5-2002

Comparison of Inheritance Evaluation Algorithms for Express Edition 3.

Judy Dawn Greer
East Tennessee State University

Comparison of Inheritance Evaluation Algorithms for Express Edition 3

A thesis
presented to
the faculty of the Department of Computer and Information Sciences
East Tennessee State University

In partial fulfillment
of the requirements for the degree
Masters of Science in Computer Science

---

by
Judy Dawn Greer
May 2002

Keywords: EXPRESS, Inheritance Hierarchy, Algorithms

ABSTRACT

Comparison of Inheritance Evaluation Algorithms for EXPRESS Edition 3

by

Judy Dawn Greer

Information exchanged between computer applications is difficult, thus the need for data exchange standards. The ISO STEP project defines data exchange standards using the EXPRESS language, which supports inheritance. Currently there are two algorithms used to evaluate an inheritance hierarchy: the Test and Generate algorithms. In this thesis, enhancements are made to both algorithms to support the Total Over Constraint, which is proposed for the third edition of EXPRESS. A formal algorithm is derived for the Test algorithm. The two enhanced algorithms are compared and shown to be result equivalent. However, it is shown that the Test algorithm is the more efficient of the two.

ACKNOWLEDGEMENTS

I would like to thank my Thesis committee, Dr. Terry Countermine, Dr. James Pleasant, and, especially, Dr. Donald Sanderson. Dr. Sanderson has been extremely helpful and fun to work with. I have learned a tremendous amount of information about EXPRESS from Dr. Sanderson. Thanks so much.

I would also like to thank my family and friends for their support and help, especially my father, Andy, my mom, Georgia, and my brother, Jeff. Without their love and support, I would not have been able to complete this degree. I love all of you very much.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Exchanging information between different types of computer applications is a significant problem faced today by all users. Different organizations, each using its own applications, often find it difficult to share information electronically, and, as software has become more complex, the data to be exchanged has also become more complex. In response to this problem, data exchange standards have been developed. These are specifications of the information to be exchanged, how it is to be interpreted, and how it is organized. One approach taken is (STEP) [6] the Standard for the Exchange of Product Data an ISO data exchange standard, which defines the EXPRESS Modeling Language [6].

EXPRESS is an information requirements specification language that is both easily interpretable and complete enough to model complex data problems. One of the key features of EXPRESS is inheritance among data types, which is used to simplify complex problems. This work looks at the existing algorithms used to validate EXPRESS inheritance hierarchies, and also proposes updates to these algorithms to accommodate proposed changes to the EXPRESS language.

To implement the EXPRESS Language, the original authors developed an algorithm to generate all possible legal combinations of data types based on the constraints in the inheritance hierarchy called the "Generate Algorithm." Knowing all of the legal generated instances of an inheritance hierarchy and its constraints is a necessity for automatic ERROR checking of EXPRESS schemas. Shortly thereafter Gunter Staub, Frank Schonefeld, and Markus Maier [7] developed another algorithm that tests if a given data type is allowable in a specific inheritance hierarchy instance. The "Test Algorithm" allows the developer to test one instance for membership in a set instead of creating the entire set and then searching the results. There was never any work done to show the equivalence of these two algorithms. The proof of equivalence of enhanced versions of these algorithms will be one result of this thesis

Since EXPRESS was first standardized in 1993, users of the language have been requesting extensions to the language. The group that originally developed EXPRESS made some minor changes to the language (EXPRESS-2) [4] and have recently proposed more drastic changes and extensions to the EXPRESS Modeling Language to create another new edition of EXPRESS, which is referred to as EXPRESS-3 [4] in this paper. The new edition of EXPRESS is fully described in the document EXPRESS Language Reference Manual ISO 10303 WG11-N105 [4]. The changes include several enhancements to the inheritance hierarchy. The Generate Algorithm has been extended rather inefficiently to support these changes, while the Test Algorithm has not been extended. Enhanced versions of both of these algorithms will be presented, analyzed for their strengths and weaknesses, proved to be correct, and shown to be equivalent.

Chapter 2 gives background information on data modeling and information models and presents the definitions of information models that are used in this work. Chapter 3 is an overview of the concepts of STEP EXPRESS Edition 1 Modeling Language. This forms the basis for the presentation of the inheritance algorithms and leads into Chapter 4, which focuses on the changes that were made to EXPRESS-3, the major change being addition of the Total Over constraint to the inheritance hierarchy. Chapter 5 describes Algorithms and ways to analyze them. This chapter explains the different approaches that will be used to analyze the two inheritance algorithms in this work. At this point the background information has been presented, and the rest of the work is devoted to a detailed analysis of the existing algorithms, their extensions to handle the Total Over Constraint, and a proof of their equivalence. Specifically, Chapter 6 covers the Generate Algorithm and its extension to support the Total Over constraint. It also presents multiple examples of applications of this algorithm. Chapter 7 covers the same topics for the Test Algorithm. In addition a formal specification of the algorithm is given. Chapter 8 examines the relative strengths and weaknesses of the Generate and Test Algorithms, comparing them over multiple factors. Chapter 9 presents the testing done on the extensions to the Generate and Test Algorithms to show they support the Total Over constraints correctly. It also proves that the Test and Generate Algorithms are equivalent. The final chapter will give a summary of the work done and suggest directions for new research.

CHAPTER 2

DATA MODELING

Understanding the EXPRESS Modeling Language requires knowledge of data modeling in general. Data is considered to be isolated facts and real world objects, which can be anything from a company to a rubber ball, while information is considered to be processed data based on implicit or explicit interpretation rules. Schenck [5] defines information to be the knowledge of ideas, facts, and/or processes that are placed in context. The need to specify information for exchange has brought about the development of information models, which are formal descriptions of types of ideas, facts, and processes. Information models are read and processed by computers and, as such, must be complete, precise, and unambiguous. They contain both the data that represents the information and the rules that are used to interpret the information. Interpretation rules allow developers to decide on a common way to view the information. The reliable exchange of data requires that developers use the same rules. An example of an information model could be the process of maintaining a library.

There are several ways to organize information: ordering, categorizing, and grouping [5]. Ordering is putting the information in a sequence (example: chapters in a book or names in a telephone book). Categorizing is the process of putting the information into subsets or grouping items into categories or classes [5]. In categorizing, the terms generalization and specialization are used to describe the categorization of information. When giving just general facts about information, it is a generalized type; when more details are given about information, it is considered a specialized type. An example of Categories is that the person category is a more general classification of male and female categories, while the male category is a specialization of the person category. Grouping is the partitioning of items into sets based on a specific set of properties. An example of grouping is dividing computers by their manufacturers. Thus, a computer would be grouped by whether the computer is a Macintosh, Dell, or IBM.

Information can be represented in two basic ways, lexically (or text based) and graphically (or icon based). A graphical representation uses symbols or icons to show information. It is easier to read and more of a model can be viewed on a single page. Graphical models, however, often do not show all of the constraints that can be specified in a lexical representation for a given model. Thus, graphical representations of a model may not be considered complete. Two popular graphical representations of information are Entity-Relationship (ER) [5] and Shlaer-Mellor [5], while SQL [5] is a typical lexical representation used for information models.

Peter Chen introduced the Entity-Relationship (ER) [5] graphical representation in 1976. The ER representation includes the entity, attribute, and relationship constructs, each denoted by a different symbol. The ER format maps real world objects as entities, the specific characteristics and details of an object as attributes attached to the entity or relationship construct, and the interactions between objects as relationships. Graphically entities are represented as a rectangle attributes as an oval that is attached to an entity or relationship, and a relationship as a diamond with connecting ends at two or more entities. Figure 2.1 is an example of an ER diagram.

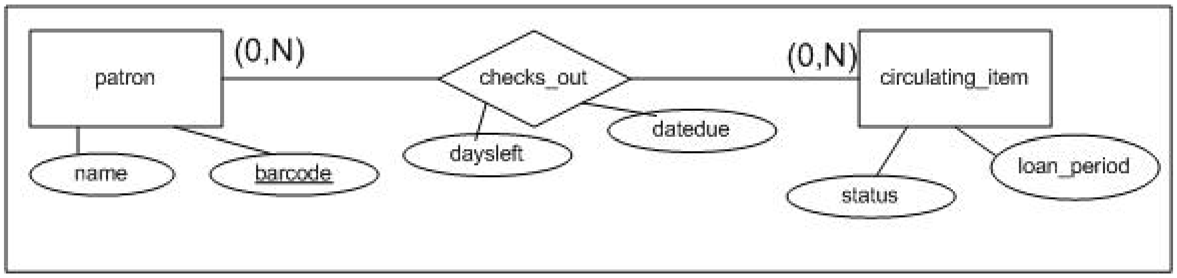


Figure 2-1 – ER Diagram

The *patron* Entity has two Attributes, which are *name* and *barcode*. The key of the *patron* Entity is the *barcode* Attribute; this is denoted by underling the attribute name. The *circulating_item* Entity has *status* and *loan_period* Attributes. The Entities participate in a Relationship called *checks_out*, which has two Attributes, *daysleft* and *datedue*. The *patron* Entity participates in the *checks_out* relationship zero or many times. The cardinality of the *circulating_item* Entity with respect to the *checks_out* Relationship depends on what information is being tracked. If the number of loans over time for a certain circulating item were tracked, then the cardinality would be zero or many times. This means that the model would be tracking how many times the circulating item was checked out for a certain amount of time. However, if the circulating item was just being tracked on whether it was loaned at the current time or not, then the cardinality would be zero or one. This means that one person could only check out the circulating item at a time.

Another example of graphical representation is the Shlaer-Mellor representation developed by S. Shlaer and S. J. Mellor [5]. The Shlaer-Mellor representation consists of objects, attributes, and relationships. Real world objects are represented as objects, which are similar to ER's entities. As with the ER format, attributes describe an object and relationships are the interactions between objects. Graphically, objects are shown as a rectangle, attributes are listed inside the object symbol, and relationships are shown as a connecting arrow between objects. Figure 2-2 is an example of the Shlaer-Mellor representation.

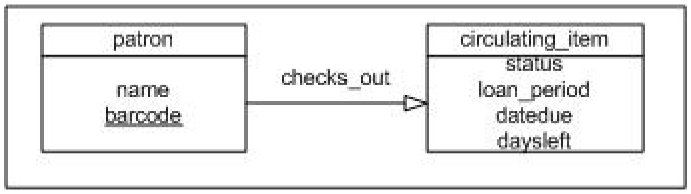


Figure 2-2 – Shlaer – Mellor Diagram

In a lexical notation, words and mathematical symbols are used to denote the different constructs that represent the information being modeled. The lexical notation has rules about the syntax and structure of the statements that are usually more detailed than those found in a graphical notation. Using lexical notation, developers are able to define both models and algorithms that can check the validity of that model. Lexical models can also be easily checked by parsers to make sure they conform to the rules of the specific lexical notation used. Being able to have a computer program read and compile a model is useful in the development of systems. However, Lexical notation is not always easy to read and understand by humans.

An example of lexical representation of information is the Structured Query Language (SQL) [5]. SQL is an ANSI and ISO standard that defines a set of statements used to define and manipulate data models. The information being modeled is stored in data structures called "tables." The table's columns are specific attributes of the object and one column called a key is used to identify one object data. Duplicating keys in other tables, which are known as foreign keys, represents relationships between objects. Figure 2-3 shows the example using SQL statements.

```
CREATE TABLE patron
        (name           CHAR(20) NOT NULL,
        barcode         INTEGER NOT NULL,
        PRIMARY KEY (barcode) )

CREATE TABLE circulating_item
        (status                 CHAR(10) NOT NULL,
        loan_period             INTEGER NOT NULL,
        checked_out_by          INTEGER NOT NULL,
        datedue                 DATE NOT NULL,
        daysleft                INTEGER NOT NULL,
        FOREIGN KEY (checked_out_by) REFERENCES patron)
```

Figure 2-3 – SQL Statements

In this example, two tables are being created; one named *patron* and the other named *circulating_item*. The *patron* table has two fields, *name* and *barcode*, while the *circulating_*item table has five fields, *status*, *loan_period*, *checked_out_by*, *datedue*, and *daysleft*. Also, the example shows the relationship between the two tables by labeling which fields are primary keys and foreign keys. The *barcode* field in the *patron* table is a unique key and is referenced by the *checked_out_by* field in the *circulating_item* table. This allows the system to track which patron has checked out a circulating item.

This chapter introduced the concept of data or information modeling. Data is described as being a real world object. Expanding on the characteristics, how the objects relate to other objects, and what rules the object follows in the situation being modeled is a key element in designing a model using EXPRESS. If the model consists of different objects or variations of the objects, then organizing the information is done by determining if it can be ordered, categorized, or grouped. Representing information can be done by text or graphics. There are already many representations available for developers to use. Entity Relationship (ER) and Shlaer-Mellor are graphical representations of information models. A common lexical representation is Standard Query Language (SQL). The EXPRESS Modeling Language has both the graphical (EXPRESS-G) and the lexical (EXPRESS) representations. The next chapter will discuss the basic concepts of the EXPRESS Modeling Language that will be used in the algorithms that are analyzed later.

CHAPTER 3

EXPRESS MODELING LANGUAGE

The EXPRESS Modeling Language (ISO 10303) [6] is an information requirement specification language. The language contains structured statements that specify what type of information is to be stored in the data model, such as numbers or strings, and the interactions between objects.

EXPRESS is used to create information models for manufacturing applications. These models are often very large and cumbersome to evaluate or review manually. One of the important features of EXPRESS is that it is a computer interpretable representation of an information model allowing developers to use programs to validate their models. Another important feature is EXPRESS's ability to build new information models from components of existing models. As models become large and complex, developers are able to subdivide models to simplify them and make the components reusable. This helps developers avoid having to "re-invent the wheel" and keeps the results in simpler, more readable and more maintainable models. Appendix B contains an information model that describes items in a library. This model is not complete but demonstrates the features of EXPRESS that are relevant to this project. The rest of this chapter discusses these constructs in detail.

One of the most important constructs in EXPRESS is the Schema; "Schemas incorporate all the other constructs and define a collection of objects that are related by meaning or purpose" [5]. A schema contains entities, constants, functions, procedures, rules, and type declarations. It often requires several schemas to model a real world situation. Schemas may include other schemas or specific parts of other schemas. An example of a Schema definition is

```
SCHEMA Library;
        (* Entity, Function, and Rules definitions *)
END_SCHEMA;
Example 3-1
```

The next important construct in EXPRESS is the Entity. An Entity describes a real world or conceptual object by listing its properties. These properties are characteristics of the objects and represent values that describe it, constraints on these values, and behavior of an object. An Entity can also be used as the type of a property in another Entity. An example Entity:

```
ENTITY patron;
        (* Characteristic definitions *)
END ENTITY;
Example 3-2
```

The characteristics of entities are described by using a construct called the Attribute. Attributes are physical properties of entities and have values that are within a specified domain defined by the type of the Attribute. Attributes can be Explicit, Derived, or an Inverse. Explicit Attributes have values that are static and independent. The optional keyword indicates that null values can be assigned to an Attribute. The following example shows the attributes *street, city, state,* and *zip* as explicit Attributes of the Entity *address*.

```
ENTITY address;
        street  : STRING;
        city    : STRING;
        state   : STRING;
        zip     : INTEGER;
END_ENTITY;
Example 3-3
```

The Entity *address* is a type declaration. A type declaration is information about what kinds of information is going to be stored in the Entity. The actual values assigned to the Attributes are not necessary for a type declaration. When the Attributes of an Entity have specific values filled in, the information is considered to be an instance. The number of instances can vary depending on the number of values that can be assigned to the attributes. For example, the following values are instances of the *address* Entity.

101 Any Lane	111 Cowboy Rd
San Jose	Dallas
CA	TX
55555	88888

A Derived Attribute holds information that is calculated from other Attribute values or on an expression. The following is an example of Derived Attribute declarations:

```
ENTITY circulating_item
        status          : STRING;
        loan_period     : INTEGER;
        checked_outby   : patron;
DERIVE
        datedue         : date := current_date() + loan_period;
        daysleft        : INTEGER := datedue-current.date();
END_ENTITY;
Example 3-4
```

The Derived Attribute *datedue* is calculated from the current date supplied by the Function *current_date()* and the *loan_period* attribute. Similarly, the *daysleft* Attribute is calculated by subtracting the *current date* from the *datedue*.

A Relationship, which can be implied or explicit, is an association between two constructs in a model. The Relationship idea does not have a construct in the EXPRESS Modeling Language. It is shown by an Attribute construct in one Entity that matches to an Inverse Attribute in another Entity. This is shown in Example 3-5 in the *circulating_item* Entity. The Attribute *checked_outby* has the *patron* Entity as its type. The *patron* Entity has an Inverse Attribute called *checks_out*, which points to the *circulating_item* Entity by the Attribute *checked_outby*. This links the two Entities to each other thus creating the Relationship between them. The *patron* and *circulating_item* Entities are related when a Patron checks out an item.

```
ENTITY circulating_item
        status          : STRING;
        loan_period     : INTEGER;
        checked_outby  : patron;
DERIVE
        datedue         : date := current_date() + loan_period;
        daysleft        : INTEGER := datedue-current.date();
END_ENTITY;

ENTITY patron;
        name            : STRING;
        mail_address: address;
        card_expired : date;
        barcode         : INTEGER;
UNIQUE
        patron_single : barcode;
INVERSE
        checks_out : BAG [1:?] OF UNIQUE circulating_item FOR checked_outby;
END_ENTITY;
Example 3-5
```

The number of times an Entity participates in a Relationship with another Entity is called cardinality. There are three cardinality ratios: One to One, One to Many, Many to Many. One to One (1..1) is where every instance of one entity requires one instance of the second Entity, and vice versa. When One to One is specified in a lexical notation, both the regular and the Inverse Attribute are of a single Entity. Modifying the Library Schema slightly by changing the barcode Attribute to an Entity, the One to One relationship can be shown by requiring that only one item have a single barcode and that a single barcode can only be assigned to one item as show in the following example.

```
ENTITY item
        title           : STRING;
        callnum         : STRING;
        unique_number       : barcode;
INVERSE
        contained_by : collection OF contains_item;
        located_on   : physical_shelf OF is_located;
END_ENTITY;

ENTITY barcode;
        number : INTEGER;
        assignedto : item;
END_ENTITY;
Example 3-6
```

The Attribute *unique_number* in the *item* Entity has the Entity *barcode* as its type. The Attribute *assignedto* in the *barcode* Entity barcode has the Entity *item* as its type.

One to Many (1..?) relationships are where an instance in one Entity is related to one or more instances of another Entity. The following example shows an Entity declaration with an Inverse Attribute that shows the cardinality of One to Many.

```
ENTITY circulating_item
        status          : STRING;
        loan_period     : INTEGER;
        checked_outby  : patron;
END_ENTITY;

ENTITY patron;
        name            : STRING;
        mail_address: address;
        card_expired : date;
        barcode         : INTEGER;
UNIQUE
        patron_single : barcode;
INVERSE
        checks_out : BAG [1:?] OF UNIQUE circulating_item FOR checked_outby;
END_ENTITY;
Example 3-7
```

In the *circulating_item* Entity the Attribute *checked_outby* has the *patron* Entity as its type. This links one *circulating_item* Entity to one *patron* Entity. In the *patron* Entity the Inverse Attribute *checks_out* has the link back to the *circulating*_item Entity. The *checks_out* Attribute is a collection of circulating items. These two Entities and Attributes together create a One to Many relationship with each other. This means that a patron can checkout one or many circulating items, and each item is checked out by at most one patron.

In a Many to Many (?..?) relationship, an instance of an Entity A is related to one or more instances of Entity B and each instance of B can be related to one or more instances of A. If the Library Schema presented here tracked the author names of the books, then the Schema would have a Many to Many relationship where many authors can write a book and a book can be written by many authors. The following example shows an Entity declaration with a Many to Many relationship.

```
ENTITY book_item;
        size    : INTEGER;                    -- size in square feet of an item
        writers : BAG [1..?] OF author OF writes;
END_ENTITY;

ENTITY author;
        name : STRING;
INVERSE
        writes  : BAG [1..?] OF book_item OF writers;
END_ENTITY;
Example 3-8
```

The Attribute, *writers* in the *book_item* Entity, contains the list of authors for the book item. The Inverse Attribute *writes,* in the *author* Entity, contains the list of book items that the authors have written.

Developers usually want to include only those situations that are appropriate for a given model. Therefore, there are rules that can be applied to the model that restrict or constrain the attribute values present in a model. The rule has three states: TRUE, FALSE, or unknown. Local rules are specified in the Entity declaration and constraint attribute values in one Entity type. There are two types of rules: Uniqueness and Domain. The Uniqueness Rules specify a single

Attribute or a list of Attributes and constrain them to be unique across all instance of the Entity. These Attributes follow the UNIQUE keyword in an Entity declaration as shown in this example:

```
ENTITY item
        title           : STRING;
        callnum         : STRING;
        barcode         : INTEGER;
UNIQUE
        item_single     : barcode;
INVERSE
        contained_by : collection OF contains_item;
        located_on      : physical_shelf OF is_located;
END_ENTITY;
Example 3-9
```

Domain Rules are constraints on values on one or more Attributes. They specify a logical expression that evaluates to TRUE, FALSE or unknown. A Domain Rule can't depend on any other Entity declaration, but it can reference Functions and Constants as long they don't have other entity declarations in them. The SELF-keyword is used to reference attributes in the Entity that is being declared. The Domain Rule follows the WHERE keyword in an Entity declaration, as shown in the example:

```
ENTITY oversize_book_item;
        SUBTYPE OF (book_item);
WHERE
        big     : size >= 2     -- larger that 2 square feet is big
END_ENTITY;
Example 3-10
```

A more advanced concept in the EXPRESS Modeling Language is the Inheritance hierarchy, which is a way to build specialized versions of Entities from other existing Entities. The Entities used in the Inheritance hierarchy are referred to as Supertypes if they are the more general Entity or Subtypes if they are the more specific. The Subtype and Supertype hierarchy functions as a directed graph. In the directed graph the nodes are considered the Entity types and the links move you from Supertypes to Subtypes.

In an Inheritance hierarchy a Supertype is an Entity that has a more general classification than that of a Subtype Entity. An Entity is declared Supertype explicitly if the definition contains the SUPERTYPE OF clause, or implicitly, when other Entities list it in their SUBTYPE clauses.

```
ENTITY circulating_item;
  SUPERTYPE OF (loaned_circulating_item ANDOR available_circulating_item)
END_ENTITY;
Example 3-11
```

The Entity *circulating_item* is the Supertype of Entities *loaned_circulating_item* and *available_circulating_item*. A Subtype is an Entity that has a more specific classification than a Supertype Entity. A Subtype instance is an instance of its Supertypes and inherits all of the characteristics of the Supertype. The Supertype construct doesn't need a SUPERTYPE OF clause in the Entity definition but is a Supertype when a Subtype specifies a SUBTYPE OF clause in the Entity declaration like in Example 3-12. The example shows that the *item* Entity doesn't have the SUPERTYPE OF clause but the *noncirculating_item* Entity specifies the *item* Entity in the SUBTYPE OF clause. This makes *item* the Supertype of *noncirculating_item*.

```
ENTITY noncirculating_item;
        SUBTYPE OF (item);
END_ENTITY;

ENTITY item;
END_ENTITY;
Example 3-12
```

Supertypes and Subtypes can be classified into two categories, simple and complex Entity data types. A simple Entity data type is an Entity declaration that defines all the significant properties. A complex Entity data type is an Entity declaration that has the characteristics from the inheritance relationships and any additional characteristics that it may have. An example complex Entity data type is the concept of a date. A *date* Entity can be constructed to have the properties month, day, and year. Then the *circulating_item* Entity would contain a property called *datedue*, which is of the *date* type.

Constraints are used with Supertypes and Subtypes and are ways to show how Subtypes and Supertypes behave in relation to each other. Constraints are restrictions on other properties of an Entity as a whole [5]. The legal combinations of Supertypes and Subtypes are based on the constraints of the Supertype and Subtypes. The constraints are ABSTRACT, ONEOF, AND, and ANDOR.

An Abstract Supertype constraint is an Entity declaration that needs more information to be complete. The extra information is defined in the Subtypes of the Abstract Supertype. Abstract Supertypes can't be instantiated, meaning there can't be an instance of an Abstract Supertype. An example of an Abstract Supertype is the *items* Entity in the example in Appendix B. More information about *items* is needed to describe the *item*. The *items* Entity*,* which is a Supertype, can be a *book_item*, *journal_item*, *circulating_item,* or *noncirculating_item.*

The ONEOF constraint is when only one group of Subtypes can be considered in a Supertype/Subtype combination. The Subtypes are considered to be mutually exclusive. The AND constraint is when two or more Subtypes are mutually inclusive. Subtype combinations must include all of the Subtypes. The ANDOR constraint is the default constraint. If the constraint is not specified then the ANDOR constraint is applied. The Subtype combinations include all Subtypes, only one of the Subtypes, or one or more Subtypes. The following example shows the above constraints on Supertypes and Subtypes.

```
ENTITY item;
        ABSTRACT SUPERTYPE OF (ONEOF (book_item, journal_item) AND
(ONEOF (circulating_item, noncirculating_item)));
        title           : STRING;
        callnum         : STRING;
        barcode         : INTEGER;
UNIQUE
        items_single  : barcode;
INVERSE
        contained_by : collection OF contains_items;
        located_on    : physical_shelf OF is_located;
END_ENTITY;

ENTITY book_item;
SUPERTYPE OF (oversize_book_item ANDOR reference_book_item)
        SUBTYPE OF item;
END_ENTITY;
Example 3-13
```

The Supertypes and Subtypes make up the Inheritance hierarchy. In the Inheritance hierarchy, Subtypes have all the properties of its Supertypes. The Subtypes can inherit all the Attributes, Labels, Local Rules, and Global Rules that pertain to its Supertype. EXPRESS can handle several different types of inheritance including Normal Inheritance, Multiple Inheritance, Attribute Inheritance, and Rule Inheritance.

When a Supertype has one or more Subtypes, it is called Normal Inheritance. This is the basic Inheritance type and the default constraint, ANDOR, is used with the Subtypes in this type of Inheritance. An example of Normal Inheritance is the Supertype/Subtype tree is shown in Example 3-14. Here *book_items* is the Supertype and the *oversize_book_items* and *reference_book_items* are the Subtypes.

```
ENTITY book_item;
        SUPERTYPE OF (oversize_book_item ANDOR reference_book_item)
        SUBTYPE OF item;
END_ENTITY;

ENTITY oversize_book_item;
        SUBTYPE OF (book_item);
        size    : INTEGER;   -- size in square feet
WHERE
        big     : size >= 2     -- larger that 2 square feet is big
END_ENTITY;

ENTITY reference_book_item
        SUBTYPE OF (book_item);
END_ENTITY;
Example 3-14
```

Multiple Inheritance is when a Subtype inherits attributes and other properties from two or more Supertypes. In Figure 3-1, Entity A is a Supertype of Entity C, and Entity B is also a Supertype of Entity C. Entity C inherits all the Attributes and other properties from both Entity A and Entity B.

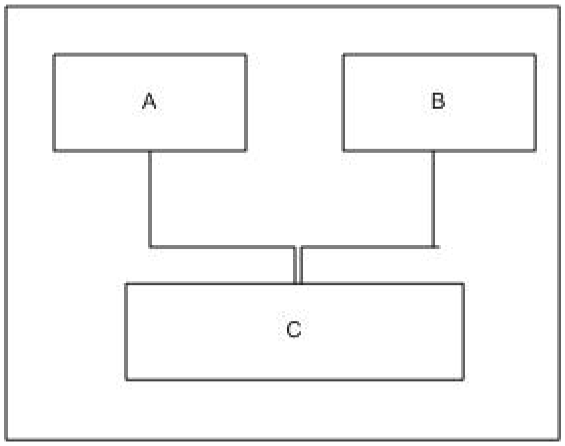


Figure 3-1 – Multiple Inheritance

In Attribute Inheritance Subtypes can see the names of all their Supertypes' attributes. The Subtypes inherits all of the Supertype attributes even if there are multiple Supertypes. Multiple Inheritance can produce multiple Attributes with the same name. In this case Supertype prefixing is used to distinguish between Attributes. Here the name of the attribute is composed of the name of the Supertype followed by a period is inserted and then the Attribute name. A Subtype may inherit the same Attribute from different Supertypes who inherited from a common ancestor [5]. In this case, the Attribute is only inherited once. In the Attributes re-declaration a Subtype re-declares an inherited Attribute to be of a different type. Re-declaration of attributes is useful when more restrictions are required on the Attribute than were applied by the Supertype. Optional values can be changed to mandatory but mandatory values cannot be changed to optional. Using the example in Appendix A, the optional attribute barcode in the item Entity may be re-declared in the circulating_item Entity as a mandatory attribute.

An EXPRESS Schema may re-use types that have been defined in another schema. This is called Interfacing. A schema model may reference some or all parts of another schema. Interfacing is useful to developers when they are able to break up large projects. Dividing large projects cuts down on duplication of work. There are two types of Interface, Use and Reference. The USE Interface treats the types being used from another schema as if they were local definitions in the current schema. Schemas can use a whole schema or just selected types. After the Schema name the keyword USE is used to import a Schema. When the USE statement specifies Entity names, only those Entities are being interfaced to the current Schema. Instances of the interfaced Schema can be created when the USE statement is included. An example of the USE statement is:

```
SCHEMA schema_name;
        USE another_schema_name;
END_SCHEMA;
Example 3-15
```

The REFERENCE mechanism also interfaces types from another schema. These items are visible in the current Schema; however, instances of the interfaced types cannot be created, because the type is not considered a local declaration. The Schema that has interfaced a type can declare an Attribute in an Entity of the interfaced item. The USE clause takes precedence when both the REFERENCE and USE statements interface the same object into one Schema. It is not possible to chain items interfaced through the REFERENCE clause. The use of the REFERENCE statement is as follows.

```
SCHEMA library;
        REFERENCE FROM calendar;
END_SCHEMA;
Example 3-16
```

In addition to the lexical notation of EXPRESS, there is also a graphical notation called EXPRESS-G, which is used to display EXPRESS models symbolically. These models can be viewed in a Schema level view or an Entity level view. The Schema level view shows only the Schema and not the Entities. The Schema diagram shows how multiple Schemas are related to each other. The Entity level view shows all the Entities in a Schema and how they relate to each other. Appendix A contains an EXPRESS-G example of the Library Schema using EXPRESS edition 1, and Appendix C is the same Schema only using EXPRESS edition 3 notation. The *calendar.date* symbol shows how to reference another Schema. Entity constructs are shown with a rectangle such as the *patron* Entity. Types, such as STRING and INTEGER, are shown with a rectangle and the line connecting the type to an Entity indicates an Attribute. An example would be the *patron* Entity's *name* Attribute that is of type STRING. There isn't a graphical way to show local rules; however, in EXPRESS-G an asterisk is used to note a Uniqueness or Domain rule or constraint.

This chapter has presented an overview of the basic concepts of EXPRESS Edition 1. EXPRESS models can be as simple or as complex as needed to model the real world situation. One major construct from EXPRESS is the inheritance hierarchy defined Entity types. Another construct is the constraints that show how each Supertype and Subtype Entity behaves with other Supertypes or Subtypes. The understanding of these constructs is crucial in when looking at the algorithms that will be analyzed later.

CHAPTER 4

EXPRESS EDITION 3

With the popularity of EXPRESS growing, developers realized that EXPRESS could be used for more than manufacturing applications. Developers started making new demands on the modeling language as well as uncovering some ambiguities and problems in the first edition of EXPRESS. Users wanted to build EXPRESS definitions for things such as parts libraries, manufacturing management data, and interface specifications [3]. Some corrections, in the form of a point release, were made and these formed EXPRESS edition 2 [4]. More extensive corrections, clarifications, and significant changes to the inheritance mechanism are proposed for edition 3.

The changes to the Inheritance Hierarchy include lexical changes to how Subtypes are defined, the addition of Connotational and Denotational Subtypes, and the addition of the Total Over constraint. In edition 1 Subtypes had the *Subtype Of* clause in the Entity definition and that has been replaced with a clause called *Subtype_Constraint*. This clause is not declared in the Entity definition but outside of the Entity declaration. An example of the notation for an Entity and a SUBTYPE_CONSTRAINT declaration is given in Example 4-1; this is the same declaration as the Example 3-11 in Chapter 3.

```
ENTITY circulating_item;
        SUBTYPE OF item;
END_ENTITY;

SUBTYPE_CONSTRAINT circulating_type FOR circulating_item;
        (loaned_circulating_item ANDOR available_circulating_item);
END_SUBTYPE_CONSTRAINT;
```
Example 4-1

This enhancement is useful in that it allows different Subtype Constraints to be applied when Supertypes are “used” in different schemas.

Connotational Subtypes are Entity types that have the same data signatures as their Supertypes but contain constraints and rules for attribute values that are not present in their

Supertypes. "An instance of a Supertype may be considered as an instance of the Subtype if it obeys all the constraints of the Subtype" [3]. During the lifetime of an instance, changes to its attribute values can cause it to move into and out of several Connotational Subtypes [3]. The following is an example of a Connotational Subtype.

```
ENTITY oversize_book_item;
        CONNOTATIONAL SUBTYPE OF (book_item);
WHERE
        big     : size >= 2;    -- larger that 2 square feet is big
END_ENTITY;
Example 4-2
```

Connotational Subtypes are useful in allowing Entity instances that fit into a certain category based on their data values to behave as a member of that category.

Another feature added to edition 3 is the Total Over constraint. When an instance of a Supertype is required to be an instance of at least one of its Subtypes, the Total Over constraint is specified in the SUBTYPE_CONSTRAINT clause. A Total Over constraint lists a group of Subtypes $T_1...T_n$ of a Supertype S. It constrains all instances that are Subtypes of S to also be one of these types $T_1…T_n$. A Subtype constraint can have more than one Total Over constraint. An example would be having *book_item* combined with *circulating_item*.

```
SUBTYPE_CONSTRAINT item_types FOR item;
ABSTRACT SUPERTYPE;
        ONEOF (book_item, journal_item) AND
        ONEOF (circulating_item, noncirculating_item);
        TOTAL_OVER (book_item, journal_item);
        TOTAL_OVER (circulating_item, noncirculating_item)
END_SUBTYPE_CONSTRAINT;
Example 4-3
```

Here an instance of *item_types* must be either *book_item* or *journal_item*. It also must be *circulating_item* or *noncirculating_item*.

This change to EXPRESS inheritance led to the need for major changes to the algorithms that check the validity of Subtype/Supertype combinations. An analysis of these changes forms the core of this thesis. To perform this analysis, the techniques presented in the next chapter will be used.

CHAPTER 5

ALGORITHM INFORMATION

Developers using EXPRESS [6] wanted to determine if specific combinations are valid for the inheritance hierarchy in their EXPRESS Schema. To do this, algorithms were developed to make the evaluation easier and manageable. To completely understand how the Test and Generate algorithms work, a basic understanding of algorithms is needed. This chapter gives background information on algorithms and how to analyze them.

An Algorithm contains a sequence of clearly specified statements or instructions that are followed to solve a problem or compute a function [1]. Designers and programmers use them as a guide to follow when creating programs and systems. Algorithms are important ways to organize a problem, give the user direction, and make a problem more manageable. The statements can be executed any number of times if the repetition is specified in the algorithm. Algorithms must terminate no matter what input values that they may have to process and are sometimes written with a combination of programming languages and English phrases.

Before algorithms are converted to a program, it is useful to evaluate the algorithm for correctness, completeness, and sometimes equivalence between similar algorithms. This is called an analysis of the algorithm. Additionally analysis consists of ways to determine the amount of work done by an algorithm and the amount of storage space it uses. Algorithms can also be analyzed for simplicity, clarity, and optimality. Developers use analysis of algorithms to improve and refine algorithms and to be assured that they produce expected results. Analysis of an algorithm can help developers decide which algorithm to use in a specific situation. Developers can find the "strengths and weaknesses of different algorithms that do the same thing, and figure out which set of characteristics most closely matches the needs of the situation." [2]

Algorithms can be proven correct by verifying that the results from it are correct. The first step is to determine what correct means. This requires a clear and precise statement about the legal inputs and the expected outputs for all input. One method is to use mathematical induction and loop invariants, which are conditions and relationships that are satisfied by the variables and data structures at the end of an iteration of a loop [1]. Induction on the number of iterations of a loop is a way to construct a loop invariant, which are used to show that an algorithm has produced the expected results. Analysis on long and complex algorithms can be hard to do and time consuming but can show that the program being developed from the algorithm will work.

Another way to analyze algorithms is to determine the amount of work done by the algorithm. The measure of work should be both precise and general enough to develop a theory that is useful for many algorithms and applications. The number of passes through a loop or the number of executions of a certain type of operation can determine the work done by an algorithm. Counting a basic operation in an algorithm is sometimes considered the complexity measure, which is how the complexity or the amount of work done by an algorithm is determined. The amount of work done can be different for the same algorithm based on the amount, order, or type of the input. The behavior of an algorithm can be described by finding the worst-case analysis and average-case analysis. Worst-case analysis is the process of finding the maximum number of operations performed by an algorithm given any input size and usually gives the upper bound for the work done. Baase [1] states the formula for computing the worst-case complexity as the following;

> Let D_n be the set of inputs of size n for the problem under consideration, and let I be an element of D_n. Let $t(I)$ be the number of basic operations performed by the algorithm on input I. Then the function is $W(n)=max\{t(I) \mid I \in D_n\}$.

In some cases it might be useful to determine the average amount of work done by an algorithm, which is called average-case analysis. This type of analysis computes number of operations performed for each input size and then takes the average. Some input may occur more frequently than others may and so a weighted-average can be used to compute the average-case. Baase [1] states the formula for computing the average-case complexity as the following:

> Let $p(I)$ be the probability that input I occurs. Then the average behavior of the algorithm is defined as $A(n)= \sum_{I \in D_n} p(I)\, t(I)$.

The function *p* is determined by information known about the inputs or by experience or by assuming that all inputs are equally likely to occur for the given problem.

Another way to analyze algorithms is to determine how much space is used by the algorithm for storing the data. There may be times when the algorithm may need to have a area to store the information required to process and store the inputs. Space usage depends on the implementation of the algorithm; however, there are some ways to examine an algorithm to decide how much storage space is needed. Assignment statements, constants, variables, and the inputs to the algorithm are going to require storage space as well as space for doing computations while the algorithm is processing the inputs. Space used by the variables and inputs can be different depending on the type of the data for example that arrays can take up more space than an integer value. "If the amount of space used depends on the particular input, worst-case and average-case analysis can be done" [1].

Another way to analyze algorithms is to determine if they are simple. This may not mean that the algorithm is more efficient but may make it easier to verify for correctness. Additionally, simple algorithms are usually easier to understand, write, debug, or modify when it is developed into a program. Simplicity makes the process of solving a problem more straightforward.

Algorithms may also be tested for optimality. This classifies algorithms based on the number of operations performed to do some task. When an algorithm has the least number of operations needed to solve a problem in its class of algorithms, the algorithm is considered to be optimal in its class in the worst-case. A class of algorithms includes all algorithms that have been developed and those that have not been discovered yet that perform the same basic operations. In order to prove an algorithm is optimal, theorems are used to prove a lower bound on the number of operations needed to solve a problem. Baase [1] says there are two tasks that need to be done to find an optimal algorithm. "The first step is to devise what seems to be an efficient algorithm and call it *A*. Analyze *A* and find a function *W* such that, for inputs of size *n*, *A* does at most *W(n)* steps in the worst case. The second step is for some function *F*, prove a theorem stating that, for any algorithm in the class under consideration, there is some input of size *n* for which the algorithm must perform at least *F(n)* steps. If the functions *W* and *F* are equal, then the algorithm *A* is optimal for the worst case. If not, there may be a better algorithm or a better lower bound."

Algorithm equivalence is another important criterion that is used when comparing algorithms. Showing equivalence can be done by showing that the output from one algorithm produces a TRUE result from another algorithm and vice versa. This paper evaluates the Test and Generate algorithms for equivalence using some of the techniques discussed in this chapter. The next two chapters will describe these two algorithms in detail with comparisons between the two in subsequent chapters.

CHAPTER 6

GENERATE ALGORITHM

This chapter presents the information on how the Generate Algorithm was developed and why. It also discusses how the Generate Algorithm works and the proposed improvements made to the algorithm.

Legal Supertype and Subtype combinations in EXPRESS Schemas are constructed based on the constraints of the inheritance hierarchy. These combinations represent the legal Entity instance that may exist in a file defined by EXPRESS Schema. For a Supertype/Subtype graph, there may be a large number of Entity data types that can be instantiated. The two algorithms that deal with these types checking are called the Generate [6] and Test [7] Algorithms.

This chapter examines the Generate algorithm in depth. First a set of simplification rules for inheritance expressions are given, then the Generate Algorithm from the EXPRESS manual is given, followed by a step-by-step example. Then a detailed analysis of the step (G) in the algorithm that deals with the Total Over constraint is given with an example. A new version of this step is then considered. The chapter concludes with a discussion of the strengths and weaknesses of both approaches.

The latest version of the Generate Algorithm is from the EXPRESS Language Reference Manual [4] version ISO TC184/SC4/WG11 N105. The Generate Algorithm generates all legal possible combinations of Entity types in the Supertype and Subtype hierarchy. The input to the algorithm is the EXPRESS inheritance hierarchy and the constraints defined in the EXPRESS Schema being evaluated. The output from this algorithm is the set of all legal combinations of Entity types. These are called partial complex Entity data types and are denoted by the names of each of the component Entity data types separated by the ampersand (&) character. An evaluated set is defined to be a mathematical set of the partial complex Entity data types that is denoted by the partial complex Entity data types separated by a comma ‘,’ enclosed in square brackets ‘[]’. An example would be [circulating_item, book_item]. An empty evaluated set is denoted by [ ], which means that there are no partial complex Entity data types. The partial complex Entity data types may be combined to form other partial complex Entity data types. The ≡ symbol is used to show that two items are equivalent. An example would be {circulating_item & book_item ≡ book_item& circulating_item}. To support this algorithm many identities, operators, and constraints need to be defined as shown in the following tables. These identities, operators, and constraints are rules that can be used to reduce a partial complex Entity data type to a simpler canonical form.

IDENTIES	EXAMPLES
1. An Entity type can occur only once in a partial complex Entity data type.	{A&A≡A}
2. The grouping of the partial complex Entity data types is commutative.	{A&B≡B&A}
3. The grouping of the partial complex Entity data types is associative; the parentheses indicate evaluation precedence, which in this case makes no difference to the evaluation.	{A&(B&C)≡(A&B)&C≡(A&B&C)}

Table 6-1 – Identities [4]

WAYS TO FORM EVALUATED SETS (OPERATORS)	EXAMPLES
4. The '+' operator adds the partial complex Entity data type to the evaluated set as a new member of the set.	{A+[B1,B2] ≡ [B1,B2]+A≡[A,B1,B2]}
5. The '&' operator adds the partial complex Entity data type to all the partial complex Entity data types within the evaluated set. It is therefore distributive over evaluated set.	{A&[B1,B2] ≡[B1,B2]&A≡[A&B1,A&B2]}
6. An evaluated set can be formed to contain all of the elements of two combined sets. This is the union of the two sets.	{[A1,A2]&[B1,B2] ≡[A1&B1, A1&B2, A2&B1, A2&B2]}
7. An evaluated set can be formed by repeated application of the distribution rule for & for each element of the first evaluated set over the second evaluated set.	{[A1,A2]&[B1,B2] ≡[A1&B1,A1&B2,A2&B1,A2&B2]}
8. An evaluated set can be found by filtering by a complex type, the new evaluated set contains only those elements in the original evaluated set, which contain the given partial complex Entity data type.	{[A,A&B,A&C,A&B&D,B&C,D]/A ≡ [A,A&B,A&C,A&B&D]}

Table 6-2 - Operators [4]

WAYS TO FORM EVALUATED SETS (OPERATORS)	EXAMPLES
9. A new evaluated set can be formed by repeated filtering an evaluated set, ONE, by each partial complex Entity data type in the second evaluated set, TWO, and combining the results using the '+' symbol. The new set will contain only those partial complex Entity data types from ONE that contain at least one of the Entity types from TWO.	ONE= {[A,A&B,A&C,A&B&D,B&C,D] TWO=[B,D] ONE/TWO ≡ [A&B,A&B&D,B&C,D]}
10. An evaluated set can be formed, which contains all the elements in the first evaluated set except for those in the second evaluated set.	{[A1,A2,B1,B2]-[A2,B1]≡[A1,B2]}
11. An evaluated set may be reordered, the order does not matter.	{[A,B] ≡[B,A]}
12. An evaluated set can only have a particular complex Entity data type once. The complex Entity data type appears only once in the set.	{[A,A,B] ≡[A,B]}
13. An evaluated set can be nested.	{[A,[B,C]] ≡[A,B,C]}

Table 6-2 - (continued)

CONSTRAINTS	EXAMPLES
14. The ONEOF constraint is converted to an evaluated set containing each individual entity type in the ONEOF as a partial complex Entity data type.	{ONEOF (A,B) becomes [A,B]}
15. The AND constraint is equivalent to the '&' operator in an evaluated set.	{A AND B becomes [A&B]}
16. The ANDOR constraint generates an evaluated set, which contains each of the operands separately as well as combined using the '&' operator.	{A ANDOR B becomes [A,B,A&B]}

Table 6-3 - Constraints [4]

To generate the evaluated set, the following algorithm is given in the EXPRESS manual [4]. The algorithm will be followed by a step-by-step example. The steps for the Generate Algorithm [4] are as follows:

The evaluated set R of complex entity datatypes is computed by the following process:

a) Identify all entity declarations, which form the subtype/supertype graph.

NOTE 1 This may require multiple iterations in cases with complex subtype/supertype graphs.

b) For each supertype i in the subtype/supertype graph, identify all types j1, j2, ... jk in the subtype/supertype graph that are defined as subtypes of i that either: Do not occur in any subtype_constraint defined for i or occur only in a TOTAL_OVER clause of a subtype_constraint defined for i. Construct a subtype_constraint of the form:Qi

```
SUBTYPE_CONSTRAINT i_othersubtypes FOR i;
  j1 ANDOR j2 ANDOR ... ANDOR jk;
END_SUBTYPE_CONSTRAINT;
```

Consider this constraint as part of the schema for the purposes of this algorithm.

c) For each supertype i in the subtype/supertype graph identify all subtype constraints sc_1 , sc_2 , ... sc_k , that have i in their FOR clause. At this point, the parts of subtype constraints that contain TOTAL_OVER or ABSTRACT restrictions are ignored. Combine the subtype expressions sx_i of these constraints into a single SUBTYPE_CONSTRAINT

```
SUBTYPE_CONSTRAINT st_i FOR i;
  sx_1 ANDOR sx_2 ANDOR ... ANDOR sx_k;
END_SUBTYPE_CONSTRAINT;
```

NOTE 2 A singleton constraint, which contains only the name of a subtype is a legal constraint.

d) For each supertype i in the subtype/supertype graph, generate the evaluated set which represents the constraints between its immediate subtypes by applying the reductions in annex B.3.1.1 and the identities in annex B.2.1.1 to the SUBTYPE_CONSTRAINT st_i given by step c above. Combine i with the result using the & operator. If i is not defined as an ABSTRACT supertype in its entity declaration or in any SUBTYPE_CONSTRAINT of i then add i to the result using +. Call this set E_i .

e) For each root supertype r in the subtype/supertype graph, expand E_r as follows:
   1) For each subtype s of r, replace every occurrence of s (including those within partial complex entity datatypes) in E_r with E_s , if available, and apply reductions in annex B.3.1.1 and the identities in annex B.2.1.1.

   2) Recursively apply step 1 to each s, expanding subtypes of s until leaf entities are reached (for which no E_s is available).

   NOTE 3 This recursion must terminate, since there are no cycles in the subtype/supertype graph.

f) Combine root sets. Create $R = \sum_r E_r / E_{r1} + E_{r2} + \ldots$, i.e. R is the union of the sets produced in step e.

g) For each supertype s in R, for each TOTAL_OVER SUBTYPE_CONSTRAINT T of the form $t_1 , t_2 , \ldots t_k$ defined for s
   1) Define t to be: (t_1 ANDOR t_2 ... ANDOR t_k)

   2) For all immediate subtypes s_i of s not in $\{t_1 , t_2 , \ldots t_k \}$ replace each occurrence of s_i in R by the expression derived from (s_i AND t) using the definitions in annex B.3.1.1

   3) Reduce R according to the reductions in annex B.3.1.1 and the identities in annex B.2.1.1.

h) For each multiply inheriting subtype m, do the following:
   1) For each of its immediate supertypes s, generate the set R/m/s which contains exactly those complex datatypes in R which include both m and s.

   2) Generate the evaluated set of supertype combinations permitted by m, Pm = R/m/s1 & R/m/s2 &..., i.e., combine the evaluated sets produced in step 1 using &.

   3) Generate the evaluated set of supertype combinations which may not include all the supertypes of m, $X_m = \sum_s R/m/s$, i.e., union together the evaluated sets produced in step 1.

   4) Put $R = (R - X_m) + P_m$.

i) For each subtype constraint expression k (including those generated in steps b and g of the form ONEOF (S_1, S_2, ...), do the following:
   1) For each pair of subexpressions S_i, S_j controlled by k ($i < j$), compute the set of combinations disallowed by ONEOF (S_i, S_j): $D_k^{i,j} = [S_i \& S_j]$. Reduce $D_k^{i,j}$ according to the reductions in annex B.3.1.1 and the identities in annex B.2.1.1.

   2) Set $D_k = \sum_{i,j} D_k^{i,j}$, i.e. D_k is the union of the sets computed in step 1.

   3) Put $R = R - (R / D_k)$.

j) For each subtype constraint expression (including those generated in steps b and g) k of the form S_1 AND S_2, do the following:
   1) Compute the set of required combinations dictated by k, $Q_k = [S_1 \& S_2]$. Reduce Q_k according to the reductions in annex B.3.1.1 and the identities in annex B.2.1.1.

   2) For each entity datatype i named in k, compute the set of invalid entity combinations containing i which are disallowed by k, $D_k^{1} = R/i - R/(Q_k /i)$.

3) Set $D_k = \sum_i D_k{}^i$, i.e., D_k is the union of the sets computed in step 2.

4) Put $R = R - D_k$.

k) The final evaluated set R is the evaluated set for the input subtype/supertype graph.

Using the EXPRESS edition 3 example of the Library model, the Generate Algorithm starts off with step a by identifying all of the Entity declarations that form the Supertype/Subtype graph. The root Supertype is the Abstract Entity item, which has book_item, journal_item, circulating_item, and noncirculating_item as Subtypes. The Entity book_item is a Supertype of oversize_book_item and reference_book_item. The Entity circulating_item is a Supertype of loaned_circulating_item and overdue_circulating_item. Steps b and c define SUBTYPE_CONSTRAINT clauses for each Supertype in the Supertype/Subtype graph and combining SUBTYPE_CONSTRAINT clauses that have the same Supertype in the FOR clause into a single SUBTYPE_CONSTRAINT clause using the ANDOR constraint. For the Library example, step d generates an evaluated set for item of :

```
{book_item,
journal_item,
book_item &journal_item,
circulating_item,
noncirculating_item,
circulating_item&noncirculating_item}
```

Applying the constraint rules, the set can be reduced to

```
{book_item,
journal_item,
circulating_item,
noncirculating_item}.
```

The Supertype needs to be combined with each element in the set using the & operator. Now the set, E_{item}, will be

```
{item&book_item,
item&journal_item,
item&circulating_item,
item&noncirculating_item}.
```

Because item is an Abstract Supertype, it is not added to the set. The same is done for the other Supertypes, which are book_item and circulating_item. The set, E_{book_item}, for book_item is

> {book_item,
> book_item&oversize_book_item,
> book_item&reference_book_item,
> book_item&oversize_book_item&reference_book_item}.

The set, $E_{circulating\_item}$, for circulating_item is

> {circulating_item,
> circulating_item&loaned_circulating_item,
> circulating_item&overdue_circulating_item,
> circulating_item&loaned_circulating_item &overdue_circulating_item}.

Step e generates the following evaluated set, E_{item},

> {item&book_item,
> item&book_item&oversize_book_item,
> item&book_item&reference_book_item,
> item&book_item&oversize_book_item &reference_book_item,
> item&journal_item,
> item&circulating_item,
> item&circulating_item&loaned_circulating_item,
> item&circulating_item&overdue_circulating,
> item&circulating_item&loaned_circulating_item&overdue_circulating_item,
> item&noncirculating_item}.

There is only one root Supertype in this graph; therefore, step f sets R equal to the E_{item} set. Since all of the immediate Subtypes of item are the only Subtypes that are part of the TOTAL_OVER constraints, step g is not necessary. The lack of Multiply inheriting Subtypes in the Library model obviates the need for step h. Step i looks at each ONEOF Subtype constraint and creates a set that containing the combinations that are disallowed. D_{item_types} is the set

> {book_item&journal_item, circulating_item&noncirculating_item).

The resulting set, R, from this step, i, is

> {item&book_item,
> item&book_item&oversize_book_item,
> item&book_item&reference_book_item,
> item&book_item&oversize_book_item&reference_book_item,

item&journal_item,
item&circulating_item,
item&circulating_item&loaned_circulating_item,
item&circulating_item&overdue_circulating_item,
item&circulating_item&loaned_circulating_item&overdue_circulating_item,
item&noncirculating_item}.

Because there are not any Subtypes that have the AND constraint, step j is not necessary. The final set, R, is

{item&book_item,
item&book_item&oversize_book_item,
item&book_item&reference_book_item,
item&book_item&oversize_book_item &reference_book_item,
item&journal_item,
item&circulating_item,
item&circulating_item&loaned_circulating_item,
item&circulating_item&overdue_circulating_item,
item&circulating_item&loaned_circulating_item&overdue_circulating_item,
item&noncirculating_item}.

The remainder of this chapter compares two versions of the Generate Algorithm for EXPRESS-3, the original version of the algorithm is described in the EXPRESS Language Reference Manual ISO TC184/SC4/WG11 N105 [4] and the new version was developed as part of this project. Both versions will be explained using Example 6-1a below.

```
SUBTYPE_CONSTRAINT item_types FOR item;
        TOTAL_OVER (book_item, journal_item);
END_SUBTYPE_CONSTRAINT;
```
Example 6-1a

Suppose that *book_item, journal_item, circulating_item,* and *noncirculating_item* are Subtypes of *item*. Both algorithms are the same through step F. They differ in Step G that handles the Total Over constraint. The result set, R, after step F for both versions of the algorithm is the same:

R={item, book_item&item, journal_item&item, circulating_item&item, noncirculating_item&item, book_item&journal_item&item, book_item&circulating_item&item, book_item&noncirculating_item&item, journal_item&circulating_item&item, journal_item&noncirculating_item&item, circulating_item&noncirculating_item&item, book_item&journal_item&circulating_item&item, book_item&journal_item&noncirculating_item&item, book_item&circulating_item&noncirculating_item&item, journal_item&circulating_item&noncirculating_item&item, book_item&journal_item&circulating_item&noncirculating_item&item}
Example 6-1b

In the original version of the Generate Algorithm, step G (as shown above) processes the Total Over constraints in the inheritance hierarchy. Applying this step of the Generate Algorithm to Example 6-1a, we find that the set R has only one Supertype, the Entity named "*item*". This Supertype has only one Total Over Subtype Constraint named "*item_types*". In the first sub-step, *t* is defined to be {*book_item* ANDOR *journal_item)*. The second sub-step identifies the immediate Subtypes of *item,* which are not in the Total Over list. In this example, these Subtypes are *circulating_item* and *noncirculating_item.* Every occurrence of *circulating_item* in any complex entity datatype in the set R is replaced with the expression (*circluating_item* AND (*book_item* ANDOR *journal_item*)). Similarly, each occurrence of *noncirculating_item* is replaced by (*noncirculating_item* AND (*book_item* ANDOR *journal_item*)). In the third sub-step, R is simplified using the defined reductions and identities producing:

R={book_item&item, journal_item&item, book_item&journal_item&item, book_item&circulating_item&item, journal_item&circulating_item&item, book_item&journal_item&circulating_item&item, book_item&noncirculating_item&item, journal_item&noncirculating_item&item, book_item&journal_item&noncirculating_item&item, book_item&circulating_item&noncirculating_item&item, journal_item&circulating_item&noncirculating_item&item, book_item&journal_item&circulating_item&noncirculating_item&item, book_item&item, journal_item&item, book_item&journal_item&item, book_item&circulating_item&item, book_item&noncirculating_item&item, journal_item&circulating_item&item, journal_item&noncirculating_item&item, book_item&journal_item&circulating_item&item, book_item&journal_item&noncirculating_item&item,

book_item&circulating_item&noncirculating_item&item,
journal_item&circulating_item&noncirculating_item&item,
book_item&journal_item&circulating_item&noncirculating_item&item}

At this point, duplicate elements may be present in set R. These are removed, giving the simplified set R below:

R={book_item&item, journal_item&item, book_item&journal_item&item,
book_item&circulating_item&item, book_item&noncirculating_item&item,
journal_item&circulating_item&item, journal_item&noncirculating_item&item,
book_item&journal_item&circulating_item&item,
book_item&journal_item&noncirculating_item&item,
book_item&circulating_item&noncirculating_item&item,
journal_item&circulating_item&noncirculating_item&item,
book_item&journal_item&circulating_item&noncirculating_item&item}

The new version of step G of the Generate Algorithm given in Figure 6-1 looks at each Total Over constraint T of each Supertype S in the Schema. It evaluates each partial complex entity datatype, r, in the set R. If r contains a Subtype of S but does not contain one of the types specified in T then r is inserted into a discard set for the Supertype. The elements of the discard set are removed from the result set R, which is then simplified using the identities, operators, and constraint rules [4] to further reduce and simplify the set.

```
g) For each supertype $s$ in R where s has Subtypes s1…sn,
     For each TOTAL_OVER SUBTYPE_CONSTRAINT T of the form $t_1$ ... $t_k$ defined for $s$
   1) For each Subtype si where i=1 to n of s, excluding those in the list $t_1 \ldots t_k$, Compute the set
      $D_{si}^T$ as follows:
            For each complex entity datatype r in set R
            If r contains si but does not contain at least one element of $t_1 \ldots t_k$
            Then r is inserted into $D_{si}^T$

      $D_{si}^T$ thus contains the elements of R, which include type si but not any of the types in the
      list $t_1 \ldots t_k$.

   2) Let $D^T = \sum_{i=1}^{n} D_{si}^T \equiv D_{s1}^T + D_{s2}^T + \ldots + D_{sn}^T$, i.e. the union of all the sets from part 1
      above.
   3) Let $R = R - D^T$. Simplify R using the reductions in annex B.3.1.1 and the identities in
      annex B.2.1.1.
```

Figure 6-1 – Improved Generate

Using the same EXPRESS definition and result set from Example 6-1a and 6-1b, the new version of the Generate Algorithm handles the Total Over differently. Applying the new Step G to Example 6-1a, there is only one Supertype named *item* with the four Subtypes, book_item, journal_item, circulating_item, and noncirculating_item, to consider. This Supertype has only one Total Over Subtype Constraint named "item_types" with the elements of book_item and journal_item.

Step 1, calculates the two discard sets for item.

$$D^{\text{item\_types}}_{\text{circulating\_item}} = \{\text{circulating\_item\&item, circulating\_item\&noncirculating\_item\&item}\}$$

$$D^{\text{item\_types}}_{\text{noncirculating\_item}} = \{\text{noncirculating\_item\&item, noncirculating\_item\&circulating\_item\&item}\}$$

In step 2 these are combined to become

$D^{\text{item\_types}}$ = {circulating_item&item, noncirculating_item&item, circulating_item&noncirculating_item&item}

In step 3, the elements of set D^{item_types} are then removed from the result set R giving:

> R={book_item&item, journal_item&item, book_item&journal_item&item, book_item&circulating_item&item, book_item&noncirculating_item&item, journal_item&circulating_item&item, journal_item&noncirculating_item&item, book_item&journal_item&circulating_item&item, book_item&journal_item&noncirculating_item&item, book_item&circulating_item&noncirculating_item&item, journal_item&circulating_item&noncirculating_item&item, book_item&journal_item&circulating_item&noncirculating_item&item}.

It is clear by inspection that both versions of Step G of the Generate Algorithm give the same result set R in the example above.

It can be shown that the algorithms are in fact equivalent. In the original version the elements in the Total Over are combined together with an ANDOR constraint. Then they are combined with other elements in the set R using an AND constraint. This insures that all of the elements in the set R comply with the Total Over constraint. However, this produces several duplicate and invalid elements that have to be discarded by applying the rest of the algorithm, which is where all the steps that remove invalid elements are applied. When there are several elements in the Total Over constraint, this process will be time consuming and cumbersome. In the new version of Step G, a set is created for each Subtype that is not an element in the Total Over constraint. The elements in this set are elements of the set R that have the Subtype in question but don't have one of the elements in the Total Over list. Once all of the Subtypes have been evaluated, the sets are unioned together into a subset. The elements in this subset are then removed from the set R giving the final set R for Step G. The new version of Step G identifies the elements of R that do not have an element from the Total Over constraint, which would mean the element was invalid, and removes it from the set R. This is not as time consuming or complex as the original version of Step G because the sets are smaller than those generated by the original version. A second advantage is that duplicate and invalid elements are not introduced into the set R, only to be discarded by later steps in the algorithm.

Both versions of Step G of the Generate Algorithm have advantages and disadvantages. The key advantage of the original version of Step G of the algorithm was that it worked by reusing existing parts of the original Generate Algorithm. It used the ANDOR constraint to generate a set based on the Subtypes in the Total Over constraint. This allowed the Total Over constraint to be evaluated without implementing any new methods. This version of the algorithm step G is easy to implement because of the reuse of existing code. The major disadvantage of the original version of Step G is the number of iterations needed to generate the large set of partial Entity data type combinations, and its introduction of extraneous elements to R which had to be removed in later steps.

The advantage of the new version of the Generate Algorithm is that rather than adding partial complex Entity instances to the result set, and then simplifying, it generates the much smaller set of Entity types constrained by the Total Over constraint, and then identifies the elements in the evaluated set that must be removed. The new version thus is much faster an conceptually simpler. One disadvantage is that a new algorithm will need a new implementation, not just the re-use of existing code.

CHAPTER 7

TEST ALGORITHM

The second algorithm used to validate inheritance in EXPRESS is the Test Algorithm developed by Staub, Maier, and Schonefeld [7]. This algorithm checks a partial complex entity type for membership in an inheritance hierarchy without generating the set. In this chapter this algorithm will be explained with examples. Then a formal version of the algorithm will be presented. This algorithm had to be developed as no formal specification of the Test Algorithm has ever been published. The algorithm is given in 2 phases and a proposed extension to the Test Algorithm is given to handle the Total Over constraint, and a revised algorithm is presented.

The Test Algorithm has two phases; in the first phase an evaluation graph is generated based on the inheritance constraints in a given Schema. Once this graph is generated, the second phase traverses this graph to determine if a given partial complex entity type is valid. The graph represents the Supertype/Subtype graph including the Supertype constraints and contains two types of nodes. One type of node is the constraint node that corresponds to the Supertype constraint expressions AND, ANDOR, and ONEOF. A type node exists to represent each Supertype or Subtype Entity type. Every type node is connected to exactly one constraint node. The algorithm checks each node using a pre-order traversal. This traversal starts with the root Supertype node and follows the graph until the left most child at the lowest level is found, at which the algorithm starts evaluating the nodes in the graph. The algorithm has two inputs, the evaluation graph and the partial complex entity type instance. The output from the algorithm is TRUE, FALSE, or ERROR. TRUE is obtained when the type is shown to be legal by satisfying the Supertype constraints, FALSE is obtained when type is not part of the current type but does not violate the Supertype constraints, and ERROR is obtained when the type violates at least one of the Supertype constraints. There are four rules that the algorithm check while it is processing each node. Let *T* be the set of the Entity types of the evaluation graph [7].

1. If *n* is a type node with $n \in T$, ensure that all its Supertypes are also elements of *T*;
2. If *n* is an abstract type node with $n \in T$, ensure that there is at least one direct Subtype node *m*, with $m \in T$;

3. If *n* is an AND-node the following constraint must be satisfied for all direct Subtype nodes m_I: $m_I \in T$ *or* $m_I \notin T$;
4. If *n* is an ONEOF-node, ensure that there is at most one direct child type node *m*, with $m \in T$.

The following graph in Figure 7-1 is an evaluation graph for the Library model.

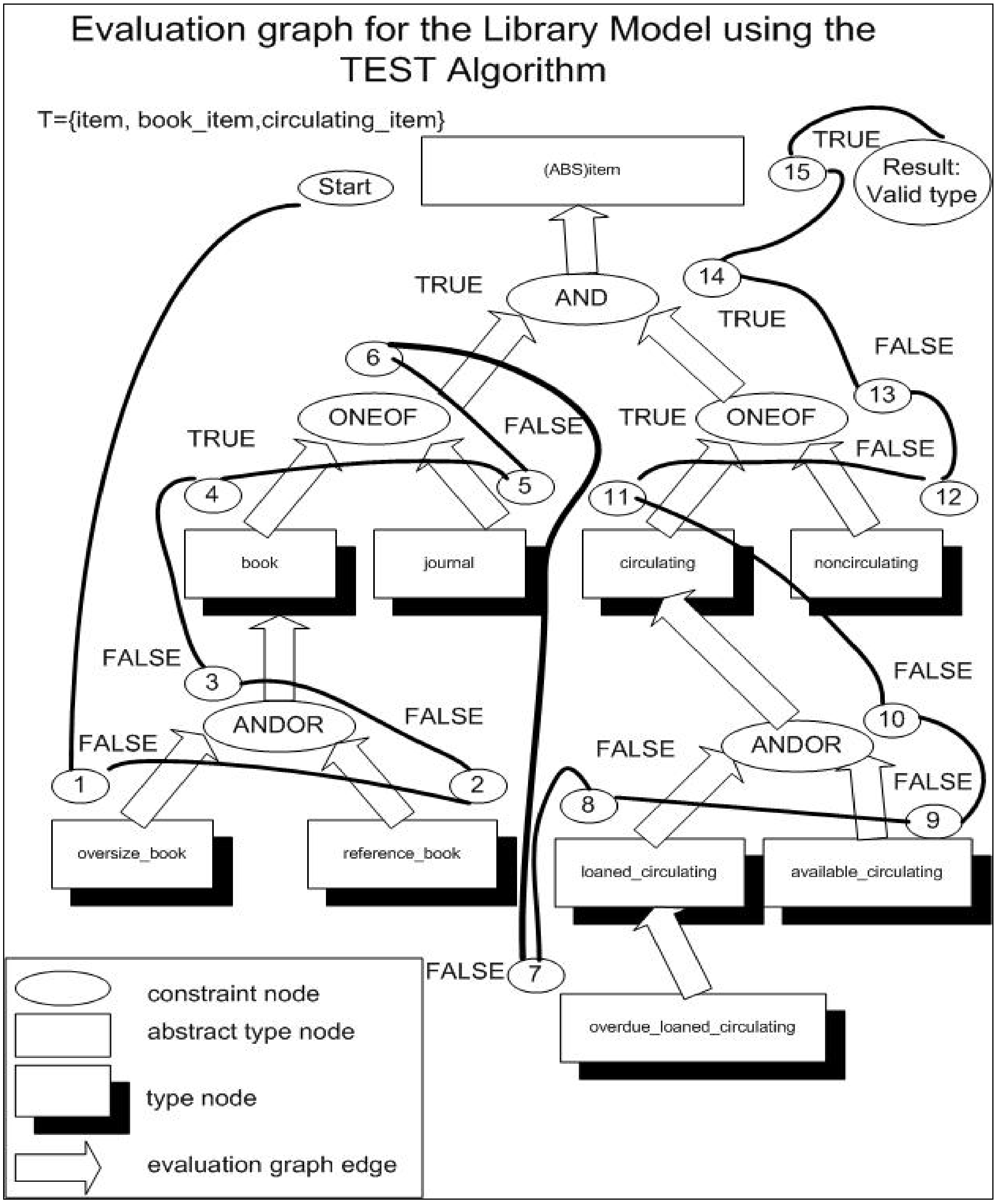


Figure 7-1 – Evaluation Graph with TRUE Result

The graph in Figure 7-1 demonstrates how the Test Algorithm checks the partial Entity data type of [item, book_item, circulating_item]. The algorithm starts with item and finds the left most child, oversize_book_item, and then moves on to reference_book_item. Because neither of these are part of the partial entity data type being tested and the node generates a FALSE. The next check is at the book_item Subtype, which produces a TRUE while the check at the journal_item Subtype produces a FALSE. The check at the ONE OF constraint produces a TRUE because neither book_item nor journal_item are listed in the partial Entity data type being tested. The rest of the Subtypes are checked in a similar fashion. The example in the graph is a valid type in this Supertype/Subtype graph.

A FALSE result is generated when the partial Entity data type doesn't violate any of the Supertype/Subtype constraints but is not a valid data type. If the partial Entity data is [patron], the evaluation graph would produce a FALSE result at each step because the *patron* Entity is not part of the Supertype/Subtype hierarchy. The following graph in Figure 7-2 demonstrates how the Test Algorithm produces a FALSE result.

An ERROR result is generated when the partial Entity data type violates a Supertype/Subtype constraint. For this example, lets assume that the Entity *loaned_circulating_item* is an ABSTRACT Supertype and the partial Entity data type that we need to check is [item, book_item, circulating_item, loaned_circulating_item]. The Test Algorithm would fail when the *overdue_loaned_circulating_item* Entity is checked because it is not included in the partial Entity data type, and the Abstract Entity can not exist without a Subtype. Figure 7-3 demonstrates how the ERROR condition is obtained in this case.

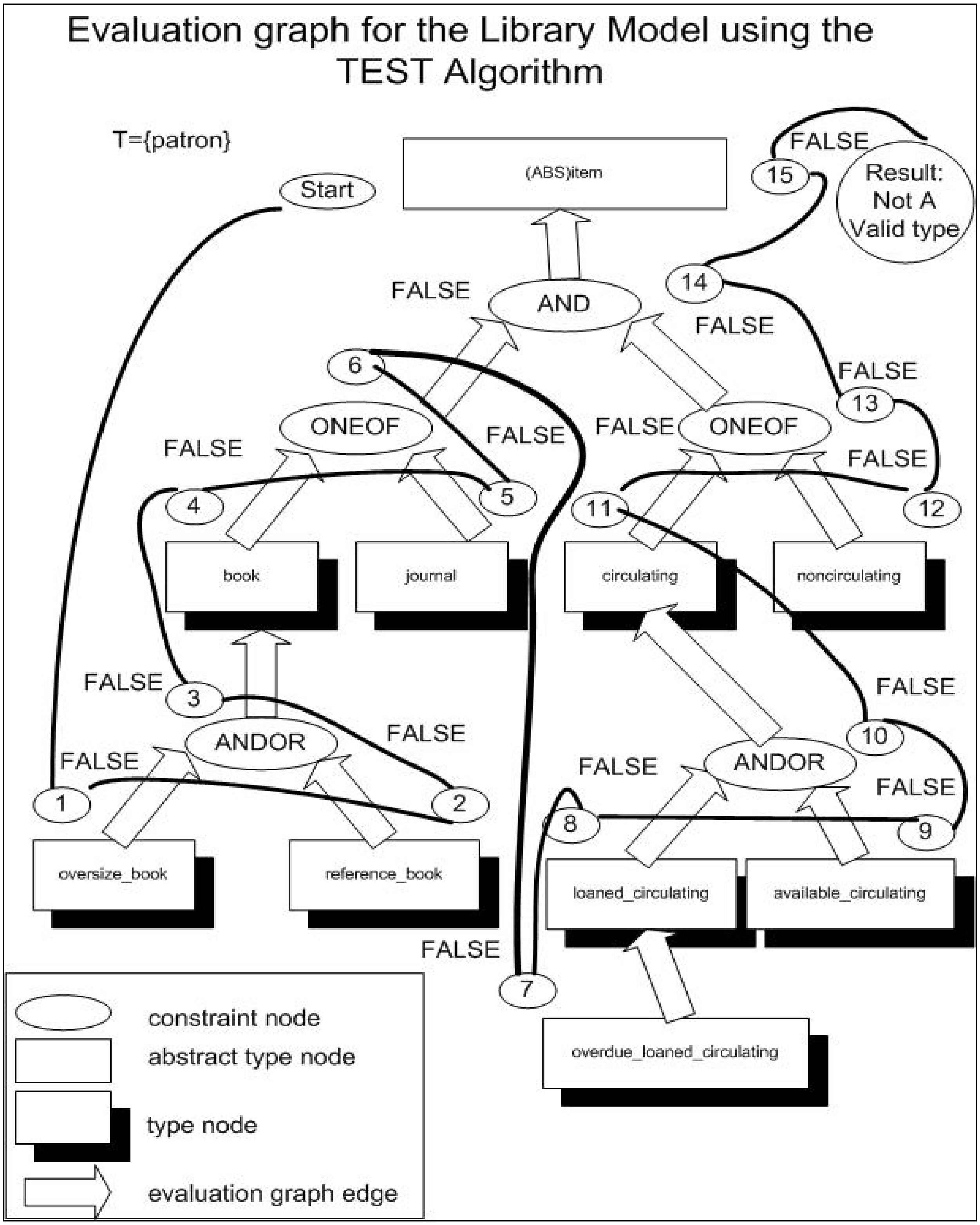


Figure 7-2 – Evaluation Graph with FALSE Result

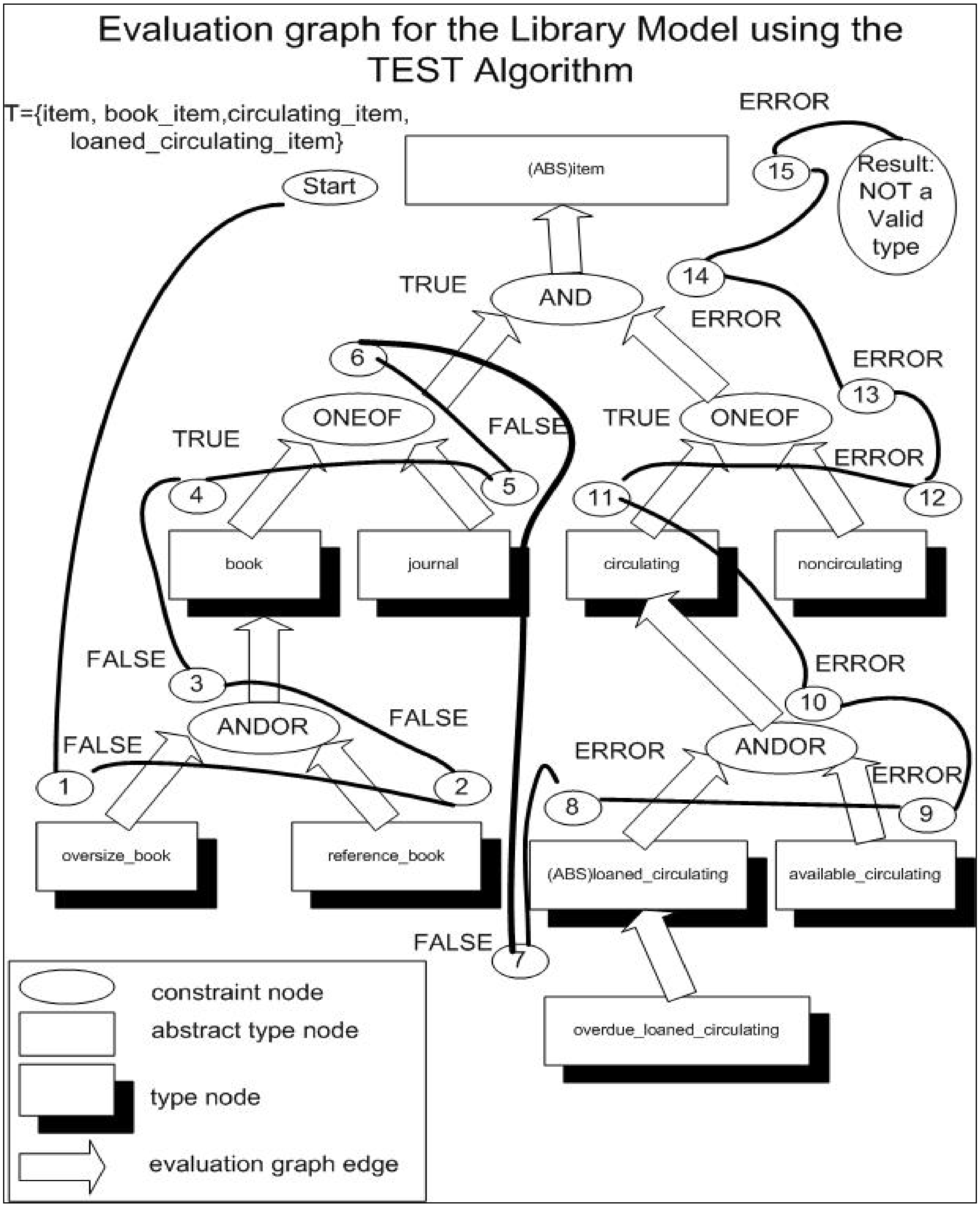


Figure 7-3 – Evaluation Graph with ERROR Result

For this thesis, the Test Algorithm was formalized based on the paper *Multiple Class Membership and Supertype Constraint Handling - Concepts and Implementation Aspects* by Staub, Maier, and Schonefeld [7]. The general description of the approach was converted to a step-by-step formal algorithm that specifies two phases. Phase 1 describes how to create the Evaluation graph. Phase 2 details how to evaluate a complex entity instance using the Evaluation graph.

Phase 1

```
I.  Generate Subgraphs for all Supertypes:
    1. For every Entity type, S, which is a Supertype in the Schema
          Generate the parse tree for S's Entity declaration using the grammar rules
          in Annex A starting with the Subtype_declaration in the EXPRESS
          Language Reference Manual [4]

          Create a type node for the Entity declaration, which is the root of the tree.

    2. For each terminal node in the parse tree
          Case: AND
                Create an AND constraint node with the same parent and children
          Case: ANDOR
                Create an ANDOR constraint node with the same parent and
                children
          Case: ONEOF
                Remove the node
          Case: SUPERTYPE
                Remove the node
          Case: OF
                Remove the node

    3. For each non-terminal node in the parse tree
          Case: Entity_Ref
                Create a type node for the type whose name is identified by the
                letters, which are the descendants of entity_ref
          Case: Oneof
                Create an ONEOF constraint node with the same parent and
                children
          Case: default (all others)
                Remove the node and connect its children to its parent.
```

4. If any two type nodes are directly connected, replace the edge with two edges and add an ANDOR constraint node connecting the two.

The parse trees have now been converted into subgraphs of the evaluation graph.

II. Union all of the subgraphs with respect to the Entity nodes

III. For each Entity, E, which is a Subtype of a Supertype, S,
   1. If Entity E is not represented in the graph by a type node, T, create one.
   2. Create an ANDOR constraint node as the parent of T
   3. Set the child of the node representing, S, to be the other child of this ANDOR node.
   4. Make the node representing the Supertype, S, the parent of this ANDOR node.

IV. For each Entity, which is neither a Subtype/Supertype
   Create a type node.

V. For each Abstract Supertype node
   Convert all type nodes in the graph, which represent Abstract Supertypes to be Abstract type nodes.

Using the Entity declaration in Example 7-1, the step I.1 of the first phase of the algorithm generates the parse tree in Figure 7-4a.

```
ENTITY item;
   SUPERTYPE OF ONEOF (book_item, journal_item)
END_ENTITY;
```
Example 7-1

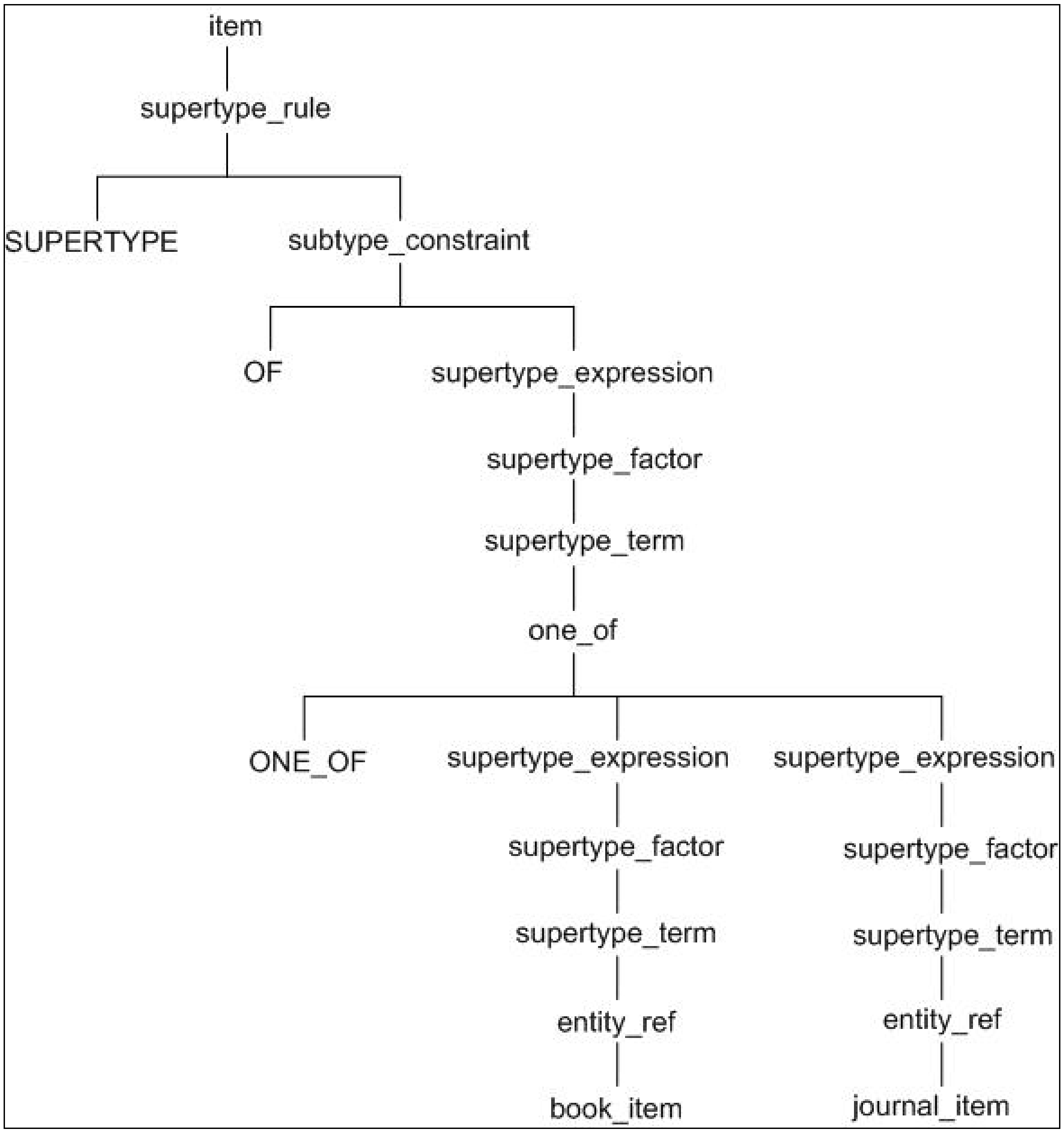


Figure 7-4a – Parse Tree

Applying the cases in step I.2, the algorithm modifies the parse tree as shown in Figure 7-4b. The SUPERTYPE_OF, and ONEOF nodes are removed from the tree.

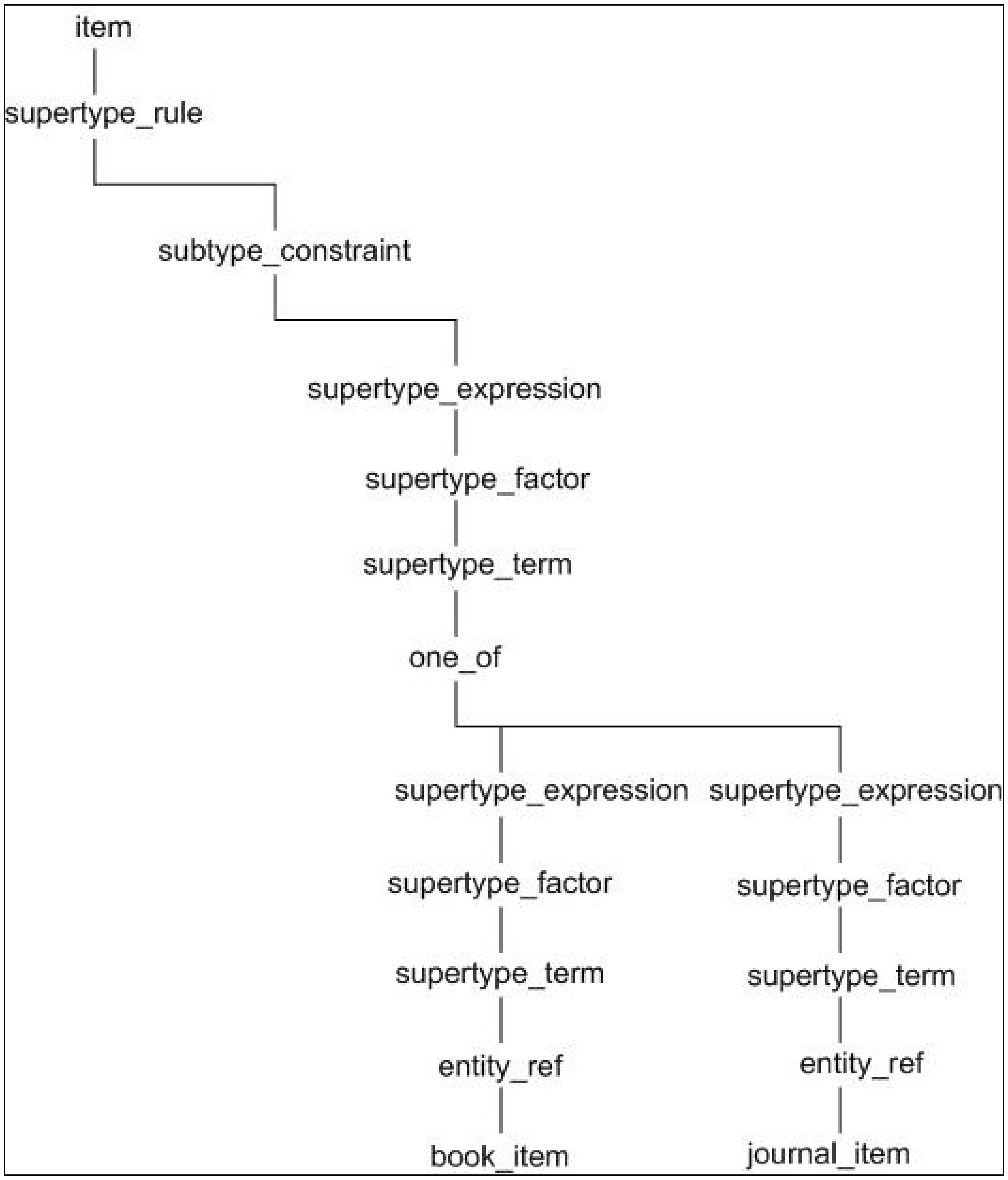


Figure 7-4b – Parse Tree Modified

Applying the cases in step I.3, the algorithm generates the subgraph shown in Figure 7-4c. The supertype_rule, subtype_constraint, supertype_expression, supertype_factor, and supertype_term, nodes are removed from the tree. The entity_ref nodes are converted to type nodes labeled with the names of their direct children. The one_of node is converted to a ONEOF constraint node.

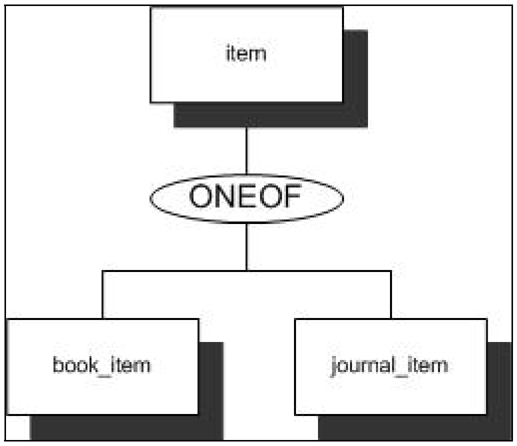


Figure 7-4c – Evaluation Graph

Steps I.4 through V don't apply to this example, so the result of the first phase of the Test Algorithm is Figure 7-4c making the graph the Evaluation graph for this example.
The second phase is to evaluate a complex entity type. The input to the second phase is the Evaluation graph generated in phase 1 and the complex entity instance, T, to be tested. The output is the result of the evaluation, which could be TRUE, FALSE, or ERROR. TRUE is obtained when the type satisfies the Supertype constraints, FALSE is obtained when the type is not part of the current type but does not violate any of the Supertype constraints, and ERROR is obtained when the type violates at least one of the Supertype constraints. The specific evaluation algorithm is:

```
Phase 2
For each disjoint subgraph, evaluate recursively starting at the root node

To evaluate a node, n
    Evaluate all of its children from left to right

Case 1: the node, n, is a leaf type node
    If $n \notin T$ then
      Evaluate to FALSE
    Else
      If all Supertypes of $n \in T$
        If n is not an ABSTRACT type node
          Evaluate to TRUE
        Else
          Evaluate to ERROR
      Else
        Evaluate to ERROR

Case 2: the node is a non-leaf type node
        If n $\notin$ T
      If the child node evaluates to TRUE
        Evaluate to ERROR
      Else if the child node evaluates to FALSE
        Evaluate to FALSE
      Else
        Evaluate to ERROR
        Else /* n $\in$ T */
      If all Supertypes of n $\in$ T
        If n is not an ABSTRACT type node
          If the child node evaluates to TRUE
            Evaluate to TRUE
          Else if child evaluates to FALSE
            Evaluate to TRUE
          Else
            Evaluate to ERROR
        Else /* n is ABSTRACT */
          If the child node evaluates to TRUE
            Evaluate to TRUE
          Else if the child node evaluates to FALSE
            Evaluate to ERROR
          Else
            Evaluate to ERROR
    Else /* all Supertypes of n$\notin$ T */
      Evaluate to ERROR

Case 3: the AND Constraint node
    If all the child nodes evaluate to TRUE
      Evaluate to TRUE
    Else if any child node evaluates to ERROR
      Evaluate to ERROR
    Else
      Evaluate to FALSE

Case 4: the ONEOF constraint node
    If all the child nodes are FALSE
      Evaluate to FALSE
    Else if only one child is TRUE and there are no ERROR evaluations
      Evaluate to TRUE
```

```
Else /* if more than one TRUE or ERRORs */
  Evaluate to ERROR
```

Using the result of phase 1 in Figure 7-4c and evaluating complex entity instance type, T={*book_item&item*}, the following is the result of phase 2 of the Test Algorithm. Evaluating all of the children of the root node, in this case *item*, we start with the leaf node represented by *book_item*. Case 1 evaluates this node as TRUE because the node is part of the T complex entity instance typeT. The next node evaluated is *journal_item*, which is evaluated to FALSE in case 1 because *journal_item* is not part of the complex entity instance type T. The next node evaluated is the ONEOF constraint node, which will be evaluated by using Case 4 of the algorithm. This node is evaluated to TRUE because only one of its children has evaluated to TRUE. The final node to be evaluated is the *item* type node, which will be evaluated by using Case 2. The child node of *item* evaluated to TRUE so this node will evaluate to TRUE. Therefore, the complex entity instance type T tested using the evaluation graph in Figure 7-4c is a valid instance.

In its present form, the Test Algorithm does not support the Total Over constraint. This project extends the evaluation graph and algorithm to include Total Over constraint nodes and a symbol for a collection point called ANDCP. The collection point is where the constraints are combined to perform a final evaluation of several constraints. The ANDCP collection point node evaluates the same way as the AND constraint node but is used to combine two constraints together. This behaviour is needed because the Total Over definition says that the Entity combination has to contain at least one of the elements in the Total Over list. If there are more than one Total Over constraints then all Total Over constraints need to be satisfied before the Entity combination is considered valid. All Total Over constraints that are defined for the same Supertype are combined into an ANDCP collection point node which is then combined using another ANDCP collection point node with the other constraints for the same Supertype. In the evaluation graph, the ANDCP collection point node is an oval shape to show that it is a constraint node. The formal definition of this revised algorithm is:

```
Phase 1 - Revised
I.  For each Supertype, S, in the Schema
```

1. Build a Subtype Constraint for Entity, S, for all Subtypes E1..Ej not in any other Subtype Constraint of S of the form

```
SUBTYPE_CONSTRAINT name FOR S;
  E1 ANDOR E1 ANDOR… ANDOR Ej;
END_SUBTYPE_CONSTRAINT;
```

```
2. For each Subtype Constraint, SCi ..SCn of S
   2.1 Generate the parse tree using the grammar rules in Annex A starting with
       the Subtype_declaration in the EXPRESS-3 Language Reference Manual
       [4]

       Replace the Subtype_Constraint_declaration non-terminal node with a
       type node representing S.

   2.2 For each terminal node in the parse tree
       Case: AND
               Create an AND constraint node with the same parent and children

       Case: ANDOR
               Create an ANDOR constraint node with the same parent and
               children

       Case: ONEOF
               Remove the node

       Case: SUPERTYPE
               Remove the node

       Case: OF
               Remove the node

       Case: TOTAL_OVER
               Remove the node

       Case: END_SUBTYPE_CONSTRAINT
               Remove the node

   2.3 For each non-terminal node in the parse tree
       Case: Entity_Ref
               Create a type node for the type whose name is identified by the
               letters, which are the descendants of entity_ref
```

```
          Case: Oneof
                    Create an ONEOF constraint node with the same parent and
                    children

          Case: Total_Over
                    Create a TOTAL_OVER constraint node with the same parent and
                    children

          Case: Abstract
                    Remove the node and it's children

          Case: Subtype_Constraint_Head
                    Remove the node and it's children

          Case: default (all others)
                    Remove the node and connect its children to its parent.

     2.4 If any two type nodes are directly connected, replace the edge with two
          edges and add an ANDOR constraint node connecting the two.

3.  For each Subtype Constraint, SC, for Supertype S
    /* Union Subgraphs together in two phases */
    3.1 For each Subtype Constraint, SCi ..SCn-1 of S {
          /* Merge the branch of the graphs that have Total Over Constraints */
                Check for Total Over Constraints in the subgraph for SCi and SCi+1

                Case: Neither graph has a Total Over Constraint
                          Result is null

                Case: Only SCi has a Total Over Constraint
                          The result is SCi from which we have removed the subtree that is
                          rooted at the ANDOR, ONEOF, or AND constraint node, which is
                          a direct child of the type node S.

                Case: Only SCi+1 has a Total Over Constraint
                          The result is SCi+1 from which we have removed the subtree that
                          is rooted at the ANDOR, ONEOF, or AND constraint node, which
                          is a direct child of the type node S.

                Case: Both SCi and SCi+1 have Total Over Constraints
                          The result is a tree with a root node of S.  The only child of S is an
                          ANDCP constraint node whose first child is the subtree of SCi,
                          which is rooted at the TOTAL_OVER constraint node.  The
                ANDCP node's second child is the subtree of SCi+1, which is
                rooted at the TOTAL_OVER constraint node.

            Consider Result as SCi+1 graph for the next iteration
    } /* End For*/

3.2 For each Subtype Constraint, SCi ..SCn-1 of S {
    /* Merge the branch of the graphs that have Subtype Expression Constraints */
            Check for Subtype Expression constraints in the subgraph for SCi and
            SCi+1

            Case: Neither graph has a Subtype Expression Constraint
                Result is null

            Case: Only SCi has a Subtype Expression Constraint
                The result is SCi from which we have removed the subtree that is
                rooted at the TOTAL_OVER constraint node, which is a direct
                child of the type node S.

            Case: Only SCi+1 has a Subtype Expression Constraint
                The result is SCi+1 from which we have removed the subtree that
                is rooted at the TOTAL_OVER constraint node, which is a direct
                child of the type node S.

            Case: Both SCi and SCi+1 have Subtype Expression Constraints
                The result is a tree with a root node of S.  The only child of S is an
                ANDCP constraint node whose first child is the subtree of SCi,
                which is rooted at the ANDOR, ONEOF, or AND constraint node.
                The ANDCP node's second child is the subtree of SCi+1, which is
                rooted at the ANDOR, ONEOF, or AND constraint node.

            Consider Result as SCi+1 graph for the next iteration
    } /* End For*/
3.3 For each Supertype, S,
    /* Merge the subgraphs from steps 3.1 and 3.2 adding an ANDCP constraint node
    if necessary. */

    If both steps 3.1 and 3.2, generate graphs, then do the following:
            Create an ANDCP node whose parent is the type node for S.
            The first child of the ANDCP node is the subtree generated from step 3.1
            and the second child is the subtree generated from step 3.2 when the root
            nodes are removed.
```

The parse tree has now been converted into a subgraph of the evaluation graph.

```
II. Union all of the subgraphs with respect to the Entity nodes

IV. For each Entity which is neither a Subtype/Supertype
        Create a type node.

V. For each Abstract Supertype node
                Convert all type nodes in the graph, which represent Abstract Supertypes
                to be Abstract type node.
```

Given the revised algorithm above, an example of its use is now given using the EXPRESS in Example 7-2. Step I.1.1 of the first phase of the algorithm generates the parse trees shown in Figure 7-5a

```
SUBTYPE_CONSTRAINT item_types_1 FOR item;
        ONE OF (book_item, journal_item);
        TOTAL_OVER (book_item, journal_item);
END_SUBTYPE_CONSTRAINT;

SUBTYPE_CONSTRAINT item_types_2 FOR item;
        TOTAL_OVER (circulating_item, noncirculating_item);
END_SUBTYPE_CONSTRAINT;
```

Example 7-2

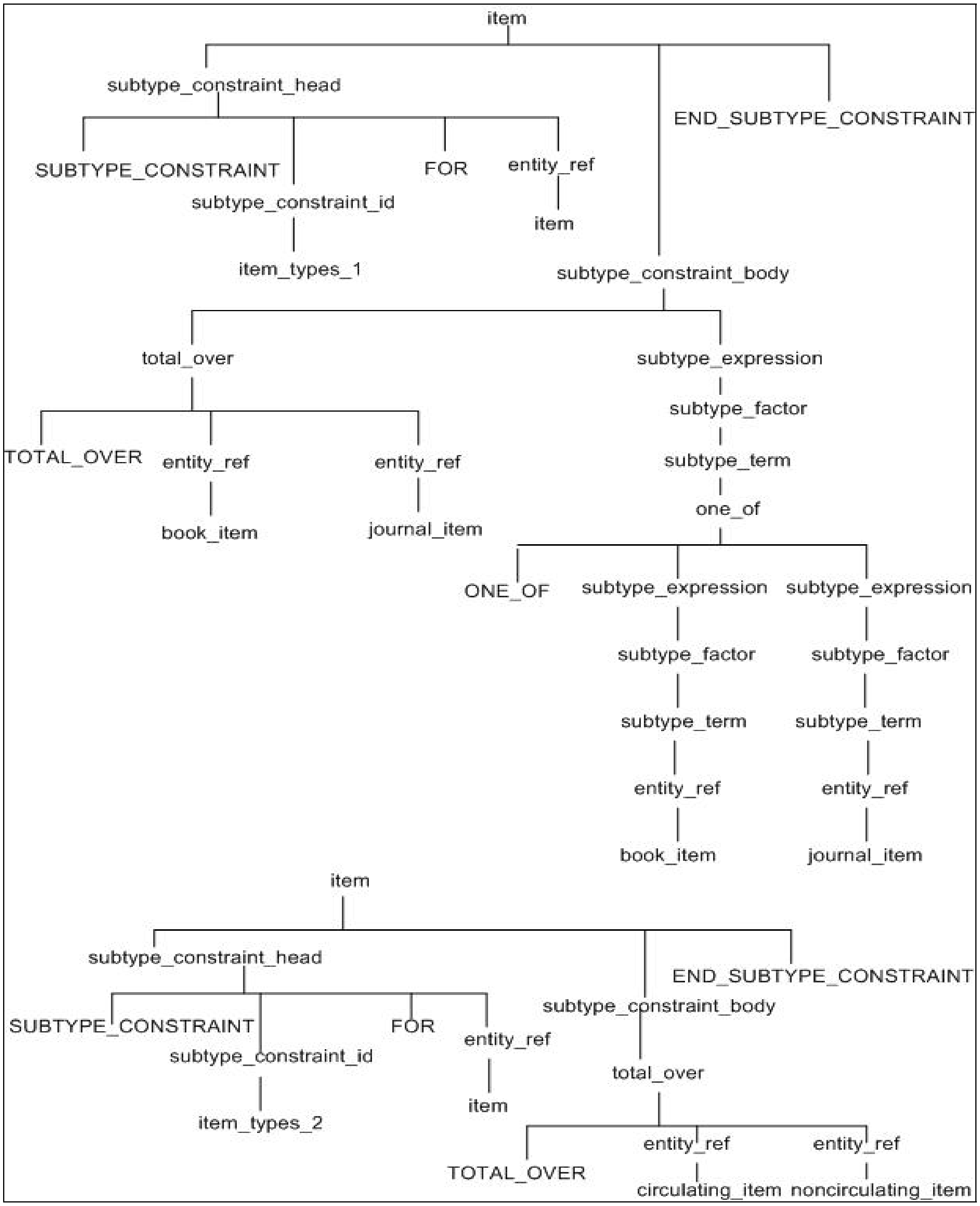


Figure 7-5a – Example 7-2 – Parse Tree

Applying the cases in step I.1.2, the algorithm modifies the parse trees to those shown in Figure 7-5b. The ONE OF, TOTAL_OVER, and END_SUBTYPE_CONSTRAINT nodes are removed from the tree.

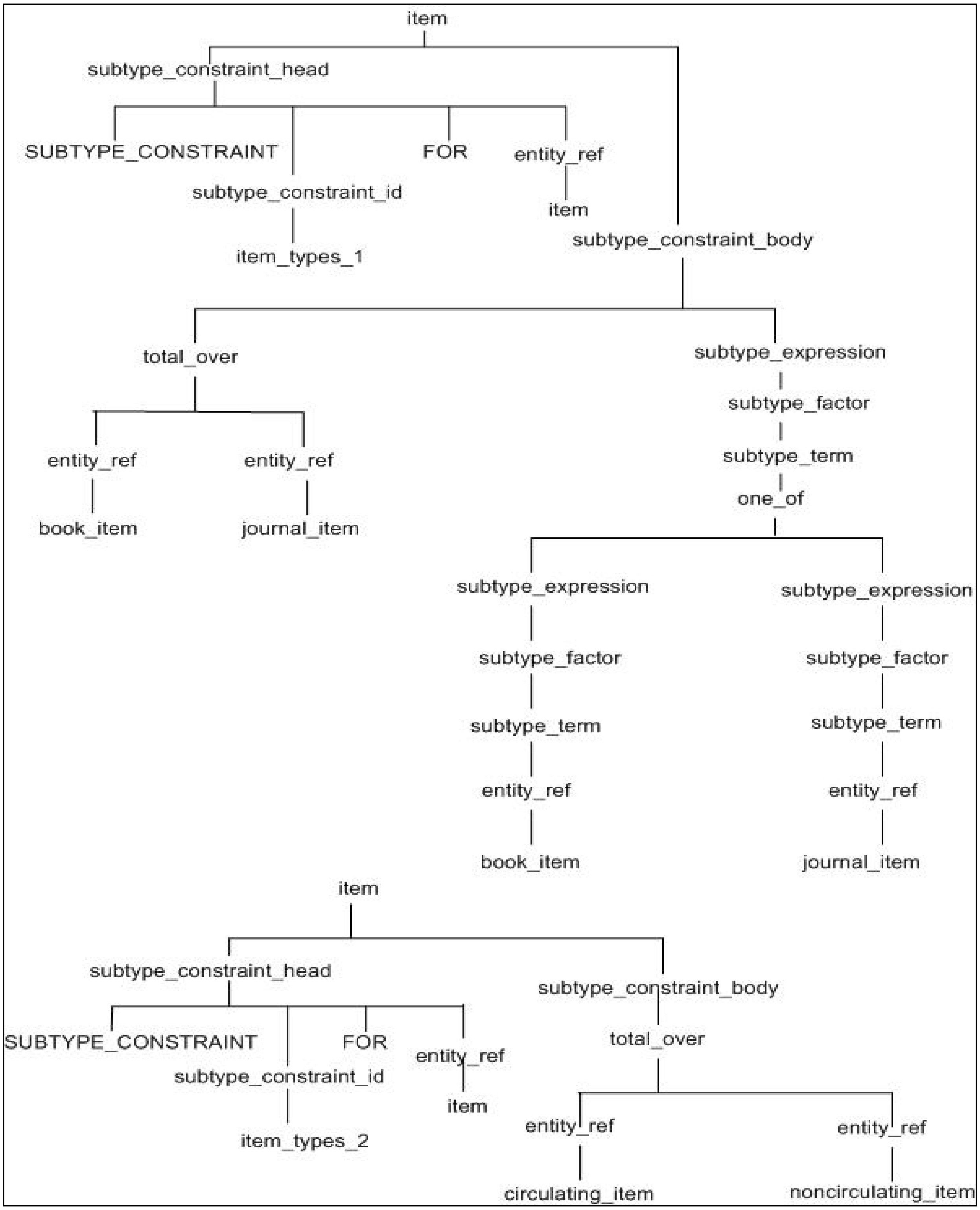


Figure 7-5b – Example 7-2 – Parse Tree After Step 1.2

Applying the cases in step I.1.3, the algorithm generates the subgraph shown in Figure 7-5c. The subtype_constraint_head subtree is removed completely from the parse tree. Also the subtype_constraint_body, subtype_expression, subtype_factor, subtype_term, nodes are removed from the tree each time making its children the children of its parent. The entity_ref nodes are converted to type nodes labeled with the name of their direct child. The one_of node is converted to a ONEOF constraint node and the total_over node is converted to a TOTAL_OVER constraint node.

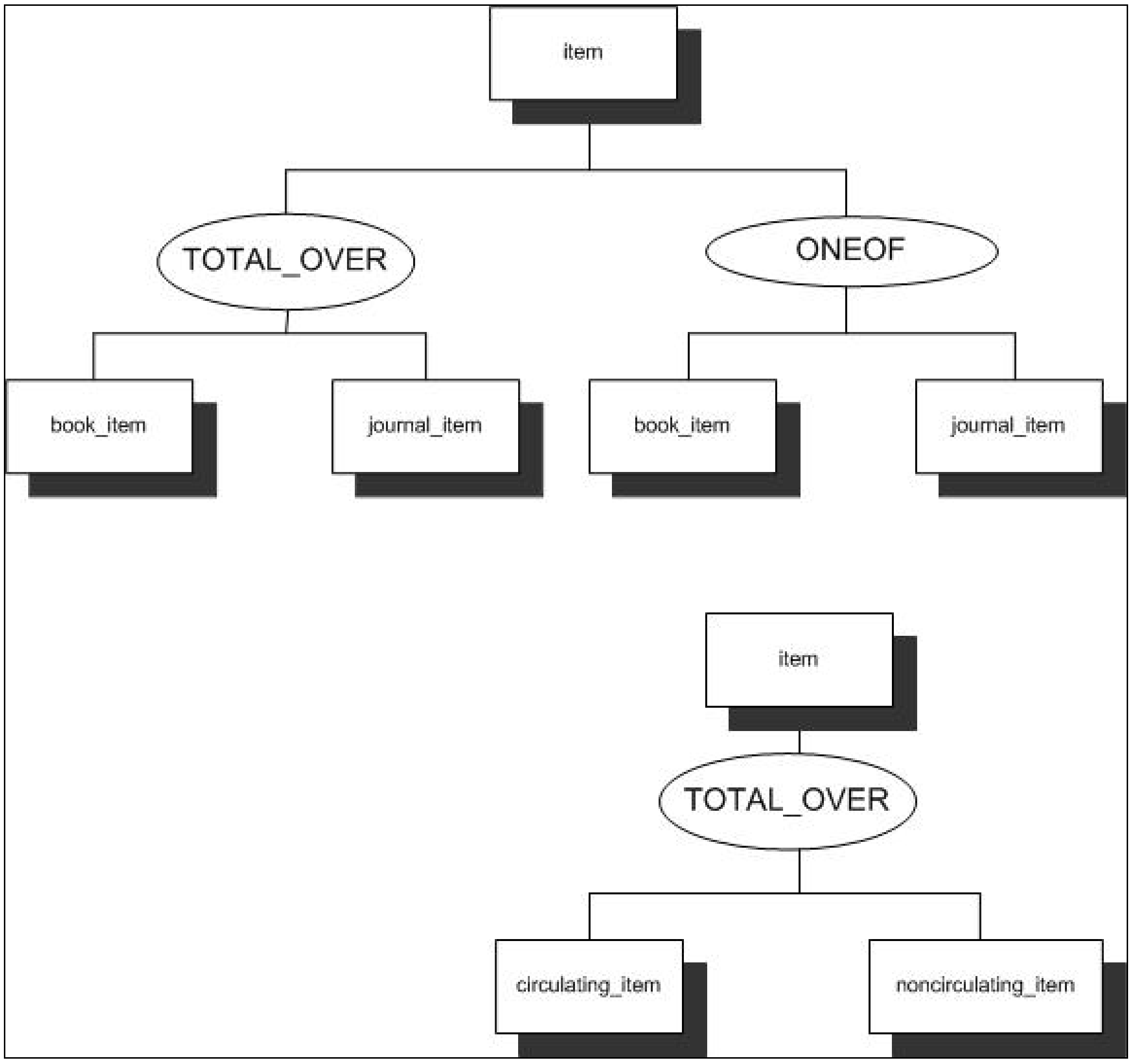


Figure 7-5c – Example 7-2 – Parse Tree After Step 1.3

Applying the cases in step I.2.1, the algorithm generates the subgraph as in Figure 7-5d. The Total Over constraints from the two parse trees are merged together with an ANDCP constraint node.

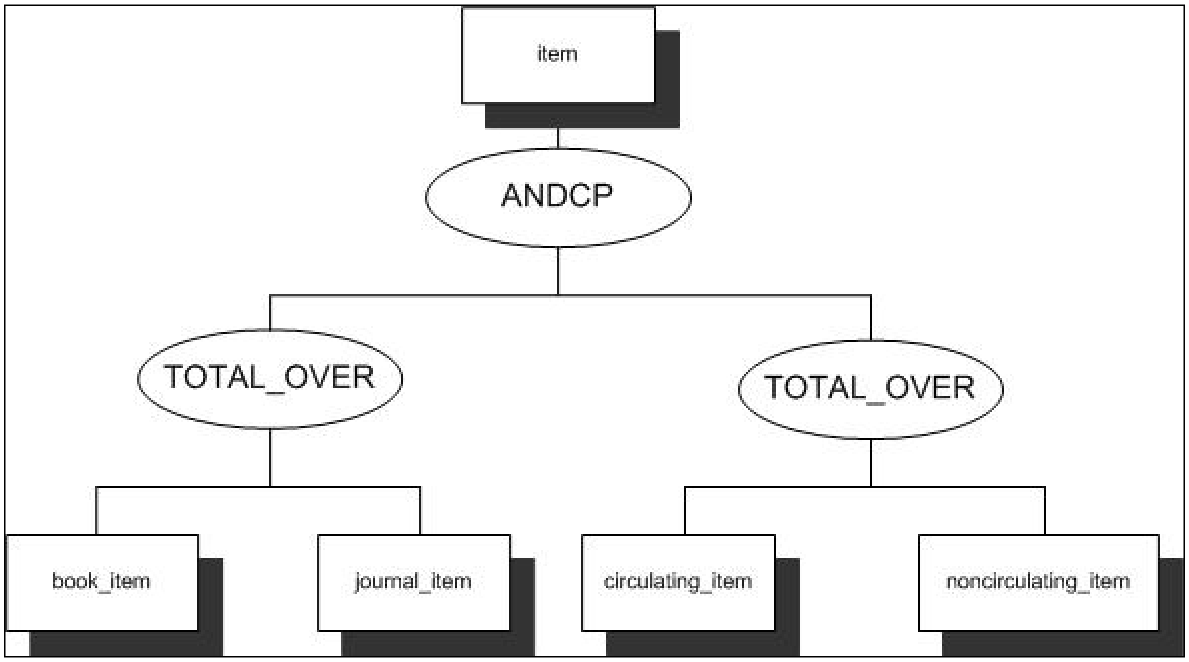


Figure 7-5d – Example 7-2 – Parse Tree After Step 2.1

Applying the cases in step I.2.2, the algorithm generates the subgraph as in Figure 7-5e. The subtype expression constraints from the two parse trees are merged together with an ANDCP constraint node. In this example only one of the trees has a subtype expression constraint.

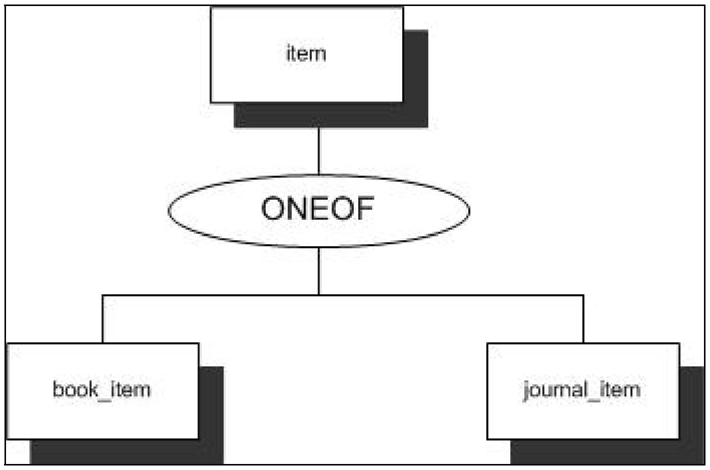


Figure 7-5e – Example 7-2 – Parse Tree After Step 2.2

Applying the cases in step I.2.3, the algorithm merges the subgraph in Figure 7-5d and the subgraph in Figure 7-5e into one subgraph creating an ANCP node to combine the Total Over constraints with the subtype express constraints. Figure 7-5f shows these two subgraphs merged together.

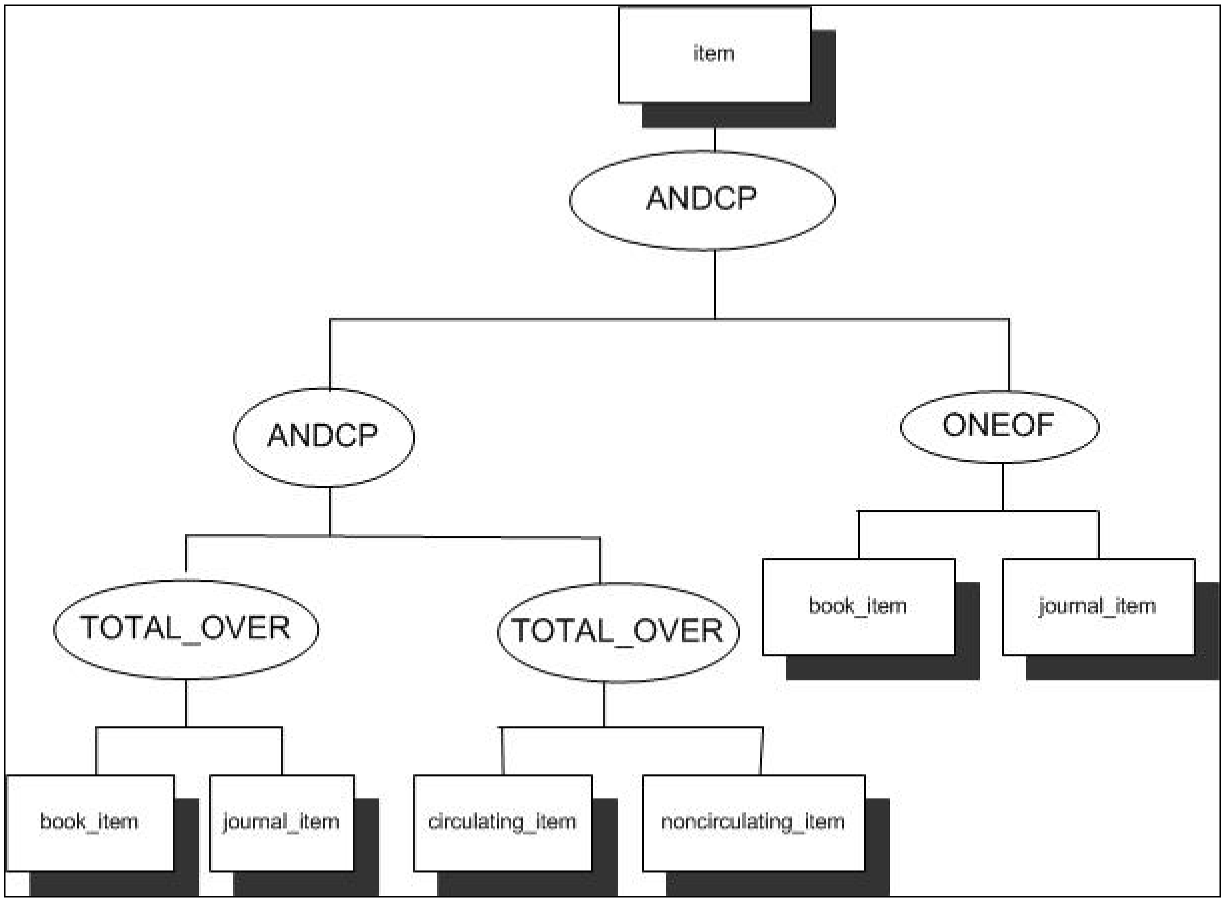


Figure 7-5f – Example 7-2 – Parse Tree After Step 2.3

Applying step II, the algorithm merges the subgraphs with respect to the Entity nodes. This step will remove the multiple type nodes that may be present in a subgraph. Figure 7-5g shows the result of this step.

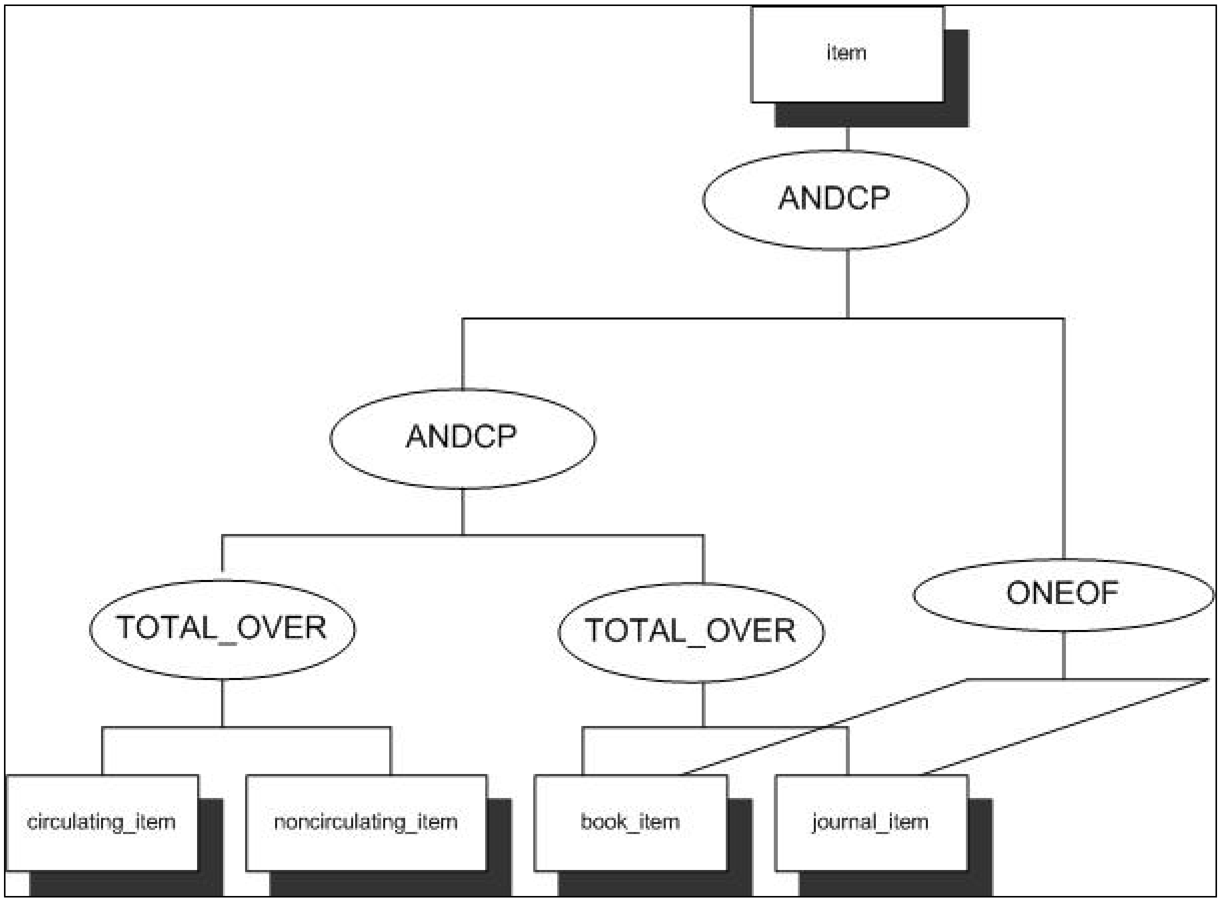


Figure 7-5g – Example 7-2 – Evaluation Graph

Steps III through V don't apply to this example, so the result of the first phase of the Test Algorithm is Figure 7-5g making the graph the evaluation graph for this example.

The second phase is to evaluate a complex entity instance type. The input to the second phase is the Evaluation graph generated in phase 1 and the complex entity instance type T to be tested. The output is the result of the evaluation, which could be TRUE, FALSE, or ERROR. TRUE is obtained when the constructed combination satisfies the Supertype constraints, FALSE is obtained when the combination is not part of the current type but does not violate the Supertype constraints, and ERROR is obtained when the combination violates at least one of the Supertype constraints. Phase 2 has been revised to include the handling of Total Over constraints. The specific evaluation algorithm is:

Phase 2 - Revised
For each disjoint subgraph, evaluate recursively starting at the root node

```
To evaluate a node, n
        Evaluate all of the children from left to right

Case 1: the node, n, is a leaf type node
        If n $\notin$ T then
          Evaluate to FALSE
        Else
          If all Supertypes of n $\in$ T
            If n is not an ABSTRACT type node
              Evaluate to TRUE
            Else
              Evaluate to ERROR
          Else
            Evaluate to ERROR

Case 2: the node is a non-leaf type node
        If n $\notin$ T
          If the child node evaluates to TRUE
            Evaluate to ERROR
          Else if the child node evaluates to FALSE
            Evaluate to FALSE
          Else
            Evaluate to ERROR
        Else /* n $\in$ T */
          If all Supertypes of n $\in$ T
            If n is not an ABSTRACT type node
              If the child node evaluates to TRUE
                Evaluate to TRUE
              Else if child evaluates to FALSE
                Evaluate to TRUE
              Else
                Evaluate to ERROR
            Else /* n is ABSTRACT */
              If the child node evaluates to TRUE
                Evaluate to TRUE
              Else if the child node evaluates to FALSE
                Evaluate to ERROR
              Else
                Evaluate to ERROR
        Else /* all Supertypes of n$\notin$ T */
```

```
            Evaluate to ERROR

Case 3: the AND Constraint node
        If all the children nodes evaluate to TRUE
            Evaluate to TRUE
        Else if any child node evaluates to ERROR
            Evaluate to ERROR
        Else
            Evaluate to FALSE

Case 4: the ONEOF constraint node
        If all the children nodes are FALSE
            Evaluate to FALSE
        Else if only one child is TRUE and there are no ERROR evaluations
            Evaluate to TRUE
        Else /* if more than one TRUE or ERRORs */
            Evaluate to ERROR

Case 5: the TOTAL OVER constraint node
        If at least one of the children nodes is TRUE
            Evaluate to TRUE
        Else /* none of the children are TRUE */
            Evaluate to FALSE

Case 6: the ANDCP constraint node
        If all the children nodes evaluate to TRUE
            Evaluate to TRUE
        Else if any child node evaluates to ERROR
            Evaluate to ERROR
        Else
            Evaluate to FALSE
```

Using the result of phase 1 in Figure 7-5g and evaluating complex entity instance, T={*book_item&item*}, the following is the result of phase 2 of the Test Algorithm. Evaluating all of the children of the root node, in this case *item*, we start with the leaf node represented by *circulating_item*. Case 1 evaluates this node as FASLE because the node is not part of the complex entity instance type T. The next node evaluated is *noncirculating_item*, which is evaluated to FALSE in case 1 because *circulating_item* is not part of the complex entity instance type T. The next node evaluated is the TOTAL_OVER constraint node, which will be evaluated by using Case 5 of the algorithm. This node is evaluated to FALSE because none of its children has evaluated to TRUE. The next node to be evaluated is the *book_item* node, which is a leaf node and is evaluated to TRUE using Case 1 because it is part of the complex entity instance type T. The *journal_item* node is evaluated to FALSE because it is not part of the complex entity instance type T. The TOTAL_OVER constraint node is evaluated next and is evaluated to TRUE using Case 5 because at least one of its children evaluated to TRUE. The next node to be evaluated is the ANDCP node, which is the parent of the Total Over nodes. Using Case 6, this node evaluates to FALSE because not all of its children evaluated to TRUE. This means that the Total Over constraints was not satisfied. The next node to be evaluated is the child node of the ONEOF constraint node. The *book_item* node is evaluated to TRUE using Case 1 because it is part of the T complex entity instance. The next node is the *journal_item*, which evaluates to FALSE because it is not part of complex entity instance type T. The ONEOF node evaluates to TRUE using Case 4. The ANDCP node, which is a direct child of the *item* node evaluates to FALSE using Case 6. One child of the ANCP node evaluated to FALSE and one evaluated to TRUE thus making the ANDCP node FALSE. The final node to be evaluated is the *item* type node, which will be evaluated by using Case 2. The child node of *item* evaluated to FALSE so this node will evaluate to FALSE. Therefore, the complex entity instance tested using the evaluation graph in Figure 7-5g is an invalid instance.

This chapter has presented a formal algorithm for the Test Algorithm and extended it to include support for the Total Over constraint. This gives two algorithms, both of which process all the inheritance constraints in EXPRESS edition 3. What remains to do is to test these algorithms for correctness, analyze their strengths and weaknesses, and prove their equivalence, which is addressed in the following chapters.

CHAPTER 8

COMPARISON OF ALGORITHMS

Both the Generate [4] and Test [7] Algorithms have similarities, advantages, and disadvantages, which will be considered in this chapter. One similarity between the two algorithms is that they both have two phases. The Generate Algorithm's first phase generates a set of complex entity instances and the second phase searches for a specific instance in that set. The first phase of the Test Algorithm builds an evaluation graph and the second phase traverses the graph for each instance to be tested. Another similarity between the two algorithms is that the first phase is performed only once for each inheritance hierarchy while the second phase for each algorithm is done for every instance being checked or tested. The first phase of the Generate Algorithm is more complex than the first phase of the Test Algorithm. The second phase of the Test Algorithm is more complex than the second phase of the Generate Algorithm. The main and more complex processing for the Generate Algorithm takes place in the first phase when the result set is generated. The main and more complex processing for the Test Algorithm takes place in the second phase when the nodes in the evaluation graph are being tested.
A major advantage of the Generate Algorithm is that the set of all legal combinations is available to the developer for a given inheritance hierarchy. It only takes one-iteration of the first phase of the algorithm to generate this set. After the first phase is complete, each comparison of complex entity datatypes with those in the set is done quickly by searching just the result set for the presence of the datatype. This is very helpful when there is a large subset of complex entity datatypes that need to be evaluated.

An advantage of the Test Algorithm over the Generate Algorithm is the ability to check only one instance instead of generating all possible combinations. This allows developers to quickly check the relatively small set of combinations that are important to the model they are working on and not worry about the much larger set of legal, but probably meaningless combinations. Also, the algorithm provides a quick result because only one iteration is needed to test a combination. A disadvantage of the Test Algorithm is the time it takes to test several combinations in the same hierarchy because the relatively complex second phase of the algorithm must be executed for each combination.

However, the main disadvantage to using the Generate Algorithm is the time required to generate all of the combinations. The Generate Algorithm requires many operations to evaluate each Entity type in the hierarchy thus making it very time consuming, and also results in a large complexity measure. Several operations in the algorithm require the creation of sub-sets, which are used to evaluate the resulting set, thus greatly increasing the space required for the algorithm. In addition, maintaining multiple sets requires more time to verify that each set is correct as well as the time needed to add or remove them from the resulting set as required. This is exacerbated by the fact that only a few of the legal combinations are of any practical value, and that a developer is usually only concerned with a small subset of these practical combinations. This drawback is further complicated if the inheritance hierarchy is large thus making the generated set large. The time required in computing the result set increases with both the number of levels and number of Entities in an inheritance hierarchy. Consider example 8.1a below:

```
SUBTYPE_CONSTRAINT item_types FOR item;
        book_item ANDOR journal_item;
END_SUBTYPE_CONSTRAINT;
Example 8-1a
```

The output from the Generate Algorithm for this example is:

{item, book_item&item, journal_item&item, book_item&journal_item&item}

Here three Entities (one Supertype and two Subtypes) have resulted in four elements in the resulting set. Suppose that the Entity, *circulating_item* is added as another Subtype of *item* using the ANDOR constraint as seen in Example 8-1b.

```
SUBTYPE_CONSTRAINT item_types FOR item;
        book_item ANDOR journal_item ANDOR circulating_item;
END_SUBTYPE_CONSTRAINT;
Example 8-1b
```

The output from the Generate Algorithm would be:

R={item, book_item&item, journal_item&item, circulating_item&item, book_item&journal_item&item, book_item&circulating_item&item, journal_item&circulating_item&item, book_item&journal_item&circulating_item&item}

By adding one more Subtype, the size of the result set doubled to 8. Compare this to the performance of the Test Algorithm, using Example 8-1a and testing the combination *book_item&journal_item&item*. The evaluation graph in Figure 8-1a shows the instance combination is valid and it takes 4 steps to evaluate the Entity instance.

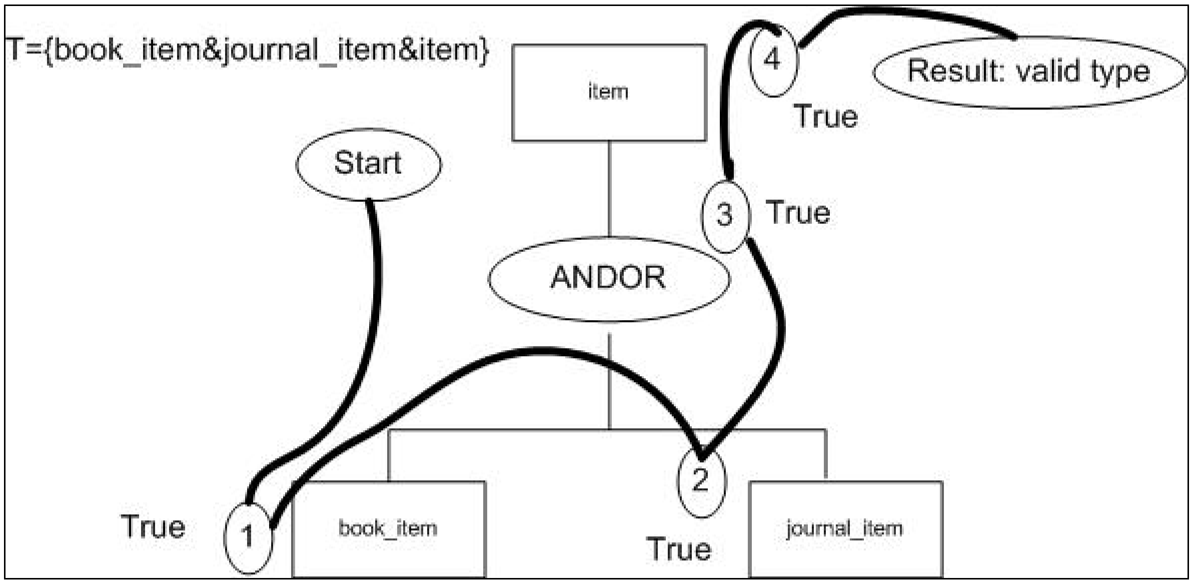


Figure 8-1a – 3 Entity Evaluation Graph

Further if Example 8-1b is used and the test combination is *book_item&journal_item&item*. The evaluation graph in Figure 8-1b shows that the instance combination is valid and it only takes 6 steps to evaluate the Entity instance.

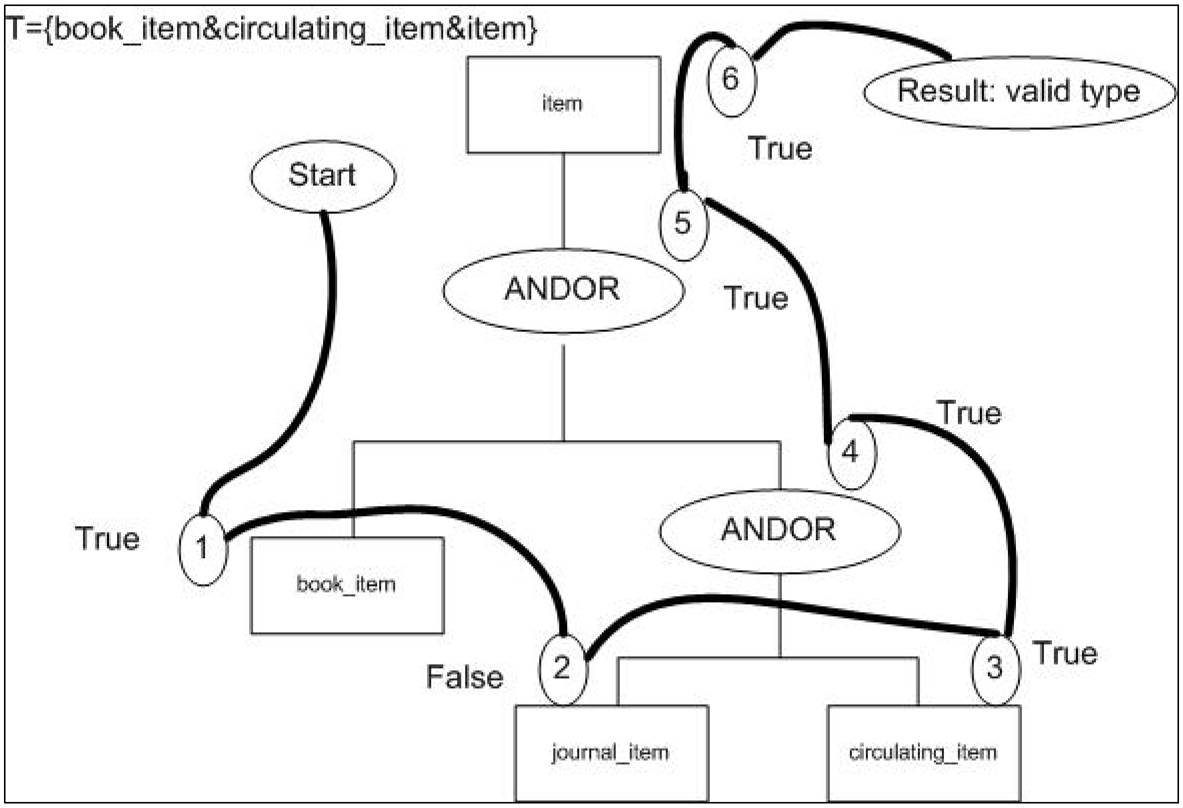


Figure 8-1b – 4 Entity Evaluation Graph

Furthermore, the seven other elements in the set created by the Generate Algorithm are not needed to evaluate this one complex entity instance. Taking this a step farther, adding another Entity to the inheritance hierarchy would cause the Generate Algorithm to generate a set of 16 items, while it would add only two more steps to the Test Algorithm. By adding more Entities to the inheritance hierarchy, the Generate Algorithm will require more work to use than the Test Algorithm.

Determining which algorithm a developer should use depends on the situation. If the developer needs to check several instances at once, the Generate Algorithm would be the preferred algorithm. Once the set is generated during the first phase of the algorithm, the developer can use a search method as many times as needed to look for each instance in the set. If only a few instances are to be checked, then the efficiency of the second phase is more than offset by the complexity of the first phase. Thus if the developer only wants to check a few instances, the Test Algorithm would be the preferred algorithm. This algorithm generates the evaluation graph in the first phase, which is performed only once for each inheritance hierarchy. The second phase of the algorithm is repeated for each instance to be tested. This can be very time consuming if the number of instances to be tested is large and the evaluation graph is large and complex. To further analyze and compare the algorithms, the next chapter will show that the Generate and Test algorithms are truly comparable by proving that they both produce the same results for any tested partial complex entity instance type.

CHAPTER 9

ALGORITHM TESTING

Testing is required to ensure that algorithms give correct results in a timely fashion and are equivalent. It will be shown in this thesis that if the Test and Generate algorithms were equivalent for EXPRESS Edition 1, then they are equivalent for EXPRESS Edition 3. The equivalence of the algorithms for Edition 1 has not been formally proven, but there is a large body of empirical evidence that demonstrates this. The proof method will be based on the structure of the Test Algorithm. Because the only new type introduced was the Total Over (the ANDCP being a specialized use of the existing AND node), only the Total Over needs be considered. The structure of the evaluation graph also simplifies the proof. Because a node is evaluated to TRUE, FALSE, or ERROR before its parent is evaluated the performance of a node is unaffected by the types of nodes that are its children. Thus all that needs to be evaluated for the proof is the different ways in which a Total Over constraint can be built, and the possibility of the algorithms 'colliding' on multiple Total Over constraints. To show that the modifications to the Generate Algorithm and additions to the Test Algorithm are correct they will be compared using the following test cases:

1. An inheritance hierarchy with One Total Over
2. An inheritance hierarchy with More than one Total Over on the same level
3. An inheritance hierarchy with Multiple Total Overs on multiple levels
4. An inheritance hierarchy with Total Overs with elements on multiple levels
5. An inheritance hierarchy with Total Over that contains a Multiple Inheritance Subtype

For each test case the original Generate Algorithm will be used to build the set of legal entity combinations, L. To show the correctness of the modified Generate Algorithm, it will be used to build the set of legal combinations, R. Then R and L will be compared, if they are identical the modified Generate Algorithm will be considered correct. To test the Test Algorithm, the Evaluation graph will be built for the test case. Then the power set, P, of all possible combinations of Entity types in the test case will be generated. Each instance p in P will be tested using the Evaluation graph. If all instances p that are in L evaluate to TRUE, and all instances p that are not in L evaluate to FALSE or unknown then the Test Algorithm will be considered correct.

The five tests will be sufficient to test the correctness and equivalence of these algorithms. The first test case (One Total Over) is the basis case that proves the algorithm will work with one set of Subtypes in a Total Over constraint. The second test case (More than one Total Over on the same level) shows that multiple (because any group of N Total Overs can be operated on pair wise as N-1 Total Over pairs) declarations can exist on the same level in a schema without interfering with each other. The third test case (Multiple Total Overs on multiple levels) proves that a developer can constrain multiple Subtypes on several levels. These tests show that the Total Over constraints will evaluate correctly although they are on different levels. Again, because any N Total Over, constraints can be considered as N-1 pairs of constraints this proves that multiple constraints on different levels will be correctly evaluated.

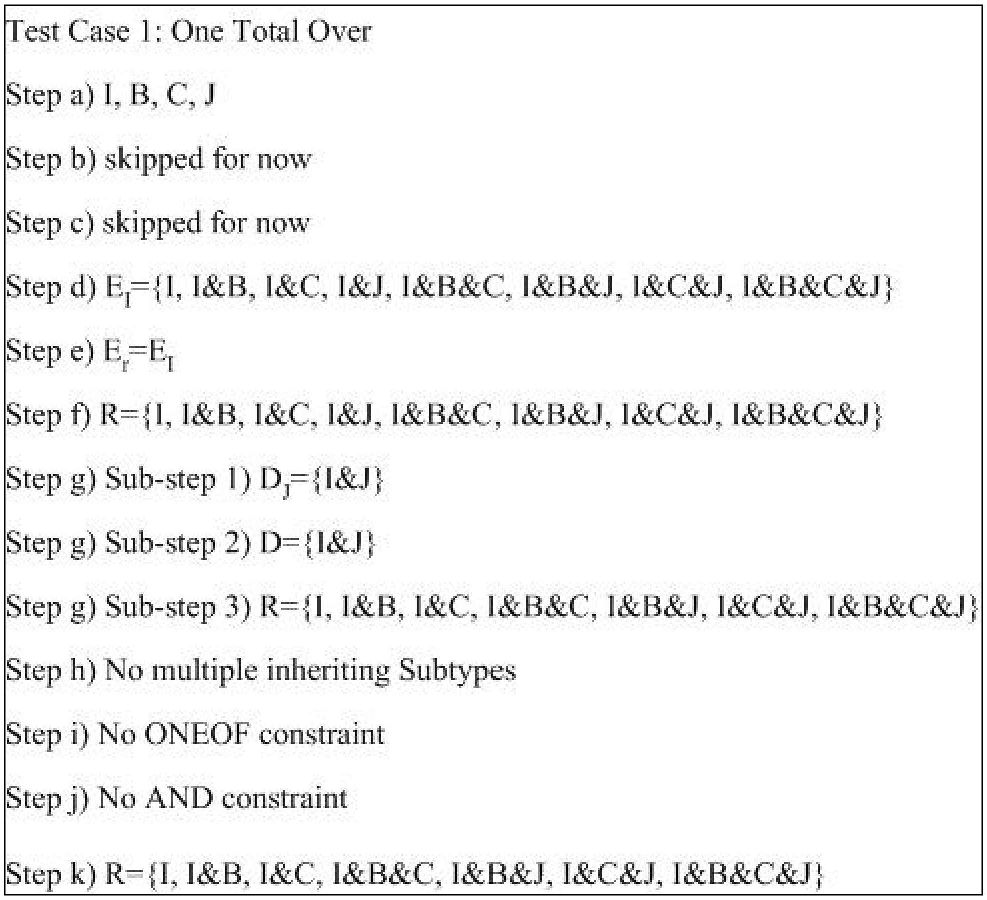
The fourth test case (Total Overs with elements on multiple levels) is similar to the third case. They show that the Entity types constrained by the Total Over need not be on the same level of the Inheritance Hierarchy. The fifth test case (Total Overs that contain a Multiple Inheritance Subtype) demonstrates that a Total Over constraint containing an Entity type that inherits from multiple Supertypes is handled correctly.

Taken together the first test shows the correct handling of a single Total Over with all constrained types on one level. The next two cases show support for multiple Total Overs in the schema either on the same or different levels of the Inheritance Hierarchy. The fourth case demonstrates the correct handling of Total Over constraints regardless of where its constituents appear in the hierarchy, and the final case shows that multiple inheritance in a Total Over constraint is also treated correctly. Because all cases have been considered, if they are proved correct then the algorithms can be considered correct. In the sections below, examples of the proof for each test case will be shown.

The first test case is illustrated by the EXPRESS code in Example 9-1, which contains one Total Over case. This example also shows the short-form names that will be used throughout the text.

```
SUBTYPE_CONSTRAINT item_types FOR item;
        book_item ANDOR journal_item ANDOR circulating_item;
        TOTAL_OVER (book_item, circulating_item);
END_SUBTYPE_CONSTRAINT;

I=item
B=book_item
C=circulating_item
J=journal_item
Example 9-1
```

The original Generate Algorithm generates the set L = {I, I&B, I&C, I&B&C, I&B&J, I&C&J, I&B&C&J}. The steps taken by the new Generate Algorithm are shown in Figure 9-1a.


Test Case 1: One Total Over

Step a) I, B, C, J

Step b) skipped for now

Step c) skipped for now

Step d) E_I={I, I&B, I&C, I&J, I&B&C, I&B&J, I&C&J, I&B&C&J}

Step e) E_r=E_I

Step f) R={I, I&B, I&C, I&J, I&B&C, I&B&J, I&C&J, I&B&C&J}

Step g) Sub-step 1) D_J={I&J}

Step g) Sub-step 2) D={I&J}

Step g) Sub-step 3) R={I, I&B, I&C, I&B&C, I&B&J, I&C&J, I&B&C&J}

Step h) No multiple inheriting Subtypes

Step i) No ONEOF constraint

Step j) No AND constraint

Step k) R={I, I&B, I&C, I&B&C, I&B&J, I&C&J, I&B&C&J}

Figure 9-1a – First Test Case – Generate Algorithm

The final result set R shown in step k in Figure 9-1a is identical to set L above showing the correctness of the Generate Algorithm in this case.

Using the Test Algorithm and Example 9-1, the set of Entity instances to be considered are: P = {I, B, J, C, I&B, I&J, I&C, B&J, B&C, J&C, I&B&J, I&B&C, I&J&C, B&J&C, I&B&J&C}. It was found that the items {I, I&B, I&C, I&B&C, I&B&J, I&C&J, I&B&C&J} gave a result of TRUE, while the items {B, J, C, I&J, B&J, B&C, J&C, B&J&C} gave a result of FALSE or Unknown. This behavior proves the correctness of this algorithm in this case. Figure 9-1b is annotated to show the steps taken in checking the item {I&B&C}. Note that I&B&C is found in both sets L and R.

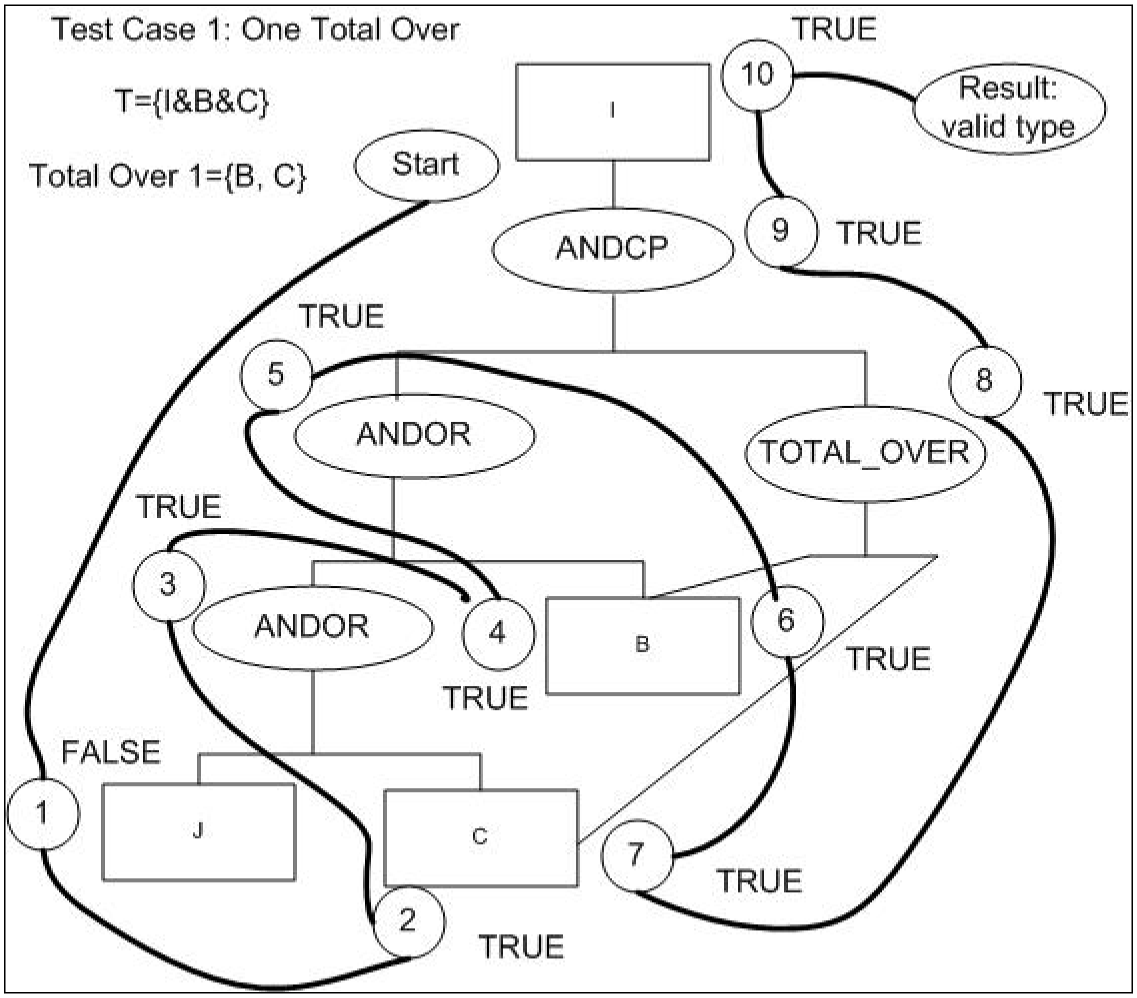


Figure 9-1b – First Test Case – Test Algorithm

The second test case is illustrated by the EXPRESS code in Example 9-2 that contains more than one (specifically two) Total Over constraints on the same level.

```
SUBTYPE_CONSTRAINT item_types_1 FOR item;
        book_item ANDOR journal_item ANDOR circulating_item ANDOR
        media_item ANDOR noncirculating_item;
        TOTAL_OVER (book_item, circulating_item);
END_SUBTYPE_CONSTRAINT;

SUBTYPE_CONSTRAINT item_types_2 FOR item;
        TOTAL_OVER (media_item, noncirculating_item);
END_SUBTYPE_CONSTRAINT;

I=item
B=book_item
C=circulating_item
J=journal_item
M=media_item
N=noncirculating_item
```

Example 9-2

The original Generate Algorithm generates the set L = {I, I&B&M, I&B&N, I&C&M, I&C&N, I&B&C&M, I&B&C&N, I&C&J&M, I&C&J&N, I&B&C&J&M, I&B&C&J&N, I&C&J&M&N, I&B&C&J&M&N}. Figure 9-2a shows the outcomes for each step of the new Generate Algorithm.

Test Case 2: Multiple Total Overs on the Same Level

Step a) I, B, C, J, M, N
Step b) skipped for now
Step c) skipped for now

Step d) E_I={I, I&B, I&C, I&J, I&M, I&N, I&B&C, I&B&J, I&B&M, I&B&N, I&C&J, I&C&M, I&C&N, I&J&M, I&J&N, I&M&N, I&B&C&J, I&B&C&M, I&B&C&N, I&B&J&M, I&B&J&N, I&B&M&N, I&C&J&M, I&C&J&N, I&C&M&N, I&J&M&N, I&B&C&J&M, I&B&C&J&N, I&B&C&M&N, I&B&J&M&N, I&C&J&M&N, I&B&C&J&M&N}

Step e) E_r=E_I

Step f) R={I, I&B, I&C, I&J, I&M, I&N, I&B&C, I&B&J, I&B&M, I&B&N, I&C&J, I&C&M, I&C&N, I&J&M, I&J&N, I&M&N, I&B&C&J, I&B&C&M, I&B&C&N, I&B&J&M, I&B&J&N, I&B&M&N, I&C&J&M, I&C&J&N, I&C&M&N, I&J&M&N, I&B&C&J&M, I&B&C&J&N, I&B&C&M&N, I&B&J&M&N, I&C&J&M&N, I&B&C&J&M&N}

Step g) Sub-step 1-Total Over 1) D_J={I&J, I&J&M, I&J&N, I&J&M&N}
D_M={I&M, I&J&M, I&M&N, I&J&M&N}
D_N={I&N, I&J&N, I&M&N, I&J&M&N}

Step g) Sub-step 2-Total Over 1) D={I&J, I&M, I&N, I&J&M, I&J&N, I&M&N, I&J&M&N}

Step g) Sub-step 3-Total Over 1) R={I, I&B, I&C, I&B&C, I&B&J, I&B&M, I&B&N, I&C&J, I&C&M, I&C&N, I&B&C&J, I&B&C&M, I&B&C&N, I&B&J&M, I&B&J&N, I&B&M&N, I&C&J&M, I&C&J&N, I&C&M&N, I&B&C&J&M, I&B&C&J&N, I&B&C&M&N, I&B&J&M&N, I&C&J&M&N, I&B&C&J&M&N}

Step g) Sub-step 1-Total Over 2) D_B={I&B, I&B&C, I&B&J, I&B&C&J}
D_C={I&C, I&B&C, I&C&J, I&B&C&J}
D_J={I&J, I&B&J, I&C&J, I&B&C&J}

Step g) Sub-step 2-Total Over 2) D={I&B, I&C, I&J, I&B&C, I&B&J, I&C&J, I&B&C&J}

Step g) Sub-step 3-Total Over 2) R={I, I&B&M, I&B&N, I&C&M, I&C&N, I&B&C&M, I&B&C&N, I&B&J&M, I&B&J&N, I&B&M&N, I&C&J&M, I&C&J&N, I&C&M&N, I&B&C&J&M, I&B&C&J&N, I&B&C&M&N, I&B&J&M&N, I&C&J&M&N, I&B&C&J&M&N}

Step h) No multiple inheriting Subtypes
Step i) No ONEOF constraint
Step j) No AND constraint

Step k) R={I, I&B&M, I&B&N, I&C&M, I&C&N, I&B&C&M, I&B&C&N, I&B&J&M, I&B&J&N, I&B&M&N, I&C&J&M, I&C&J&N, I&C&M&N, I&B&C&J&M, I&B&C&J&N, I&B&C&M&N, I&B&J&M&N, I&C&J&M&N, I&B&C&J&M&N}

Figure 9-2a – Second Test Case – Generate Algorithm

The final result set R shown in step k in Figure 9-2a is identical to set L above showing the correctness of the Generate Algorithm in this case.

Using the Test Algorithm and Example 9-2, the set of Entity instances to be considered are: P = {I, B, C, J, M, N, I&B, I&C, I&J, I&M, I&N, B&C, B&J, B&M, B&N, C&J, C&M, C&N, J&M, J&N, M&N, I&B&C, I&B&J, I&B&M, I&B&N, I&C&J, I&C&M, I&C&N, I&J&M, I&J&N, I&M&N, B&C&J, B&C&M, B&C&N, B&J&M, B&J&N, B&M&N, C&J&M, C&J&N, C&M&N, J&M&N, I&B&C&J, I&B&C&M, I&B&C&N, I&B&J&M, I&B&J&N, I&B&M&N, I&C&J&M, I&C&J&N, I&C&M&N, I&J&M&N, B&C&J&M, B&C&J&N, B&C&M&N, B&J&M&N, C&J&M&N, I&B&C&J&M, I&B&C&J&N, I&B&C&M&N, I&B&J&M&N, I&C&J&M&N, B&C&J&M&N, I&B&C&J&M&N}.

It was found that the items {I, I&B&M, I&B&N, I&C&M, I&C&N, I&B&C&M, I&B&C&N, I&B&J&M, I&B&J&N, I&B&M&N, I&C&J&M, I&C&J&N, I&C&M&N, I&B&C&J&M, I&B&C&J&N, I&B&C&M&N, I&B&J&M&N, I&C&J&M&N, I&B&C&J&M&N} gave a result of TRUE, while the items {B, C, J, M, N, I&B, I&C, I&J, I&M, I&N, B&C, B&J, B&M, B&N, C&J, C&M, C&N, J&M, J&N, M&N, I&B&C, I&B&J, I&C&J, I&J&M, I&J&N, I&M&N, B&C&J, B&C&M, B&C&N, B&J&M, B&J&N, B&M&N, C&J&M, C&J&N, C&M&N, J&M&N, I&B&C&J, I&J&M&N, B&C&J&M, B&C&J&N, B&C&M&N, B&J&M&N, C&J&M&N, B&C&J&M&N} gave a result of FALSE or Unknown. This behavior proves the correctness of this algorithm in this case.

Figure 9-2b is annotated to show the steps taken in checking the item {I&B&C} using the Test Algorithm and Example 9-2. Note that I&B&C is absent in both sets L and R.

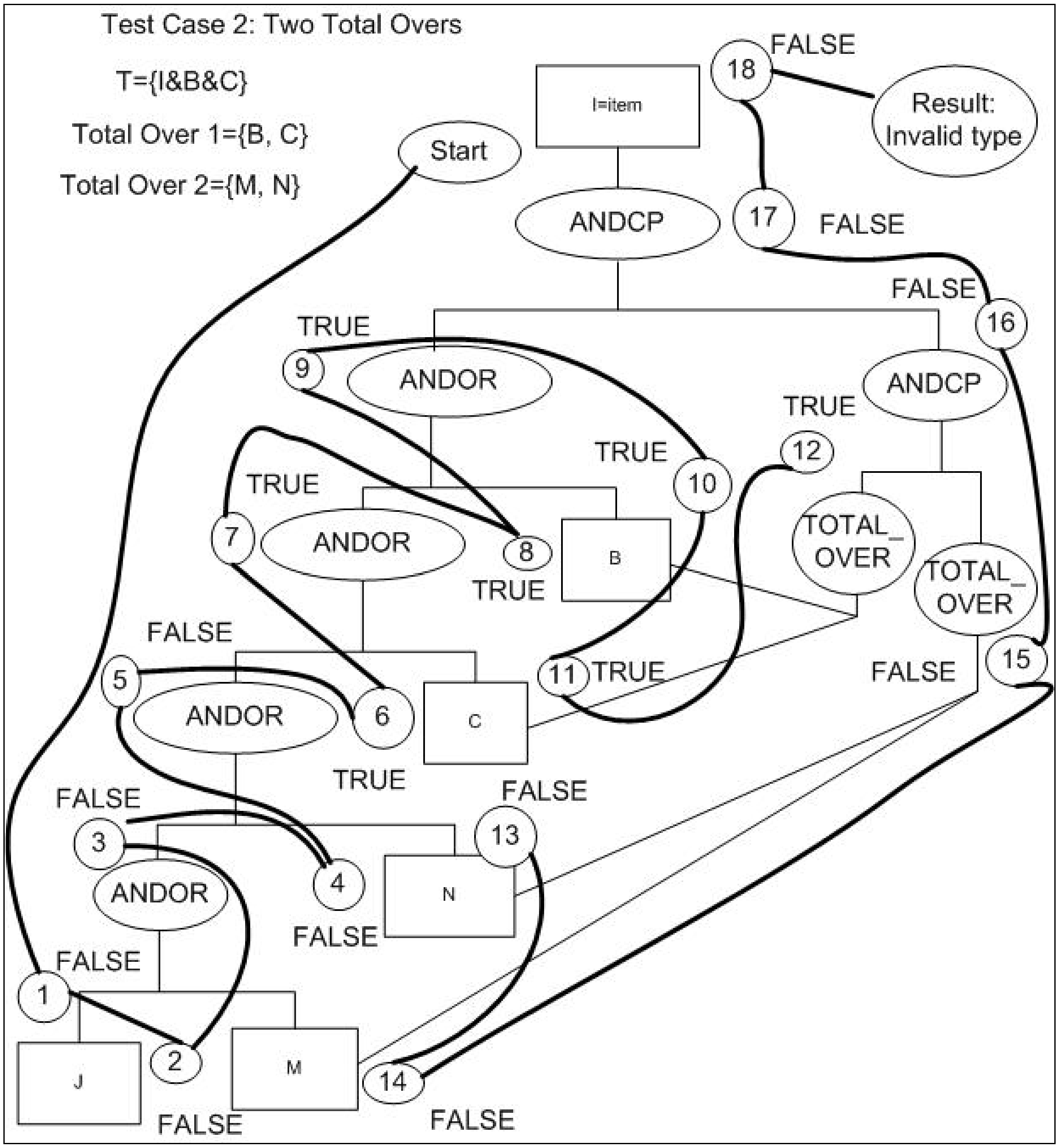


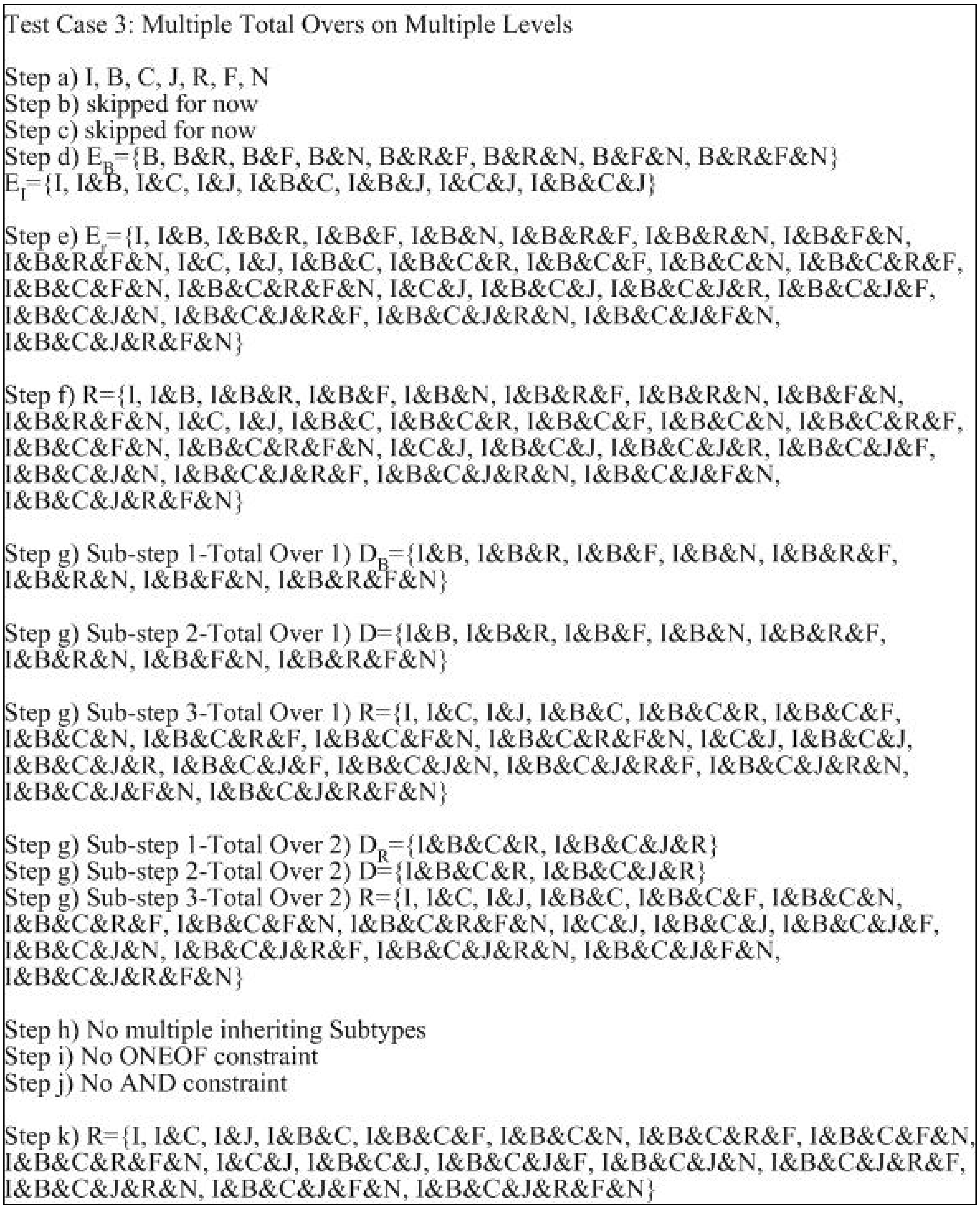
Figure 9-2b – Second Test Case – Test Algorithm

The third test case (Multiple Total Overs on multiple levels) proves that a developer can constrain multiple Subtypes on several levels. This test shows that the developer can add more complexity to the Total Over constraints. These tests show that the Total Over constraints will evaluate correctly although they are on different levels. The third test case is illustrated by the EXPRESS code in Example 9-3, which contains multiple Total Overs on multiple levels.

```
SUBTYPE_CONSTRAINT item_types_1 FOR item;
        book_item ANDOR circulating_item ANDOR journal_item;
        TOTAL_OVER (circulating_item, journal_item);
END_SUBTYPE_CONSTRAINT;

SUBTYPE_CONSTRAINT book_item_types_1 FOR book_item;
        reserve_item ANDOR fiction_item ANDOR noncirculating_item;
        TOTAL_OVER (fiction_item, noncirculating_item);
END_SUBTYPE_CONSTRAINT;

I=item
B=book_item
C=circulating_item
J=journal_item
R=reserve_item
F=fiction_item
N=noncirculating_item
```

Example 9-3

The original Generate Algorithm generates the set L={I, I&C, I&J, I&B&C, I&B&C&F, I&B&C&N, I&B&C&R&F, I&B&C&F&N, I&B&C&R&F&N, I&C&J, I&B&C&J, I&B&C&J&F, I&B&C&J&N, I&B&C&J&R&F, I&B&C&J&R&N, I&B&C&J&F&N, I&B&C&J&R&F&N}. Using Example 9-3, Figure 9-3a shows the results of each step in the new Generate Algorithm.



Test Case 3: Multiple Total Overs on Multiple Levels

Step a) I, B, C, J, R, F, N
Step b) skipped for now
Step c) skipped for now
Step d) E_B={B, B&R, B&F, B&N, B&R&F, B&R&N, B&F&N, B&R&F&N}
E_I={I, I&B, I&C, I&J, I&B&C, I&B&J, I&C&J, I&B&C&J}

Step e) E_I={I, I&B, I&B&R, I&B&F, I&B&N, I&B&R&F, I&B&R&N, I&B&F&N, I&B&R&F&N, I&C, I&J, I&B&C, I&B&C&R, I&B&C&F, I&B&C&N, I&B&C&R&F, I&B&C&F&N, I&B&C&R&F&N, I&C&J, I&B&C&J, I&B&C&J&R, I&B&C&J&F, I&B&C&J&N, I&B&C&J&R&F, I&B&C&J&R&N, I&B&C&J&F&N, I&B&C&J&R&F&N}

Step f) R={I, I&B, I&B&R, I&B&F, I&B&N, I&B&R&F, I&B&R&N, I&B&F&N, I&B&R&F&N, I&C, I&J, I&B&C, I&B&C&R, I&B&C&F, I&B&C&N, I&B&C&R&F, I&B&C&F&N, I&B&C&R&F&N, I&C&J, I&B&C&J, I&B&C&J&R, I&B&C&J&F, I&B&C&J&N, I&B&C&J&R&F, I&B&C&J&R&N, I&B&C&J&F&N, I&B&C&J&R&F&N}

Step g) Sub-step 1-Total Over 1) D_B={I&B, I&B&R, I&B&F, I&B&N, I&B&R&F, I&B&R&N, I&B&F&N, I&B&R&F&N}

Step g) Sub-step 2-Total Over 1) D={I&B, I&B&R, I&B&F, I&B&N, I&B&R&F, I&B&R&N, I&B&F&N, I&B&R&F&N}

Step g) Sub-step 3-Total Over 1) R={I, I&C, I&J, I&B&C, I&B&C&R, I&B&C&F, I&B&C&N, I&B&C&R&F, I&B&C&F&N, I&B&C&R&F&N, I&C&J, I&B&C&J, I&B&C&J&R, I&B&C&J&F, I&B&C&J&N, I&B&C&J&R&F, I&B&C&J&R&N, I&B&C&J&F&N, I&B&C&J&R&F&N}

Step g) Sub-step 1-Total Over 2) D_R={I&B&C&R, I&B&C&J&R}
Step g) Sub-step 2-Total Over 2) D={I&B&C&R, I&B&C&J&R}
Step g) Sub-step 3-Total Over 2) R={I, I&C, I&J, I&B&C, I&B&C&F, I&B&C&N, I&B&C&R&F, I&B&C&F&N, I&B&C&R&F&N, I&C&J, I&B&C&J, I&B&C&J&F, I&B&C&J&N, I&B&C&J&R&F, I&B&C&J&R&N, I&B&C&J&F&N, I&B&C&J&R&F&N}

Step h) No multiple inheriting Subtypes
Step i) No ONEOF constraint
Step j) No AND constraint

Step k) R={I, I&C, I&J, I&B&C, I&B&C&F, I&B&C&N, I&B&C&R&F, I&B&C&F&N, I&B&C&R&F&N, I&C&J, I&B&C&J, I&B&C&J&F, I&B&C&J&N, I&B&C&J&R&F, I&B&C&J&R&N, I&B&C&J&F&N, I&B&C&J&R&F&N}

Figure 9-3a – Third Test Case – Generate Algorithm

The final result set R shown in step k in Figure 9-3a is identical to set L above showing the correctness of the Generate Algorithm in this case.

Using Example 9-3, the Test Algorithm evaluates the multiple Total Overs on multiple levels correctly. The set of Entity instances to be considered are: P={I, B, C, J, R, F, N, I&B, I&C, I&J, I&R, I&F, I&N, B&C, B&J, B&R, B&F, B&N, C&J, C&R, C&F, C&N, J&R, J&F, J&N, R&F, R&N, F&N, I&B&C, I&B&J, I&B&R, I&B&F, I&B&N, I&C&J, I&C&R, I&C&F, I&C&N, I&J&R, I&J&F, I&J&N, I&R&F, I&R&N, I&F&N, B&C&J, B&C&R, B&C&F, B&C&N, B&J&R, B&J&F, B&J&N, B&R&F, B&R&N, B&F&N, C&J&R, C&J&F, C&J&N, C&R&F, C&R&N, C&F&N, J&R&F, J&R&N, J&F&N, R&F&N, I&B&C&J, I&B&C&R, I&B&C&F, I&B&C&N, I&B&J&R, I&B&J&F, I&B&J&N, I&B&R&F, I&B&R&N, I&B&F&N, I&C&J&R, I&C&J&F, I&C&J&N, I&C&R&F, I&C&R&N, I&C&F&N, I&J&R&F, I&J&R&N, I&J&F&N, I&R&F&N, B&C&J&R, B&C&J&F, B&C&J&N, B&C&R&F, B&C&R&N, B&C&F&N, B&J&R&F, B&J&R&N, B&J&F&N, B&R&F&N, C&J&R&F, C&J&R&N, C&J&F&N, C&R&F&N, J&R&F&N, I&B&C&J&R, I&B&C&J&F, I&B&C&J&N, I&B&C&R&F, I&B&C&R&N, I&B&C&F&N, I&B&J&R&F, I&B&J&R&N, I&B&J&F&N, I&B&R&F&N, I&C&J&R&F, I&C&J&R&N, I&C&J&F&N, I&C&R&F&N, I&J&R&F&N, B&C&J&R&F, B&C&J&R&N, B&C&J&F&N, B&C&R&F&N, B&J&R&F&N, C&J&R&F&N, I&B&C&J&R&F, I&B&C&J&R&N, I&B&C&J&F&N, I&B&C&R&F&N, I&B&J&R&F&N, I&C&J&R&F&N, B&C&J&R&F&N, I&B&C&J&R&F&N}.

It was found that items {I, I&C, I&J, I&B&C, I&B&C&F, I&B&C&N, I&B&C&R&F, I&B&C&F&N, I&B&C&R&F&N, I&C&J, I&B&C&J, I&B&C&J&F, I&B&C&J&N, I&B&C&J&R&F, I&B&C&J&R&N, I&B&C&J&F&N, I&B&C&J&R&F&N} gave a result of TRUE, while the items {B, C, J, R, F, N, I&B, I&R, I&F, I&N, B&C, B&J, B&R, B&F, B&N, C&J, C&R, C&F, C&N, J&R, J&F, J&N, R&F, R&N, F&N, I&B&J, I&B&R, I&B&F, I&B&N, I&C&R, I&C&F, I&C&N, I&J&R, I&J&F, I&J&N, I&R&F, I&R&N, I&F&N, B&C&J, B&C&R, B&C&F, B&C&N, B&J&R, B&J&F, B&J&N, B&R&F, B&R&N, B&F&N, C&J&R, C&J&F, C&J&N, C&R&F, C&R&N, C&F&N, J&R&F, J&R&N, J&F&N, R&F&N, I&B&C&R, I&B&J&R, I&B&J&F, I&B&J&N, I&B&R&F, I&B&R&N, I&B&F&N,

I&C&J&R, I&C&J&F, I&C&J&N, I&C&R&F, I&C&R&N, I&C&F&N, I&J&R&F, I&J&R&N, I&J&F&N, I&R&F&N, B&C&J&R, B&C&J&F, B&C&J&N, B&C&R&F, B&C&R&N, B&C&F&N, B&J&R&F, B&J&R&N, B&J&F&N, B&R&F&N, C&J&R&F, C&J&R&N, C&J&F&N, C&R&F&N, J&R&F&N, I&B&C&J&R, I&B&C&R&N, I&B&J&R&F, I&B&J&R&N, I&B&J&F&N, I&B&R&F&N, I&C&J&R&F, I&C&J&R&N, I&C&J&F&N, I&C&R&F&N, I&J&R&F&N, B&C&J&R&F, B&C&J&R&N, B&C&J&F&N, B&C&R&F&N, B&J&R&F&N, C&J&R&F&N, I&B&J&R&F&N, I&C&J&R&F&N, B&C&J&R&F&N} gave a result of FALSE or Unknown. This behavior proves the correctness of this algorithm in this case. Figure 9-3b shows the steps taken in checking the item {I&B&C}, using the Test Algorithm and Example 9-3. Note that doing a scan of the result set R and L, the test case of {I&B&C} is found both sets.

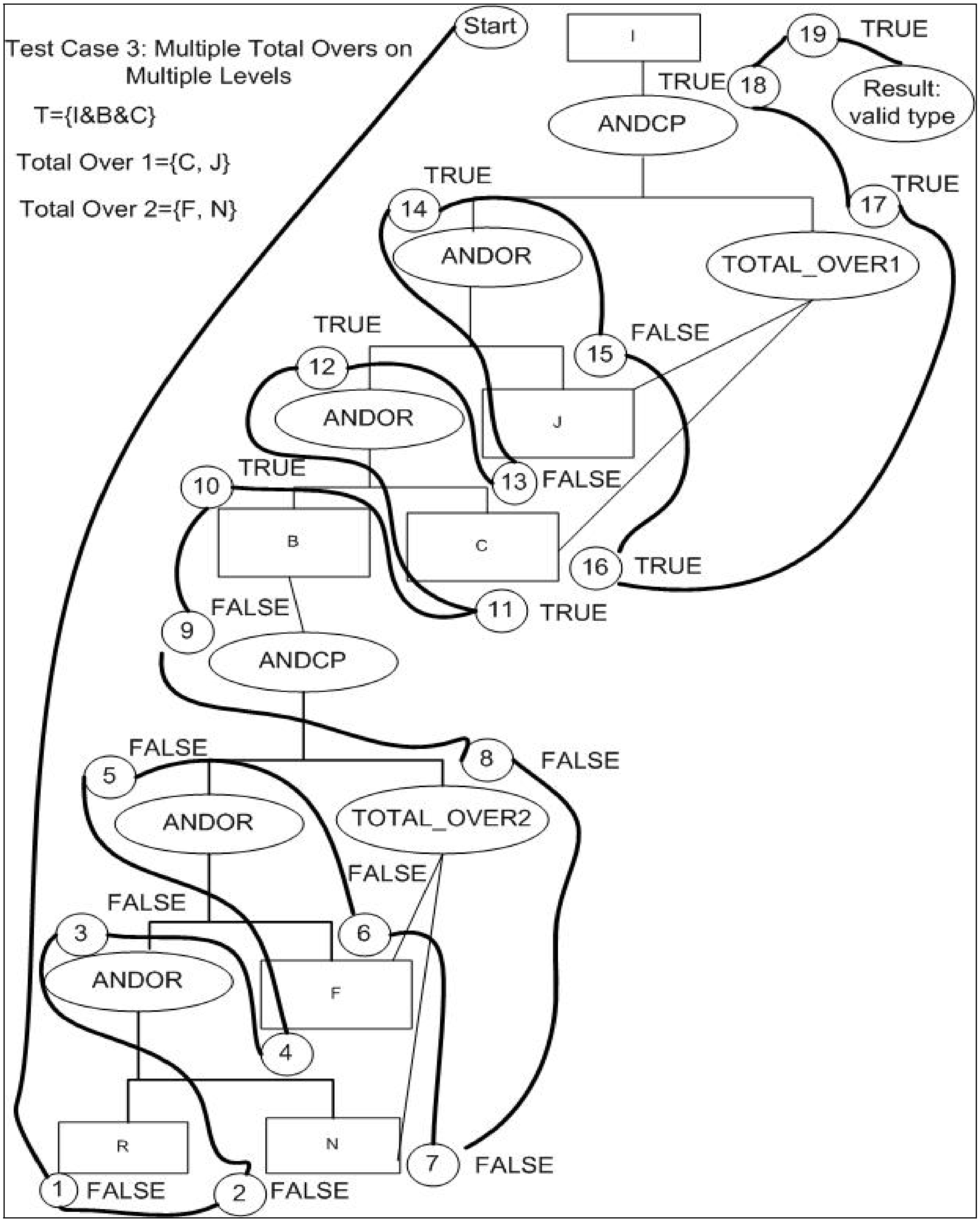


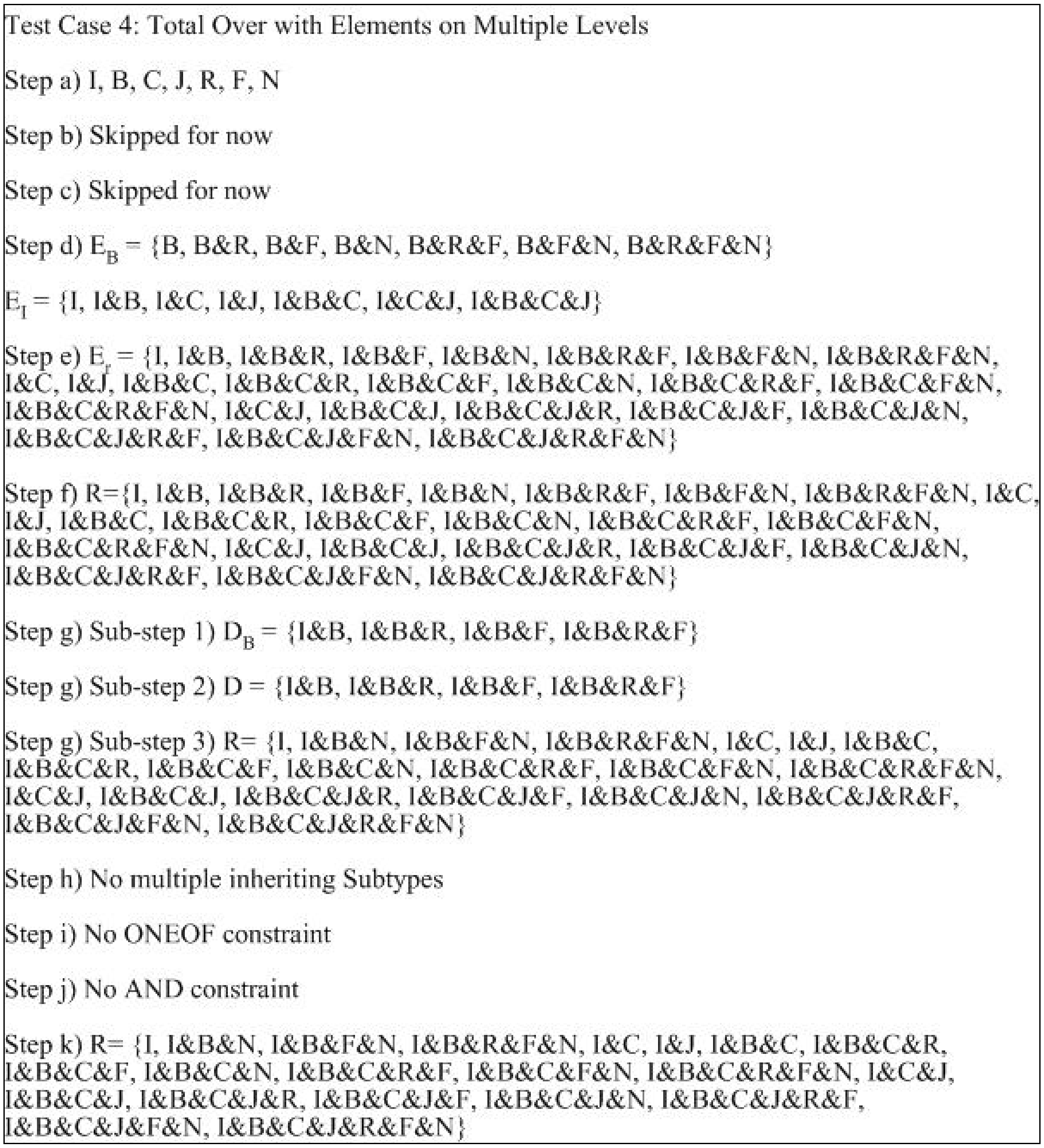
Figure 9-3b – Third Test Case – Test Algorithm

The fourth test case (Total Overs with elements on multiple levels) is useful if there is a need to constrain Subtypes that are on different levels. Because inheritance hierarchies can be multiple levels depending on their complexity, showing that a Total Over constraint can have elements on different levels and have the Total Over evaluate correctly is an important test.

```
SUBTYPE_CONSTRAINT item_types_1 FOR item;
        book_item ANDOR journal_item ANDOR circulating_item
END_SUBTYPE_CONSTRAINT;

SUBTYPE_CONSTRAINT book_item_types_1 FOR book_item;
        reserve_item ANDOR fiction_item ANDOR noncirculating_item;
END_SUBTYPE_CONSTRAINT;

SUBTYPE_CONSTRAINT item_types_2 FOR item;
        TOTAL_OVER (circulating_item, journal_item, reserve_item);
END_SUBTYPE_CONSTRAINT;

I=item
B=book_item
C=circulating_item
J=journal_item
R=reserve_item
F=fiction_item
N=noncirculating_item
```

Example 9-4

The original Generate Algorithm generates the set L={I, IBN, IBFN, IBRFN, IC, IJ, IBC, IBCR, IBCF, IBCN, IBCRF, IBCFN, IBCRFN, ICJ, IBCJ, IBCJR, IBCJF, IBCJN, IBCJRF, IBCJFN, IBCJRFN}. Using Example 9-4, Figure 9-4a shows the results of each step in the Generate Algorithm.


Test Case 4: Total Over with Elements on Multiple Levels

Step a) I, B, C, J, R, F, N

Step b) Skipped for now

Step c) Skipped for now

Step d) E_B = {B, B&R, B&F, B&N, B&R&F, B&F&N, B&R&F&N}

E_I = {I, I&B, I&C, I&J, I&B&C, I&C&J, I&B&C&J}

Step e) E_r = {I, I&B, I&B&R, I&B&F, I&B&N, I&B&R&F, I&B&F&N, I&B&R&F&N, I&C, I&J, I&B&C, I&B&C&R, I&B&C&F, I&B&C&N, I&B&C&R&F, I&B&C&F&N, I&B&C&R&F&N, I&C&J, I&B&C&J, I&B&C&J&R, I&B&C&J&F, I&B&C&J&N, I&B&C&J&R&F, I&B&C&J&F&N, I&B&C&J&R&F&N}

Step f) R={I, I&B, I&B&R, I&B&F, I&B&N, I&B&R&F, I&B&F&N, I&B&R&F&N, I&C, I&J, I&B&C, I&B&C&R, I&B&C&F, I&B&C&N, I&B&C&R&F, I&B&C&F&N, I&B&C&R&F&N, I&C&J, I&B&C&J, I&B&C&J&R, I&B&C&J&F, I&B&C&J&N, I&B&C&J&R&F, I&B&C&J&F&N, I&B&C&J&R&F&N}

Step g) Sub-step 1) D_B = {I&B, I&B&R, I&B&F, I&B&R&F}

Step g) Sub-step 2) D = {I&B, I&B&R, I&B&F, I&B&R&F}

Step g) Sub-step 3) R= {I, I&B&N, I&B&F&N, I&B&R&F&N, I&C, I&J, I&B&C, I&B&C&R, I&B&C&F, I&B&C&N, I&B&C&R&F, I&B&C&F&N, I&B&C&R&F&N, I&C&J, I&B&C&J, I&B&C&J&R, I&B&C&J&F, I&B&C&J&N, I&B&C&J&R&F, I&B&C&J&F&N, I&B&C&J&R&F&N}

Step h) No multiple inheriting Subtypes

Step i) No ONEOF constraint

Step j) No AND constraint

Step k) R= {I, I&B&N, I&B&F&N, I&B&R&F&N, I&C, I&J, I&B&C, I&B&C&R, I&B&C&F, I&B&C&N, I&B&C&R&F, I&B&C&F&N, I&B&C&R&F&N, I&C&J, I&B&C&J, I&B&C&J&R, I&B&C&J&F, I&B&C&J&N, I&B&C&J&R&F, I&B&C&J&F&N, I&B&C&J&R&F&N}

Figure 9-4a – Fourth Test Case – Generate Algorithm

The final result set R shown in step k in Figure 9-4a is identical to set L above showing the correctness of the Generate Algorithm in this case.

Using Example 9-4, the Test Algorithm evaluates the Total Over with elements on multiple levels correctly. The set of Entity instances to be considered are: P={I, B, C, J, R, F, N, I&B, I&C, I&J, I&R, I&F, I&N, B&C, B&J, B&R, B&F, B&N, C&J, C&R, C&F, C&N, J&R, J&F, J&N, R&F, R&N, F&N, I&B&C, I&B&J, I&B&R, I&B&F, I&B&N, I&C&J, I&C&R, I&C&F, I&C&N, I&J&R, I&J&F, I&J&N, I&R&F, I&R&N, I&F&N, B&C&J, B&C&R, B&C&F, B&C&N, B&J&R, B&J&F, B&J&N, B&R&F, B&R&N, B&F&N, C&J&R, C&J&F, C&J&N, C&R&F, C&R&N, C&F&N, J&R&F, J&R&N, J&F&N, R&F&N, I&B&C&J, I&B&C&R, I&B&C&F, I&B&C&N, I&B&J&R, I&B&J&F, I&B&J&N, I&B&R&F, I&B&R&N, I&B&F&N, I&C&J&R, I&C&J&F, I&C&J&N, I&C&R&F, I&C&R&N, I&C&F&N, I&J&R&F, I&J&R&N, I&J&F&N, I&R&F&N, B&C&J&R, B&C&J&F, B&C&J&N, B&C&R&F, B&C&R&N, B&C&F&N, B&J&R&F, B&J&R&N, B&J&F&N, B&R&F&N, C&J&R&F, C&J&R&N, C&J&F&N, C&R&F&N, J&R&F&N, I&B&C&J&R, I&B&C&J&F, I&B&C&J&N, I&B&C&R&F, I&B&C&R&N, I&B&C&F&N, I&B&J&R&F, I&B&J&R&N, I&B&J&F&N, I&B&R&F&N, I&C&J&R&F, I&C&J&R&N, I&C&J&F&N, I&C&R&F&N, I&J&R&F&N, B&C&J&R&F, B&C&J&R&N, B&C&J&F&N, B&C&R&F&N, B&J&R&F&N, C&J&R&F&N, I&B&C&J&R&F, I&B&C&J&R&N, I&B&C&J&F&N, I&B&C&R&F&N, I&B&J&R&F&N, I&C&J&R&F&N, B&C&J&R&F&N, I&B&C&J&R&F&N}.

It was found that items {I, I&B&N, I&B&F&N, I&B&R&F&N, I&C, I&J, I&B&C, I&B&C&R, I&B&C&F, I&B&C&N, I&B&C&R&F, I&B&C&F&N, I&B&C&R&F&N, I&C&J, I&B&C&J, I&B&C&J&R, I&B&C&J&F, I&B&C&J&N, I&B&C&J&R&F, I&B&C&J&F&N, I&B&C&J&R&F&N} gave a result of TRUE, while the items {B, C, J, R, F, N, I&B, I&R, I&F, I&N, B&C, B&J, B&R, B&F, B&N, C&J, C&R, C&F, C&N, J&R, J&F, J&N, R&F, R&N, F&N, I&B&J, I&B&R, I&B&F, I&C&R, I&C&F, I&C&N, I&J&R, I&J&F, I&J&N, I&R&F, I&R&N, I&F&N, B&C&J, B&C&R, B&C&F, B&C&N, B&J&R, B&J&F, B&J&N, B&R&F, B&R&N, B&F&N, C&J&R, C&J&F, C&J&N, C&R&F, C&R&N, C&F&N, J&R&F, J&R&N, J&F&N, R&F&N, I&B&J&R, I&B&J&F, I&B&J&N, I&B&R&F, I&B&R&N, I&B&F&N, I&C&J&R, I&C&J&F, I&C&J&N, I&C&R&F, I&C&R&N, I&C&F&N, I&J&R&F, I&J&R&N, I&J&F&N, I&R&F&N, B&C&J&R, B&C&J&F, B&C&J&N, B&C&R&F, B&C&R&N, B&C&F&N, B&J&R&F, B&J&R&N, B&J&F&N, B&R&F&N, C&J&R&F, C&J&R&N, C&J&F&N, C&R&F&N, J&R&F&N, I&B&C&R&N,

I&B&J&R&F, I&B&J&R&N, I&B&J&F&N, I&C&J&R&F, I&C&J&R&N, I&C&J&F&N, I&C&R&F&N, I&J&R&F&N, B&C&J&R&F, B&C&J&R&N, B&C&J&F&N, B&C&R&F&N, B&J&R&F&N, C&J&R&F&N, I&B&C&J&R&N, I&B&J&R&F&N, I&C&J&R&F&N, B&C&J&R&F&N} gave a result of FALSE or Unknown. This behavior proves the correctness of this algorithm in this case.

By doing a scan of the result set R, the test case of {I&B&C} is found in the set. Figure 9-4b shows the steps taken in checking the item {I&B&C} using the Test Algorithm and Example 9-4.

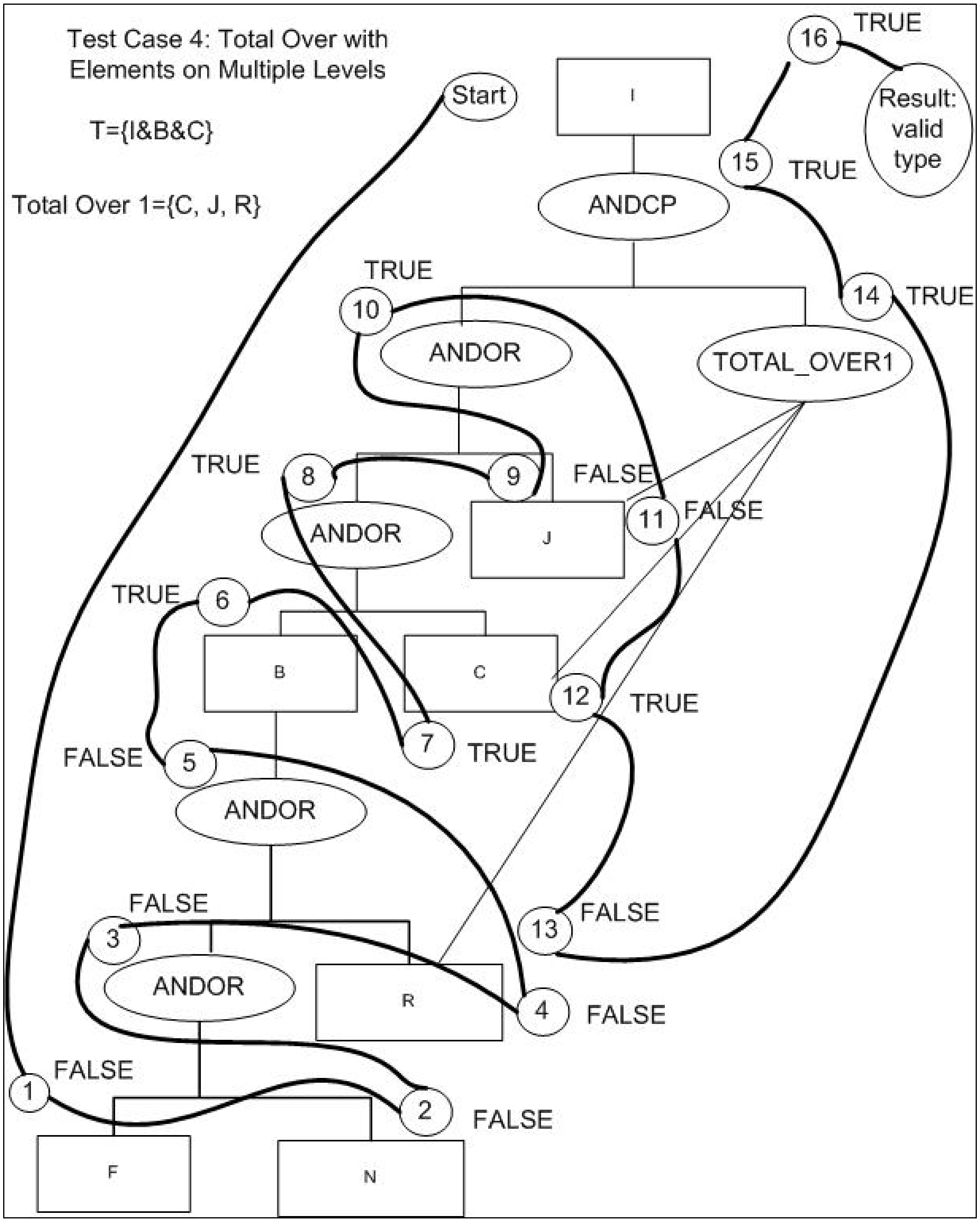


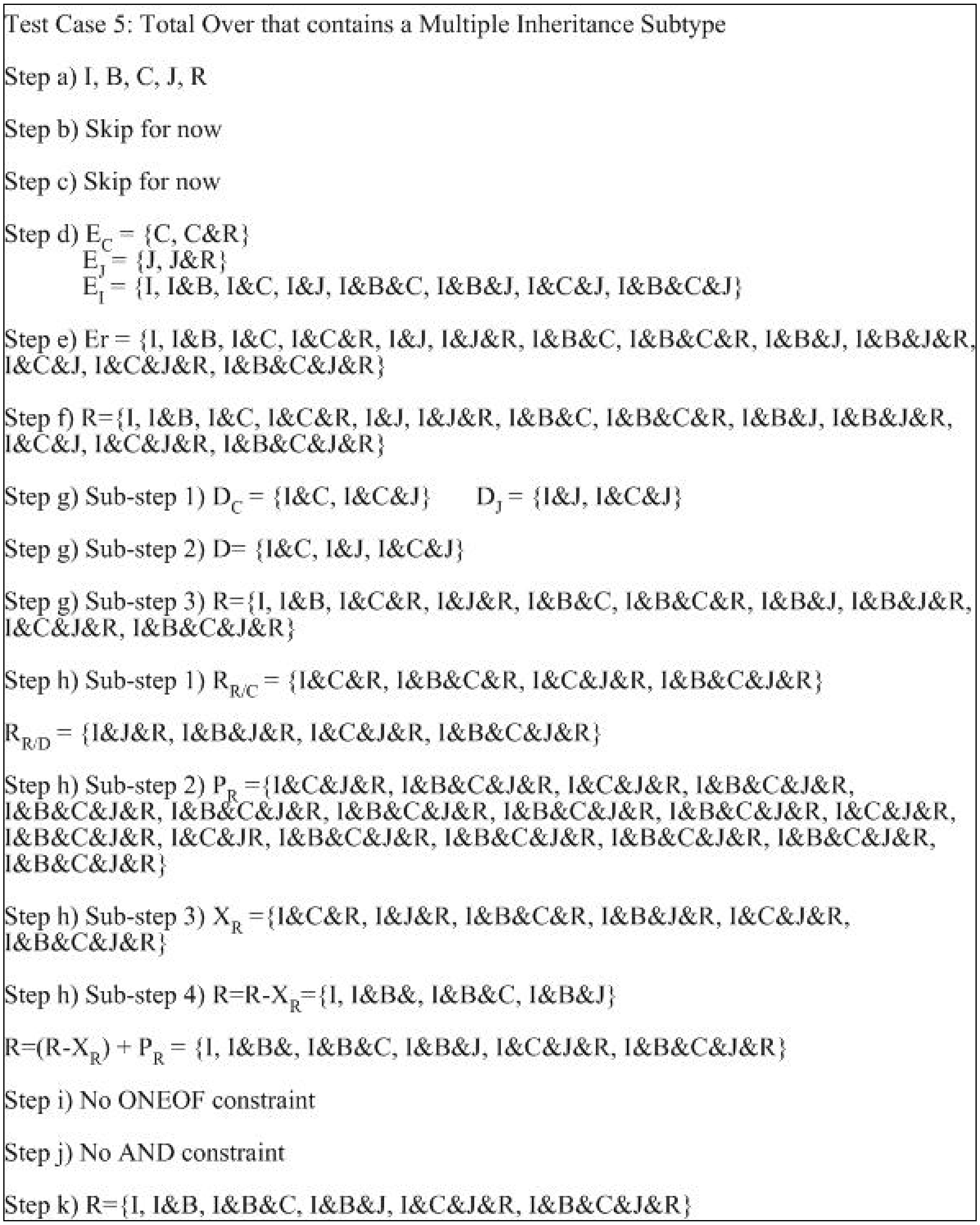
Figure 9-4b – Fourth Test Case – Test Algorithm

The fifth test case Total Overs that contain a Multiple Inheritance Subtype is useful because multiple inheritance is a possibility. Proving that a developer can use a Total Over constraint, which includes a multiply inherited Subtype, is important. These tests will show a developer that a model can be made as complex as needed for the real world situation.

```
SUBTYPE_CONSTRAINT item_types_1 FOR item;
        book_item ANDOR circulating_item ANDOR journal_item ;
END_SUBTYPE_CONSTRAINT;

SUBTYPE_CONSTRAINT circulating_types_1 FOR circulating_item;
        reserve_item;
END_SUBTYPE_CONSTRAINT;

SUBTYPE_CONSTRAINT journal_types_1 FOR journal_item;
        reserve_item;
END_SUBTYPE_CONSTRAINT;

SUBTYPE_CONSTRAINT item_types_2 FOR item;
        TOTAL_OVER (book_item, jreserve_item);
END_SUBTYPE_CONSTRAINT;

I=item
B=book_item
C=circulating_item
J=journal_item
R=reserve_item
```

Example 9-5

The original Generate Algorithm generates the set L={I, I&B, I&B&C, I&B&J, I&C&J&R, I&B&C&J&R}. Using Example 9-5, Figure 9-5a shows the results of each step in the new Generate Algorithm.



Test Case 5: Total Over that contains a Multiple Inheritance Subtype

Step a) I, B, C, J, R

Step b) Skip for now

Step c) Skip for now

Step d) E_C = {C, C&R}
E_J = {J, J&R}
E_I = {I, I&B, I&C, I&J, I&B&C, I&B&J, I&C&J, I&B&C&J}

Step e) Er = {I, I&B, I&C, I&C&R, I&J, I&J&R, I&B&C, I&B&C&R, I&B&J, I&B&J&R, I&C&J, I&C&J&R, I&B&C&J&R}

Step f) R={I, I&B, I&C, I&C&R, I&J, I&J&R, I&B&C, I&B&C&R, I&B&J, I&B&J&R, I&C&J, I&C&J&R, I&B&C&J&R}

Step g) Sub-step 1) D_C = {I&C, I&C&J} D_J = {I&J, I&C&J}

Step g) Sub-step 2) D= {I&C, I&J, I&C&J}

Step g) Sub-step 3) R={I, I&B, I&C&R, I&J&R, I&B&C, I&B&C&R, I&B&J, I&B&J&R, I&C&J&R, I&B&C&J&R}

Step h) Sub-step 1) $R_{R/C}$ = {I&C&R, I&B&C&R, I&C&J&R, I&B&C&J&R}

$R_{R/D}$ = {I&J&R, I&B&J&R, I&C&J&R, I&B&C&J&R}

Step h) Sub-step 2) P_R ={I&C&J&R, I&B&C&J&R, I&C&J&R, I&B&C&J&R, I&B&C&J&R, I&B&C&J&R, I&B&C&J&R, I&B&C&J&R, I&B&C&J&R, I&C&J&R, I&B&C&J&R, I&C&JR, I&B&C&J&R, I&B&C&J&R, I&B&C&J&R, I&B&C&J&R, I&B&C&J&R}

Step h) Sub-step 3) X_R ={I&C&R, I&J&R, I&B&C&R, I&B&J&R, I&C&J&R, I&B&C&J&R}

Step h) Sub-step 4) R=R-X_R={I, I&B&, I&B&C, I&B&J}

R=(R-X_R) + P_R = {I, I&B&, I&B&C, I&B&J, I&C&J&R, I&B&C&J&R}

Step i) No ONEOF constraint

Step j) No AND constraint

Step k) R={I, I&B, I&B&C, I&B&J, I&C&J&R, I&B&C&J&R}

Figure 9-5a – Fifth Test Case – Generate Algorithm

The final result set R shown in step k in Figure 9-5a is identical to set L above showing the correctness of the Generate Algorithm in this case.

Using Example 9-5, the Test Algorithm evaluates the Total Over that contains a multiple inheritance subtype correctly. The set of Entity instances to be considered are: P={I, B, C, J, R, I&B, I&C, I&J, I&R, B&C, B&J, B&R, C&J, C&R, J&R, I&B&C, I&B&J, I&B&R, I&C&J, I&C&R, I&J&R, B&C&J, B&C&R, B&J&R, C&J&R, I&B&C&J, I&B&C&R, I&B&J&R, I&C&J&R, B&C&J&R, I&B&C&J&R}.

It was found that items {I, I&B, I&B&C, I&B&J, I&C&J&R, I&B&C&J&R} gave a result of TRUE, while the items {B, C, J, R, I&C, I&J, I&R, B&C, B&J, B&R, C&J, C&R, J&R, I&B&J, I&B&R, I&C&J, I&C&R, I&J&R, B&C&J, B&C&R, B&J&R, C&J&R, I&B&C&J, I&B&C&R, I&B&J&R, B&C&J&R} gave a result of FALSE or Unknown. This behavior proves the correctness of this algorithm in this case. By doing a scan of the result set R, the test case of {I&B&C} is found in the set. Figure 9-5b shows the steps taken in checking the item {I&B&C} using the Test Algorithm and Example 9-5.

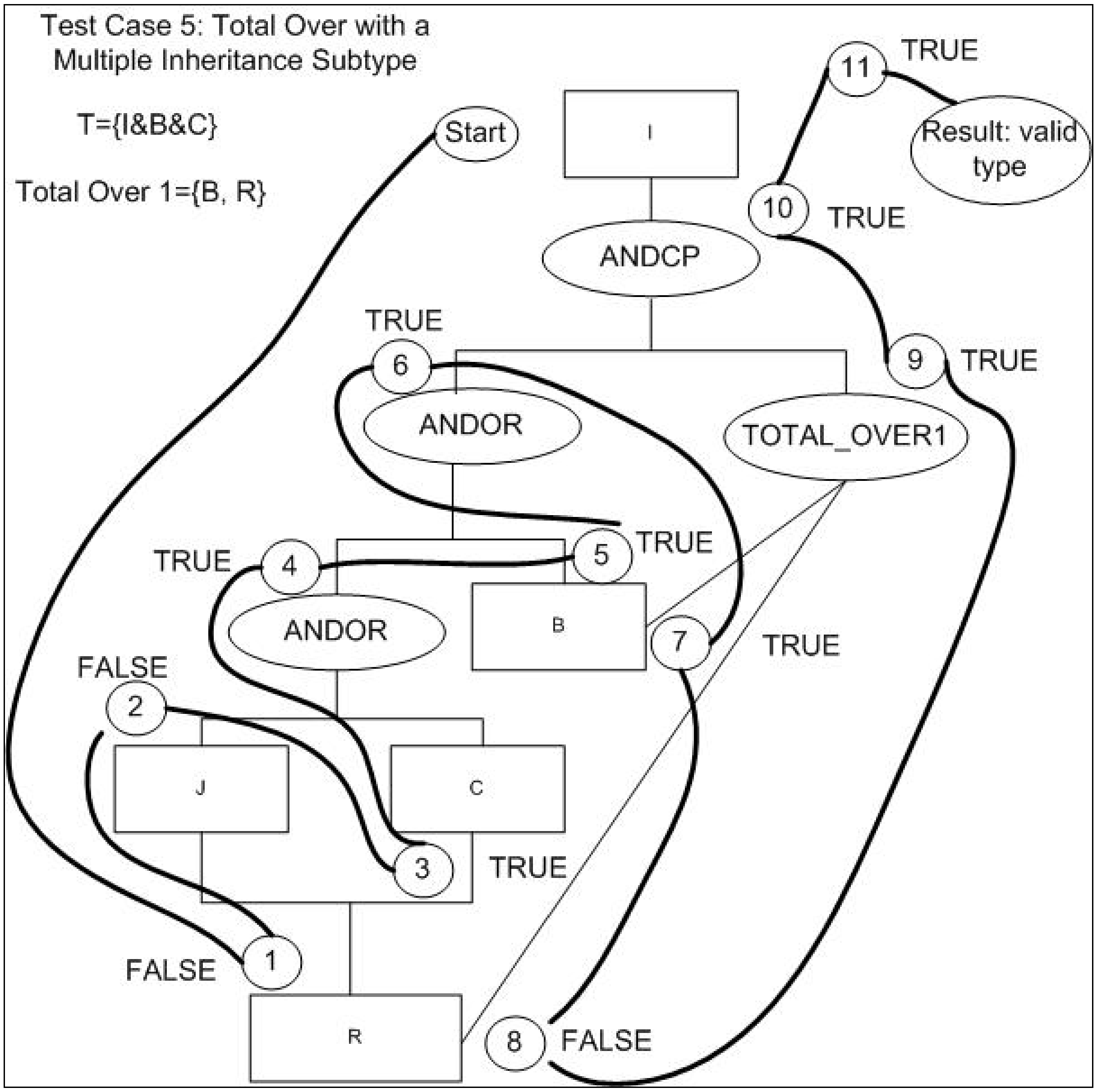


Figure 9-5b – Fifth Test Case – Test Algorithm

Testing an algorithm for correctness is an important step in determining if the algorithm is useful to the developer. The test cases used in this chapter show that the output from the Generate Algorithm produces a TRUE result from the Test Algorithm. Furthermore, an instance combination that results as a FALSE or ERROR from the Test Algorithm will not be in the set generated from the Generate Algorithm. This proves that the Generate Algorithm is equivalent to the Test Algorithm. Also, the test cases showed that the output from the modified Generate

Algorithm produced the same results as the original Generate algorithm. This proves that the modified Generate Algorithm is equivalent to the original Generate algorithm. The modified Generate Algorithm produces the results in a more efficient manner and is, therefore, the better version for the developers to use. Developers can be assured the results will be correct by using either of these algorithms to test an instance combination.

CHAPTER 10

CONCLUSION

The main focus of this paper was to compare the Generate and Test inheritance hierarchy algorithms for the EXPRESS language. Due to the updating of EXPRESS planned for the third edition, both of the algorithms lacked efficient support for the new Total Over constraint. This work demonstrated efficient extensions for both algorithms. In the case of the Generate Algorithm the ISO proposed extension while correct was inefficient, a much more efficient extension has been proposed here and demonstrated to be equivalent to the ISO extension. In the case of the Test Algorithm no extension had yet been proposed. In the process of defining the extension it was found that the Test Algorithm only existed as a general description. A full formal algorithm was reverse engineered from this. The strengths and weaknesses of both algorithms were analyzed and presented. In addition both of these enhanced algorithms were compared for correctness and equivalence by using test cases based on how the new Total Over type was used in the inheritance hierarchy. Each of these accomplishments will be reviewed below, and some proposed extensions to the work will be discussed.

The Generate Algorithm from the EXPRESS Language Reference Manual 4] generated all legal possible combinations of Entity types in the inheritance hierarchy. The Generate Algorithm had been extended in edition three with a step to handle the Total Over constraint, but this was not an efficient method. The original version of the Generate Algorithm generated a subset of partial complex entity data types by combining all of the Entity types in the Total Over set with all of the combinations in the result set. This required that duplicates and invalid combinations be discarded from the result set in later steps, which makes this algorithm inefficient. The new version proposed here generates a subset of partial complex entity data types that are invalid based on the Total Over constraint. This subset is then removed from the result set that doesn't allow for invalid combinations to be introduced. This makes it easier for the user to generate a legal set of instances that they need and makes the algorithm more efficient.

Gunter Staub, Frank Schonefeld, and Markus Maier [7] developed the Test Algorithm. This algorithm tests one instance for membership in an inheritance hierarchy without generating the entire set of legal combinations. There are two phases to this algorithm: creating the evaluation graph and testing the instance. The input to the first phase is the inheritance hierarchy and the output is the evaluation graph. The input to the second phase is the evaluation graph and the partial complex entity instance to be tested and the output is a result of TRUE, FALSE, or ERROR. The TRUE result means it is valid based on the inheritance hierarchy, while FALSE and ERROR means that it did not conform to one or more constraints in the hierarchy. This algorithm has been published as a general description, but no formal algorithm of the steps needed to evaluate a partial complex entity data type was ever disclosed. This formal model was created as part of this work. The first step was to document the steps needed to generate the evaluation graph, followed by those needed to traverse that graph to evaluate the validity of a type combination. The Test Algorithm did not support the proposed Total Over constraint, this paper has presented such an extension to the algorithm.

The revised Test and Generate Algorithms were compared on their similarities, advantages, and disadvantages and tested to for correctness and equivalence. The Generate Algorithm takes a long time to generate all possible legal combinations but is very efficient at testing if a specific combination is valid or not, as all that is required is a lookup in the set. The Test Algorithm exhibits the opposite behavior. It can generate the evaluation graph quickly, but it takes a fair amount of time to evaluate the graph for the validity of a specific combination. This makes the Generate Algorithm very efficient when a large number of type combinations need to be tested, and the Test Algorithm is more efficient when just a few number of combinations need to be tested. Because developers usually have just a few instance combinations to test, the Test Algorithm is usually the more efficient. Also, as the number of Entity types increases by one for the Generate Algorithm, the number of partial complex entity data type in the set is doubled. While increasing the number of Entity types for the Test Algorithm by one only required two more steps in the evaluation of the partial complex entity data type. This makes the Test Algorithm more efficient as the inheritance hierarchy becomes more complex.

The tests performed in this paper show that both the algorithms produced correct results and that the results from both are in fact equivalent. The tests cases covered all possible ways the new Total Over constraint could be added into an existing inheritance hierarchy. The proof showed that all combinations generated by the Generate Algorithm were accepted by the Test Algorithm, and that no combination of types not present in the generated set were accepted by the Test Algorithm. It also showed that all combinations accepted by the Test Algorithm were present in the generated set, and that no item rejected by the test algorithm was present in the generated set. These tests were performed over several cases each of which demonstrated one of the possible ways a Total Over constraint could be added into an existing inheritance hierarchy. With all possible cases evaluated the proof was considered complete.

This work has compared and extended the Generate and Test Algorithms, further work in this field is possible. Specifically, the author has considered: An induction proof of the equivalence of the Test and Generate Algorithms based on the number of nodes present in the evaluation graph and a comparison of the Test and Generate algorithms in terms of space efficiency. An empirical study to confirm this theoretical one, timing runs of implementations of both the Test and Generate algorithms over a group of schemas used for STEP AIP's.

REFERENCES

1. Baase, Sara. 1988. *Computer Algorithms: Introduction to Design and Analysis.* 2d ed. Reading: Addison-Wesley Publishing Company.

2. Hofri, Micha. 1995. Algorithm analysis. *Dr. Dobb's Journal*. (March): 125-26.

3. Sanderson, Donald, and Phil Spiby. 1998. *EXPRESS 2.*

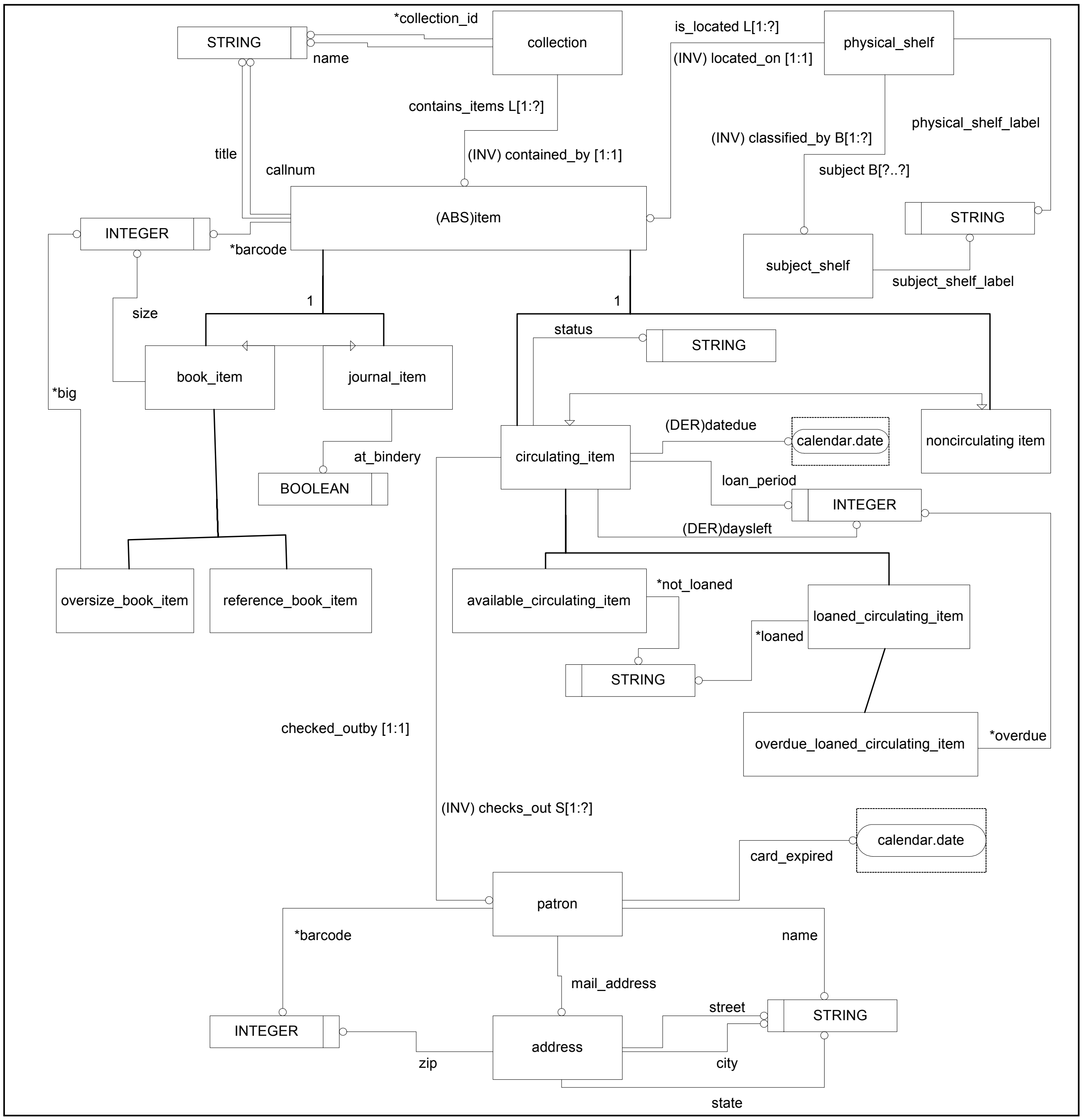
4. Sanderson, Donald, and Phil Spiby. 2000. *ISO 10303 TC184/SC4/WG11 N105 Express Language Reference Manual*. Technical Paper.

5. Schenck, Douglas, and Peter Wilson. 1994. *Information Modeling the EXPRESS Way*. New York: Oxford University Press.

6. Spiby, Phil, and Douglas Schenck. 1994. *ISO 10303 Part 11 Express Language Reference Manual*.

7. Staub, Günter, Markus Maier, Frank Schönefeld. 1994. *Multiple Class Membership and Supertype Constraint Handling - Concepts and Implementation Aspects.*

APPENDIX A
EXPRESS EDITION 1 – LIBRARY SCHEMA



APPENDIX B

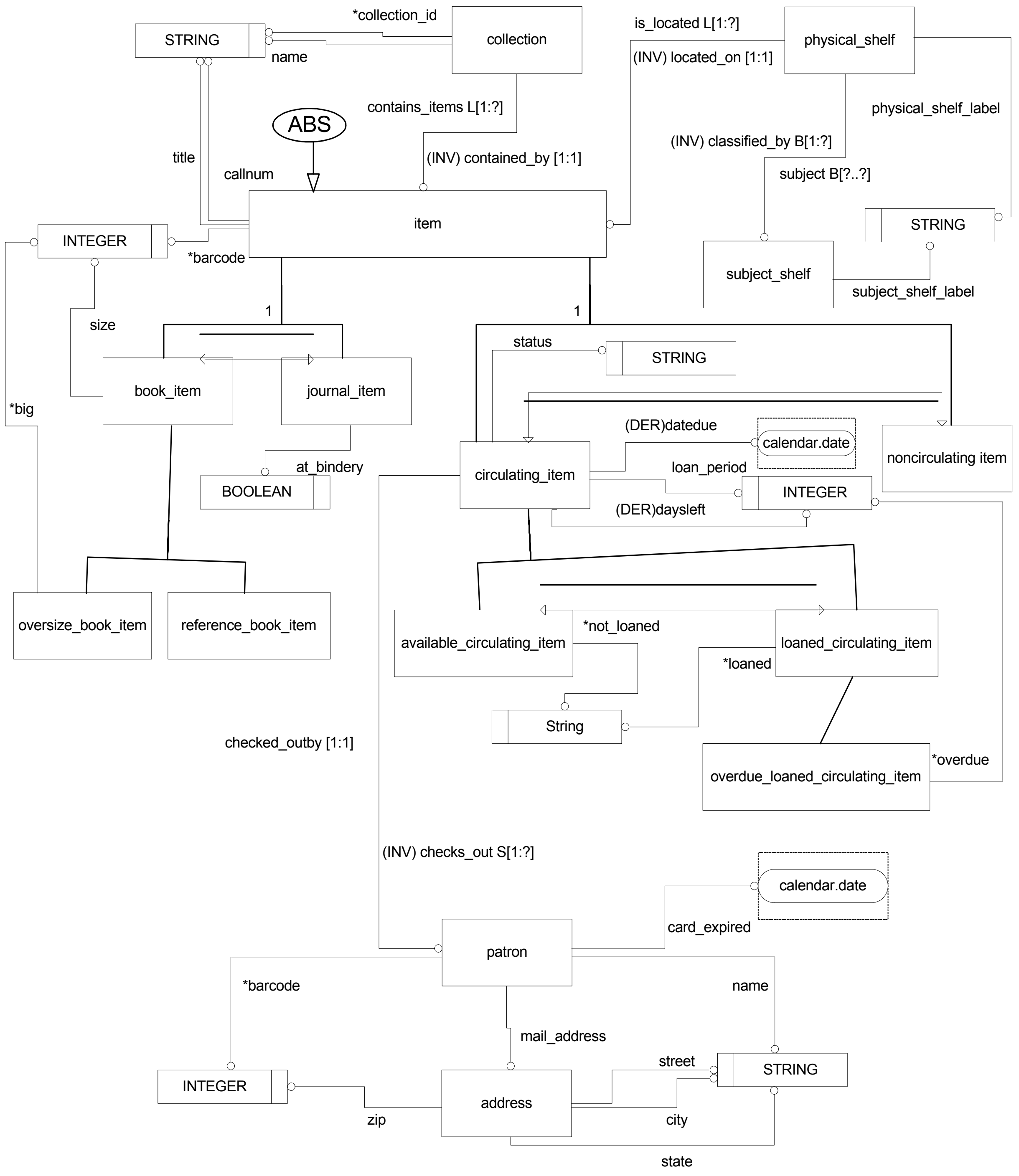
EXPRESS EDITION 1 – LIBRARY SCHEMA

```
SCHEMA library;
        (* EXPRESS Edition 1 model of a library by J. Dawn Greer *)
        REFERENCE FROM calendar;

ENTITY collection;                      -- library collection
        name            : STRING;
        contains_item :  LIST [1..?] OF item;
        collection_id  : STRING;
UNIQUE
        collect         : collection_id;
END_ENTITY;

ENTITY item                             -- item in the library
ABSTRACT SUPERTYPE OF (ONEOF (book_item, journal_item) AND
(ONEOF (circulating_item, noncirculating_item)));
        title           : STRING;
        callnum         : STRING;
        barcode         : OPTIONAL INTEGER;             -- uniquely identifies the item
INVERSE
        contained_by : collection OF contains_item;
        located_on    : physical_shelf OF is_located;
UNIQUE
        item_single   : barcode;
END_ENTITY;

ENTITY book_item
        SUPERTYPE OF (oversize_book_item ANDOR reference_book_item)
        SUBTYPE OF (item);
        size    : INTEGER;              -- size in square feet of an item
END_ENTITY;

ENTITY journal_item                     -- an item that is a journal
        SUBTYPE OF (item);
        at_bindery    : BOOLEAN;        -- an item is being bound
END_ENTITY;

ENTITY circulating_item                 -- an item that can be checked out by a person
        SUPERTYPE OF (loaned_circulating_item ANDOR available_circulating_item)
        SUBTYPE OF (item);
        status : STRING;                    -- status of the item checked out or on the shelf
        loan_period   : INTEGER;            -- length of time the item can be checked out
        checked_outby  : patron;            -- link to a person who checks out the item
DERIVE
        datedue : date := current_date() + loan_period; -- date the item is due
        daysleft : INTEGER := datedue-current.date();  -- datedue minus current date
END_ENTITY;

ENTITY loaned_circulating_item              -- item is checked out to a person
        SUPERTYPE OF (overdue_loaned_circulating_item);
        SUBTYPE OF (circulating_item);
WHERE
        loaned : status = "Checked out";
END_ENTITY;

ENTITY overdue_loaned_circulating_item          -- item is overdue
        SUBTYPE OF (loaned_circulating_item);
WHERE
        overdue : daysleft < 0; -- daysleft is negative (datedue is before the current date)
END_ENTITY;

ENTITY available_circulating_item           - an item is available for check out
        SUBTYPE OF (circulating_item);
WHERE
        not_loaned : status = “Not Checked out”;
END_ENTITY;

ENTITY noncirculating_item                  -- an item that does not get checked out
        SUBTYPE OF (item);
END_ENTITY;

ENTITY oversize_book_item
        SUBTYPE OF (book_item);
WHERE
        big     : size >= 2;    -- an item larger that 2 square feet is considered big
END_ENTITY;

ENTITY reference_book_item
        SUBTYPE OF (book_item);
END_ENTITY;

ENTITY patron;                              -- person who uses the library
        name          : STRING;
        mail_address: address;
        card_expired : date;         -- expiration date of the person's library card
        barcode      : INTEGER;  -- barcode number that uniquely identifies a person
INVERSE                              -- patron is allowed to check out circulating items
        checks_out   : SET [1:?] OF circulating_item FOR checked_outby;
UNIQUE
        parton_single : barcode;
END_ENTITY;

ENTITY physical_shelf;                       -- physical shelf where items are stored
        physical_shelf_label : STRING;
        subject : BAG [1..?] OF subject_shelf;
        is_located : LIST [1..?] OF item;
END_ENTITY;

ENTITY subject_shelf;                        -- items are arranged by subject
        subject_shelf_label : STRING;
INVERSE
        classified_by : BAG [1..?] OF physical_shelf FOR subject;
END_ENTITY;

ENTITY address;                      -- address which holds street, city, state, and zip
        street  : STRING;
        city    : STRING;
        state   : STRING;
        zip     : INTEGER;
END_ENTITY;

END_SCHEMA;
```

```
SCHEMA calendar;
(* Taken from Information Modeling: The EXPRESS Way by Douglas Schenck and
Peter Wilson [5] *)

TYPE months = ENUMERATION OF
(January, February, March, April, May, June, July, August, September, October,
November, December);

ENTITY date;
        day    : INTEGER;
        month : months;
        year   : INTEGER;
WHERE
        days_ok : {1 <= day <= 31};
        years_ok : year > 0;
END_ENTITY;

FUNCTION current_date() : date;
(* This function returns the date when it is called.  Typically, it will be implemented via a
system provided procedure within the information base *)
END_FUNCTION;

END_SCHEMA; -- calendar
```

APPENDIX C

EXPRESS EDITION 3 – LIBRARY SCHEMA



APPENDIX D

EXPRESS EDITION 3 – LIBRARY SCHEMA

```
SCHEMA library version 3;
        (* EXPRESS Edition 3 model of a Library by J. Dawn Greer *)
        REFERENCE FROM calendar;

ENTITY collection;                          -- library collection
        name            : STRING;
        contains_items :  LIST [1:?] OF item;
        collection_id  : STRING;
UNIQUE
        collect         : collection_id;
END_ENTITY;

ENTITY item;                        -- item in the library
        title           : STRING;
        callnum         : STRING;
        barcode         : OPTIONAL INTEGER;            -- uniquely identifies the item
UNIQUE
        items_single  : barcode;
INVERSE
        contained_by : collection OF contains_items;
        located_on     : physical_shelf OF is_located;
END_ENTITY;

SUBTYPE_CONSTRAINT item_type FOR item;
ABSTRACT SUPERTYPE;
ONEOF (book_item, journal_item) AND
ONEOF (circulating_item, noncirculating_item);
TOTAL_OVER (book_item, journal_item);
TOTAL_OVER (circulating_item, noncirculating_item);
END_SUBTYPE_CONSTRAINT;

ENTITY book_item;
        SUBTYPE OF (item);
        size     : INTEGER;                  -- size in square feet of an item
END_ENTITY;

SUBTYPE_CONSTRAINT book_type FOR book_item;
        (oversize_book_item ANDOR reference_book_item);
END_SUBTYPE_CONSTRAINT;
```

```
ENTITY journal_item;                    -- an item that is a journal
        SUBTYPE OF (item);
        at_bindery    : BOOLEAN;        -- an item is being bound
END_ENTITY;

ENTITY circulating_item;                -- an item that can be checked out by a person
        SUBTYPE OF (item);
        status        : STRING;         -- status of the item checked out or on the shelf
        loan_period   : INTEGER;        -- length of time the item can be checked out
        checked_outby  : patron;        -- link to a person who checks out the item
DERIVE
        datedue : date := current_date() + loan_period; -- date the item is due
        daysleft : INTEGER := datedue – current_date();  -- datedue minus current date
END_ENTITY;

SUBTYPE_CONSTRAINT circulating_type FOR circulating_item;
        (loaned_circulating_item ANDOR available_circulating_item);
        TOTAL_OVER (loaned_circulating_item, available_circulating_item);
END_SUBTYPE_CONSTRAINT;

ENTITY loaned_circulating_item;         -- item is checked out to a person
        CONNOTATIONAL SUBTYPE OF (circulating_item);
WHERE
        loaned : status = “Checked out”;
END_ENTITY;

ENTITY overdue_loaned_circulating_item;              -- item is overdue
        CONNOTATIONAL SUBTYPE OF (loaned_circulating_item);
WHERE
        overdue : daysleft < 0;  -- daysleft is negative (datedue is before the current date)
END_ENTITY;

ENTITY available_circulating_item       -item is available for check out
        CONNOTATIONAL SUBTYPE OF (circulating_item);
WHERE
        not_loaned : status =”Not Checked out”;
END_ENTITY;

ENTITY noncirculating_item;       -- an item that does not get checked out
        SUBTYPE OF (item);
END_ENTITY;
```

```
ENTITY oversize_book_item;
        SUBTYPE OF (book_item);
WHERE
        big     : size >=2;    -- an item larger than 2 square feet is considered big
END_ENTITY;

ENTITY reference_book_item;
        SUBTYPE OF (book_item);
END_ENTITY;

ENTITY patron;                      -- person who uses the library
        name            : STRING;
        mail_address    : address;
        card_expired    : date;         -- expiration date of the person's library card
        barcode         : INTEGER;  -- barcode number that uniquely identifies a person
UNIQUE
        parton_single : barcode;
INVERSE                             -- patron is allowed to check out circulating items
        checks_out   : BAG [1:?] OF UNIQUE circulating_item FOR checked_outby;
END_ENTITY;

ENTITY physical_shelf;                      -- physical shelf where items are stored
        physical_shelf_label : STRING;
        subject : BAG [1:?] OF subject_shelf;
        is_located : LIST [1:?] OF item;
END_ENTITY;

ENTITY subject_shelf;                       -- items are arranged by subject
        subject_shelf_label : STRING;
INVERSE
        classified_by : BAG [1:?] OF physical_shelf OF subject;
END_ENTITY;

ENTITY address;                     -- address which holds street, city, state, and zip
        street  : STRING;
        city    : STRING;
        state   : STRING;
        zip     : INTEGER;
END_ENTITY;

END_SCHEMA;
```

```
SCHEMA calendar;
(* Taken from Information Modeling: The EXPRESS Way by Douglas Schenck and
Peter Wilson [5] *)

TYPE months = ENUMERATION OF
(January, February, March, April, May, June, July, August, September, October,
November, December);

ENTITY date;
        day     : INTEGER;
        month : months;
        year    : INTEGER;
WHERE
        days_ok : {1 <= day <= 31};
        years_ok : year > 0;
END_ENTITY;

FUNCTION current_date() : date;
(* This function returns the date when it is called.  Typically, it will be implemented via a
system provided procedure within the information base *)
END_FUNCTION;

END_SCHEMA; -- calendar
```

VITA

JUDY DAWN GREER

Personal Data:	Date of Birth: February 25, 1970
	Place of Birth: Atlanta, Georgia
Education:	Public Schools, Gray, Tennessee
	East Tennessee State University, Johnson City, Tennessee; Computer and Information Science, B.S., 1992
	East Tennessee State University, Johnson City, Tennessee; Computer and Information Science, M.S., 2002
Professional Experience:	Systems Analyst, Sherrod Library-East Tennessee State University; Johnson City, Tennessee, 1994-2000
	Network Analyst, ETSU Physicians and Associates; Johnson City, Tennessee, 2000-Present