `::=`
  function-identifier { { `(` *sort-list* `)` } `->` *sort* }

restricts algebras in the concept to those in which all elements of the set corresponding to the sort named in the clause are finitely generated by the functions named in the clause. This property serves as a basis for developing induction schemata for proving properties by induction for all values associated with the named sort.

Example:

```
Definition: Natural
  introduces naturals,
  ...
  generates naturals freely using 0, succ;
  ...
```

The complete definition is given in Section 3.1. From the `generates` clause we obtain the usual simple induction schema for natural numbers,

$$\frac{P(0) \qquad (\text{for } n : \text{naturals})\ P(n) \text{ implies } P(\text{succ}(n))}{(\text{for } n : \text{naturals})\ P(n)}$$

To state precisely the meaning of the generates clause, let $\psi = \{s_1, \ldots, s_k\}$ be all of the declared subsorts of s, including s itself. In

[5] This valuation will not actually be used since there are no free variables allowed—every variable must be declared in the quantifiers.

`generates` s `using` $\sigma_1, \ldots, \sigma_n$

the function descriptors must be from among the available function declarations and must have range sort in ψ. For an algebra $(\mathcal{S}, \mathcal{F})$ to belong to the concept it must have $\mathcal{S}(s) = U$ where every value in U is denotable by a sort-correct expression in which

- every function identifier with range sort in ψ refers to a member of $\{\sigma_1, \ldots, \sigma_n\}$,
- there is no variable whose sort is in ψ.

If the word `freely` is included in the clause, there is an additional requirement that any two such expressions must denote distinct values of U if they cannot be made identical by replacement of subexpressions whose sort is not in ψ by variables. This results in additional properties: For distinct $\sigma_i = f_i(d_{j1}, \ldots, d_{jn}) \to r_i$ and $\sigma_j = f_j(e_{j1}, \ldots, e_{jm}) \to r_j$,

$$f_i(x_1, \ldots, x_n) \neq f_j(y_1, \ldots, y_m)$$

and for every $\sigma_i = f_i(d_{j1}, \ldots, d_{jn}) \to r_i$,

$$f_i(x_1, \ldots, x_n) = f_i(y_1, \ldots, y_n) \text{ implies } x_1 = y_1 \text{ and } x_2 = y_2 \text{ and } \ldots \text{ and } x_n = y_n.$$

Thus for the above natural numbers example, we obtain the properties

$$\mathrm{succ}(n) \neq 0$$

and

$$\mathrm{succ}(n) = \mathrm{succ}(m) \text{ implies } n = m.$$

Without the word `freely` in the `generates` clause the concept would include not only the natural number algebra but also the the residue class rings Z_m, for every $m > 0$.

2.2 Concept instances

A *concept instance* is a concept name optionally followed by a clause specifying replacements:

concept-instance `::=` *concept-name* { `[ with` *replacement-list* `]` }
concept-name `::=` *concept-identifier* { `by` *concept-identifier* }

The concept name part of a concept instance is called the base of the concept instance. The form A `by` B is introduced in a realization statement, as discussed in Section 2.8.

There are three kinds of replacements allowed:

replacement `::=` *sort* `as` *sort*
`|` *function-descriptor* `as` *function-descriptor*
`|` *concept-name* `as` *concept-name*

A function-descriptor is usually just a function-identifier, but the longer form can also be used, and must be used when it is necessary to distinguish among different instances of an overloaded function-identifier.

Examples:

```
Binary-op [with * as +]

Binary-op [with * as +(domain, domain) -> domain]

Indexed-circuit [with Boolean as State, true as high, false as low].
```

The semantics of a concept instance is defined in terms of relabellings of algebras. Let c be a string or sequence of strings and $a_1, \ldots, a_n, b_1, \ldots, b_n$ be strings. Then by $c[a_1 \leftrightarrow b_1, \ldots, a_n \leftrightarrow b_n]$, called a *relabelling* of c, we denote the result of simultaneous exchange of a_1 for b_1, …, a_n for b_n in c. Note that if only one of a_i or b_i occurs in c (the usual case) the exchange is just a replacement of the one that does occur with the one that doesn't.

If $A = (\mathcal{S}, \mathcal{F})$ is an algebra, let $A[a_1 \leftrightarrow b_1, \ldots, a_n \leftrightarrow b_n]$ denote the algebra $(\mathcal{S}', \mathcal{F}')$, in which the simultaneous exchanges are applied to the domains of $\mathcal{S}$ and $\mathcal{F}$:

$$\mathcal{S}' = \{(s[a_1 \leftrightarrow b_1, \ldots, a_n \leftrightarrow b_n], S) : (s, S) \in \mathcal{S}\}$$

and

$$\mathcal{F}' = \{(\sigma[a_1 \leftrightarrow b_1, \ldots, a_n \leftrightarrow b_n], f) : (\sigma, f) \in \mathcal{F}\}.$$

Let B be the base of a concept instance C, and let x_1 `as` y_1, …, x_n `as` y_n be all of the sort and function descriptor replacements in C, where those function descriptors that are just identifiers are expanded to include the arity of the (assumed unique) function description with that identifier. Then define the concept B^* by

$$A[x_1 \leftrightarrow y_1, \ldots, x_n \leftrightarrow y_n] \in B^* \quad \text{if and only if} \quad A \in \mathrm{sem}\, B.$$

Let the concept name replacements in C be B_1 `as` A_1, …, B_n`as` A_n. For each i, let B_i^* be the concept

$$A[x_1 \leftrightarrow y_1, \ldots, x_n \leftrightarrow y_n] \in B_i^* \quad \text{if and only if} \quad A \in \mathrm{sem}\, B_i$$

Then

$$\mathrm{sem}\, C = B^* \cap B_1^* \cap \ldots \cap B_n^*.$$

(If there are no concept identifier replacements in C, then $\mathrm{sem}\, C = B^*$).

In each kind of replacement x `as` y, a declaration of at least one of x or y must be available. If x and y are function descriptors and declarations of both are available, they must have the same arity. In the case of concept name replacements there are further requirements, as discussed in a Section 2.4.

2.3 Inheritance from previous concept descriptions

The refines and uses clauses of a concept definition allow reuse of previously described concepts.

refines-clause `::= refines` *superior-list*
superior `::=` *concept-instance*
uses-clause `::= uses` *part-list*
part `::=` *concept-instance*

Examples:

```
Definition: Identity
  refines Binary-op;
  introduces 0 -> domain;
  obeys (for x: domain) x + 0 = x.

Definition: Monoid
  refines Associative, Identity.

Definition: Abelian-monoid
  refines Monoid, Commutative.

Definition: Left-distributive
  refines Binary-op, Binary-op [with * as +];
  obeys (for x, y, z: domain)
     x * (y + z) = x * y + x * z.
```

To give the semantics of a definition D of concept C with refines or uses clauses, let $C_1, \ldots, C_n$ be all of the parts and superiors that appear in D. An algebra is in $\mathrm{syn}\, C$ if and only if it is in $\mathrm{syn}\, C_1 \cap \ldots \cap \mathrm{syn}\, C_n$, and the sorts introduced in D belong to $\mathrm{dom}\, \mathcal{S}$ and the function descriptions introduced in D belong to $\mathrm{dom}\, \mathcal{F}$. A algebra X is in $\mathrm{sem}\, C$ if and only if $X \in \mathrm{syn}\, C \cap \mathrm{sem}\, C_1 \cap \ldots \cap \mathrm{sem}\, C_n$ and each property in the obeys list of D is a true statement about the sets and functions of X.

There is a semantic requirement on uses clauses, which is part of the distinction between uses and refines clauses. Intuitively, when B has A as a part, we want to require that the meaning of the symbols of A be unchanged by the description of B. Formally, we can state the requirement as follows. If A is a concept identifier and X is an algebra in $\mathrm{syn}\, A$, define *the restriction of* X *to* A, written $X|_A$, to be the subalgebra of X that has exactly the sets and functions required by A. Then when B has A as a part, we require

$$\{Y|_A : Y \in \mathrm{sem}\, B\} = \{X|_A : X \in \mathrm{sem}\, A\}$$

Example: In

```
Definition: Set
    uses Boolean;
    introduces domain,
        =(domain, domain) -> bool.
```

the requirement is that

$$\{Y|_{\text{Boolean}} : Y \in \text{sem Set}\} = \{X|_{\text{Boolean}} : X \in \text{sem Boolean}\}$$

where the set on the left is the concept formed by restricting each algebra in sem Set to just those indexed sets and functions required by Boolean, and the set on the right is the concept formed by restricting each algebra in sem Boolean to just those indexed sets and functions required by Boolean. Equality between these two concepts holds trivially in this example because there is no obeys clause in the description of Set.

There is no such requirement on concept instances in the refines clause; in fact, when we include a concept instance A in the refines clause of B we usually intend to modify the semantics associated with A, thereby eliminating some algebras from the semantic concept of B.

A proof theory for concept definitions with refines and uses clauses can be based on unioning the theory directly stated in the definition (the properties in the obeys clause and an induction principle corresponding to the generates clause) with the theories associated with each part or superior. We do not however go into the details in this paper.

2.4 Legality of concept instances

Another difference between the uses and refines clauses has to do with legal concept instances. In a concept instance

B `[with ...` C `as` A `...]`

where A, B, and C are concept-names, it is required that

$$\text{sem } B \subseteq \text{sem } A$$

and, if

$$\{Y|_A : Y \in \text{sem } B\} = \{X|_A : X \in \text{sem } A\}$$

then

$$\text{sem } C \subseteq \text{sem } A$$

else

$$\text{sem } C = \text{sem } A$$

A sufficient condition for the first of these conditions to hold is that A be a part or superior in the concept description named by B. As discussed in the previous section, if A is a part then the equality between restrictions will hold and it is only required that $\text{sem}\, C \subseteq \text{sem}\, A$; but if it is a superior, then we require equality between the two concepts.

In effect, the superiors and parts mentioned in the base concept description are formal parameters and can be replaced using other concept names as actual parameters. The concept named by an actual parameter must be contained in the concept of the corresponding formal parameter that is a part, or must be the same concept as that denoted by the formal parameter if the latter is a superior.

Example: In

```
Indexed-circuit [with Boolean as State, true as high, false as low]
```

it suffices for the concept `State` to appear as the base of a part in the concept description of `Indexed-circuit` and the semantic concept of `Boolean` to be contained in the semantic concept of `State`. (See Section 4 for the full context of this example.)

2.5 Abbreviations

Abbreviations are concept descriptions which allow either introduction of a new concept name for a concept instance or modification of an existing concept to include one or more abbreviations of expressions by new functions.

abbreviation `::=` `Abbreviation:` *concept-name* `is` *concept-instance* `.`
`|` `Abbreviation:` *concept-name* `has` *expression-abbreviation-list* `.`
expression-abbreviation `::=`
    function-declaration-with-variables `=` *expression*

The form

`Abbreviation:` C `is` A

is called a concept abbreviation and is just a way of naming a concept instance A with a concept name C. We define $\text{syn}\, C = \text{syn}\, A$ and $\text{sem}\, C = \text{sem}\, A$.

Examples:

```
Abbreviation: Semigroup is Associative.

Abbreviation: Range is Set [with codomain as domain].
```

```
Abbreviation: All
   is Generalized-reduction [with bool as codomain, p as f, and as +].

Abbreviation: Exists
   is Generalized-reduction [with bool as codomain, p as f, or as +].
```

The second form introduces one or more expression abbreviations.

Examples:

```
Abbreviation: Group has -(x: domain, y:domain) = x + -(y).

Abbreviation: Partial-order
   has <(x: domain, y: domain) = (x <= y) and not(x = y).

Abbreviation: Array
   has swap(a: arrays, i: naturals, j: naturals)
         = assign(assign(a, j, access(a, i)), i, access(a, j)).
```

As with lemmas and extensions (defined below), the subject of this second form of abbreviation must be a previously defined concept name, and for each expression-abbreviation the function description must not have previously been available in the definition. In all subsequent uses of the concept name, the functions defined in the abbreviation are treated as though they had been given in the original definition. The meanings associated with all other function descriptions are not changed.

2.6 Lemmas

A lemma is a statement with one of three forms

lemma `::=`
   `Lemma:` *concept-name* { *generates-clause* } { ; }
      { *obeys-clause* } .
 `|` `Lemma:` *concept-instance* `=` *concept-instance* .
 `|` `Lemma:` *concept-instance* `is` *concept-instance* .

The first form of a lemma is used to state properties of a previously defined concept that are logical consequences of the properties stated in the definition of or extensions to that concept. At least one of the generates or obeys clauses must be present.

Examples:

```
Lemma: Group obeys (for x: domain) 0 + x = x.

Lemma: Group obeys (for x: domain) -(x) + x = 0.

Lemma: Sequence generates sequences freely using null, |-.
```

The same syntactic requirements are placed on the quantifiers and properties as in a definition. Each property in the property list of a lemma must be a true statement about the named sets and functions of every algebra in the concept, and the `generates` clause, if present, must be satisfied.

There is, obviously, a proof obligation associated with a lemma, to show that the stated properties follow from the definition of and extensions to the concept.

The subject of a lemma is the concept name it begins with. A lemma does not change the concept associated with its subject. (However, a Tecton implementation should provide queries about concept names which would list all lemmas whose subject was a given concept name.)

The second and third forms of a lemma state a relation between concept instances. The second form

`Lemma:` A `=` B.

asserts that A and B are the same concept; that is, sem A = sem B. The third form

`Lemma:` A `is` B

asserts that sem $A \subseteq$ sem B. In either of these two forms, the subject is the base of the first of the two concept instances.

Examples:

```
Lemma: Integer is Total-order [with integers as domain].

Lemma: Second-order-sequence
   is Reduction [with Sequence-monoid as Monoid].

Lemma: Natural is Monoid [with naturals as domain].

Lemma: Natural is Monoid [with naturals as domain, * as +, 1 as 0].

Lemma: Natural [with domain as naturals, + as *, 0 as 1] is Monoid.
```

2.7 Extensions

An extension is a concept description of the form

extension `::=`
    `Extension` *concept-name* { *uses-clause* } { ; }
      { *introduces-clause* } { ; } { *obeys-clause* } .

At least one of three clauses must be present. An extension is used to introduce a new sorts and function identifiers and state properties about them. As in an expression abbreviation, an extension associates the new information with a previously defined concept name. In subsequent references to the concept name the sorts and functions introduced in the extension are available, as though they had actually been introduced in the statement defining the name.

Example:

```
Extension: Sequence
  introduces
    reverse(sequences) -> sequences;
  obeys (for s: sequences; x: elements)
    reverse(null) = null,
    reverse(x -| s) = reverse(s) |- x.
```

A semantic requirement on an extension is that the meaning of any identifiers already available in the concept is not changed (i.e., it is a *conservative extension*). As with the requirement on parts in a definition, this requirement can be formally stated as follows: letting A be the subject of the extension, $C_1 = \text{sem}\, A$ before the extension, and $C_2 = \text{sem}\, A$ after the extension, then

$$\{Y|_A : Y \in C_2\} = \{X|_A : X \in C_1\}$$

must hold (where the restrictions refer to the identifiers available before the extension).

2.8 Realizations

Realizations are concept descriptions of the form:

realization `::=`
  `Realization:` *concept-identifier* `by` *concept-identifier*
      { *refines-clause* } { ; } { *uses-clause* } { ; }
      { *introduces-clause* } { ; } { *obeys-clause* } .

In

`Realization:` A `by` B `...`

the concept-identifiers A and B must already have been the subject of previous statements. The concept name A `by` B is the subject of this concept description. The concept C associated with A `by` B is the same as given by a definition with the same clauses, except that all sorts and function declarations available in A are also available (but the semantics of A is not inherited), and B is treated as a part (as though added to the uses clause).

Example:

```
Realization: Stack by Sequence
     introduces
         rep(sequences) -> stacks (private),
     obeys (for s, u: sequences; t: nonempty-sequences; e: elements)
         empty = rep(null),
         push(e, rep(s)) = rep(s |- x),
         top(rep(t)) = last(t),
         pop(rep(t)) = rep(butlast(t)),
         size(rep(s)) = length(s),
         (rep(s) = rep(u)) = (s = u).
```

There is a substantial semantic requirement associated with a realization: the concept relationship that could be stated by a lemma of the form

A `by` B `is` A.

The corresponding proof obligation is to prove this lemma, which involves proof of each of the properties in the theory of A from the new information in the realization, including the properties of equality for the `=` function. For assistance with this proof obligation, one may obtain from the induction principle for A a corresponding induction principle that applies to terms built with a representation function such as `rep` in the above example, as discussed for example in [13, Section 4].

3 Simple examples of concept descriptions

3.1 Boolean and Natural concepts

```
Definition: Boolean
  introduces bool,
```

```
      true -> bool,
      false -> bool;
  generates bool freely using true, false.
```

Remark: the above concept is actually predefined by the semantics.

```
Extension: Boolean
  introduces
      not(bool) -> bool,
      and(bool, bool) -> bool,
      or(bool, bool) -> bool,
      xor(bool, bool) -> bool,
      implies(bool, bool) -> bool;
  obeys (for x, y: bool)
      not(true) = false,
      not(false) = true,
      (true and x) = x,
      (false and x) = false,
      (x or y) = not(not(x) and not(y)),
      (x xor y) = not(x = y),
      (x implies y) = (not(x) or y).

Definition: Set
    uses Boolean;
    introduces domain.

Definition: Natural
    refines Set [with naturals as domain];
    introduces
       0 -> naturals,
       1 -> naturals,
       succ(naturals) -> naturals,
       +(naturals, naturals) -> naturals,
       -(naturals, naturals) -> naturals,
       *(naturals, naturals) -> naturals,
       <=(naturals, naturals) -> bool;
    generates naturals freely using 0, succ;
    obeys (for n, m: naturals)
       1 = succ(0),
```

```
      (0 = 0) = true,
      (succ(n) = 0) = false,
      (0 = succ(n)) = false,
      (succ(m) = succ(n)) = (m = n),
      n + 0 = n,
      n + succ(m) = succ(n + m),
      n - 0 = n,
      0 - n = 0,
      succ(n) - succ(m) = n - m,
      i * 0 = 0,
      i * succ(j) = i * j + i,
      0 <= n,
      not(succ(n) <= 0),
      succ(m) <= succ(n)) = (m <= n).
```

3.2 Some algebraic concepts

```
Definition: Binary-op
  refines Set;
  introduces +(domain, domain) -> domain.

Definition: Associative
  refines Binary-op;
  obeys (for x, y, z: domain) x + (y + z) = (x + y) + z.

Abbreviation: Semigroup
  is Associative.

Definition: Commutative
  refines Binary-op;
  obeys (for x, y: domain) x + y = y + x.

Definition: Left-identity
  refines Binary-op;
  introduces 0 -> domain;
  obeys (for x: domain) 0 + x = x.

Definition: Right-identity
  refines Binary-op;
```

```
  introduces 0 -> domain;
  obeys (for x: domain) x + 0 = x.

Definition: Identity
  refines Left-identity, Right-identity.

Definition: Monoid
  refines Semigroup, Identity.

Definition: Abelian-monoid
  refines Monoid, Commutative.

Definition: Left-inverses
  refines Left-identity;
  introduces -(domain) -> domain;
  obeys (for x: domain) -(x) + x = 0.

Definition: Right-inverses
  refines Right-identity;
  introduces -(domain) -> domain;
  obeys (for x: domain) x + -(x) = 0.

Definition: Inverses
  refines Left-inverses, Right-inverses.

Definition: Group
  refines Monoid, Inverses.

Definition: Abelian-group
  refines Group, Commutative.

Definition: Exponentiation
   uses Natural, Monoid [with * as +, 1 as 0];
   introduces ^(domain , naturals) -> domain;
   obeys (for x: domain; n: naturals)
     x ^ 0 = 1,
     x ^ (n + 1) = (x ^ n) * x.

Abbreviation: Natural-mult-monoid
```

```
   is Natural [with naturals as domain, * as +, 1 as 0].

Lemma: Natural-mult-monoid is Monoid.

Remark: The following extends Natural to have ^ as exponentiation.

Extension: Natural
   uses Exponentiation [with Natural-mult-monoid as Monoid].
```

3.3 Relations and ordering

```
Definition: Unary-relation
  refines Set;
  introduces p(domain) -> bool.

Definition: Binary-relation
  refines Set;
  introduces r(domain, domain) -> bool.

Definition: Reflexive
  refines Binary-relation;
  obeys (for x: domain) x r x.

Definition: Anti-reflexive
  refines Binary-relation;
  obeys (for x: domain) not(x r x).

Definition: Symmetric
  refines Binary-relation;
  obeys (for x, y: domain) x r y implies y r x.

Definition: Anti-symmetric
  refines Binary-relation;
  obeys (for x, y: domain) (x r y and y r x) implies x = y.

Definition: Transitive
  refines Binary-relation;
  obeys (for x, y, z: domain) (x r y and y r z) implies x r z.
```

```
Definition: Equivalence-relation
  refines Reflexive [with = as r],
          Symmetric [with = as r],
          Transitive [with = as r].
```

3.4 Binary-trees

```
Abbreviation: Index-set
   is Set [with indices as domain].

Definition: Binary-tree-index
  refines Index-set;
  uses Natural;
  introduces
    nonroot < indices,
    root -> indices,
    left(indices) -> indices,
    right(indices) -> indices,
    parent(nonroot) -> indices,
    depth(indices) -> naturals;
  generates indices freely using root, left, right;
  obeys (for i: indices)
    i: nonroot = not(i = root),
    depth(root) = 0,
    depth(left(i)) = depth(i) + 1,
    depth(right(i)) = depth(i) + 1,
    parent(left(i)) = i,
    parent(right(i)) = i.

Extension: Binary-tree-index
  introduces level-order-index(indices) -> naturals,
  obeys (for i: indices)
    level-order-index(root) = 1
    level-order-index(left(i)) = 2 * level-order-index(i),
    level-order-index(right(i)) = 2 * level-order-index(i) + 1.
```

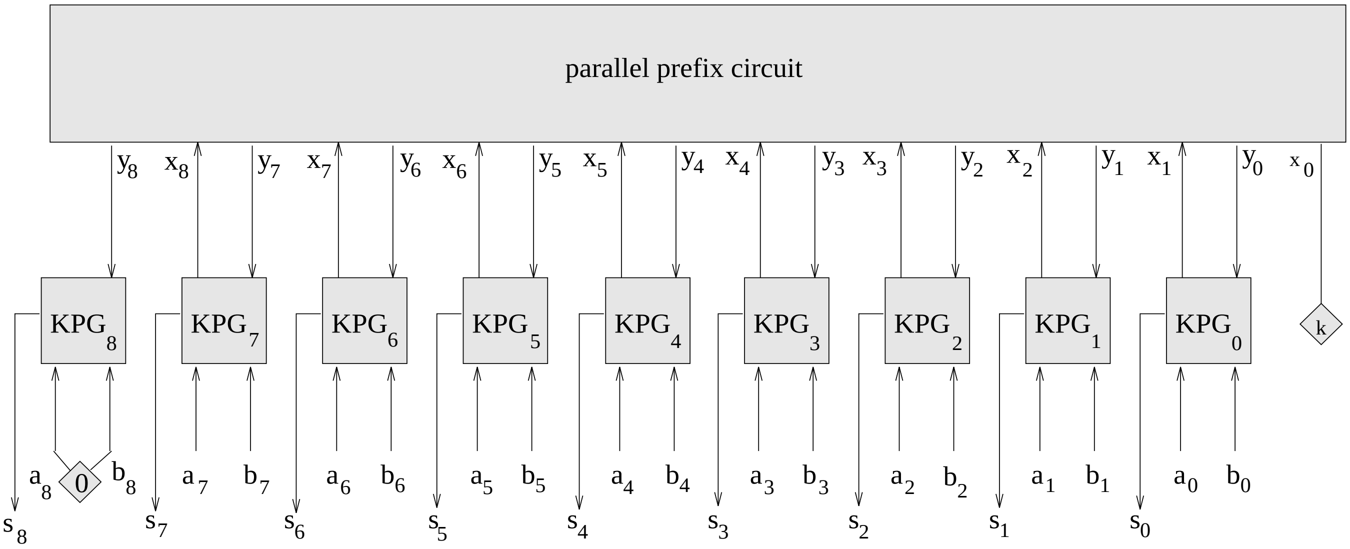

Figure 1: A carry-lookahead adder circuit (based on Figure 29.9-10 in [4])

4 An extended example: carry-lookahead adder

This section contains an attempt to formalize the behavioral specification of a generic parallel-prefix circuit and its use in realizing a carry-lookahead adder circuit. The terminology and organization used in the example closely follows the discussion in [4, Ch. 29]. Figures 1, 2, and 3 are based on figures in [4]. Some of the concepts used in the example are defined in the previous section.

4.1 Basic circuit elements

```
Definition: State
   refines Set [with states as domain]
   introduces
      low -> states,
      high -> states;
   obeys
      not(low = high).

Remark: In State we deliberately refrain from requiring that it
generates states using low, high, as we want to allow for additional
states (as in Kpg-state in the next section).
```

```
Abbreviation: Boolean
   has parity(x: bool, y: bool, z: bool)
         = (x xor y xor z),
       majority(x: bool, y: bool, z: bool)
         = (x and y) or (y and z) or (x and z).

Abbreviation: Time is Natural [with time as naturals, tick as succ].

Definition: Indexed-circuit
   uses Natural, Time, Index-set, State;
   introduces
      in(naturals, indices, time) -> states,
      out(naturals, indices, time) -> states;
   obeys (for n: naturals; i: indices)
      out(n, i, 0) = low.

Remark: Indexed-circuit is a concept of a circuit to be replicated
some number of times with each copy indexed by a value from indices.
Each copy has potentially infinitely many input and output lines,
indexed by natural numbers, but we only ever use a finite number.

Abbreviation: Linearly-indexed-circuit
   is Indexed-circuit [with Natural as Index-set, naturals as indices].

Abbreviation: Binary-tree-indexed-circuit
   is Indexed-circuit [with Binary-tree-index as Index-set].
```

4.2 Carry statuses

```
Remark: In the following, k, p, and g stand for "kill," "propagate,"
and "generate," respectively.

Definition: Kpg-state
  refines State [with kpg-states as states, k as low, g as high],
          Identity [with p as 0, kpg-states as domain];
  introduces g -> kpg-states;
  generates kpg-states freely using k, p, g;
  obeys  k + k = k,  k + g = g,  g + k = k,  g + g = g.
```

```
Lemma: Kpg-state [with domain as kpg-states] is Associative.

Definition: Carry-status
   uses Boolean, Kpg-state;
   introduces carry-status(bool, bool) -> kpg-states;
   obeys
     carry-status(false, false) = k,
     carry-status(false, true ) = p,
     carry-status(true,  false) = p,
     carry-status(true,  true ) = g.

Definition: Indexed-mixed-circuit
   refines
     Indexed-circuit [with Boolean as State, true as high, false as low];
   uses State;
   introduces
     state-in(naturals, indices, time) -> states;
     state-out(naturals, indices, time) -> states.

Abbreviation: Linearly-indexed-mixed-circuit
   is Indexed-mixed-circuit [with Natural as Index-set, naturals as domain].

Definition: Linearly-indexed-KPG-box
   refines Linearly-indexed-mixed-circuit
       [with Kpg-state as State, k as low, g as high];
   uses Carry-status;
   introduces -- to allow use of the notation in Figure 1
     a(n: naturals, t: time) = in(1, n, t),
     b(n: naturals, t: time) = in(2, n, t),
     s(n: naturals, t: time) = out(1, n, t),
     x(n: naturals, t: time) = state-out(1, n, t),
     y(n: naturals, t: time) = state-in(1, n, t);
   obeys (for n: naturals; t: time)
     (for some delay: time)
       x(n + 1, t + delay) = carry-status(a(n, t), b(n, t)),
     (for some delay: time)
       s(n, t + delay) = parity(a(n, t), b(n, t), carry-in)
          where carry-in = (y(n, t) = g).
```

4.3 Prefix computation

```
Definition: Prefix-computation
  refines Associative;
  uses Natural;
  introduces
     prefix(naturals, naturals) -> domain;
  obeys (for i, j, k: naturals)
     0 <= i and i <= j and j + 1 <= k implies
       prefix(i, k) = prefix(i, j) + prefix(j + 1, k).
```

```
Remark: The + function identifier between prefix expressions in
the last line above is from the Associative concept.
```

```
Definition: Limit
  uses Natural;
  introduces limit -> naturals.
```

```
Definition: Prefix-circuit
   refines Linearly-indexed-circuit [with Associative as State];
   uses Prefix-computation, Limit [with length as limit];
   introduces
     x(n: naturals, t: time) = in(1, n, t),
     y(n: naturals, t: time) = out(1, n, t);
   obeys (for n: naturals; t: time)
     n < length implies x(n, t) = prefix(n, n),
     n < length implies
       (for some delay: time)  y(n, t + delay) = prefix(0, n).
```

```
Remark: The following realizes a generic parallel prefix circuit.
First we define the internal computation of each node of a generic
parallel prefix circuit, which involves two uses of the generic
binary operation, here called +. (See Figure 2.)
```

Next, we specify the overall binary tree structure of the parallel prefix circuit (see Figure 3).

```
Definition: Binary-tree-indexed-double-node
   refines Binary-tree-indexed-circuit
      [with Binary-op as State, states as domain];
```

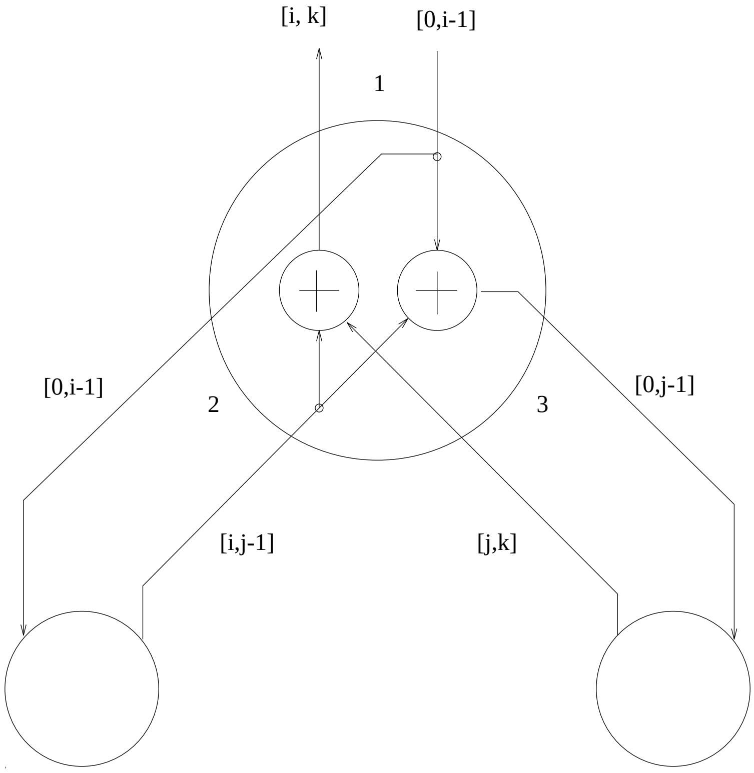

Figure 2: Organization of a each node of a parallel prefix circuit (based on Figure 29.8 in [4])

```
   obeys (for i: indices; t: time)
     (for some delay: time)
        out(1, i, t + delay) = in(2, i, t) + in(3, i, t) and
        out(2, i, t) = in(1, i, t) and
        out(3, i, t + delay) = in(1, i, t) + in(2, i, t).

Abbreviation: Binary-tree-indexed-associative
   is Binary-tree-indexed-double-node [with Associative as Binary-Op];

Realization: Prefix-circuit by Binary-tree-indexed-associative
   uses Prefix-computation, Limit;
   obeys (for i: indices; t: time)
     -- tree connections (see Figures 2 and 3)
     depth(i) < limit implies
```

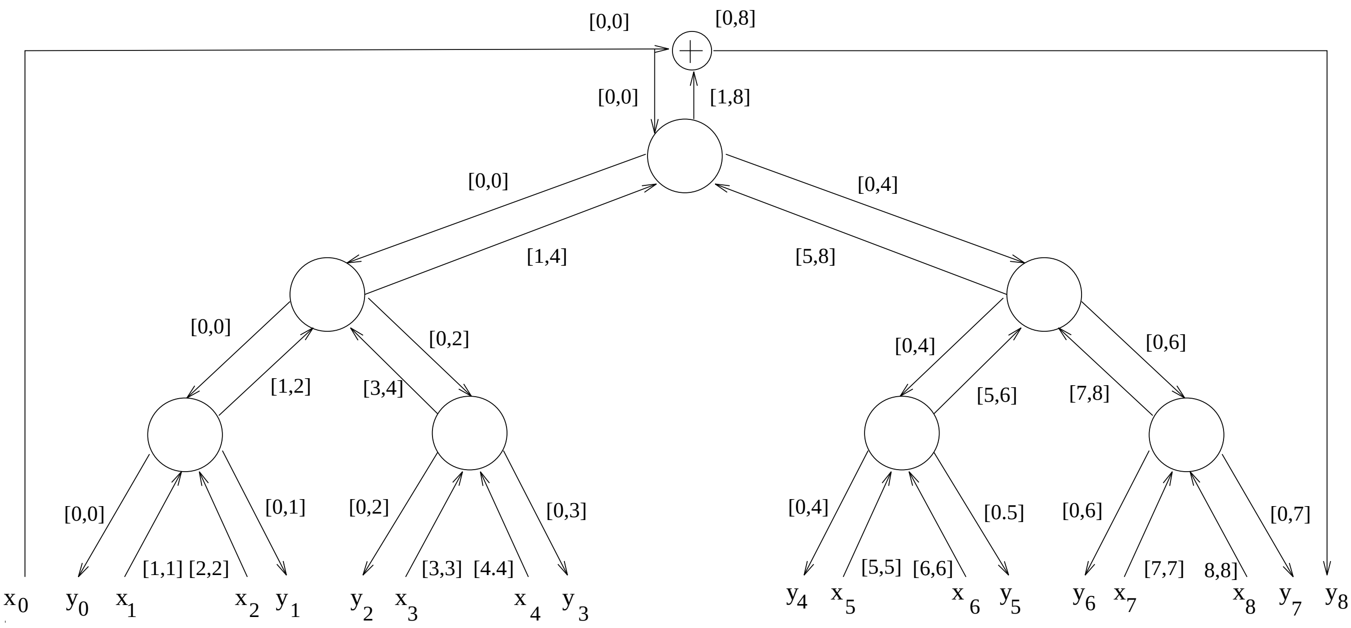

Figure 3: A parallel prefix circuit for $n = 8$ (based on Figure 29.9(a) in [4])

```
      in(2, i, t) = out(1, left(i), t) and
      out(2, i, t) = in(1, left(i), t) and
      in(3, i, t) = out(1, right(i), t) and
      out(3, i, t) = in(1, right(i), t),
   -- connections to linearly indexed inputs and outputs
   depth(i) = limit and length = 2 ^ (limit + 1)
      implies in(2, i, t)  = x(n + 1, t)
          and out(2, i, t) = y(n, t)
          and in(3, i, t)  = x(n + 2, t)
          and out(3, i, t) = y(n + 1, t)
        where n = 2 * level-order-index(i) - length,
   -- other boundary connections
   in(1, root, t) = x(0, t),
   (for some delay: time)
      y(length, t + delay) = x(0, t) + out(1, root, t).

Abbreviation: Parallel-prefix-circuit
   is Prefix-circuit by Binary-tree-indexed-associative.
```

4.4 Carry-lookahead adder

```
Definition: Bitlevel-addition
   uses Natural, Limit [with length as limit];
   introduces
       a(naturals) -> bool,
       b(naturals) -> bool,
       c(naturals) -> bool,
       s(naturals) -> bool;
   obeys (for n: naturals)
       n <= length implies
            c(0) = false
        and c(n + 1) = majority(a(n), b(n), c(n))
        and s(n) = parity(a(n), b(n), c(n)).

Abbreviation: Linearly-indexed-boolean-circuit
   is Linearly-indexed-circuit
       [with Boolean as State, false as low, true as high];

Definition: Adder
   refines Linearly-indexed-boolean-circuit;
   uses Bitlevel-addition;
   introduces
     a(n: naturals, t: time) = in(1, n, t),
     b(n: naturals, t: time) = in(2, n, t),
     s(n: naturals, t: time) = out(1, n, t),
   obeys (for some hold, delay: time)
     (for n: naturals; t: time)
        n < length and
           ((for u: time) t <= u and u < t + hold implies
                 a(n, u) = a(n)
             and b(n, u) = b(n))
        implies
           s(n, t + delay) = s(n).

Abbreviation: Kpg-parallel-prefix-circuit
   is Parallel-prefix-circuit
       [with Kpg-state as Associative, kpg-in as in, kpg-out as out];
```

```
Remark: The following interconnects the kpg boxes with the kpg parallel
prefix circuit. (See Figure 1.)

Realization: Adder by Kpg-parallel-prefix-circuit
   uses Linearly-indexed-kpg-box [with state-out as in, state-in as out];
   obeys (for n: naturals, t: time)
     x(0, t) = k,
     a(length, t) = false,
     b(length, t) = false.

Abbreviation: Carry-lookahead-adder
   is Adder by Kpg-parallel-prefix-circuit.
```

5 Concluding remarks

We have presented the syntax and semantics of Tecton concept descriptions and illustrated their use in development of a generic parallel prefix circuit, in terms of which a carry-lookahead adder is realized. We conclude with brief observations about the main points of some of the proofs that must be done to complete this example, with emphasis on how a theorem prover can take advantage of the organizational structure of the concept descriptions.

The lemma given at the end of Section 3.2,

```
Lemma: Natural-mult-monoid is Monoid.
```

which justifies the extension of Natural to have an exponentiation operator, can be proved by induction in the Tecton Proof System [18] (or in RRL [20], on which Tecton's proof capabilities are based) using the cover set method or test set method. In such proofs much of the work is done as applications of equations as rewrite rules. Since these rules are kept in a rule-base, a kind of (dynamically updated) data base of rewrite rules, and invoked by a matching algorithm, it is unnecessary to have them as explicit assumptions in the formulas to be proved. This is a well-known advantage of rewrite-rules, but the advantage can be enhanced by compartmentalization of the rule base according to concepts and restricting the matching algorithm to use only the concepts that are referenced (in this case, Natural and Monoid directly, and Boolean indirectly).

The realization in Section 4.3,

```
Prefix-circuit by Binary-tree-indexed-associative
```

gives rise to proof obligations for showing that each of the properties of Prefix-circuit follow from those of Binary-tree-indexed-associative and Prefix-computation, along with the connection properties stated directly in the realization. These properties involve quantifiers and cannot be handled purely by rewrite rules. Straightforward generation of the proof obligations as first order logic formulas would result in large and unwieldy formulas since there would large numbers of assumptions. As noted above for the case of rewrite-rules, we would like to have instead a data base of formulas whose organization is based on the concept organization built up with the definitions, abbreviations, etc. The proof obligations for the above realization would then be set up with just the properties of Prefix-circuit as goals, and the prover would access the formula base as necessary to obtain properties from Binary-tree-indexed-associative, Prefix-computation, and their ancestors. In contrast to the case with rewrite rules, there is no simple algorithm such as matching to guide the selection of formulas, but we note that the search space is much more limited than if one has to deal with a large unstructured global base of formulas. In some cases, rewrite rule matching *can* be used, using polynomial representations and refutational proof (methods which are available in RRL [19]).

Similarly, the realization `Adder by Kpg-parallel-prefix-circuit` gives rise to proof obligations in which the goals are the properties of Adder and the formula base consists of the properties from Kpg-parallel-prefix-circuit and Linearly-indexed-kpg-box [with Kpg-state as Associative, kpg-in as in, kpg-out as out]. One important issue is whether the data base entries for a concept instance such as this could be set up as minor variants on the entries for the base concept, Linearly-indexed-kpg-box, rather than as independent properties. We are currently exploring this and other issues in making effective use, within proofs, of the organizational strengths of Tecton concept descriptions.

References

[1] R. S. Boyer, D. M. Goldschlag, M. Kaufman, and J S. Moore. "Functional Instantiation in First-Order Logic," in *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, edited by Vladimir Lifschitz, Academic Press, 1991.

[2] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[3] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1991.

[5] W. M. Farmer, J. D. Guttman, F. J. Thayer, "Little Theories," in D. Kapur, ed., *Proc. of 11th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence 607*, June 1992, 567-581.

[6] S. J. Garland, J. V. Guttag, and J. J. Horning, "Debugging Larch Shared Language Specifications," *IEEE Trans. Software Engineering*, Vol 16, No. 9, September 1990, 1044-1057.

[7] M. Gordon, "A proof generating system for Higher-Order Logic," in: G. Birtwistle and P. Subrahmanyam, eds., *VLSI Specification, Verification and Synthesis*, Kluwer Academic Publishers, Boston, 1988, pp. 73-128.

[8] J. A. Goguen and R. Burstall, "Putting theories together to make specifications," in Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045-1058.

[9] J. A. Goguen, T. Winker, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud, "Introducing OBJ," to appear in J.A. Goguen, D. Coleman, and R. Gallimore (editors). *Applications of Algebraic Specification using OBJ*, Cambridge University Press, 1992.

[10] J. A. Goguen "OBJ as a theorem prover, with application to hardware verification." In P. Subrahmanyam and Graham Birtwistle, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pp. 218-267, Springer-Verlag, 1989.

[11] J. A. Goguen and José Meseguer. *Order sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations.* Technical Report SRI-CSL-89-10, SRI International, Computer Science Lab, July 1989.

[12] J. V. Guttag and J. J. Horning, "Report on the Larch Shared Language," *Sci. Comput. Program.*, vol. 6, no. 2, pp. 103-134, Mar. 1986.

[13] J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract Data Types and Software Validation," *Comm. ACM*, 21, December 1978, 1049-1064.

[14] J. Joyce. *Multi-Level Verification of Microprocessor-Based Systems.* PhD Thesis, University of Cambridge, December 1989.

[15] D. Kapur and D. R. Musser, *Examples of Tecton concept descriptions*, working paper, May, 1992.

[16] D. Kapur, D. R. Musser, and Alexander A. Stepanov, "Operators and algebraic structures," *Proc. of Conference on Functional Programming Languages and Computer Architecture*, Portsmouth, New Hampshire, October 1981.

[17] D. Kapur, D. R. Musser, and A. A. Stepanov, "Tecton: a language for manipulating generic objects," *Proc. of Program Specification Workshop, University of Aarhus, Denmark*, August 1981, *Lecture Notes in Computer Science*, Springer-Verlag, Vol. 134, 1982.

[18] D. Kapur, D. R. Musser, and X. Nie, "An overview of the Tecton Proof System," *Proc. of a Workshop on Formal Methods in Databases and Software Engineering*, Concordia University, Montreal, May 15-16, 1992.

[19] D. Kapur and P. Narendran, "An equational approach to theorem proving in first-order predicate calculus," *Proc. of 8th IJCAI*, Los Angeles, Calif, 1985.

[20] D. Kapur and H. Zhang, "RRL: Rewrite Rule Laboratory," *Proc. of Ninth International Conference on Automated Deduction (CADE-9)*, Argonne, IL, May 1988.

[21] T. F. Melham, "Abstraction mechanisms for hardware verification," in: G. Birtwistle and P. Subrahmanyam, eds., *VLSI Specification, Verification and Synthesis*, Kluwer Academic Publishers, Boston, 1988, pp. 267-291.

[22] J. Palsberg and M. Schwartzbach, "Type substitution for object-oriented programming," *OOPSLA ECOOP '90 Proceedings*, Ottawa, Canada, October, 1990, 151-168.

[23] J. Rushby and F. W. von Henke. Formal verification of the interactive convergence clock synchronization algorithm using EHDM. Technical Report SRI-CSL-89-3, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1989.

[24] L. Théry, Y. Bertot, and G. Kahn, "Real theorem provers deserve real user-interfaces," University of Cambridge report, May, 1992.

[25] F. W. von Henke, J. S. Crowe, R. Lee, J. M. Rushby, and R. A. Whitehurst, "EHDM Verification Environment: An Overview," in *11th National Computer Security Conference*, Baltimore, Maryland, 1988.