Date: January 2011

Architecture-driven Modernization: Abstract Syntax Tree Metamodel (ASTM)

Version 1.0

OMG Document Number: formal/2011-01-05
Standard document URL: http://www.omg.org/spec/ASTM
Associated Schema Files:

ptc/2009-09-06 -- http://www.omg.org/spec/ASTM/20090901
ptc/2009-07-08 -- http://www.omg.org/spec/ASTM/20090701

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

Table of Contents

Preface

About the Object Management Group

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A catalog of all OMG Specifications is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

Business Modeling Specifications

- Business Rules and Process Management Specifications

Language Mappings

- IDL/Language Mapping Specifications
- Other Language Mapping Specifications

Middleware Specifications

- CORBA/IIOP
- CORBA Component Model
- Data Distribution
- Specialized CORBA

Modeling and Metadata Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications.

Modernization Specifications

- KDM

Platform Independent Model (PIM), Platform Specific Model (PSM), and Interface Specifications

- CORBAservices
- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) All specifications are available in PostScript and PDF format and may be obtained from the Specifications Catalog cited above. Certain OMG specifications are also available as ISO standards. Please consult http://www.iso.org

OMG Contact Information

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
http://www.omg.org/
Email: pubs@omg.org

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier - 10 pt. Bold:` Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

Note – Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/technology/agreement.htm*.

1 Scope

The Abstract Syntax Tree Metamodeling (ASTM) and the Knowledge Discovery Metamodeling (KDM) are two complementary modeling specifications developed by the OMG Architecture Driven Modernization Task Force. Their relationship can be clearly understood by recognizing that the KDM establishes a specification for abstract semantic graph models, while the ASTM establishes a specification for abstract syntax tree models. Thus, in contrast to other software representation standards, such as the Knowledge Discovery Metamodel or the Unified Modeling Language, the ASTM supports a direct 1-to-1 mapping of all code-level software language statements into low-level software models.

This mapping is intended to provide a framework for:

1. A high-fidelity invertible representation of code written in any software language (achieved by supplementation of the ASTM with concrete syntax specifications).
2. Attachment of low-level software semantics produced by a constraint analysis, specifically scope analysis and the defRef and refTo association between definitions and identifier usage.

This separation within the ASTM between high fidelity low-level syntax models and low-level semantic models defines the two compliance points defined in the Conformance section.

In combination the ASTM and the KDM provide a comprehensive modeling framework for modeling software syntax and semantics. Within this framework the ASTM provides high-fidelity low-level syntax models and their basic semantics, and the KDM provides the higher level semantic models of software.[1] The UML and many other OMG specifications can also be considered semantic graphs for modeling the semantic properties of many other aspects of software. Thus, in essence the ASTM acts as the lowest level foundation for modeling of software within the OMG ecosystem of standards, while the KDM serves as a gateway to the higher-level OMG models.

Unlike the KDM, UML, and other OMG specifications, the ASTM does not itself provide constructs for refinement of low-level software conceptions into higher-level conceptual abstractions, as does the core KDM Compositional, Conceptual, and Behavioral Packages; however, it does complement the KDM by providing a continuous framework for mapping between low-level software models that are represented in the ASTM and higher-level conceptual views of software that are represented by the KDM and other OMG modeling standards, such as UML. The complementary relationship between the ASTM and the KDM is illustrated in Figure 1.1, with the ASTM depicted as the light blue sphere and the KDM depicted as the light yellow sphere.

1. Please note that basic semantics for scope, defRef, type and relationship are included in this ASTM Specification.

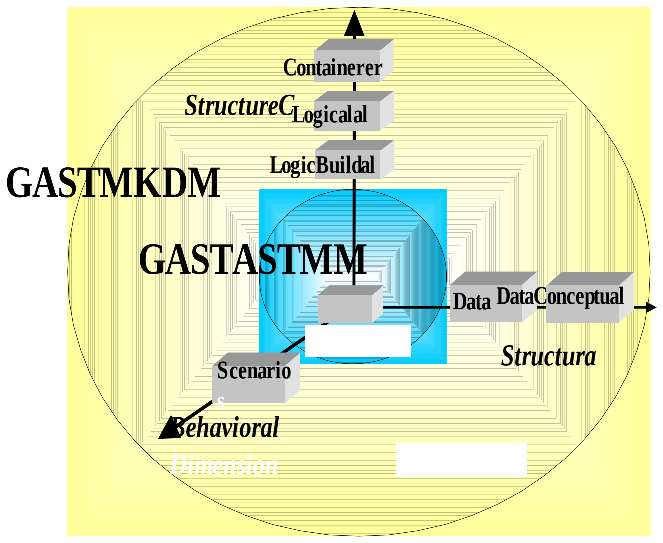


Figure 1.1 - ASTM + KDM

ASTM itself serves as a universal high-fidelity gateway for modeling code at its most fundamental syntactic level. The ASTM respects the scope of the KDM and the UML for modeling the semantics of higher-level software concepts and it therefore includes only the most basic semantics for modeling low-level semantics that are closely associated with code (namely, code location, scope, reference, and type). These forms of basic semantics are the foundation for most other forms of higher level semantics and must be available at the very high-level of fidelity that only the ASTM provides. By contrast the KDM provides the higher-level conceptual models for capturing the behavioral compositional and structural semantics of software. The KDM is by intent more general and more abstract than the ASTM.

To provide for uniformity as well as a universal framework for extension, the ASTM is composed of a core specification, the Generic Abstract Syntax Meta-Model (GASTM), and a set of complementary specifications that extend the core, called the Specialized Abstract Syntax Meta-Models (SASTMs). Figure 1.2 illustrates this concept by depicting the GASTM as the core and two separate SASTMs whose elements derive and extend the GASTM.

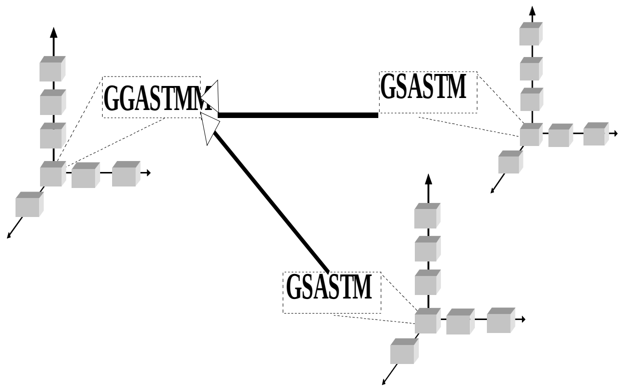


Figure 1.2 - ASTM = GASTM + SASTMs

Support for syntactic modeling and basic semantics of machine-language, functional and logic programing, object-oriented and rule-based languages will be added incrementally in the future by the supplementation of the ASTM with SASTMs for additional families of languages, as illustrated in Figure 1.3.

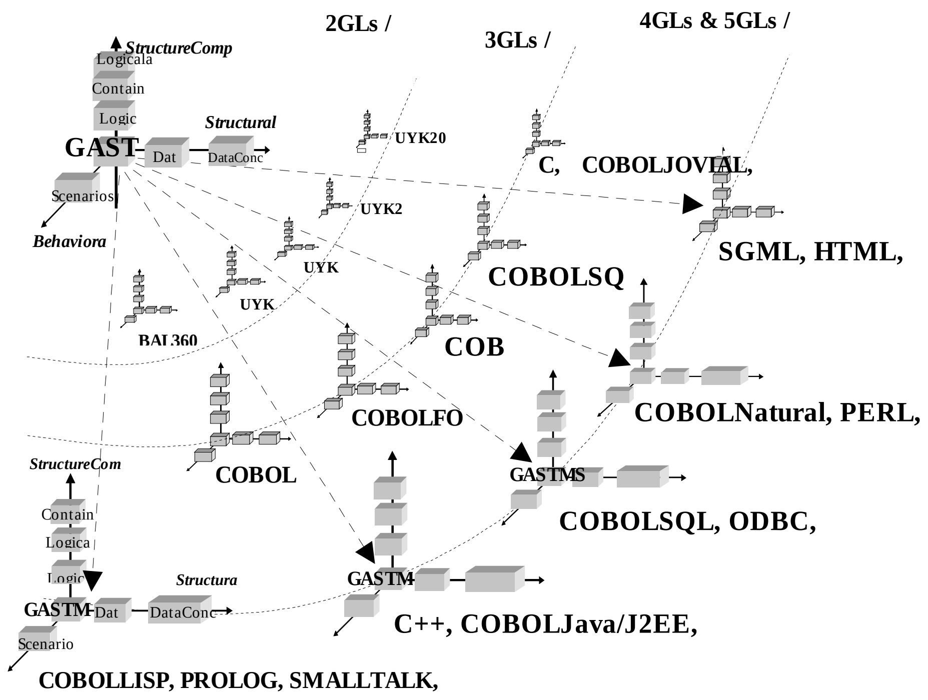


Figure 1.3 - Universal Software Modeling Framework (GASTM+SASTMs + KDM)

Annex A contains an example of a SASTM for modeling Relational Data Base (RDB) manipulation languages. The RDB SASTM provided in Annex A is a non-normative SASTM provided solely to illustrate how the GASTM is to be extended by SASTMs such as the RDB. The RDB SASTM illustrates the process by which the core the ASTM, the GASTM can be elegantly extended without redundancy or overlap to support comprehensive modeling of other software language families, thus illustrating the process by which the ASTM's GASTM and SASTMs will be extended to provide a universal framework for modeling any and all software languages at a low-level of abstraction.

2 Conformance

The purpose of the ASTM is to provide a framework that allows tool vendors and tool clients to build and use tools that conform to commonly agreed upon modeling specifications for the interchange of abstract syntax models of software. Interoperability is achieved when models can be interchanged using modeling elements that conform to those specified in the ASTM specification. The internal proprietary models of tools need not conform the ASTM for a tool to be considered compliant with the ASTM. To be considered compliant a tool need only adhere to the ASTM as a model interchange specification. Tool conformance is concerned solely with the ability of tools to interchange models that conform to the ASTM.

- For a GAST model to conform with the ASTM it must conform to the GAST Metamodel provided by this specification.
- For a SAST model to conform to the ASTM it must conform to both the GASTM model provided with this specification as well as the SASTM model provided by some future SASTM specification.

The ASTM is a bi-dimension multi-layered modeling specification. The two dimensions of the ASTM define both syntactic as well as the semantic properties of software. The layers of the ASTM define a core set of modeling elements, the GASTM, that are common to many programming languages as well as a set of extensions, the SASTMs, that extend from the core for and are used in concert with the GASTM for defining models specialized to particular programming languages. Table 2.1 illustrates the Compliance Points of the ASTM.

Table 2.1 - ASTM Compliance Points

Level 0 Compliance - Syntactic	Level 1 Compliance - Semantic
GASTM Syntax	GASTM Semantics
SASTM Syntax	SASTM Semantics

Compliance points are defined for the syntactic and semantic dimensions as well as for the GASTM and SASTM layers as follows.

To achieve Level 0 Compliance (Syntactic Compliance)

- All the elements of the AST must conform to the syntactic properties of software defined by the GASTM, specifically they must be expressed using the elements of the GASTMSyntaxObject and its subclasses. A model is compliant syntactically with the ASTM if its syntactic properties are expressed in the syntactic modeling elements of the GASTMSyntaxObject and the syntax elements of future SASTM syntax objects.

To achieve Level 1 Compliance (Semantic Compliance)

- The elements of the AST must conform to the semantic properties of software defined by the GASTM, specifically they must be expressed using the elements of the GASTMSemanticObject and its subclasses. A model is compliant semantically with the ASTM if its semantic properties are expressed in the semantic modeling elements of the GASTM and/or the semantic elements of future SASTM semantic objects.

Tools may work together to achieve both syntactic and semantic conformance. The diagram below illustrates the interoperability of two ASTM compliant tools that conform to Compliance Level 0 and Level 1 respectively and exchange information via XMI documents.

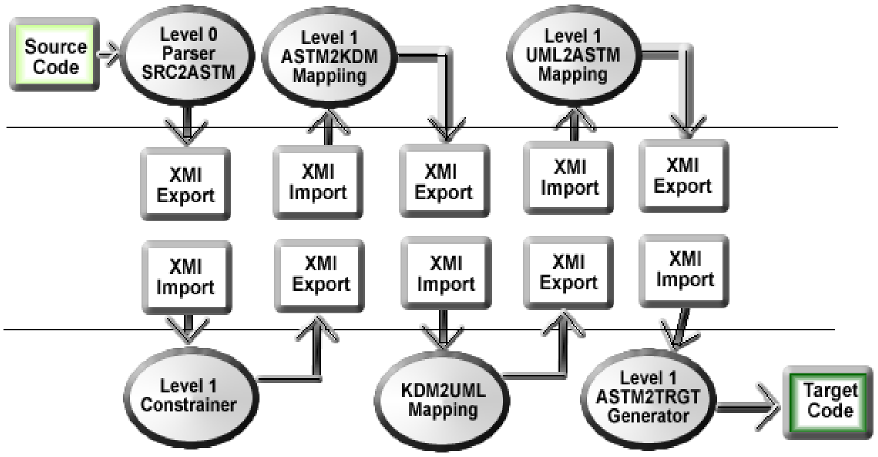


Figure 2.1 - ASTM Tool Interoperability

Figure 2.1 illustrates a classic ADM to MDA mapping scenario, in which a tool chain composed of ADM and MDA tools cooperatively carry out a complex transformation mapping from source code into target code interoperatively by interchanging a series of OMG XMI models.

In this diagram the parser is a Level 0 compliant tool. The parser analyzes source code to generate the abstract syntax elements for a language. The parser generates an ASTM model compatible with the syntactic elements of the GASTM. The parser exports the syntactic GASTM model via XMI through an Export API. In this diagram the Level 1 compliant tool is a constrainer. The constrainer imports the Syntactic XMI model produced by the parser through an Import API. The constrainer analyzes the abstract syntax tree produced by the parser and augments the abstract syntax with semantic elements. Subsequently, a Level 1 compliant tool ASTM2KDM maps the ASTM to the KDM model and exports its model via XMI to the KDM2UML tool via XMI interchange. Subsequently a Level 1 compliant UML2ASTM tool converts a UML model of a system into a Level 1 ASTM model, which is exported to a Level 1 ASTM2TRGT Generator via XMI. The ASTM2TRGT Generator generates Target Code.

Model Interchange via XMI generally needs to be undertaken only when a tool requires the services of some other tool. Agreement upon a common interchange format facilitates construction of tool chains that cooperate by interchanging models in commonly agreed upon formats. Model interchange via XMI can be skipped if a single tool can accomplish a complex task without the need for the services of some other tool.

Highly complex software tasks often require cooperation between tools, and tool chains can be rapidly assembled to accomplish complex tasks collaboratively within a tool ecosystem that interchanges models via the OMG ADM MRD.

Table 2.2 - Compliance Statements

Compliance Statement			
Compliance Level	Manipulation	Import API	Export API
L0	The capability to analyze and transform GASTM and SASTM syntactic models of existing applications that conform to the UML model.	The capability to import GASTM and SASTM syntax models based on the XMI schema.	The capability to export GASTM and SASTM syntax models based on the XMI schema.
L1	The capability to analyze and transform GASTM and SASTM syntactic models and semantic models of existing applications that conform to the UML model.	The capability to import GASTM and SASTM syntactic and semantic models based on the XMI schema.	The capability to export GASTM and SASTM syntactic and semantic models based on the XMI schema.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

- Unified Modeling Language (UML) 2. Infrastructure Specification
- Meta Object Facility (MOF) 2.0 Specification
- Architecture Driven Modernization Knowledge Discovery Meta-Model (KDM) 1.0 Specification

4 Terms and Definitions

See Annex B for this information.

5 Additional Information

5.1 How to Proceed

The rest of this document contains the technical content of this specification. Chapter 6 contains the Overview of the ASTM specification, including how the ASTM is envisioned to support the OMG Architecture Driven Modernization (ADM) Road Map (as defined in numerous published ADM whitepapers). It describes how the ASTM is intended to support other relevant OMG standards, and describes how the ASTM is envisioned to support the ADM Scenarios.

Chapter 7 develops and refines the core concepts and expresses them in a compact BNF format favored by computer language theorists for defining programming languages abstract syntax.

Chapter 8 continues to develop and refine the ASTM core concepts with UML diagrams with detailed description of the structure, meaning, and intended usage of the elements.

Annex A illustrates RDBMS Extensions, Annex B provides a relatively comprehensive glossary of terms and definitions that are widely used or understood within the computer science community to refer to the concepts that the ASTM models. Annex C is a bibliography of sources for the terms defined in Annex B.

5.2 Acknowledgments

The following companies submitted and/or supported parts of this specification.

Submitted by:

- EDS
- IBM
- Software Revolution
- TCS

Supported by:

- 88 Solutions
- Adaptive Technologies
- Blue Phoenix
- Composable Logic
- KDM Analytics
- Kestrel Institute
- Northrop Grumman
- SAIC
- Tactical Strategy Group

The following contributors either wrote or reviewed this specification.

AUTHORS

- Philip H. Newcomb, The Software Revolution, Inc.
- Ravindra Naik, Tata Consultancy Services

SUBMISSION TEAM CONTRIBUTORS

- Robert Couch, Senior Computer Scientist, The Software Revolution, Inc.
- Mark Purtill, Senior Computer Scientist, The Software Revolution, Inc.
- Luo Nguyen, Operations Manager, The Software Revolution, Inc
- Roger Knapp, VP Operations, The Software Revolution, Inc.
- Howard Ramsdell, Computer Scientist, The Software Revolution, Inc.
- Shrawan Kumar, Senior Computer Scientist, Tata Consultancy Services
- Hitesh Sajnani, Computer Scientist, Tata Consultancy Services
- Djenana Campara, KDM Analytics (formerly Klocwork)

- Nikolai Mansurov, KDM Analytics (formerly Klocwork)
- Tim W.Wilson, Distinguished Engineer, IBM Research
- Howard Hess, Distinguished Engineer, IBM Research
- Larry M. Hines PhD, Austin Innovation Centre, EDS.
- Michael K. Smith PhD, EDS Distinguished SE, Austin Innovation Centre, EDS
- Rich Cohen, Rich PhD, EDS Distinguished SE , Austin Innovation Centre, EDS.
- Barabara Erikson-Conner, EDS

ADM TASK FORCE REVIEWERS AND CONTRIBUTORS

- William Ulrich, Tactical Strategy Group, Inc.
- Smith, Jeff PhD, Composable Logic
- Vitaly Khusidman PhD, Director Enterprise Modernization, Unisys Corporation
- Vadim Pevzner, PhD, Director of Business Transformation, Unisys Corporation
- Mike Oara, CTO, Relativity Technologies, Inc
- Ira Baxter PhD, CEO, Semantics Designs
- Douglas Smith, PhD, Kestrel Institute
- Kimbrell, Roy E., Northrup Grumman
- Chris Caputo, Blue Phoenix
- Alain Picard, CTO, Benchmark Consulting,

OMG ARCHITECTURE REVIEWERS

- Sridhar Iyengar, IBM Distinguished Engineer, IBM
- Sumeet. S Malhotra, Unisys
- Peter Rivett, CTO, Adaptive Inc.
- Manfred R Koethe, CTO, 88solutions Corporation

DISTINGUISHED REVIEWERS

- Cordell Green PhD, Director, Kestrel Institute
- Chikofsky, Elliot, Entineering Management & Integration, Inc.
- David S. Frankel, David Frankel Consulting

6 Overview

6.1 Abstract Syntax Tree

Compilers, converters, and transformation tools represent programming language constructs, such as expressions, statements and loops, as a tree structure known as an "Abstract Syntax Tree" or AST. An AST provides a means for creating a representation of the executable software artifact. The AST is a formal representation of the syntactical structure of software that is more amenable to formal analysis techniques than is the concrete or surface syntax of software. Construction of ASTs typically involves the use of parsing technologies, but ASTs can also be constructed by means of a generation or derivation process from some other form or specification. AST model structures permit the expression of compositional relationships to other language constructs and provide a means of expressing a set of direct and derived properties associated with each such language construct.

The data structures from which the abstract syntax trees are composed provide an exhaustive collection of formal compositional elements for a language. These language constructs (or model elements) are generally defined in a type (or class) hierarchy. There are many ways to define these ASTs. The AST may be derived from an analytical process that can be applied to the surface syntax of the software asset or may be captured through a process that involves the application of rewrite rules to other data structures. For instance, a common or language-neutral AST model might be generated by the application of rewrite rules that generate a language specific AST model of some application, or a generic AST model might be generated directly from a UML class diagram or action diagram by means of a series of refinement rules. An AST may be an invertible representation. In other words, it may be possible to traverse the AST and reconstruct the "surface syntax" of the legacy system or reconstitute it in textual form from the abstract structures. An AST may be augmented, it may be analyzed and updated using additional structures that describe other properties about the software. Common analyses that augment an AST with additional properties include constraint analysis, data-flow analysis, control-flow analysis, axiomatic analysis, and denotational analysis. ASTs are generally augmented with additional analyses layers, such as type analysis, control-flow analysis, or data-flow analysis (to support code optimization). Augmentation may also support capture of software engineering metrics and documentation. Having a standard metamodel to represent ASTs will facilitate interchange at a foundational level for all architecture-driven modernization work. Hence, the AST provides an appropriate formalism for the derivation of properties required for detailed knowledge discovery.

Formally, in computer science, an abstract syntax tree (AST) is a finite, labeled, directed tree, where the internal nodes are labeled by operators, and the leaf nodes represent the operands of the node operators. Thus, the leaves have nullary operators, i.e., variables or constants. In computing, it is used in a parser as an intermediate between a parse tree and a data structure, the latter which is often used as a compiler or interpreter's internal representation of a computer program while it is being optimized and from which code generation is performed. The range of all possible such structures is described by the abstract syntax. An AST differs from a parse tree by omitting nodes and edges for syntax rules that do not affect the semantics of the program. The classic example of such an omission is grouping parentheses, since in an AST the grouping of operands is explicit in the tree structure.

In contrast, an abstract semantic graph (ASG) is a data structure used in representing or deriving the semantics of an expression a formal language (for example, a programming language). An abstract semantic graph is a higher level abstraction than an abstract syntax tree (or AST), which is used to express the syntactic structure of an expression or program. An abstract semantic graph is typically constructed from an abstract syntax tree by a process of enrichment and abstraction. The enrichment can for example be the addition of back-pointers, edges from an identifier node (where a variable is being used) to a node representing the declaration of that variable. The abstraction can entail the removal of details which are relevant only in parsing, not for semantics.

6.2 Abstract Syntax Tree Metamodeling Specification

The Abstract Syntax Tree Metamodel (ASTM) is the subject of this specification. The main purpose of Abstract Syntax Tree Metamodeling specification is to enable easy interchange of detailed software metadata between software development and software modernization tools, platforms, and metadata repositories in distributed heterogeneous environments. The Abstract Syntax Tree Metamodel defines a specification for modeling elements to express abstract syntax trees (AST) in a representation that is sharable among multiple tools from different vendors.

This specification defines a metamodel for representing information about existing software assets in the form of abstract syntax trees for those software assets. An ASTM Metamodel describes the elements used for composing AST models. An AST model is a model of how the statements of a software asset are structured and thus reflect the grammar of the particular programming language. An AST is thus a model of the formal structure, but not the language- specific form of expression of the program statements.

The ASTM specification mainly consists of definitions of metamodels software application artifacts in the following domains:

- Generic Abstract Syntax Tree Metamodel (GASTM): A generic set of language modeling elements common across numerous languages establishes a common core for language modeling, called the Generic Abstract Syntax Trees. In this specification the GASTM model elements are expressed as UML class diagrams.
- Language Specific Abstract Syntax Tree Metamodels (SASTM) for particular languages such as Ada, C, Fortran, Java, etc. are modeled in Meta Object Facility (MOF) or MOF compatible forms and expressed as the GASTM along with modeling elment extensions sufficient to capture the language.
- Proprietary Abstract Syntax Tree Metamodels (PASTM) express ASTs for languages such as Ada, C, COBOL, etc. modeled in formats that are not consistent with MOF, the GSATM, or SASTM. For such proprietary AST this specification defines the minimum conformance specifications needed to support model interchange.

The GAST may be derived from rewrite rules applied to a Specific Abstract Syntax Tree (SAST) or refinement rules applied to UML class diagrams. Proprietary formats (PASTs) may be mapped into the GASTM or GASTM/SASTM as illustrated in Figure 6.1.

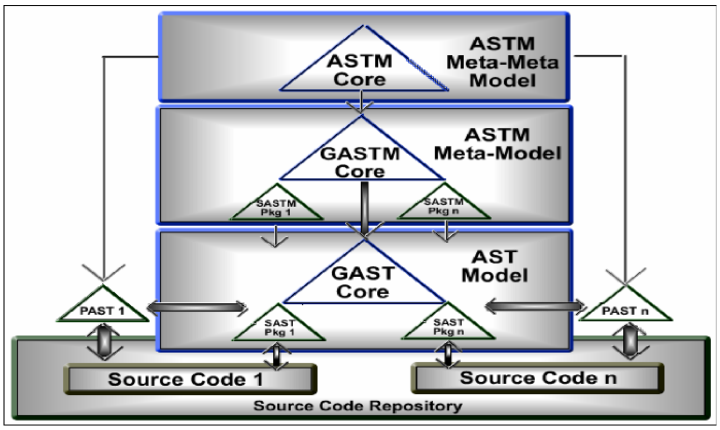


Figure 6.1 - ASTM Modeling Framework

6.3 ASTM Support and Complementation for MDA

Another key goal of the ASTM is to both support and complement the MDA by establishing a standard set of platform independent abstract syntax language elements as a common framework for analysis and for interchange of platform and language specific programming language metamodels. The ASTM is based on three key industry specifications:

- UML - Unified Modeling Language, an OMG modeling standard
- MOF - Meta Object Facility, an OMG metamodeling and metadata repository standard
- XMI - XML Metadata Interchange, an OMG metadata interchange standard

Collectively these three specifications provide foundation for object services, interchange services, and common repository facilities as illustrated in Figure 6.2.

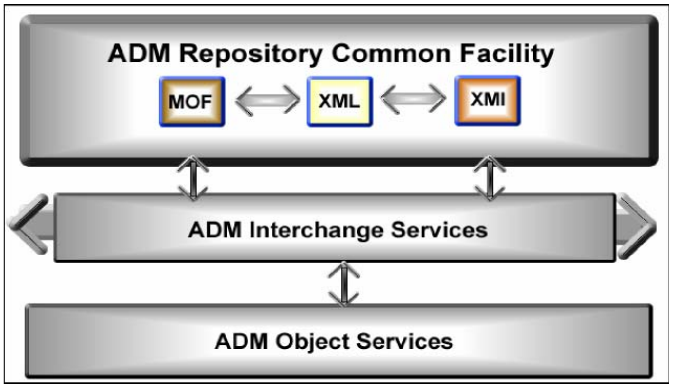


Figure 6.2 - OMG ADM Repository, Interchange, and Object Services

The OMG base architecture for Repository Common Facilities, Model Interchange Services, and Object Services are obtained from the OMG MOF model and supporting MOF based extensible frameworks, such as Eclipse. These three specifications form the core of the OMG Architecture Driven Modernization (ADM) Metadata repository (MDR) architecture illustrated in Figure 6.2. The ADM repository Metadata Repository architecture follows and replicates the OMG Metadata Repository architecture except as specialized for abstract modeling of software source code artifacts, support for the ADM Roadmap packages and the ADM Modernization Scenarios to be described below. The roles of the key architectural components of the OMG ADM Metadata Repository are described in greater detail in the section below.

6.4 ADM Meta Data Repository

The UML specification defines a rich, object oriented modeling language that is supported by a range of graphical design tools. The MOF specification defines an extensible framework for defining models for metadata, and providing tools with programmatic interfaces to store and access metadata in a repository. The ASTM extends MOF modeling to encompass several existing families of languages in a uniform way. The XMI specification allows metadata to be interchanged as streams or files with a standard format based on XML. XMI in particular lowers the barrier to entry for the use of OMG metadata specifications.

The complete architecture offers a wide range of implementation choices to developers of tools, repositories, and object frameworks. Key aspects of the architecture include:

- A four layered metamodeling architecture for general purpose manipulation of ADM metadata in distributed object repositories. See the MOF and UML specifications for more details.

- The use of UML notation for representing ADM metamodels and models.
- The use of standard information models (UML) to describe the semantics of object analysis and design models for ADM models.
- The use of MOF to define and manipulate ADM metamodels programmatically using fine grained CORBA interfaces. This approach leverages the strength of CORBA distributed object infrastructure.
- The use of XMI for stream based interchange of ADM metadata.

Collectively these specifications support an extensible framework for the definition of metamodels, access to, and interchange of these metamodels as various forms and levels of metadata by ADM Modernization Tools and Repositories. ADM Interchange Services provide mechanisms for the exchange of Metamodels and Models metadata. ADM Object Services provide mechanisms for persistent storage of this metadata (as illustrated in Figure 6.3).

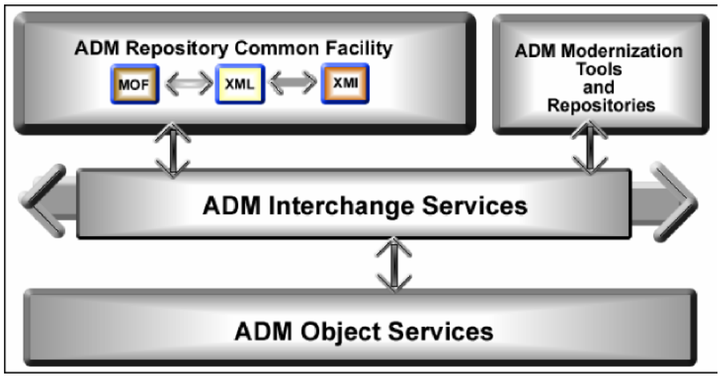

ADM Repository Common Facility
MOF
XML
XMI
ADM Modernization Tools and Repositories
ADM Interchange Services
ADM Object Services

Figure 6.3 - OMG ADM Meta Data Repository (ADM MDR)

6.5 ASTM Relationship to MOF

In the OMG Meta Object Facility (MOF) Abstract Syntaxes are nested meta-modeling levels with each layer defining a set of elements and conformance constraints for each subordinate layer. The nesting of the MOF metadata enables each layer in a MOF model to serve as the meta layer for its subordinate model layers. Thus, M3 is the Abstract Syntax of M2 Abstract Syntax Trees. M2 is the Abstract Syntax of M1 Abstract Syntax Trees. M1 is the Abstract Syntax of M0 Abstract Syntax Trees MOF uses UML Class diagrams to define Abstract Syntax. MOF platform independence comes from its use of generators that produce software for managing models that conform to meta-models.

The ASTM in this specification is defined as a MOF model. While MOF models are constrained by the compositional, relational, and Object Constraint Language constraints, ASTM meta-models are additionally constrained by grammar specification constraint checkers that impose restrictions upon the relationships between language constructs derived from the compositional constraints defined upon syntactic model elements of the language. Thus, many perfectly acceptable MOF Abstract Syntax models would be considered malformed by the grammar specification system constraint checkers that are commonly used for composing ASTM models. While most well-formed grammar specifications acceptable to compilers or parser generators are likely to be MOF compatible, MOF permits more flexible construction of models that would be regarded as incomplete, inconsistent, or contradictory by constraint checkers for ASTM models.

Table 6.1 - The ASTM MOF Relationship

Layer	Description	ADM Examples
Meta-metamodel M3	MOF (i.e., the set of constructs used to define metamodels)	MOF Class, MOF Attribute, MOF Association, etc.
Metamodel M2	Metamodels consisting of instances of MOF constructs	KDM UML profiles GASTM UML profile SASMT UML profile
Model M1	Models consisting of instances of AS Model of COBOL language M2 metamodel constructs	KDM Data Model
Instances examples) M0	Objects and data (i.e., instances of M1 model constructs)	AST model instances of source code of real applicationKDM Data models instance of data base or data files

Combining MOF and the Language Parsing Approaches to Abstract Syntax and ASTs is highly powerful. Abstract Syntaxes defined for Language Parsers can usually be modeled as MOF Models using MOF Tools. Once the MOF Models for language models exist, MOF has generators to create XML, Java, and CORBA API support for these models.

MOF allows the model instances (ASTs) of software applications to be modeled and interchanged via MOF Repository technology. MOF allows the exchange of AS (metadata) and ASTs (i.e., enterprise applications treated as data) with full machine automation.

Without MOF, the manipulation of applications in Abstract Syntax form will remain proprietary with limited penetration. With MOF, the manipulation of programs as data (ASTs) will become universal and many hard software problems will be solved efficiently and economically. ASTM extends MOF modeling to encompass several existing families of languages in a uniform way. A generic set of language modeling elements common across numerous languages establishes a common core for language modeling, called the Generic Abstract Syntax Trees (GAST). Language Specific Abstract Syntax Trees (SAST) for particular languages such as Ada, C, Fortran, Java, etc. must be modeled in MOF or MOF compatible forms. The transformation between SAST GAST must be demonstrated without loss of meaning even though their abstract syntax model changes during transformation between languages. The ASTM supports transformation between GAST to SAST and transformation between SAST and GAST, and replicates and extends the MDA approach of generating platform specific model from platform independent models.

6.6 ASTM Support for ADM Roadmap

The ASTM is supportive of the following ADM Roadmap specifications:

- KDM - Architecture-Driven Modernization Knowledge Discovery Meta-Model, an OMG modeling specification

The ASTM is supportive of five additional ADM Roadmap specifications that are undergoing definition, or will be defined in future years. The ASTM also provides a language-neutral meta-model to support these future ADM specifications.

1. The Analysis Package (AP) facilitates the examination of structural meta-data with the intent of deriving behavioral meta-data about systems that may take the form of business rules or other aspects of a system that are not part of the structure of the system, but are rather semantic derivations of that structure and the data.

2. The Metrics Package (MP) derives metrics from the ASTM and AP to describe various system attributes. These metrics convey technical, functional, and architectural issues for the data and the procedural aspects of the applications of interest.
3. The Visualization Package (VP) depicts application meta-data stored within the AP and the KDM in any variety of views as may be appropriate or useful for planning and managing modernization initiatives.
4. The Refactoring Package (RP) defines ways in which the AP, MP, VP, KDM, and ASTM can be used to refactor applications. This includes structuring, rationalizing, modularizing, and other ways of improving existing applications without redesigning those systems or otherwise deriving model-driven views of those systems.
5. The Target Mapping & Transformation Package (TMTP) defines mappings between the KDM, ASTM, AP, MP, and RP models. This specification defines the mappings and transformations that may occur between existing applications and top down, target models.

The overall structure of the ADM Metadata Repository Architecture is depicted in Figure 6.4. The ASTM is highlighted to illustrate its position within this structure.

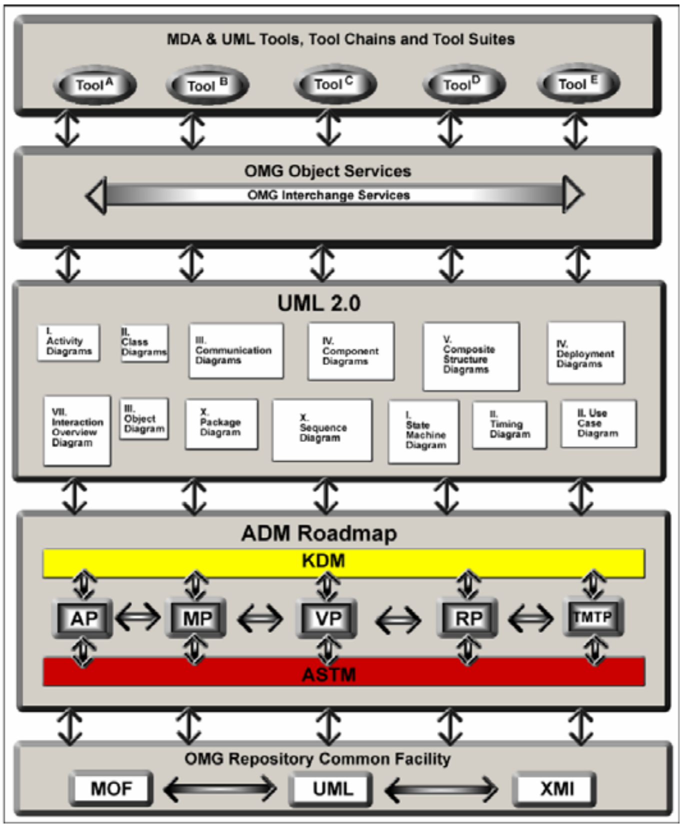


Figure 6.4 - ADM Metadata Repository Architecture

6.6.1 ASTM Complements the KDM

The first specification, the Knowledge Discovery Metamodel, provides a means for capture of information from multiple sources and its meta-model provides a comprehensive view of application structure and data, but does not represent software below the procedure level.

The Architecture-Driven Modernization Knowledge Discovery Meta-Model (KDM) specification provides a comprehensive high-level view of the application behavior, structure, and data, but does not represent application models below the procedural level. The ASTM models constructs within programming languages, while the KDM models structural, behavioral, and data information about application portfolios. ASTM is one of the sources of information for the KDM. The ASTM is expected to complement the KDM as an element of the ADM Roadmap to express fully detailed models of application artifacts.

The ASTM complements the KDM by modeling detailed syntactic structures using generic model elements common to numerous languages, as well as using specialized modeling elements specific to particular languages. Combining the KDM and the ASTM enables the exchange of metadata for applications written in multiple programming languages with full machine automation. The ASTM establishes the necessary fine-grained models of application artifacts required for detailed analysis and visualization, refactoring, target mapping, and transformation. Figure 6.5 depicts the relationship between the system itself, the ASTM, and the KDM.

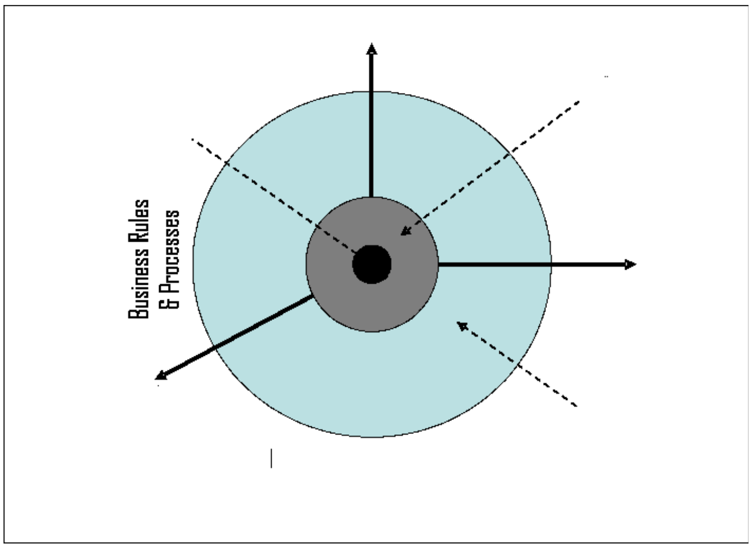


Figure 6.5 - ASTM & KDM Complementarity

In the figure above, the level of abstraction increases as you travel out from the center of the diagram. The ASTM does not duplicate the KDM, but complements it. Further, the ASTM is one of many sources of information for populating the KDM. The ASTM extends the KDM to support the creation of comprehensive and detailed models of systems.

Standardizing the format of AST structures, representation, and interchange of AST models will complement the KDM by completing a comprehensive model for the exchange of application models tools that would otherwise be prohibitive to justify and will insure that the aggregate of vendor tools provides a comprehensive architecture-driven modernization capability. A standard KDM complemented by a standard ASTM will enable a user of the technology to bring together a variety of best-of-breed products to analyze, visualize, re-factor, and transform legacy applications between application modernization tools. It will enable vendors to develop specialized modernization.

Table 6.2 illustrates the high-level correspondence mappings between elements of the ASTM and the KDM. Development of the detailed mapping and mechanisms for automating the mapping between the KDM and the ASTM and the ASTM and the KDM will be left to vendors who subscribe to these specifications. There are many more objects in the ASTM than listed in the table below, and there are far more elements in the ASTM than are listed in the table below. Not all objects of the ASTM will have a mapping to objects of the KDM, since the KDM models high-level concepts that are not necessarily implemented in the code, or if implemented in code, not necessarily by individual low-level statements of the code.

Table 6.2 - AST to KDM Mapping

ASTM	KDM	ASTM	KDM
Comment	CommentUnit	Block	Compound
SourceFile	SourceFile	SwitchStatement	Switch
SourceLocation	SourceRef	Add	Add
Project	Project	Subtract	Subtract
FunctionDeclaration	CallableUnit	Multiply	Multiply
FunctionDefinition	CallableUnit	Divide	Divide
VariableDeclaration	StorableUnit	Modulus	Remainder
VariableDefinition	StorableUnit	And	And
PreprocessorElement	PreprocessorDirective	Or	Or
NameSpaceDefinition	Namespace	Equal	Equals
IncludeUnit	IncludeDirective	NotEqual	NotEqual
MacroDefinition	MacroUnit	Greater	GreaterThan
MacroCall	MacroDirective	Less	LessThan
DataType	Datatype	NotLess	GreaterThanOrEqual
FunctionType	CallableKind	NotGreater	LessThanOrEqual
PrimitiveType	PrimitiveType	BitAnd	BitAnd
Void	VoidType	BitOr	BitOr
Integer	IntegerType	BitXor	BitXor
Float	FloatType	BitLeftShift	LeftShift
Character	CharType	BitRightShift	RightShift
String	StringType	Assign	Assign
Boolean	BooleanType	PostIncrement	Incr
IntegerLiteral	integer-literal	PostDecrement	Decr
StringLiteral	string-literal	ActualParameter	ParameterUnit
CharLiteral	character-literal	NewExpression	New
RealLiteral	real-literal	RangeExpression	RangeType
BooleanLiteral	boolean-literal	AnnotationExpression	Annotation
BitLiteral	bitstring-literal	EnumType	EnumeratedType
AggregateType	RecordType	ConditionalExpression	Condition
		FunctionCallExpression	Call
		ReturnStatement	Return
		JumpStatement	Goto
		ThrowStatement	Throw
		ArrayReference	ArraySelect
		FieldReference	FieldSelect
		CastExpression	TypeCast

6.6.2 ASTM Support for the KDM

From the KDM perspective the ASTM is one means of populating the KDM. The ASTM extends the KDM to support comprehensive and detailed modeling of systems. From ASTM perspective, the KDM as well as the ADM Roadmap are MDA MOF models the ASTM populates from its highly detailed and precise models of systems. The ASTM regards the KDM is one source for information about systems that can guide large-grained analysis, metrics, visualization, model mapping, transformations, and refactoring that the ASTM supports. The ASTM does not directly depend upon or intersect with the KDM for any part of its meta-model definition. Together, the ASTM and the KDM provide high fidelity support for ADM scenarios when effectively combined.

Users of the KDM and the ASTM regard modernization from somewhat different perspectives. A KDM user regards transformation as an augmentation strategy that is supported by a better understanding application data, structure and behavior, and architecture and is undertaken to make legacy systems more reliable and adaptable. Transformation is a technique needed when extracting and rationalizing data definitions, data and business rules, redesigning, and reusing legacy rules and data within the context of strategic enterprise architecture.

From an ASTM User’s perspective modernization is a direct strategy that includes model driven mapping of data, structure, and behavior between language feature sets (language translation), model driven restructuring of language feature sets with replacement of undesirable features with reliable and adaptable features (application refactoring), model driven rationalizing of data definitions, data and business rules by abstraction to MDA data views (models) that support model-driven regeneration of specific language features, model driven re-architecting of systems by abstracting design patterns and applying generation, transformation, and refactoring to regenerate redesigned and re-architected enterprise applications.

6.6.3 The ADM Metadata Repository Services

The ASTM lays a foundation of repository services for the subsequent specifications in the ADM Roadmap. Compactness and uniformity is achieved for the ADM Roadmap specifications by basing them upon the GASTM, the language-neutral set of common AST modeling constructs. Application of these services to specific languages is achieved by the definition of a standardized approach to defining mappings between the single generic GAST upon which ADM Roadmap packages are based, and the myriad specialized forms of SAST used for modeling specific languages. Extension of ADM Roadmap Services from the GAST to SAST is achieved by association relationships between the generic language constructs of the GAST and the specific language constructs of the SASTM modeling constructs. Construction of GASTM and SASTM models requires use of MOF and UML modeling technology. Internet Interchange of metadata conformant with these metamodels is typically accomplished by XML or CORBA brokerage services hypertransport facilities. The transport, management, or interchange of ADM metadata within tools (inside the box) is accomplished by vendor-specific utilities optimized for performance and efficiency. Realization of GASTM and SASTM models results in GAST and SAST instances that constitute persistent high fidelity models of software systems and applications, however this technical modeling approach is not limited to software artifacts alone.

6.6.4 The ASTM Support for Multiple Language Types

In particular, the ASTM is capable of representing a broad range of software languages and language types including 2GL, 3GL, 4GL, and 5GL languages such as Ada, Assembler, C, C#, COBOL, FORTRAN, Java, Natural, Power Builder, Refine, SQL, etc. Techniques used for defining GASTM and SASTM replicates the form of detailed language analysis required for compiler construction, but the persistent models realized from GASTM and SASTM metamodels are often much richer in information content than the models used by compilers for translation from higher level languages into machine code or byte code.

The ASTM Extensibility

Generally speaking, any OMG MOF metamodel defines an abstract syntax for conformant model instantiations. More broadly the term “domain modeling” is generally applicable to the process of defining formal languages that describe specific areas of knowledge that may be fully described by specialized vocabularies, syntax, and associated semantics. The ASTM is an open ultra-wide spectrum architecture capable of supporting AST metamodel definition and extension, metadata interchange and transformation and object services across any language that can be expressed using abstract syntax as an intermediate representation or expression of the concrete (surface level) language in which knowledge is commonly expressed.

Role of the ASTM in the ADM Metadata Repository Architecture

An essential element of the ASTM architecture is the definition of a single universal ASTM model consisting of a core, GASTM, accompanied by a set of separately defined extensions, SASTM, for expressing the specialized abstract syntaxes of specific languages. A single SASTM may provide language elements for multiple specific languages or dialects, but should be restricted to the establishment of languages elements for languages within a particular language family. A SASTM could, for example, be defined for ‘rule-based’ languages by defining a set of modeling elements for expressing cause-effect rules consisting of conditions and actions associated with conditions with the set of modeling elements used to represent ‘expressions’ of conditions provided by the ‘Expression’ elements of the GASTM.

In principle, the GASTM in combination with the set of SASTMs provides a set of universal language constructs adequate for the expression of all languages. AST modeling is a common practice, and vendors and compiler developers have developed AST models for many languages. The model definitions for all such proprietary AST models are called proprietary abstract syntax trees (PAST). The meta-models for all such PASTs are called PAST meta-models, or PASTMs. The ASTM, therefore consists of three kinds of meta models, the GASTM, SASTM, and the PASTM, where all such models must be expressed as MOF models to become interchangeable. The GASTM is the language neutral and vendor neutral core. The SASTM is a set of extensions of the GASTM, which are specialized to specific languages or families of languages. The PASTMs are vendor-specific proprietary models. If a vendor wishes to interchange his PASTM as a vendor-neutral representation it should be transformed or mapped into a GASTM+SASTM. Figure 6.6 illustrates this concept.

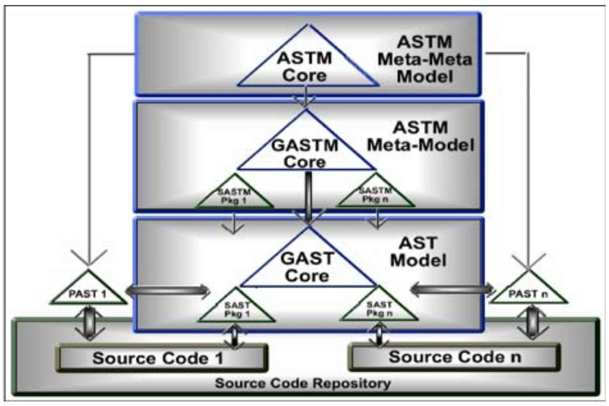


Figure 6.6 - AST Metamodel and Metadata Repository

The GAST+SAST is capable of faithfully capturing without loss of underlying meaning the specialized abstract syntax of the PASTs defined for specific languages. The figure above illustrates the relationship between AST metamodels (GASTM, SASTMs, PASTMs), models (GAST and SASTs, PASTs) and source code artifacts. This architecture requires that mappings be defined between the PAST to the GAST+SAST to re-express the language elements of a specific (or proprietary) language in terms of the language-neutral elements of the GAST+SAST. Conversely mappings must be defined between the GAST+SAST and a specific (or proprietary) PAST to re-express the language elements of the generic language in terms of the language elements of a specific (or proprietary) language. For some classes of languages the GASTM may be sufficient, in and of itself and without an SASTM, for expressing the language elements of the language.

The mapping between the SAST+GAST and the PAST must be demonstrated without loss of meaning even though their abstract syntax models changes during this transformation or mapping. Separation of concerns dictates the PASTM be distinct from the GASTM+SASTM and necessitates that this mapping or transformation process be defined between the PAST and the GAST+SASTM.

In practice, the plethora of PASTM models that already exist originated as a consequence of the separate concerns of each of their creators. Capturing the abstract syntax of a language in the form of an AST models is a highly complex undertaking that requires concern for efficiency of the parsing and constraining process. It is essential that the capture of the specific details of a particular language not be encumbered by the need for conformance to a fixed reference model such as the GAST+SAST during the definition of the PAST for any specific language. New languages are constantly being invented as part of the natural evolution and on-going innovation of programming languages. Moreover there are degenerate and obsolete syntactical forms in many language that are unique, and there are undesirable model elements that are excluded from the GAST+SAST by design or by its state of maturity. Modeling of PAST models entails the utilization of abstract syntax tree elements for modeling unique language forms for which modeling elements may not exist (and might never exist) within the GASTM and the SASTM. Such language unique elements of PAST models must be ‘transformed’ into standard element forms in the GAST+SAST.

The ASTM Provisional Extensibility

While the ASTM standard provides support specifically for several categories of languages, the standard must provide a process for community extension when GASTM constructs are required to model languages to which the GASTM has not been previously extended by the formal standardization process.

For example, Domain Specific Languages (DSL) are examples of languages with vocabularies, syntax, and supporting semantics specialized to a specific purpose to expedite the expression of solutions within a limited problem domain or used to define a particular aspect of a problem solution. While it is always possible to define a SASTM for any DSL, it is possible to conceive of a DSL containing one or more language constructs that cannot be mapped into the generic language constructs previously defined within the GASTM. Therefore it is common, in practice, for additional language constructs to be introduced into the GASTM on a frequently recurring basis. It is not possible to establish a conclusive set of GASTM constructs, nor is it desirable to do so, because the possible set of language neutral language construct types is indefinitely extensible.

Therefore the ADM TF must provide for user extensions to the GASTM, and establish a process for regular review, and publication of extended GASTM models.

Business Value of the ASTM and ADM Metadata Architecture

The ASTM greatly reduces the complexity of the technology required to support many modernization scenarios. In particular it significantly reduces the problem of translation or transformation between multiple different programming languages. Considering the set of all economically interesting source (S) and target (T) languages, the ASTM reduces the O (S * T) transformation problem to an O(S+T+G), where S is the number of source languages (or models) and T is the number of target languages (or models) + G the GASTM (G).

Standardizing the format of AST structures, representation and interchange of AST models provides a foundation for a comprehensive model for the exchange of application models between application modernization tools that enables vendors to develop specialized modernization tools that fit within MOF compliant tool frameworks and insures that the aggregate of vendor tools provides a comprehensive modernization capability. A standard ASTM establishes the foundation for the architecture of an ADM Metadata Repository that will enable a user of the technology to bring together a variety of best-of-breed products to analyze, visualize, refactor, and transform legacy applications.

6.7 ASTM Support for OMG Specifications

The GAST is used to support other OMG modeling specifications by providing a generic set of language modeling elements as the basis of OMG model (AST) derivation and as the basis for OMG language generation. The GASTM is effectively an Ultra Wide Spectrum Intermediate Language Model which 2GLs, 3GLs, 4GLs, and 5GL languages. Existing OMG generators for MOF models are limited to Java or C++ for behavior support, and XML Schema and DTDs for structure support. Retargeting OMG MOF generators into the GASTM will extend MDA support to a much broader spectrum of target languages than is currently supported by MOF technology. Application Models and Meta-Models (AST and AS) will be sharable among multiple tools from different vendors with much more uniform support for analysis, visualization, re-factoring, target mapping, and transformations across multiple languages. All ADM tools must adhere to the specifications for software modeling, as defined by the Object Management Group (OMG).

6.8 Role of the ASTM in the ADM Metadata Repository Support Services

The ADM Metadata Repository Support Services and Architecture are defined by seven interrelated specifications, starting with the KDM followed by the ASTM. At the time of this submission none of the service layers of the ADM Roadmap had been adopted as OMG specifications. The ASTM is envisioned as the kernel for the ADM Metadata Repository Support Services providing a set of modeling elements and facilities that interface directly with the source code artifacts that are the focus of the ADM Modernization Scenarios. The KDM is envisioned as the husk of the ADM Metadata Repository providing a multi-dimensional collection of modeling elements and modeling services that enable ADM Modernization to be effectively utilized by the enterprise in planning and executing ADM Modernization Scenarios. The remaining 5 ADM specifications define services for Analysis, Metrics, Visualization, Refactoring, Target Mapping & Transformation. The roles of the 7 ADM specifications and their relationship to the ASTM are defined below.

6.8.1 ASTM Support For ADM: Knowledge Discovery Meta-Model Package

The KDM Package establishes an initial meta-model that allows modernization tools to exchange application meta-data across applications, languages, platforms, and environments. This initial meta-model provides a comprehensive view of application structure and data, but does not represent software below the procedure level. The KDM RFP has been issued and six companies have submitted a total of four responses.

Abstract Syntax Tree Metamodel (ASTM)

One particular way ADM work will benefit from standardizing ASTs is by providing a foundation for other ADM sts. ASTs are one of many sources of information for populating the Knowledge Discovery Metamodel (KDM). The KDM provides a comprehensive view of application behavior, structure and data, but does not represent the detailed syntactic structures of programming artifacts.

ASTM SUPPORT: The ASTM supports the KDM by providing a language neutral framework for derivation of KDM models to the extent that these models can be derived by automation. The use of the ASTM establishes a task complexity, O, for defining KDM support for a set of languages S to be O(M(S) + KDM(G) + KDM(S)) where M(S) is the effort to map each language into the ASTM and KDM(G) is the effort to provide a set of Reusable KDM functions based upon the GASTM for the set of languages, and KDM(S) is the effort to provide language specific KDM functionality for each specific language specializations.

A set of language neutral Mapping and Transformation relationship defined between the GASTM and the KDM meta models are complemented by transitive mappings onto language specific SASTMs or directed mappings between SASTM and KDM meta models when specialization is required.

6.8.2 ASTM Support For ADM: Abstract Syntax Tree Metamodel Package

This ASTM builds upon the KDM Package in order to represent software below the procedural level. This effort will allow the KDM to fully represent applications and facilitate the exchange of granular meta-data across multiple languages. This version of the KDM establishes the foundation for subsequent analysis, visualization, and transformation specifications.

ASTM SUPPORT: This document provides a partial description of how the ASTM supports the ADM Roadmap packages.

6.8.3 ASTM Support For ADM: Analysis Package (AP)

The Analysis Package creates a standard that facilitates the examination of structural meta-data with the intent of deriving behavioral meta-data about systems. This behavioral meta-data may take the form of business rules or other aspects of a system that are not part of the structure of the system, but are rather semantic derivations of that structure and the data.

This ASTM supports the Analysis package by providing a language neutral framework for derivation of Analysis Package models to the extent that these models can be derived by automation. The use of the ASTM establishes a task complexity, O, for defining AP support for a set of languages S to be O(M(S) + AP(G) + AP(S)) where M(S) is the effort to Map each language into the ASTM, AP(G) is the effort to provide a set of language-neutral Reusable Analyses Package functions based upon the GASTM for the set of languages, and AP(S) is the effort to provide language specific Analysis Package functionality for the language unique features of each of the specific languages specializations.

The set of language neutral Mapping and Transformation relationship defined between the GASTM and the AP meta models are complemented by transitive mappings onto language specific SASTMs or directed mappings between SASTM and AP meta models when specialization is required.

6.8.4 ASTM Support for ADM: Metrics Package (MP)

The focus of the Metrics Package is to derive metrics from the KDM that can describe various system attributes. These metrics convey technical, functional, and architectural issues for the data and the procedural aspects of the applications of interest. These metrics support planning and estimating, ROI analysis and the ability of analysts to maintain application and data quality. The task force envisions gaining validation for the metrics package by working with industry and academic resources.

ASTM SUPPORT: This ASTM supports the Metrics package by providing a language neutral framework for derivation of Metrics Package models to the extent that these models can be derived by automation. The use of the ASTM establishes a task complexity, O, for defining MP support for a set of languages S to be O(M(S) + MP(G) + MP(S)), where M(S) is the effort to Map each language into the GASTM, MP(G) is the effort to provide a set of language-neutral Reusable Metrics Package functions based upon the GASTM for the set of language, and MP(S) is the effort to provide language specific Metrics Package functionality for the language unique features of each of the specific languages specializations.

The set of language neutral Mapping and Transformation relationship defined between the GASTM and the AP meta models are complemented by transitive mappings onto language specific SASTMs or directed mappings between SASTM and MP meta models when specialization is required.

6.8.5 ASTM Support For ADM: Visualization Package (VP)

The Visualization Package focuses on ways to depict application meta-data stored within the KDM. This may include any variety of views as may be appropriate or useful for planning and managing modernization initiatives. Examples include the use of graphs or charts, metric summaries, or standardized development models.

The set of language neutral Mapping and Transformation relationship defined between the GASTM and the AP meta models are complemented by transitive mappings onto language specific SASTMs or directed mappings between SASTM and MP meta models when specialization is required.

ASTM SUPPORT: This ASTM supports the ADM Visualization Package by providing a language neutral framework for derivation of Visualization Package models to the extent that these models can be derived by automation. The use of the ASTM establishes a task complexity for defining VP support for a set of languages S to be O(M(S) + G + VP(S)) where M(S) is the effort to Map each language into the ASTM, G is the effort to provide a set of Reusable Visualization Package functions based upon the GASTM for the set of all language, and VP(S) is the effort to provide language specific Visualization Package functionality for the language unique features of specific languages.

6.8.6 ASTM Support For ADM: Refactoring Package (RP)

The Refactoring Package defines ways in which the KDM can be used to refactor applications. This includes structuring, rationalizing, modularizing, and in other ways improving existing applications without redesigning those systems or otherwise deriving model-driven views of those systems. Work on the Refactoring Package STANDARD will begin after issuance of the Visualization Package STANDARD.

ASTM SUPPORT: This ASTM supports the ADM Refactoring Package by providing a language neutral framework for derivation of Refactoring Package Models to the extent that these models can be derived by automation. The use of the ASTM establishes a task complexity for defining RP support for a set of languages S to an O(M(S) + RP(G) + RP(S)) where M(S) is the effort to Map each language into the ASTM, G is the effort to provide a set of Reusable Refactoring Package functions based upon the GASTM for each language, and RP(S) is the effort to provide language specific Refactoring Package functionality for language unique features for specific languages.

The set of language neutral Mapping and Transformation relationship defined between the GASTM and the RP meta models are complemented by transitive mappings onto language specific SASTMs or directed mappings between SASTM and RP meta models when specialization is required.

6.8.7 ASTM Support For ADM: Target Mapping & Transformation Package (TMTP)

The Target Mapping & Transformation Package defines mappings between the KDM, ASTM(GASTM ⇔ SASTM) AP, MP, and RP models. This standard defines the mappings and transformations that may occur between existing applications and top down, target models. Development paradigms may vary, but will include MDA as a target. This standard will complete ADM task force efforts in providing a transformational bridge between existing systems and target architectures.

ASTM SUPPORT: ADM Target Mapping & Transformation Package defines a standard defining and deriving Target Mapping & Transformation Package models to the extent that these models can be derived by automation. The ASTM establishes a standard language neutral framework for the ADM Package Models to which the TMTP is applied to support transformations and mappings between the ADM Package Models and other OMG models.

6.9 ASTM Support for the ADM Scenarios

The ADM Modernization Scenarios have been defined by the ADM task force in order to provide guidelines for envisioning all potential ADM applications. The ADM Scenarios help a user determine the tasks, tools, and use of the ADM, provide templates for crafting project objectives, plans, and related deliverables, defines tasks necessary to complete a given modernization initiative, and omit unnecessary tasks that would not apply to such a scenario. The Scenarios allow a user to pinpoint the types of tools necessary to perform these tasks, identifies the universe of modernization scenarios and tasks, and provides a guide as to the role of the ADM within modernization in general.

The ADM Modernization Scenarios

I. Application Portfolio Management

II. Application Improvement

III. Language-to-Language Conversion

IV. Platform Migration

V. Non-Invasive Application Integration

VI. Services Oriented Architecture Transformation

VII. Data Architecture Migration

VIII. Application & Data Architecture Consolidation

XI. Data Warehouse Deployment

X. Application Package Selection & Deployment

XI. Reusable Software Assets / Component Reuse

XII. Model-Driven Architecture Transformation

The ADM Modernization Scenarios are used to

- define an approach to various application improvement, migration, and redesign initiatives that an organization may pursue,
- pinpoint the types of tools necessary to perform these tasks, and
- mix and match Scenarios based upon the organization's Modernization goals.

For example, Language-to-language conversion might be coupled with a platform migration. The ADM has defined 13 Modernization scenarios for 12 of which the ASTM relationship is outlined below. Other scenarios may be added to this list from time to time. This section discusses the support the ASTM provides for the ADM Scenarios.

The following section outlines each of the ADM Modernization Scenarios and discusses how the ASTM supports each of them.

6.9.1 Application Portfolio Management (Scenario I)

Organizations must manage application systems as business assets and this requires portfolio analysis and management. Whether driven by internal audit requirements or government regulations, accounting of information assets is essential. This scenario captures and exposes technical and functional meta-data on an organization's information systems. Further, many old systems are poorly documented and this scenario addresses this shortcoming. The need for such a scenario is characterized as follows.

- There is no documentation depicting information flows, system-wide data usage, or overall systems architecture.
- Management requires detailed accounting of information systems to fulfill audit or regulatory requirements (e.g., Sarbanes-Oxley, Basil Accord).
- IT systems will undergo major changes or are to be outsourced.
- Modernization projects require baseline information to segment a portfolio into functional work units and provide input to an overall strategy.

This scenario entails running systems analysis tools and augmenting the resulting meta-data with analyst expertise to create a reliable knowledge base on existing systems. Meta-data would be stored in the ADM Meta Data Repository in the ADM Package Models, which is central to the effective and ongoing ability to manage information systems as organizational assets. Specifically, the ADM MDR would facilitate the following portfolio management activities:

- Serve as repository of cross-systems and cross-platform application meta-data.
- Enable analysts to visualize the relationship among business processes and related business artifacts with application artifacts.
- Enable analysts to incorporate evolving application and related meta-data as it changes over time.
- Allow analysts to expose ADM Package meta-data through visualization tools.
- Support distributed or centralized meta-data viewing, including the ability to scale up or down based on organizational size, structure, or reporting requirements.
- Support the federation of the ADM Package, based on organizational structure, size, or reporting requirements.

ASTM SUPPORT: This ASTM supports the ADM Automated Portfolio Management (APM) scenario by providing a language neutral framework upon which APM Tools can provide a uniform and high level of automated support. The use of the ASTM establishes a task complexity, O, for defining APM support for a set of languages S to be O(M(S) + (APM(G) + APM(S)) where M(S) is the effort to Map each language into the ASTM, APM(G) is the effort to provide a set of Reusable APM functions based upon the GASTM for each language, and APM(S) is the effort to provide language specific APM functionality for language unique features for specific languages.

6.9.2 Application Improvement (Scenario II)

The application improvement scenario is a "super scenario" comprised of several sub-scenarios. The goal of this scenario is to improve the robustness, integrity, quality, consistency, and/or performance of applications. Activities include the correction of program or system flaws (e.g., recursion), source code restructuring, data definition rationalization, or field size standardization. Organizations may address some or all of these issues depending on their objectives and basic weaknesses in the applications of interest. This scenario involves no architectural modifications and is based on the following requirements:

- A growing upgrade request backlog cannot be met due to system quality.
- The system is experiencing high failure rates or reliability problems.
- There is a long learning curve, poor IT responsiveness, and user dissatisfaction.
- System upgrade costs are not proportionate to business returns.
- One or more systems are being prepared to be outsourced or are being brought back in-house.
- Management has given up even trying to change the applications.

- Field expansion is required based on specific business requirements (e.g., uniform bar code, phone number, or revenue growth).

The portfolio analysis and asset management scenario may precede this scenario, but this is not essential. ADM Packages supports this scenario as follows.:

- Store and associate application and business meta-data associated with the systems of interest for the purposes of planning, management and execution.
- Expose system and program weaknesses discovered by system analysis tools.
- Provide planning input to improvement tasks by linking system weaknesses with business needs, particularly across systems, platforms and languages.
- Assist analysts with the process of rationalizing system-wide data names, attributes, records, segments and tables.
- Assist with the change management throughout the improvement process.
- Facilitate the tracking of system meta-data as it changes through the system improvement process.
- Streamline the planning and execution of the validation and verification stage of the project.

ASTM SUPPORT: This ASTM supports the ADM Application Improvement (AI) scenario by providing a language neutral framework upon which AI Tools can provide a uniform and high level of automated support. The use of the ASTM establishes a task complexity, O, for defining AI support for a set of languages S to be O(M(S) + (AI(G) + AI(S)) where M(S) is the effort to Map each language into the ASTM, AI(G) is the effort to provide a set of reusable AI functions based upon the GASTM for each language, and AI(S) is the effort to provide language specific AI functionality for language unique features for specific languages.

6.9.3 Language-to-Language Conversion (Scenario III)

This scenario involves converting one or more information systems from one language to another language. The language-to-language conversion scenario addresses the physical need to move from one language to another. This may be driven by a variety of factors, but does not involve a redesign of the application functionality beyond that which is required by the language change itself. This scenario is driven by the following needs:

- A language has become obsolete, is no longer vendor supported, is no longer understood by available programming talent or is just too hard to change.
- There is a business requirement to enhance the functionality of the current system but the language is no longer supported or readily adapted to change.
- Systems must move to a new platform and the new platform does not run the existing language or particular version of that language.
- A baseline system must be established from which current applications may be migrated to a strategic architecture.

ADM Packages play the following role in the language-to-language conversion scenario:

- Tracks system artifacts through the planning, phasing, and staging of the conversion effort.
- Facilitates discovery of high risk issues such as the use of runtime libraries or language constructs not available in the target environment.
- Assists with the change management process as the conversion proceeds.
- Streamline the planning and execution of the validation and verification stage of the project.

ASTM SUPPORT: This ASTM supports the ADM Language To Language (L2L) scenario by providing a language neutral framework upon which L2L Tools can provide a uniform and high level of automated support. The use of the ASTM establishes a task complexity, O, for defining L2L support for a set of languages S to be O(M(S) + (L2L(G) + L2L(S)) where M(S) is the effort to Map each language into the ASTM, L2L(G) is the effort to provide a set of reusable L2L functions based upon the GASTM for each language, and L2L(S) is the effort to provide language specific L2L functionality for language unique features for specific languages.

6.9.4 Platform Migration (Scenario IV)

Moving systems from one platform to another is driven by platform obsolescence or the desire to standardize applications to an organizational standard. This scenario does not involve any functional or data redesign beyond that which is essential to the platform migration. This scenario may also be combined with a language-to-language conversion, although language conversion is not always required (e.g., UNIX to LINUX). The following situations typically drive a platform migration:

- The hardware and/or operating system is no longer supported or viable.
- Management has decided to "right size" a system by moving it to a distributed environment or back to a mainframe.
- The current platform does not support the accepted operating system standard.
- Management has mandated a platform change.

ADM Packages play the following role in a platform migration project:

- Tracks system artifacts through the planning, phasing and staging of the migration, particularly if it has to be phased in over time.
- Facilitates discovery of high risk issues such as the use of runtime libraries or language constructs not available in the target environment.
- Assists with the change management process as the migration proceeds.
- Streamlines the planning and execution of the validation and verification stage of the migration.

ASTM SUPPORT: This ASTM supports the ADM Platform Migration (PM) scenario by providing a language neutral framework upon which PM Tools can provide a uniform and high level of automated support. The use of the ASTM establishes a task complexity, O, for defining PM support for a set of languages S to be O(M(S) + (PM(G) + PM(S)) where M(S) is the effort to Map each language into the ASTM, PM(G) is the effort to provide a set of reusable PM functions based upon the GASTM for each language, and PM(S) is the effort to provide language specific PM functionality for language unique features for specific languages.

6.9.5 Non-Invasive Application Integration (Scenario V)

Organizations with an immediate need to bring a graphically oriented look and feel to end users can utilize middleware technology to replace existing front-ends with Web-based front-ends. While a non-invasive integration project qualifies as a modernization scenario only at the most superficial level (i.e., the user interface), this scenario is still supported by ADM Packages. The integration scenario is characterized by the following requirements:

- Business users want to replace aging front-ends with Web-based front-ends.
- Users gain value from replacing older front-ends while leaving core system functionality, data structures, and interfaces essentially unchanged.
- An integration project is seen as a stepping stone to subsequent modernization objectives such as an SOA migration.

In spite of the fact that this is a non-invasive approach (i.e., does not change underlying application) to providing new user front-ends to business users, the ADM Packages play a role in the planning, execution and post-project documentation of new user front-ends. In general, ADM Packages aid in the discovery and adaptation of existing user interfaces. ADM Packages also supports the post-implementation phase because middleware environments are becoming so complex. Because larger organizations are losing track of which interfaces and systems are connected to other systems, documentation is a key aspect of integration deployment. The following points exemplify the role of ADM Packages in a non-invasive integration scenario:

- Aids in determining how to create a new interface to existing business logic because it has high-level and granular information about application data and processing logic.
- Allows analysts to pinpoint all user interface candidates targeted for migration to Web-based front-ends.
- Helps identify existing front-ends that may be redundant with other front-ends across application environments.
- Highlights front-end consolidation opportunities for redundant back-ends.
- Facilitates the tracking of distributed user interfaces and middleware on an ongoing basis as new front-ends are deployed.
- Ensures alignment with the OMG Enterprise Application Integration (EAI) specification.

ASTM SUPPORT: This ASTM supports the ADM Non-Invasive Application Integration (NIAI) scenario by providing a language neutral framework upon which NIAI Tools can provide a uniform and high level of automated support. The use of the ASTM establishes a task complexity, O, for defining NIAI support for a set of languages S to be O(M(S) + (NIAI(G) + NIAI(S)) where M(S) is the effort to Map each language into the ASTM, NIAI(G) is the effort to provide a set of reusable PM functions based upon the GASTM for the set of languages, and NIAI(S) is the effort to provide language specific NIAI functionality for language unique features for specific languages.

6.9.6 Services Oriented Architecture Transformation (Scenario VI)

The transformation to services oriented architecture involves more than just attaching new front-ends to legacy user interfaces as some analysts mistakenly believe. Because most existing application functionality is embedded in monolithic, functionally and architecturally dated systems, application and data architectures cannot be segregated into services in any useful or meaningful way. In addition, business logic is typically intertwined with user interface and data access logic. In these situations, a true SOA cannot be created without retroactively applying modular design principles to existing, back-end systems. The SOA scenario is applicable in the following situations:

- Business functions embedded in monolithic batch or online applications need to be accessed in a modular, services oriented fashion.
- System functionality is locked into backend batch processing systems.
- Complex user interface and data access logic complicates the isolation of business logic that may be deemed a service.
- Online applications do not update data in real time, which results in a service relying on back-end batch update systems.
- Existing front-ends rely on segmented, inconsistent, and redundant functionality in back-end systems, which is not conducive to forming well-bounded services.

ADM Packages plays a key role in an SOA scenario by helping identify and track relationships between the physical system, program functionality, data usage, and user interfaces. Such a project requires the componentization of existing applications to facilitate the reuse of business logic contained within them. Based on this requirement, the KDM helps as follows:

- Facilitates the front-end planning necessary to identify redundant, inconsistent and segregated functionality that needs to be refactored to create services.
- Identifies the front-ends of interest that could serve as prototypes for creating a service that hooks into an existing system.
- Allows analysts to determine the need to consolidate and reconcile redundant and inconsistent user interfaces and program functionality into reusable services.
- Aids in discovery and extraction of the business logic as service candidates.
- Helps with the discovery and adaptation of new, component level interfaces.
- Streamlines efforts to track consolidated user interfaces and program functions throughout the service creation and deployment process.
- Facilitates the tracking of new user interfaces into back-end applications as a way to document post-project SOA architecture.
- Simplifies the planning and execution of the validation and verification stage of the project.

ASTM SUPPORT: This ASTM supports the ADM Services Oriented Architecture (SOA) scenario by providing a language neutral framework upon which SOA Tools can provide a uniform and high level of automated support. The use of the ASTM establishes a task complexity, O, for defining SOA support for a set of languages S to be O(M(S) + (SOA(G) + SOA(S)) where M(S) is the effort to Map each language into the ASTM, SOA(G) is the effort to provide a set of reusable PM functions based upon the GASTM for the set of languages, and SOA(S) is the effort to provide language specific NIAI functionality for language unique features for specific languages.

6.9.7 Data Architecture Migration (Scenario VII)

A data architecture migration moves one or more data structures from a non-relational file or database to relational data architecture. Many times this is done using a “quick and dirty” approach, leaving users with performance, reliability and data accessibility problems. Pitfalls include ignoring business requirements, sidestepping relational design techniques, not incorporating related or redundant data in the project, not utilizing qualified data analysts or treating the project as a straight conversion effort. This scenario shuns the quick and dirty approach. Typical requirements are as follows.

- Users are experiencing data consistency, accessibility, redundancy, and integrity problems.
- The business is experiencing an inability to get at the same types of data defined or calculated differently across different systems.
- Existing flat file, hierarchical or networked structures are not readily accessible to distributed or new technologies
- Users require more flexible views of business data.
- Older file or database structures are obsolete and being eliminated.

ADM Packages facilitate data architecture migration because they can assist with tracking artifacts impacted by such a project including program-based data definitions, data access logic, database definition language, and the data itself. The ADM Packages assist as follows:

- Enable analysts to define the scope of the project based on the artifacts impacted by existing data structures.
- Allow analysts to identify all relevant and impacted artifacts based on the nature and scope of the data migration effort.
- Help determine if a common basis exists, within underlying computation and data models, to identify changes that would allow different systems to produce compatible answers for the same data.

- Facilitate the isolation, consolidation, and reconciliation of data access logic as a basic step in simplifying programmatic access to the newly redesigned database.
- Streamline the planning and execution of the validation and verification stage of the project.

ASTM SUPPORT: This ASTM supports the ADM Data Architecture Migration (DAM) scenario by providing a language neutral framework upon which DAM Tools can provide a uniform and high level of automated support. The use of the ASTM establishes a task complexity, O, for defining DAM support for a set of languages S to be O(M(S) + (DAM(G) + DAM(S)) where M(S) is the effort to Map each language into the ASTM, DAM(G) is the effort to provide a set of reusable DAM functions based upon the GASTM for the set of languages, and DAM(S) is the effort to provide language specific NIAI functionality for language unique features for specific languages.

6.9.8 Application & Data Architecture Consolidation (Scenario VIII)

Many organizations have multiple systems that perform the same basic functions. For example, a merger or an acquisition may have resulted in three separate billing systems. System cloning also contributes to these redundancies.

Redundant systems and data structures contribute to usage inconsistencies, redundant business processes, customer frustration, integration problems, and excessive maintenance workloads. Another factor driving this scenario is the need to construct a single application from multiple stand-alone systems. For example, analysts may want to create a single human resources system from separate payroll, insurance, and pension systems. The following factors drive such a scenario:

- An organization has recently undergone a merger or acquisition or has not yet fully streamlined applications from a past merger or acquisition:
    - Management wishes to consolidate redundant business areas into a single functional unit.
    - High business and IT costs require the consolidation of redundant business processes and related systems.
    - Currently running several redundant systems that essentially perform the same or similar functions.
    - Several similar systems contain large segments of overlapping functionality.
    - Related, stand alone systems process similar data redundantly and inconsistently.
    - Inadequate or nonexistent sharing of data between systems is severely limiting user service levels.
    - Downstream systems have evolved to handle much of the functionality that should be defined in core systems.
    - Business users get different answers to the same questions from different systems.

The application and data architecture consolidation scenario is far reaching because it involves major retooling multiple applications. This scenario does not involve model-driven transformation, language change, or platform migration. It can, however, be combined with these scenarios. Care must be taken to make a business case for this scenario. This may not be difficult when millions of dollars are spent on redundant business units that cannot be combined due to information systems redundancies. ADM plays the supports this scenario as follows.

- Helps pinpoint which applications and data structures are within the project scope.
- Assists in determining the business process, user interface, application logic, data definition, and related redundancies as candidates for consolidation.
- Enables the tracking of application artifacts as consolidation efforts proceed and are phased into production.
- Streamlines the planning and execution of the validation and verification stage of the project.

ASTM SUPPORT: This ASTM supports the ADM Application Data Architecture Consolidation (ADAC) scenario by providing a language neutral framework upon which ADAC Tools can provide a uniform and high level of automated support. The use of the ASTM establishes a task complexity, O, for defining ADAC support for a set of languages S to be O(M(S) + (ADAC(G) + ADAC(S)) where M(S) is the effort to Map each language into the ASTM, ADAC(G) is the effort to provide a set of reusable ADAC functions based upon the GASTM for the set of languages, and ADAC(S) is the effort to provide language specific ADAC functionality for language unique features for specific languages.

6.9.9 Data Warehouse Deployment (Scenario IX)

This scenario builds a data warehouse of business data and creates ways to access this data. The warehouse contains data that has been extracted, analyzed, and transformed into a common repository that users can access, but not update, as required. This is common when organizations have the following requirements:

- Business functions require consolidated access to certain data (e.g., customer information) to streamline user tasks.
- Users require access to data that crosses organizational and application boundaries.
- Related data is defined across multiple systems, making it difficult to access user summary information.
- There is not enough time or budget to integrate or modify core applications to address these data requirements.

As discussed in the data architecture migration scenario, KDM facilitates data analysis and capture from existing systems. Further, the KDM is aligned with the OMG Common Warehouse Model (CWM), which is a standard for representing data from disparate sources for the purposes of building and maintaining a data warehouse. The KDM helps facilitate this scenario as follows:

- Helps analysts identify the relevant data and related data definitions that need to be analyzed, reconciled, validated, and loaded into the warehouse.
- Facilitates the detailed analysis needed to identify data definition discrepancies across systems or business units.
- Facilitates the tracking of data models and the physical data from which these models are derived.
- Allows analysts to continue to track multiple data sources on an ongoing basis as the data warehouse is used.

ASTM SUPPORT: This ASTM supports the ADM Data Warehouse Deployment (DWD) scenario by providing a language neutral framework upon which DWD Tools can provide a uniform and high level of automated support. The use of the ASTM establishes a task complexity, O, for defining DWD support for a set of languages S to be O(M(S) + (DWD(G) + DWD(S)) where M(S) is the effort to Map each language into the ASTM, DWD(G) is the effort to provide a set of reusable DWD functions based upon the GAS TM for the set of languages, and DWD(S) is the effort to provide language specific DWD functionality for language unique features for specific languages.

6.9.10 Application Package Selection & Deployment (Scenario X)

This scenario defines how to analyze, select, and deploy third party application packages. If management is weighing one or more packages against in-house options, this scenario assists with comparing functional requirements. It assists with the deployment of the package by helping analysts determine which portions of the package need to be implemented, integrated, discarded, or updated. It also outlines how existing systems are to be retired, integrated, or retooled to work with a package. The following requirements drive the application package scenario:

- A decision has been made to investigate a third party application software package options.
- An application package has already been acquired and needs to be implemented.
- Documentation of the package is required.

- A roadmap is needed to determine how a package can interact, interface, or integrate with existing systems or packages.
- Assist with mapping strategic requirements to the data and functional capabilities of various application packages.
- Allow analysts to compare the functionality of various packages with the functionality running in the existing application environment.
- Upon package selection, determine which portions of the package would be deployed and integrated into the current environment.
- Determine which portions of the existing application environment need to be retained, retired, or integrated into the package environment.
- Provide a data mapping, integration, and migration plan based on current data structures and those used by the package.
- Offer insights into functions that need to be added to the newly deployed application based on functions the package does not perform.
- Streamline the planning and execution of the acceptance testing process.

ASTM SUPPORT: This ASTM supports the ADM Application Package Selection & Deployment (APSD) scenario by providing a language neutral framework upon which APSD Tools can provide a uniform and high level of automated support. The use of the ASTM establishes a task complexity, O, for defining APSD support for a set of languages S to be O(M(S) + (APSD(G) + APSD(S)) where M(S) is the effort to Map each language into the ASTM, APSD(G) is the effort to provide a set of reusable APSD functions based upon the GASTM for the set of languages, and APSD(S) is the effort to provide language specific APSD functionality for language unique features for specific languages.

6.9.11 Reusable Software Assets / Component Reuse (Scenario XI)

Reuse is one of the critical ways in which an IT organization can improve productivity. Consider the massive redundancies hidden in application and data structures across software portfolios and the savings become apparent. Organizations could save significant time and money by leveraging previously implemented functionality in new development projects. Further, businesses that are now running duplicate customer management, payment, claims, inventory and other systems have a wealth of untapped information building blocks. Modernization helps identify, capture, streamline and prepare information assets for reuse. This scenario applies in the following situations:

- An organization understands and has bought into the value of reuse and component-based development.
- There is a significant installed base of application systems that contain functionality that IT wishes to turn into reusable assets or components.

The ASTM supports the reuse scenario as follows:

- Facilitates the identification of certain software assets based on data utilization, transaction access, or other criteria.
- Identifies related or duplicate functionality based on common data usage, structural considerations or other cross-reference criteria.
- Assists with tracking reusable assets as they are consolidated and populated into reuse libraries.

ASTM SUPPORT: This ASTM supports the ADM Reusable Software Assets / Component Reuse(RSA-CR) scenario by providing a language neutral framework upon which RSA-CR Tools can provide a uniform and high level of automated support. The use of the ASTM establishes a task complexity, O, for defining RSA-CR support for a set of languages S to be

O(M(S) + (RSA-CR(G) + RSA-CR(S)) where M(S) is the effort to Map each language into the ASTM, RSA-CR(G) is the effort to provide a set of reusable RSA-CR functions based upon the GASTM for the set of languages, and RSA-CR(S) is the effort to provide language specific RSA-CR functionality for language unique features for specific languages.

6.9.12 Model-Driven Architecture Transformation (Scenario XII)

Transforming a non-model-driven environment to a model-driven environment requires a series of phased tasks over a window of time. Moving to MDA, given that rewriting all of the existing applications is not an option in most cases, requires transformation of existing, hand-crafted applications into models that can be used to generate replacement applications. The following are common requirements driving this concept:

- An organization has adopted and is committed to an IT environment in which applications are built and maintained in models.
- MDA concepts and tools are accepted within the organization.
- The current data and application architecture is dated or obsolete.
- The business has changed to the degree that existing systems and data structures no longer support the organization in their current form.
- Users require functional upgrades that are difficult to add to the existing architecture. The KDM can assist with the phased transition to such an environment as follows:
    - Current data and application artifacts can be mapped to these new requirements in the ASTM to assist with defining a transformation plan.
    - Transformation projects can be defined across enterprise data and applications of interest based on ASTM mappings.
    - As transformation proceeds, ASTM serves as the vehicle for mapping current-to-target functionality, along with existing artifacts, for transformation purposes.
    - Streamline the planning and execution testing process.
    - Post-transformation documentation ensures that emerging MDA models and system artifacts are documented during and after the transformation process.

ASTM SUPPORT: This ASTM supports the ADM Model Driven Architecture Transformation (MDAT) scenario by providing a language neutral framework upon which MDAT Tools can provide a uniform and high level of automated support. The use of the ASTM establishes a task complexity, O, for defining MDAT support for a set of languages S to be O(M(S) + (MDAT(G) + MDAT(S)) where M(S) is the effort to Map each language into the ASTM, MDAT(G) is the effort to provide a set of reusable MDAT functions based upon the GASTM for the set of languages, and MDAT(S) is the effort to provide language specific MDAT functionality for language unique features for specific languages.

The ASTM establishes a standard bottom-most language modeling level for many MDA tools to generate to and derive from. It allows AST models to be sharable among multiple tools from different vendors accurately support analysis, visualization, refactoring, target mapping and transformations. It provides high levels of automation for tasks that are highly manual today, such as application rehosting, platform retargeting, legacy system replacement, database upgrade.

The SATM provides sufficient precision and generality and fineness of granularity to allow its language modeling elements to be used as a common basis for application analysis, metrics, visualization translation, and refactoring. The ASTM defines an intensely rich architecture based upon MDA principals for supporting the ADM modernization scenarios. A common use of the ASTM is to support for scenario III Language-to-Language conversions. The ASTM is architected to supports higher efficient conversion between multiple language categories with a high degree of reuse of language elements. Conversions between “language levels” are carried out within the GASTM as language-neutral conversions to achieve high-level of reuse. The platform and language specific (each source PSM) mappings into the GAS TM (one PIM) and from the GASTM (each target PSM) to the target platform specific language is defined one once per language.

The complete set of the language-to-Language transformation combinatorics supported by GASTM based language to language conversion is outlined in the Top-Down L2L and the Bottom-Up L2L conversion scenario table below.

Table 6.3 - GASTM Language to Language (L2L) Conversion Scenarios

Top Down Language To Language (L2L) Conversion Scenarios		
5GL to 4GL		
5GL to 4GL to 3GL	4GL to 3GL	
5GL to 4GL to 3GL to 2GL	4GL to 3GL to 2GL	3GL to 2GL

Bottom Up Language To Language (L2L) Conversion Scenarios		
2GL to 3GL		
2GL to 3GL to 4GL	3GL to 4GL	
2GL to 3GL to 4GL to 5GL	3GL to 4GL to 5GL	4GL to 5GL

6.9.13 ASTM Support for the MDA

MDA is typically regarded as a top-down model-driven process for new system development. The MDA Architectural models provide portability, interoperability, and re-usability through architectural separation of concerns. Its Architectural models direct the course of understanding, design, construction, deployment, operation, maintenance, and modification. ADM incorporates bottom-up extraction of architectural models followed by top down reuse in MDA processes and scenarios for legacy systems modernization and closes the gap between top down and bottom up methodologies.

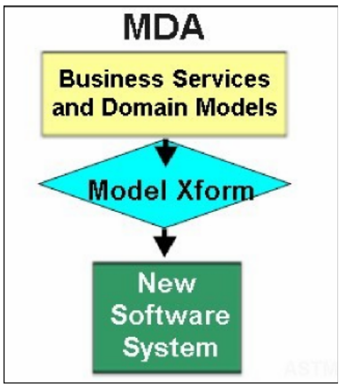


Figure 6.7 - MDA with ADM

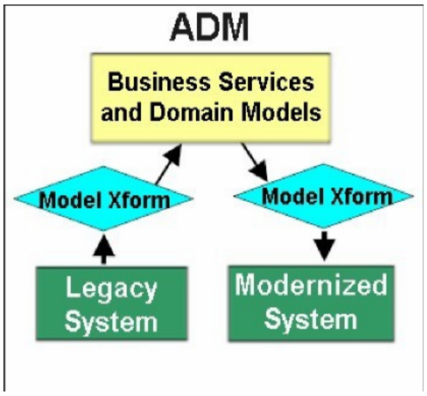


Figure 6.8 - MDA with ADM

Closure of this gap permits the application of MDA to existing legacy systems as illustrated in Figure 6.9.

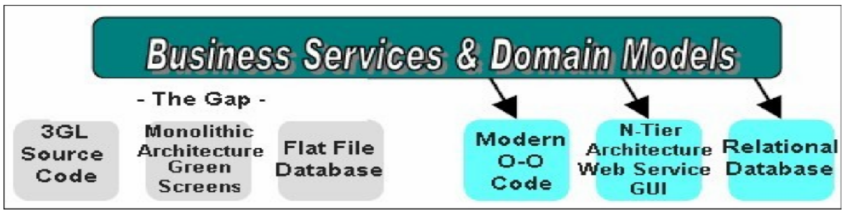

Business Services & Domain Models
- The Gap -
3GL Source Code
Monolithic Architecture Green Screens
Flat File Database
Modern O-O Code
N-Tier Architecture Web Service GUI
Relational Database

Figure 6.9 - ADM allows modern MDA tools to be applied to legacy software

7 ASTM Core Concepts

The terms and definitions in the Glossary, Annex B, are widely used or understood within the computer science community. They are provided herein as a common frame of reference for interpreting the meaning of the terminology used throughout this document and are the source references or basis of understanding for the core ASTM concepts and their definitions presented below.

The following tables categorize and classify the terminology used in the ASTM model into major syntactic and semantic categories by programming language domain (e.g., imperative, objective-oriented). The terminology was subclassified along data, functional, structural and preprocessor dimensions, similar to those used in KDM to facilitate ASTM alignment with the KDM. The names, associations and properties of the GAST modeling elements were synthesized by combining the best practices employed in commercial-grade software modeling tools of the submission teams (TSRI, TCS, Klocwork, IBM and EDS). The ASTM Core Element Concise Definitions are definitions of the core ASTM modeling elements based upon their common understood meaning within the computer science community.

7.1 ASTM Core Syntax Concepts

Table 7.1 - Generic Abstract Tree Core Terminomogy Matrix

Generic Abstract Syntax Tree Core Terminology Matrix						
Domain	**Data**		**Executable Code**		**Structure**	**Preprocessor**
Programming Paradigm	**Symbols**	**Types**	**Statements**	**Expressions**		
Imperative Paradigm	Entry Definition Enumeral Definition Label Definition Procedure Definition Template Definition Type Definition Variable Definition Formal Parameter Definition	Collection Type Enumeration Literal Enumeration Type Exception Type Label Type Pointer Type Primitive Type Range Type Reference Type Structure Type Template Type Sequence Type Dimension Type Address Of	Block Statement Break Statement Case Statement Continue Statement Default Statement Expression Statement Try Statement Jump Statement Label Statement Loop Statement Return Statement Switch Statement Throw Statement Global Declaration	Array Reference Binary Expression Cast Expression Conditional Expression Enumeration Reference Identifier Reference Label Reference Literal Operator (Name) Pointer Expression Procedure Call Qualified Identifier Reference Range Expression Reference Expression	Compilation Unit Declaration Entry Point Procedure	Include Statement Include Unit Macro \Call Macro Definition
Object-Oriented	Class Definition Method Definition Member Definition	Class Type Inherits <possible relationship>				

7.2 ASTM Core Semantic Concepts

Table 7.2 - Generic Abstract Semantic Graph Terminology

Generic Abstract Semantic Graph Terminology		
Domain	Bindings	Location
All Programming Paradigms	ProgramScopeProcedureScope Block Scope Type Scope	SourceLocation SourceFile

7.3 ASTM Core Element Concise Definitions

Table 7.3 - GASTM Element Definitions

GASTM Modeling Element	**Core Element Concise Definitions**
1. GASTMObject	The root of the GASTM class hierarchy
1.1 GASTMSourceObject	Objects related to specifying locations within source files
1.1.1 SourceLocation	Start/end line/column position information, part of a source location specification
1.1.2 SourceFile	Objects related to semantic artifacts of the modeled/analyzed system
1.2 GASTMSemanticObject	The collection of compilation units to be modeled/analyzed as a whole
1.2.1 Project	Declaration context in which names declared must be unique
1.2.2 Scope	Declaration context in which names declared must be unique
1.2.2.1 FunctionScope	The scope introduced by a function definition
1.2.2.2 AggregateScope	The scope introduced by an aggregate type
1.2.2.3 BlockScope	The scope introduced by a block statement
1.2.2.4 ProgramScope	The scope introduced by a compilation unit
1.2.2.4 GlobalScope	outermost scope, surrounding all compilation units of a project
1.3 GASTMSyntaxObject	All syntactic constructs
1.3.1 PreprocessorElement	Inclusion of a file during preprocessing
1.3.1.1 IncludeUnit	Inclusion of a file during preprocessing
1.3.1.2 MacroCall	Invocation of a preprocessor macro
1.3.1.3 MacroDefinition	Definition of a preprocessor macro

1.3.1.4 Comment	Comments appearing in source files
1.3.2 DefinitionObject	Constructs that define entities
1.3.2.1 DeclarationOrDefinition	Declarations and definitions
1.3.2.1.1 Declaration	Constructs that declare entities without defining them
1.3.2.1.1.1 FunctionDeclaration	Function declarations
1.3.2.1.1.2 VariableDeclaration	Variable declarations
1.3.2.1.1.3 FormalParameterDeclaration	Formal Parameter Declarations, appearing in function declarations
1.3.2.1.2 Definition	Constructs that declare entities as they also define them
1.3.2.1.2.1 FunctionDefinition	Subprogram definitions
1.3.2.1.2.2 EntryDefinition	Subprogram entry definitions
1.3.2.1.2.3 DataDefinition	Definitions involving data
1.3.2.1.2.3.1 VariableDefinition	Variable definitions
1.3.2.1.2.3.2 FormalParameterDefinition	Formal Parameter Declarations, appearing in function declarations
1.3.2.1.2.3.3 BitFieldDefinition	Definitions of bit-field data
1.3.2.1.2.4 EnumLiteralDefinition	Definitions of enumerals (members of enumerated types)
1.3.2.1.2.5 TypeDefinition	Definitions of types
1.3.2.1.2.5.1 NamedTypeDefinition	Definitions of types to be referred to by a specified name
1.3.2.1.2.5.2 AggregateTypeDefinition	Definitions of aggregate types
1.3.2.1.2.5.3 EnumTypeDefinition	Definitions of enumeration types
1.3.2.2 NamespaceDefinition	Definitions of namespaces
1.3.2.3 LabelDefinition	Definitions of labels
1.3.3 Type	All types
1.3.3.1 FunctionType	Function types
1.3.3.2 DataType	Types involving data
1.3.3.2.1 PrimitiveType	Primitive types; not further decomposable
1.3.3.2.1.1 Void	Void type

1.3.3.2.1.2 Boolean	Boolean Type
1.3.3.2.1.3 NumberType	Numeral type (unsigned or signed)
1.3.3.2.1.3.1 Byte	Byte Type
1.3.3.2.1.3.2 Character	Character Type
1.3.3.2.1.3.3 IntegralType	Various Integer types with optional size specification
1.3.3.2.1.3.3.1 ShortInteger	Short Integer type
1.3.3.2.1.3.3.2 Integer	Integer type
1.3.3.2.1.3.3.3 LongInteger	Long Integer type
1.3.3.2.1.3.4 RealType	Various floating-point types
1.3.3.2.1.3.4.1 Real	Short floating-point type
1.3.3.2.1.3.4.2 Double	Long floating-point type
1.3.3.2.1.3.4.3 LongDouble	Long floating-point type
1.3.3.2.2 EnumType	Enumerated types
1.3.3.2.3 ConstructedType	Types constructed from a specified base type
1.3.3.2.3.1 CollectionType	Types characterized as collections (lists, sets, bags, ...)
1.3.3.2.3.2 PointerType	Types whose values are pointers
1.3.3.2.3.3 ReferenceType	Types whose values are references
1.3.3.2.3.4 RangeType	Types whose values are ranges
1.3.3.2.3.5 ArrayType	Array types
1.3.3.2.4 AggregateType	Types composed of heterogeneous subtypes
1.3.3.2.4.1 StructureType	Simple structure types (no inheritance or function members)
1.3.3.2.4.2 UnionType	Union types (like structures but each data member occupies the same location)
1.3.3.2.4.3 ClassType	Class types
1.3.3.2.4.4 AnnotationType	Denotations that complete or extend the definitions of other types
1.3.3.2.5 ExceptionType	
1.3.3.2.6 FormalParameterType	

1.3.3.2.6.1 ByValueFormalParameterType	
1.3.3.2.6.2 ByReferenceFormalParameterType	
1.3.3.2.7 NamedType	
1.3.3.2.6 LabelType	
1.3.3.2.7 NamespaceType	
1.3.3.2.8 TypeReference	
1.3.3.2.8.1 UnnamedTypeReference	
1.3.3.2.8.2 NamedTypeReference	
1.3.4 Expression	
1.3.4.1 Literal	
1.3.4.1.1 IntegerLiteral	
1.3.4.1.2 StringLiteral	
1.3.4.1.3 CharLiteral	
1.3.4.1.4 RealLiteral	
1.3.4.1.5 BooleanLiteral	
1.3.4.1.6 BitLiteral	
1.3.4.1.7 EnumLiteral	
1.3.4.2 CastExpression	
1.3.4.3 AggregateExpression	
1.3.4.4 ArrayReference	
1.3.4.5 UnaryExpression	
1.3.4.6 BinaryExpression	
1.3.4.7 ConditionalExpression	
1.3.4.8 RangeExpression	
1.3.4.9 FunctionCallExpression	
1.3.4.10 NewExpression	
1.3.4.11 NameReference	

1.3.4.11.1 IdentifierReference	
1.3.4.11.2 QualifiedIdentifierReference	
1.3.4.11.2.1 QualifiedOverPointer	
1.3.4.11.2.2 QualifiedOverData	
1.3.4.11.2.3 TypeQualifiedIdentifierReference	
1.3.4.12 LabelAccess	
1.3.4.13 ArrayAccess	
1.3.4.14 AnnotationExpression	
1.3.4.15 CollectionExpression	
1.3.5 Statement	
1.3.5.1 ExpressionStatement	
1.3.5.2 JumpStatement	
1.3.5.3 BreakStatement	
1.3.5.4 ContinueStatement	
1.3.5.5 LabeledStatement	
1.3.5.6 BlockStatement	
1.3.5.7 EmptyStatement	
1.3.5.8 IfStatement	
1.3.5.9 SwitchStatement	
1.3.5.10 ReturnStatement	
1.3.5.11 LoopStatement	
1.3.5.11.1 WhileStatement	
1.3.5.11.2 DoWhileStatement	
1.3.5.11.3 ForStatement	
1.3.5.11.3.1 ForCheckBeforeStatement	
1.3.5.11.3.2 ForCheckAfterStatement	

1.3.5.12 TryStatement	
1.3.5.13 ThrowStatement	
1.3.5.14 DeleteStatement	
1.3.5.15 TerminateStatement	
1.3.6 MinorSyntaxObject	
1.3.6.1 Dimension	
1.3.6.2 CompilationUnit	
1.3.6.3 Name	
1.3.6.4 SwitchCase	
1.3.6.4.1 CaseBlock	
1.3.6.4.2 DefaultBlock	
1.3.6.5 CatchBlock	
1.3.6.5.1 TypesCatchBlock	
1.3.6.5.2 VariableCatchBlock	
1.3.6.6 UnaryOperator	
1.3.6.6.1 UnaryPlus	
1.3.6.6.2 UnaryMinus	
1.3.6.6.3 Not	
1.3.6.6.4 BitNot	
1.3.6.6.5 AddressOf	
1.3.6.6.6 Deref	
1.3.6.6.7 Increment	
1.3.6.6.8 Decrement	
1.3.6.6.9 PostIncrement	
1.3.6.6.10 PostDecrement	
1.3.6.7 BinaryOperator	
1.3.6.7.1 Add	
1.3.6.7.2 Subtract	

1.3.6.7.3 Multiply	
1.3.6.7.4 Divide	
1.3.6.7.5 Modulus	
1.3.6.7.6 Exponent	
1.3.6.7.7 And	
1.3.6.7.8 Or	
1.3.6.7.9 Equal	
1.3.6.7.10 NotEqual	
1.3.6.7.11 Greater	
1.3.6.7.12 NotGreater	
1.3.6.7.13 Less	
1.3.6.7.14 NotLess	
1.3.6.7.15 BitAnd	
1.3.6.7.16 BitOr 1.3.6.7.17 BitXor	
1.3.6.7.18 BitLeftShift	
1.3.6.7.19 BitRightShift 1.3.6.7.20 Assign 1.3.6.7.20.1 OperatorAssign 1.3.6.9 StorageSpecification	
1.3.6.9.1 External 1.3.6.9.2 FunctionPersistent 1.3.6.9.3 FileLocal 1.3.6.9.4 PerClassMember 1.3.6.9.5 NoDef	
1.3.6.10 VirtualSpecification	
1.3.6.10.3 NonVirtual 1.3.6.11 AccessKind 1.3.6.11.1 Public 1.3.6.11.2 Protected 1.3.6.11.3 Private 1.3.6.12 ActualParameter	
1.3.6.12.1 ActualParameterExpression	
1.3.6.12.1.1 ByValueActualParameterExpression	
1.3.6.12.1.2 ByReferenceActualParameterExpression	

1.3.6.12.2 MissingActualParameter	
1.3.6.13 FunctionMemberAttributes 1.3.6.14 DerivesFrom	
1.3.6.15 MemberObject	

7.4 ASTM Core Abbreviated BNF Definitions

The ASTM Abbreviated BFN definitions below is a compact form of description of BNFs that is commonly used by compiler developers and language theory experts to express abstract syntax trees. A BNF is a concise definition formalism used for describing ASTs in the machine readable format typically used by parser generators. The ABNF of the GASTM is provided below, as part of the ASTM Core Concept Definition section, to serve as a compact conceptual expression for the normative specification provided in Section 3.

7.5 ASTM Abreviated BNF (ABNF) Specification Notation

An Abbreviated BNF (Baccus-Nauer Format) notation is used to describe the Generic Abstract Syntax Tree Metamodel (GASTM) using objects, attributes, and associations.

Notations used in this document are described below:

- "!!" is used to denote comments.
- "!" applied as a prefix to a class denotes that class is an abstract class.
- Rules of the form A -> B mean “class A has the attributes specified in B”
- Rules of the form A => B mean “class B is a subtype of class A”
- Attributes are listed as “name : Primtiive type”
- Associations are listed as “name : Class type”
- Attributes and Associations within sqaure brackets [ ] hold semantic information as opposed to abstract- syntactic information.

7.6 The ASTM Model Hierarchy

The ASTM Class Hierarchies consist of a single GASTM Class Hierarchy and multiple SASTM Class Hierarchies. The GASTM contains a set of UML Class descriptions for syntactic and basic semantic concepts that are common across many languages. The SASTM Class Hierarchies are extensions of the GASTM that provide the specialized syntactic and basic semantic concepts that are found within a particular language or a language category. The relationship between the GASTM and SASTMs are depicted in the figure below. The GASTM provides language neutral support for the ADM Packages and the SASTMs provide language specific specialization in support of the ADM Packages.

Table 7.4 - ASTM Support for the ADM Packages and Scenarios

Vertical Support For ADM Scenarios													
ADM Repository Technologies													
ASTM AP				MP	SW/A		VP	RP		TMTP		KDM	
ASTM		**GASTM : Language Neutral Support**											
SASTM : Horizontal Language Support													
C++	Java	Ada	C#	VB/.Net	C	COBOL	FORTRAN	Jovial	VMSVAXBASIC	RDBMS	PL/1	JCL	Etc

7.7 GASTM Core Concepts

The ASTM Core as expressed in Abbreviated Baccus-Nauer Format defines a model to represent the syntactic, semantic, and source associations and properties of programming language elements as they exist in source code and in abstract model forms. The specification defines a model to represent the programming language elements and their relationships, as they exist in source code. The GASTM specifications have been split into subsections in order to group related specifications and to distinguish semantic elements from syntactic language elements.

Table 7.5 - Core Components of the ASTM

ASTM Core Semantics Specification	The ASTM Core Semantics Specification defines a set of core modeling elements used for enrichment and abstraction of the ASTM abstract syntax tree to represent or derive the *semantics* of a formal language so as to enrich an abstract syntax tree to form an abstract semantic graph (ASG).
ASTM Core Syntax Specification	The ASTM Core Syntax Specification defines a set of generic modeling elements commonly used in formal programming languages in the form of a *finite*, labeled, *directed tree*, where *internal nodes* are represented by the classes, the relationships between interior nodes are depicted by unidirectional associations, and the *leaf nodes*.
ASTM Unified Modeling Language Specification	The ASTM Unified Modeling Language Specification depicts the ASTM Semantic and Syntactic Specification as a set of UML Class diagrams.

Table 7.5 - Core Components of the ASTM

ASTM XSD Specification	The ASTM XSD Specification depicts the ASTM Semantic and Syntactic Specification as an eXtended Meta Language (XML) Schema Description (XSD). An XSD is a specification format established by the World Wide Web Consortium (W3C) for defining the structure of XML documents.
ASTM XMI Specification	The ASTM XMI Specification depicts the ASTM Semantic and Syntactic Specification as an eXtended Meta Language Metamodel Interchange Specification. XMI is a specification format established by the OMG for defining the structure of the meta models of XML documents.

Table 7.6 - ASTM Core Semantics Specification

Project	The project consisting of a set of source files containing programming language.
Scope	The scope consists of a set of symbols within some set of programming elements. The scope can contain subscopes and may be contained within a parent scope.

Table 7.7 - ASTM Core source Specification

SourceFile	The source files that physically contain the programming language elements.
SourceLocation	The physical source code location of each programming language element.
SourceFileReference	The reference to a source file; reference is contained in another source file.

7.8 GASTM Object

The core of the GASTM consists of three abstract classes (denoted with a ! prefix) for depicting the syntactic, semantic, and source properties of programming language elements.

```
GASTMObject    =>     ! GASTMSourceObject
               =>     ! GASTMSemanticObject
               =>     ! GASTMSyntaxObject
;
```

7.9 GASTM Source Object

The GASTMSourceObject has two subclasses, SourceFile and SourceLocation for modeling the source file path and SourceLocation starting line, starting column, ending line, and ending column of any ASTM modeling element. The GASTM SourceFile and SourceLocation are used for modeling the file system path and line and column location of source code that is stored in files. A GASTMObject can exist as a syntactic abstraction without need for a SourceFile or SourceLocation description, in which case these properties are not needed.

```
GASTMSourceObject    =>     SourceFile
                     =>     SourceLocation
;
SourceFile->  < path : String >
              ;
SourceFile    =>     CompilationUnit
              =>     SourceFileReference
;

CompilationUnit->    < language : String >
                     fragments : DefinitionObject* [ opensScope : ProgramScope? ]
;

SourceLocation
              ->     < startLine : Integer >
                     < startPosition : Integer >
                     < endLine : Integer >
                     < endPosition : Integer >
inSourceFile : SourceFile
;
SourceFileReference
              ->     locationInfo : SourceLocation
[ ofSourceFile : SourceFile ]
;
```

7.10 GASTMSemanticObject

This section contains a set of non-syntactic elements used for modeling certain kinds of basic semantic properties between ASTM objects. Project is a collection of Compilation Units used for depicting the set of code units that are to be modeled.

Scope is, strictly speaking, a semantic rather than a syntactic property that describes the largest declarative region (a part of a program) in which the name is valid, that is, in which the name may be used as an unqualified name to refer to the same entity. Scope is semantic property that can be derived by constraint analysis, from the syntactical elements of the AST, and hence it is considered a Level 1 Conformance Point of the ASTM model. Scope can be captured explicitly through the use of the Scope that is subclassed into several kinds of scope for depicting the scope of functions (FunctionScope), aggregates (AggregateScope), blocks (BlockScope), programs (ProgramScope), and globals (GlobalScope).

```
GASTMSemanticObject
              =>     Project
              =>     Scope
              ;

Project       ->     files : CompilationUnit +
                     [ outerScope : GlobalScope ? ]
              ;
Scope         ->     definitionObject : DefinitionObject *
                     [ childScope : Scope * ]
                     !! Two way semantic association
              ;
```

```
Scope          =>     FunctionScope
               =>     AggregateScope
               =>     BlockScope
               =>     ProgramScope
               =>     GlobalScope
               ;
```

7.11 GASTMSyntaxObject

The topmost syntactical object in the ASTM model hierarchy is the GASTMSyntaxObject. PreProcessorElements associates GASTMSyntaxObject with preprocessor elements (PreprocessorElement) with source co-ordinates in the preprocessor element denoting whether they appear before or after the GASTMSyntaxObject. Note that PreprocessorElement need not have PreProcessorElements associated with it. Similarly, AnnotationExpression need not have Annotations. The association <Parent>, which is present in many AST models is not depicted. Instead every AST association is treated as navigable association, thus making it bi-directional. Each AST association can be thought of as a navigable association (as per complete MOF), thus enabling access to parent from the child. When explicitly implemented the <Parent> association should be treated as the universal converse for AST associations described herein. Preprocessor, Annotations and SourceLocation can be attached to any syntax object. The multiplicity of PreProcsssorElements and Annotations is {0. .m}. Note that the Preprocessor annotation is provided as a convenience for capturing the syntactic representation of preprocessing directives. An individual vendor may choose to expand actual pre-processor directives in a pre-processing pass before parsing in which case their syntactic representation is lost.

```
GASTMSyntaxObject-
               >      locationInfo : SourceLocation

                      preProcessorElements : PreprocessorElement *

                      annotations : AnnotationExpression *
               ;
GASTMSyntaxObject
               =>     ! PreprocessorElement
               =>     ! DefinitionObject
               =>     ! Type
               =>     ! Expression
               =>     ! Statement
               =>     ! MinorSyntaxObject
               ;
```

7.11.1 Other Syntax Object

```
MinorSyntaxObject
               =>     Dimension
               =>     Name
               =>     SwitchCase
               =>     CatchBlock
               =>     ! UnaryOperator
               =>     ! BinaryOperator
               =>     ! StorageSpecification
               =>     ! VirtualSpecification
               =>     AccessKind
```

```
          =>     ! ActualParameter
          =>     FunctionMemberAttributes
          =>     DerivesFrom
          =>     MemberObject
          ;
```

7.11.2 Declarations and Definitions

```
Name      ->     < nameString : String >
;
DefinitionObject[1],[2]
          =>     ! DeclarationOrDefinition
          =>     TypeDefinition
          =>     NameSpaceDefinition
          =>     LabelDefinition
          =>     TypeDeclaration
          ;
DeclarationOrDefinition
          =>     ! Definition
          =>     ! Declaration
          ;
DeclarationOrDefinition
          ->     storageSpecifiers : StorageSpecification
                 accessKind : AccessKind
                 < linkageSpecifier : String >
          ;
Definition-> identifierName : Name
             definitionType : TypeReference ?
             !! To allow K&R C formal parameter defn
          ;
Definition=> FunctionDefinition
          =>     EntryDefinition
          =>     ! DataDefinition
          =>     EnumLiteralDefinition
          ;
Declaration->[ defRef : Definition ]
identifierName : Name ?
          ;
declarationType : TypeReference
          ;
Declaration
          =>     FunctionDeclaration
          =>     VariableDeclaration
          =>     FormalParameterDeclaration
```

1. Definition of variables. DefRef : associates Declaration object to the Definition object
2. The association <ReferencesOf : IdentifierReference *> - is not by Name because this relationship is captured in NameReference and NamedTypeReference

```
        ;

StorageSpecification
        =>     External
        =>     FunctionPersistent
        =>     FileLocal
        =>     PerClassMember
        =>     NoDef
        ;
FunctionDeclaration
        ->     formalParameters :
FormalParameterDeclaration *
               functionMemberAttributes :
FunctionMemberAttributes?
;
VariableDeclaration
        ->     < isMutable : Boolean >
;
FunctionDefinition
        ->     returnType : TypeReference ?
        formalParameters : FormalParameterDefinition *
        body : Statement*
        functionMemberAttributes : FunctionMemberAttributes?
        [ opensScope : FunctionScope ]
        ;
FunctionMemberAttributes[1]
        ->     < isFriend : Boolean >
               < isInline : Boolean >
               < isThisConst : Boolean >[2]
        virtualSpecifier : VirtualSpecification
        ;
VirtualSpecification
        =>     Virtual
        ;
EntryDefinition
        ->     formalParameters : FormalParameterDefinition*
               body: Statement*
        ;
DataDefinition->initialValue : Expression?
        < isMutable : Boolean >
        ;
DataDefinition=>VariableDefinition
        =>     FormalParameterDefinition
```

1. Attributes of this class apply only to member functions
2. e.g., C++ const after method

```
        =>    BitFieldDefinition
        ;
BitFieldDefinition
        ->    bitFieldSize : Expression

EnumLiteralDefinition
        -> value : Expression?
;
Type Definition,
        -> typeName : Name
;
TypeDefinition
        => NamedTypeDefinition
        => AggregateTypeDefinition
        => EnumTypeDefinition
;
NamedTypeDefinition
        -> definitionType : NamedType
;
AggregateTypeDefinition
        -> aggregateType : AggregateType
;
EnumTypeDefinition
        -> definitionType : EnumType
;
NameSpaceDefinition
        -> nameSpace : Name
body : DefinitionObject+ nameSpaceType : NameSpaceType
;
LabelDefinition
        -> labelName : Name labelType : LabelType
;
TypeDeclaration
        -> typeRef : TypeReference
;
TypeDeclaration
        => AggregateTypeDeclaration
        => EnumTypeDeclaration
;
```

7.11.3 Directives

```
PreprocessorElement
        =>    IncludeUnit
        =>    MacroCall
        =>    MacroDefinition
        =>      Comment
        ;
IncludeUnit->file : SourceFileReference

MacroCall MacroDefinition-> ;->refersTo : MacroDefinition < macroName : String >
                < body : String >
        ;
Comment->       < body :String >
```

7.11.4 Data Types

```
Type    ->      < isConst : Boolean >
        ;
Type    =>      FunctionType
        =>      ! DataType
        =>      LabelType
        =>      NameSpaceType
        =>      ! TypeReference
        ;

DataType=>      ! PrimitiveType
        =>      EnumType
        =>      ! ConstructedType
        =>      ! AggregateType
        =>      ExceptionType
        =>      ! FormalParameterType
        =>      NamedType
        ;
PrimitiveType=>! NumberType
        =>      Void
        =>      Boolean
        ;
NumberType=> !      IntegralType
        =>      ! RealType
        =>      Byte
        =>      Character
        ;

NumberType-> < isSigned : Boolean >
        ;
IntegralType=>ShortInteger
        =>      Integer
        =>      LongInteger
        ;
IntegralType->[     <size  :      Integer> ]
        ;
```

```
RealType=>=>=>RealDoubleLongDouble
       ;
RealType->   [      <precision:  Integer> ]
       ;
EnumType     ->     enumLiterals : EnumLiteralDefinition+
```

7.11.5 Constructed Type

```
ConstructedType     ->     baseType : TypeReference
                    ;
ConstructedType     =>     CollectionType
                    =>     PointerType
                    =>     ReferenceType
                    =>     RangeType
                    =>     ArrayType
                    ;
PointerType         ->     [ <size : Integer ]
                    ;
AggregateType       ->     members : MemberObject+
                    [      opensScope : AggregateScope ]
                    ;
MemberObject        ->     [< offset:   Integer > ]member : DefinitionObject
                    ;

AggregateType       =>     StructureType
                    =>     UnionType
                    =>     ClassType
                    =>     AnnotationType
                    ;
ArrayType           ->     ranks : Dimension+
                    ;
Dimension           ->     lowBound : Expression? highBound : Expression
                    ;
FunctionType        ->     returnType : TypeReference?
                           parameterTypes : FormalParameterType*
                    ;
FormalParameterType ->     type : TypeReference
                    ;
FormalParameterType => =>ByValueFormalParameterType ByReferenceFormalParameterType
                    ;
NamedType           ->     body : Type
                    ;
ClassType           ->     derivesFrom : DerivesFrom*
                    ;
DerivesFrom         ->     virtualSpecifier : VirtualSpecification ? accessKind :
                            AccessKind
```

```
                              className : NamedTypeReference
                        ;
AccessKind              =>    Public
                        =>    Protected
                        =>    Private
                        ;
TypeReference           =>    UnnamedTypeRe ference
                        =>    NamedTypeReference
                        ;
UnnamedTypeReference          ->     type : Type
                        ;
NamedTypeReference      ->    typeName : Nametype : TypeDefinition
;
```

7.11.6 Statements

```
Statement               =>    ExpressionStatement
                        =>    JumpStatement
                        =>    BreakStatement
                        =>    ContinueStatement
                        =>    LabeledStatement
                        =>    BlockStatement
                        =>    EmptyStatement
                        =>    IfStatement
                        =>    SwitchStatement
                        =>    ReturnStatement
                        =>    LoopStatement
                        =>    TryStatement
                        =>    DeclarationOrDefinitionStatement
                        =>    ThrowStatement
                        =>    DeleteStatement
                        =>    TerminateStatement
                        ;
DeleteStatement         ->    operand: Expression
;

DeclarationOrDefinitionStatement
                        ->    declOrDefn: DefinitionObject
;
ExpressionStatement
                        ->    expression : Expression
;
JumpStatement           ->    target : Expression
;
BreakStatement          ->    target : LabelAccess?
;
ContinueStatement
                        ->    target : LabelAccess?
```

```
;
LabeledStatement
                    ->     label : LabelDefinition
statement : Statement?
;
BlockStatement      ->     subStatements : Statement*
                           [ opensScope : BlockScope ]
;
EmptyStatement->
;
IfStatement         ->     condition : Expression
thenBody : Statement
elseBody : Statement?
;
SwitchStatement     ->     switchExpression : Expression
cases : SwitchCase
;
SwitchCase          ->     < isEvaluateAllCases : Boolean >
body : Statement+
;
SwitchCase          =>     CaseBlock
                    =>     DefaultBlock
;
CaseBlock           ->     caseExpressions : Expression+
;
ReturnStatement     ->     returnValue : Expression?
;
LoopStatement       ->     condition : Expression
;
LoopStatement       ;=>    body : Statement WhileStatement
                    =>     DoWhileStatement
                    =>     ! ForStatement
                    ;
ForStatement        ->     initBody : Expression*
                           iterationBody : Expression*
                    ;
ForStatement        =>     ForCheckBeforeStatement
                    =>     ForCheckAfterStatement
                    ;
TryStatement        ->     guardedStatement : Statement catchBlocks : CatchBlock*
                           finalStatement : Statement?
                    ;
CatchBlock          ->     body : Statement
                    ;

CatchBlock          =>     TypesCatchBlock
                    =>     VariableCatchBlock
```

```
                   ;
TypesCatchBlock    ->     exceptions : Type+
                   ;
VariableCatchBlock ->     exceptionVariable : DataDefinition
                   ;
ThrowStatement     ->     exception : Expression
;
```

7.11.7 Expressions

```
Expression         ->     [expressionType : TypeReference ];
Expression         =>     Literal
                   =>     CastExpression
                   =>     AggregateExpression
                   =>     UnaryExpression
                   =>     BinaryExpression
                   =>     ConditionalExpression
                   =>     RangeExpression
                   =>     FunctionCallExpression
                   =>     NewExpression
                   =>     ! NameReference
                   =>     LabelAccess
                   =>     ArrayAccess
                   =>     AnnotationExpression
                   =>     CollectionExpression
             ;
NameReference      =>     IdentifierReference
                   =>     ! QualifiedIdentifierReference
                   =>     TypeQualifiedIdentifierReference
             ;
NameReference      ->     identifierName : Name
                          refersTo : DefinitionObject
             ;
ArrayAccess        ->     arrayName : Expression
                          subscripts : Expression+
             ;
QualifiedIdentifierReference
                   ->     qualifiers : Expression
member : IdentifierReference
                   ;
QualifiedIdentifierReference
                   =>     QualifiedOverPointer
                   =>     QualifiedOverData
                   ;
TypeQualifiedIdentifierReference
                   ->     aggregateType : TypeReference +
                          member : IdentifierReference
                   ;
```

```
Literal             ->    < value : String >
;
Literal             =>    IntegerLiteral
                    =>    StringLiteral
                    =>    CharLiteral
                    =>    RealLiteral
                    =>    BooleanLiteral
                    =>    BitLiteral
                    =>    EnumLiteral
                    ;
CastExpression      ->    castType : TypeReference
                          expression : Expression
;
UnaryExpression     ->    operator : UnaryOperator
                          operand : Expression
                    ;
UnaryOperator       =>    UnaryPlus
                    =>    UnaryMinus
                    =>    Not
                    =>    BitNotAddressOfDerefIncrement
                    =>    Decrement
                    =>    PostIncrement
                    =>    PostDecrement
                    ;
BinaryExpression    ->    operator : BinaryOperator
                          leftOperand : Expression
                          rightOperand : Expression
                    ;
BinaryOperator      =>    Add
                    =>    Subtract
                    =>    Multiply
                    =>    Divide
                    =>    Modulus
                    =>    Exponent
                    =>    And
                    =>    Or
                    =>    Equal
                    =>    NotEqual
                    =>    Greater
                    =>    NotGreater
                    =>    Less
                    =>    NotLess
                    =>    BitAnd
                    =>    BitOr
                    =>    BitXor
                    =>    BitLeftShift
                    =>    BitRightShift
                    =>    Assign
                    =>     OperatorAssign
                    ;
OperatorAssign      ->     operator : BinaryOperator
                    ;
ConditionalExpression->    condition : Expression onTrueOperand : Expression
                           onFalseOperand : Expression
                    ;
RangeExpression     ->     fromExpression : Expression toExpression : Expression
                    ;
FunctionCallExpression->   calledFunction : Expression actualParams :
                           ActualParameter*
                    ;
ActualParameter
                    =>     ActualParameterExpression
                    =>     MissingActualParameter
                    ;
ActualParameterExpression
                    ->     value : Expression
                    ;
ActualParameterExpression
                    =>     ByValueActualParameterExpression
                    =>     ByReferenceActualParameterExpression
                    ;
NewExpression       ->     newType : TypeReference
                           actualParams : ActualParameter *
                    ;
LabelAccess         ->     labelName : Name
                           labelDefinition : LabelDefinition
                    ;
AnnotationExpression
                    ->     annotationType : TypeReference ?
                    ->     memberValues : Expression *
                    ;
CollectionExpression
                    ->     expressionList : Expression *
;
```

8 ASTM Core Specification

The Unified Modeling Language (UML) Diagrams and Detailed Descriptions provided in Section 8.1 High-Level (Composite) UML Diagrams (below) and in Section 8.2 Low-Level (Detailed) GASTM Class Hierarchy (below) are the normative specification of the core of the Abstract Syntax Tree Specification, the Generic Abstract Syntax Tree Meta-Model, GASTM.

8.1 High-Level (Composite) UML Diagrams

This section contains a set of diagrams whose elements are described is detail in the subsequent section.

8.1.1 ASTM Core Objects

The core of the ASTM is the GASTMObject and consists of three abstract classes for depicting the syntactic, semantic, and source properties of programming language elements. This section provides the high-level Composite UML Class Diagrams of these elements. All other ASTM elements descend from these elements. These core elements describe the characteristic properties of the source locations of the objects, abstract syntax of the objects, and the basic semantics of the objects.

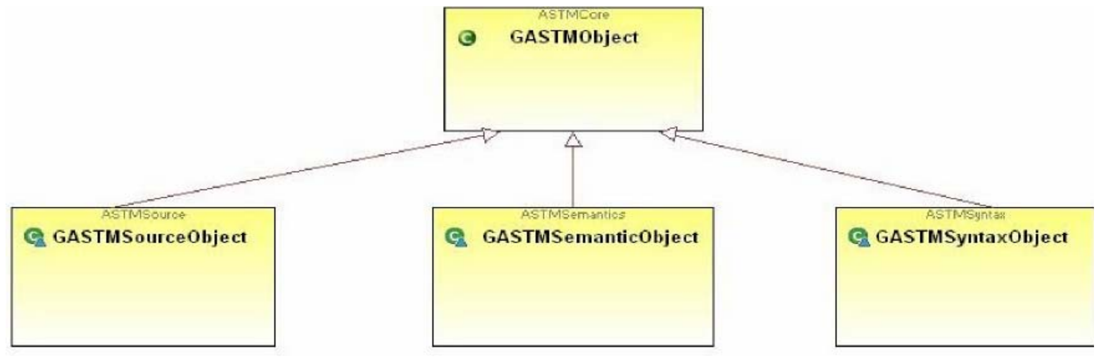


Figure 8.1 - GASTMObject Hierarchy

8.1.2 ASTM Core Semantic Object

The ASTM Core Semantics Specification defines a set of core modeling elements used for enrichment and abstraction of the ASTM abstract syntax tree to represent or derive the basic semantics of a formal language so as to enrich an abstract syntax tree to form an abstract semantic graph (ASG) for the basic semantics of code-level elements. This section provides the High-Level Composite UML Class Diagrams of the GASTMSemanticObject and its principal subclasses: Scope and Project. Project is a container for a collection of Compilation Units that contain the source code that is modeled. Scope and its subclasses are containers for Definitions that are defined in CompilationUnits.

- declOrDefn associates a Scope to the DefinitionObject it contains.
- childScope associates a parent Scope to its children Scopes.

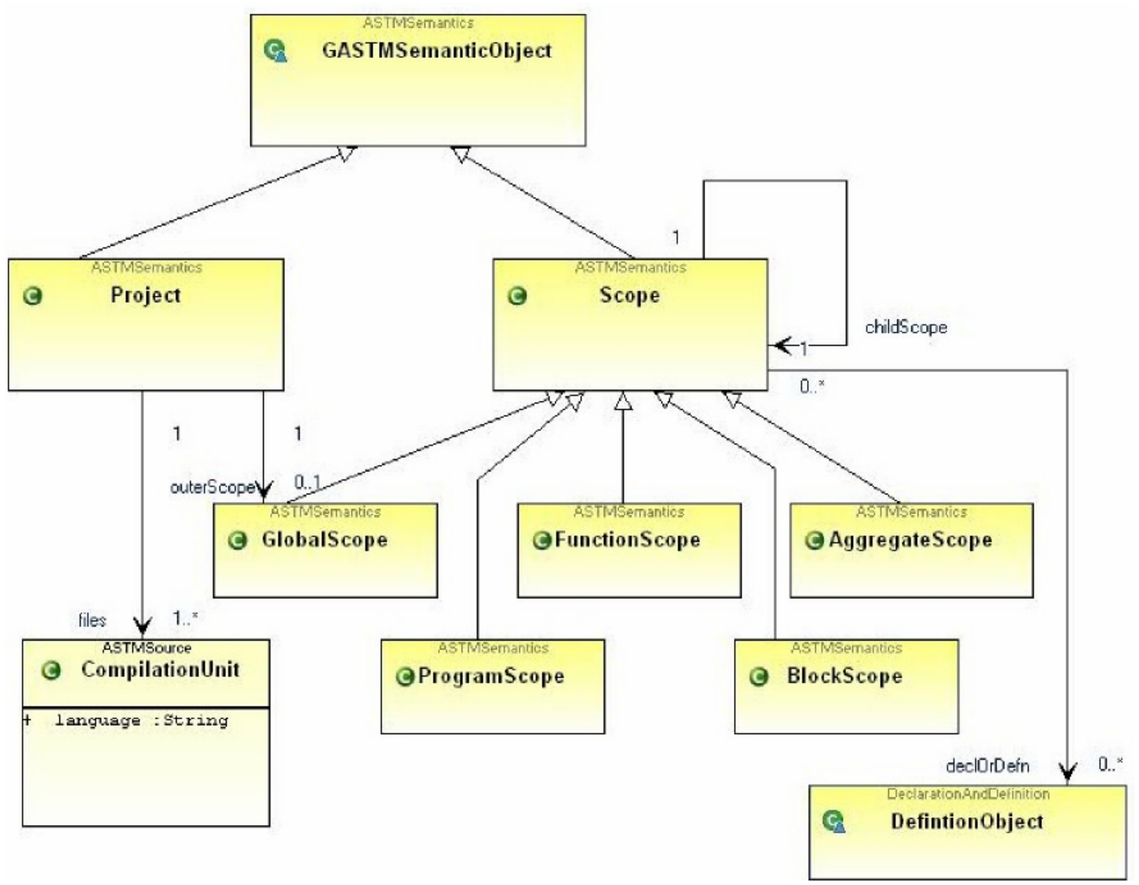


Figure 8.2 - GASTMSemantic Object

8.1.3 ASTM Core Source Object

The ASTM Core Source Specification defines a set of core modeling elements used for modeling the source file SourceFile and SourceLocation starting line, starting column, ending line and ending column of any ASTM modeling element. This section describes the High-Level Composite UML Class Diagrams of the GASTMSourceObject.

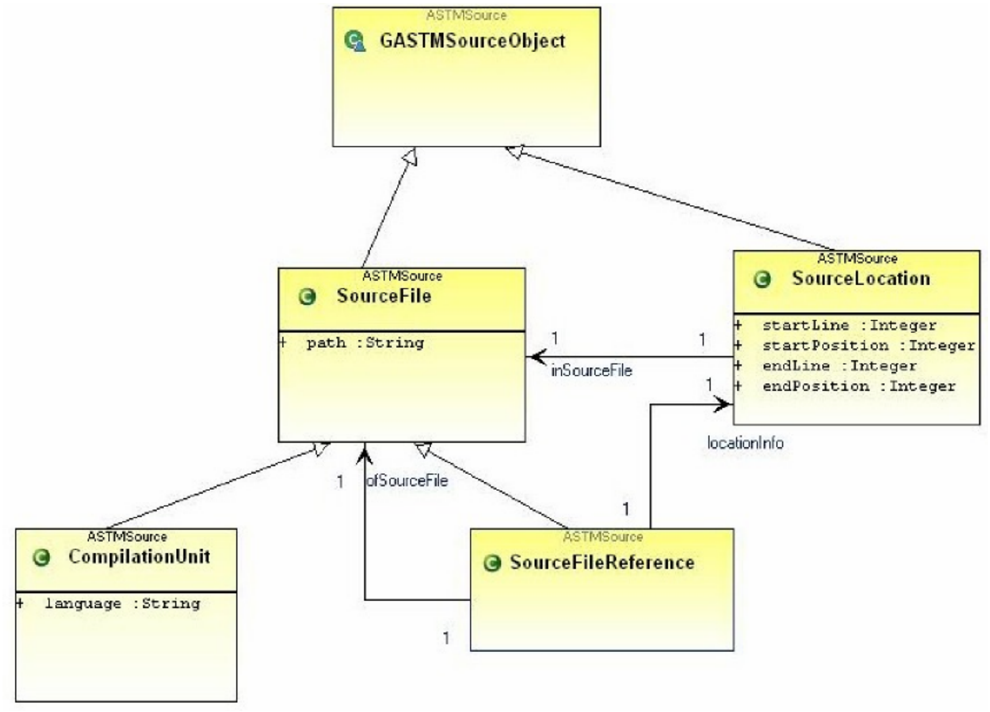


Figure 8.3 - GASTMSource Object

8.1.4 ASTM Core Syntax Object

The ASTM Core Syntax Specification defines a set of generic modeling elements commonly used in formal programming languages in the form of a finite, labeled, directed tree, where internal nodes are represented by the classes, the relationships between interior nodes are depicted by unidirectional associations, and the leaf nodes are depicted by associations to terminal primitive classes. This section provides the High-Level Composite UML Class Diagrams of the GASTMSyntaxObject.

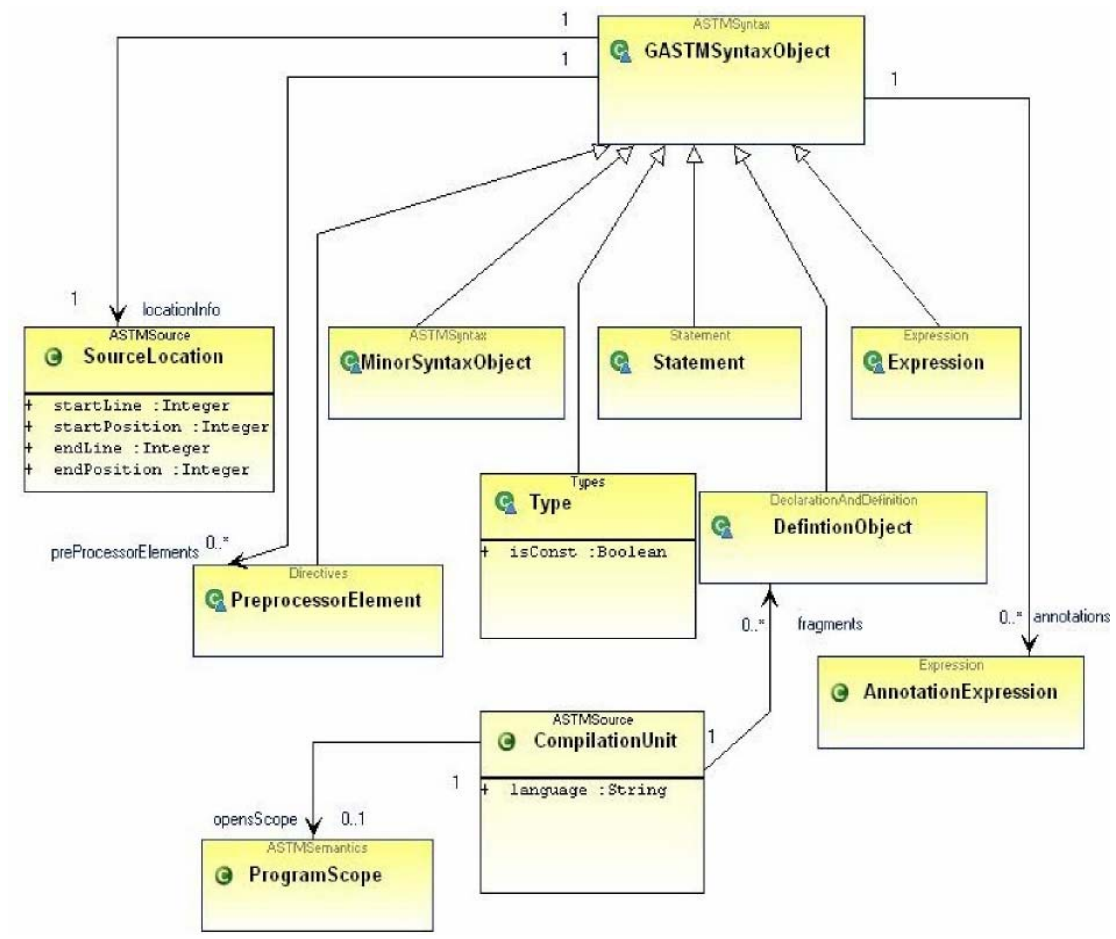


Figure 8.4 - GASTMSyntax Object

8.1.5 ASTM Core Preprocessor Objects

This section describes the High-Level Composite UML Class Diagrams of the PreprocessorElement. The PreprocessorElement subclasses MacroCall, MacroDefinition, Comment, and IncludeUnit are used for modeling the properties of preprocessor elements. Preprocessor elements model source code that is typically processed by a preprocessor and converted into the code that is to be processed by a parser or a compiler.

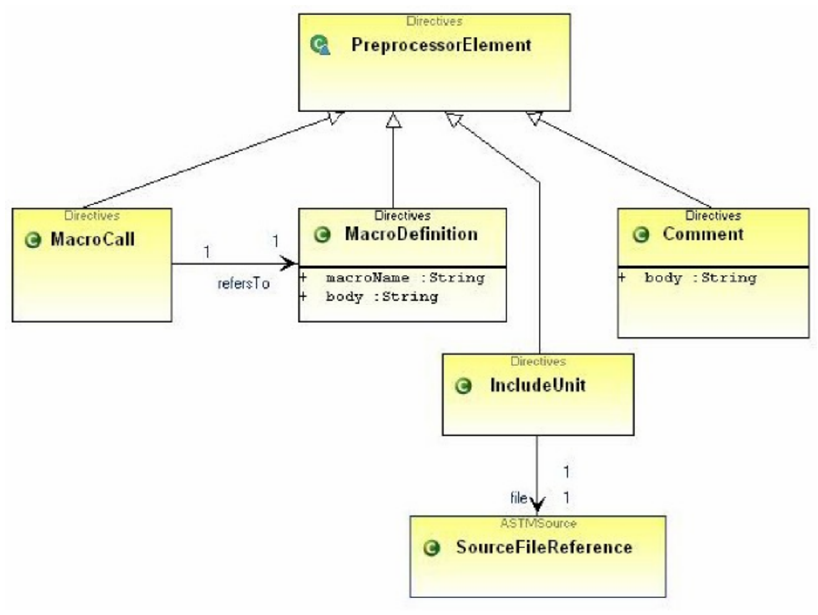


Figure 8.5 - Preprocessor Element

8.1.6 ASTM Core Definition Unit

This section describes the High-Level Composite UML Class Diagrams of the DefinitionObject and its principal subclasses LabelDefinition, TypeDefinition, DeclarationOrDefinition, NameSpaceDefinition, MinorSyntaxObject, and their subclasses.

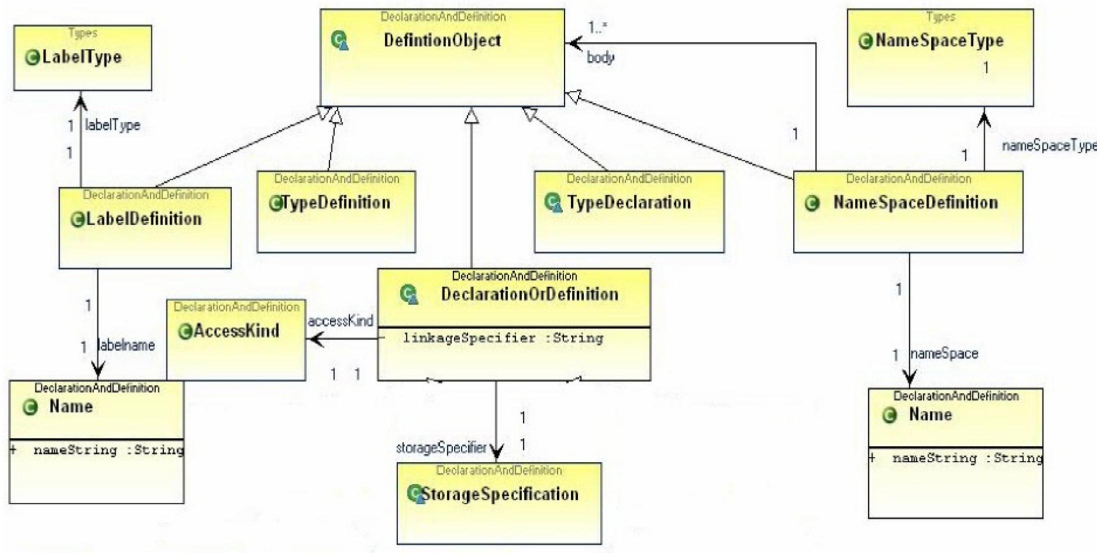


Figure 8.6 - DefinitionObject and its Subclasses

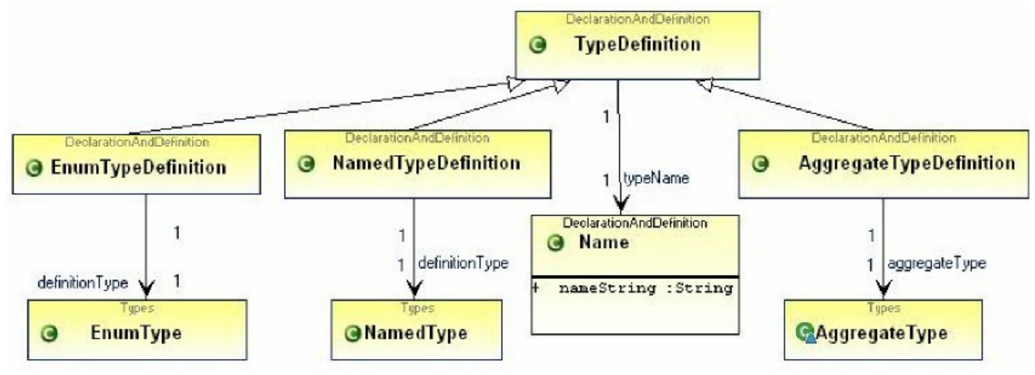


Figure 8.7 - TypeDefinition and its Subclasses

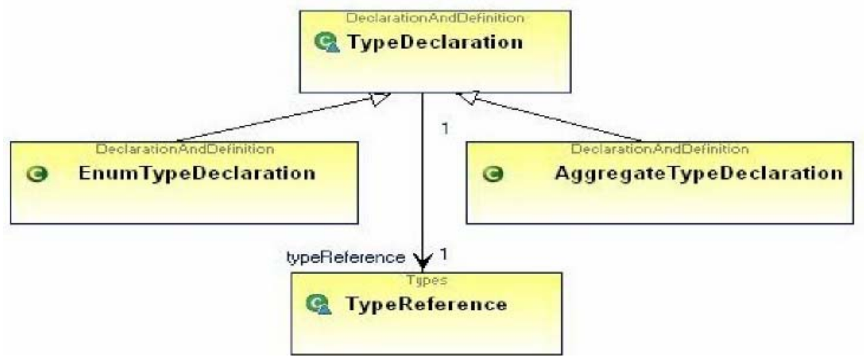


Figure 8.8 - TypeDeclaration Object

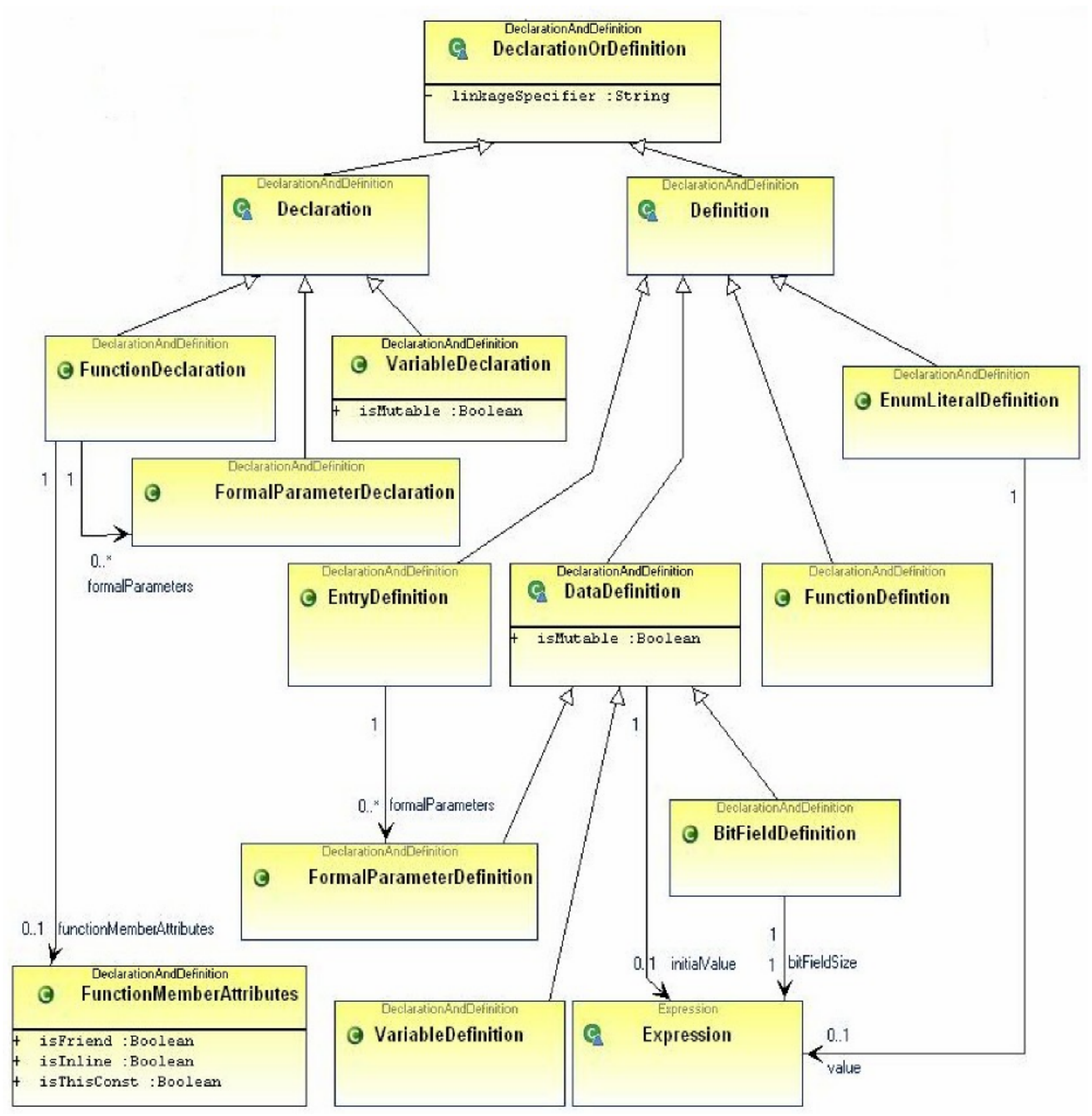


Figure 8.9 - DeclarationOrDefinitionObject and its Subclasses

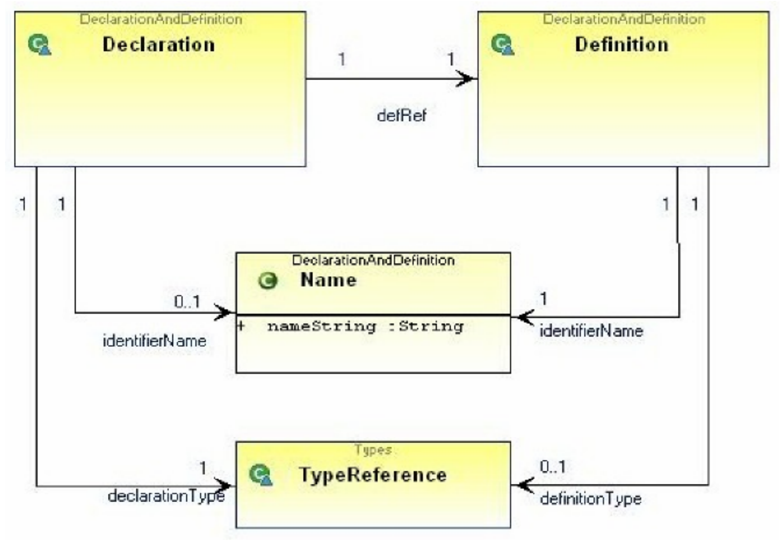


Figure 8.10 - Declaration and Definition Objects

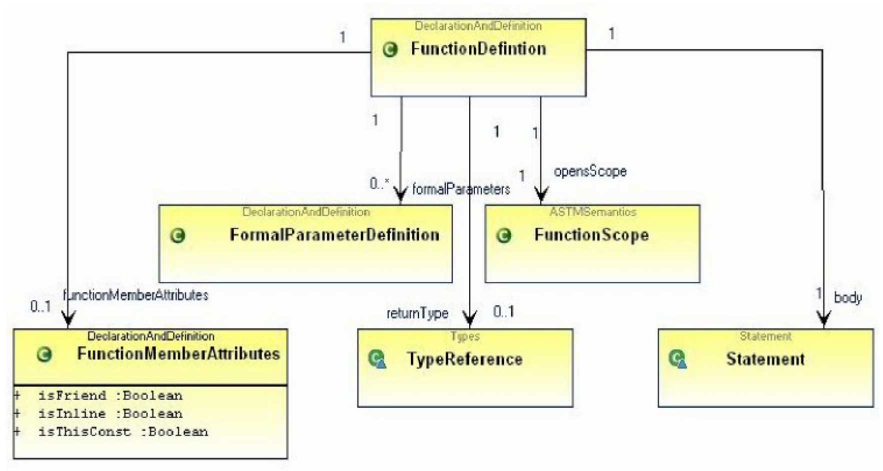


Figure 8.11 - FunctionDefinition

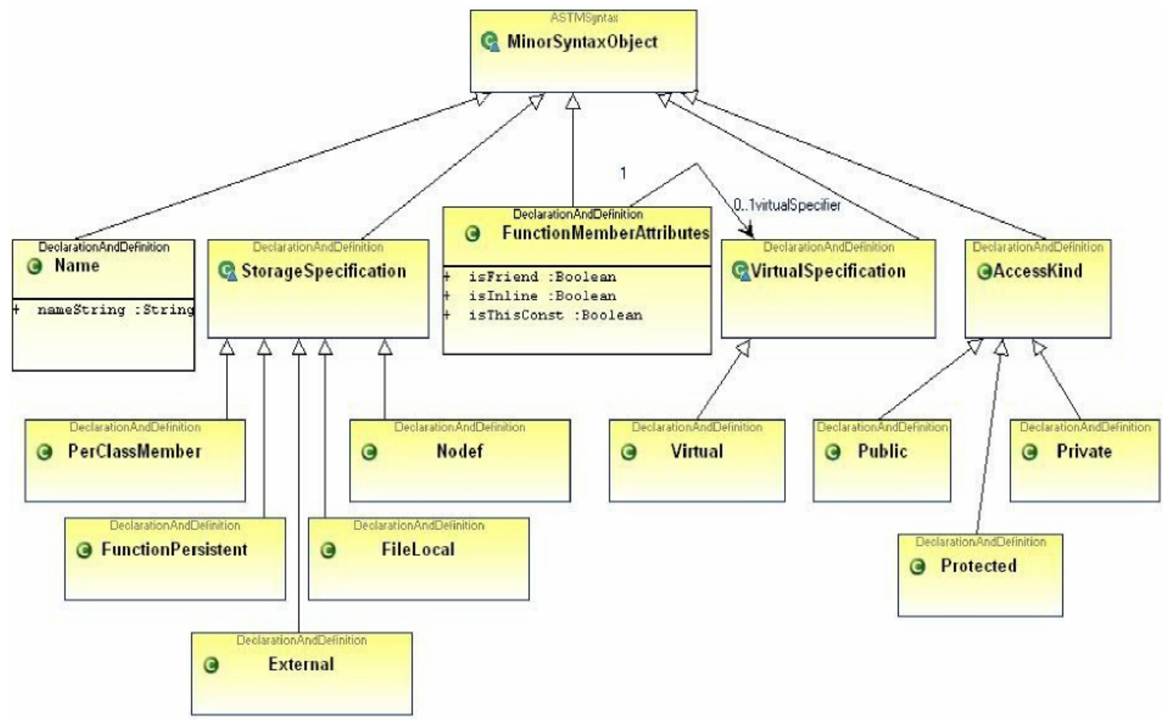


Figure 8.12 - MinorSyntaxObject (subclass belonging to DeclataionAndDefinition)

8.1.7 ASTM Core Types

This section describes the High-Level Composite UML Class Diagrams of Type, its principal subclasses DataType and their subclasses.

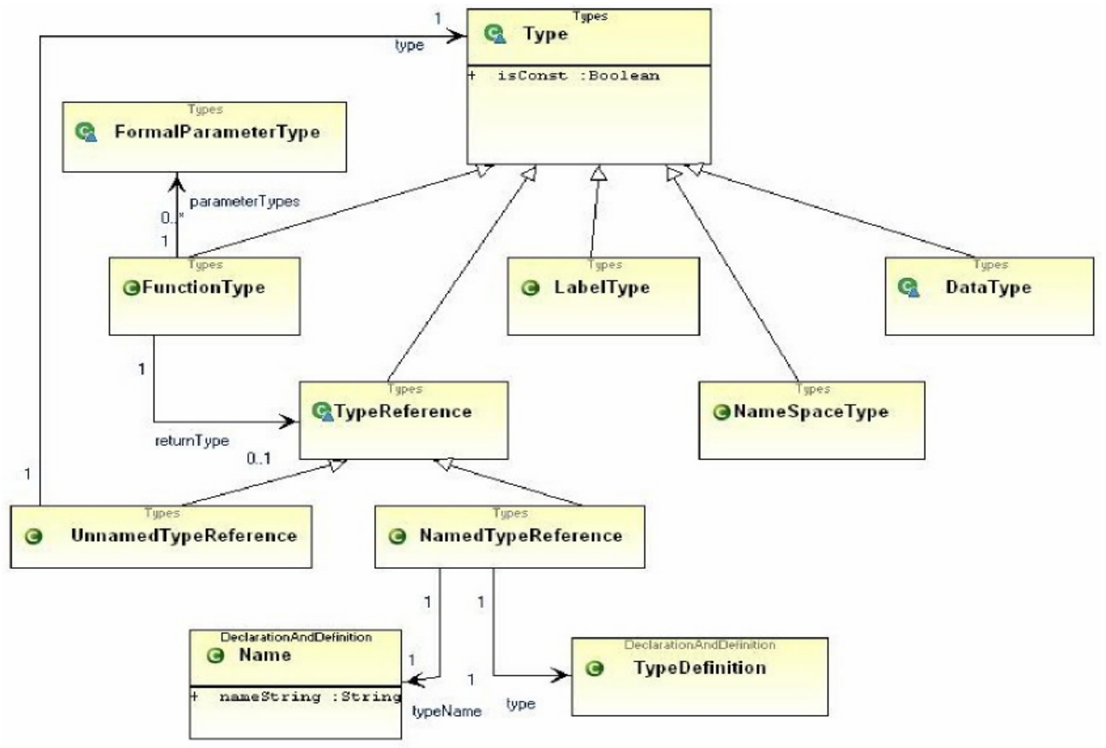


Figure 8.13 - Types

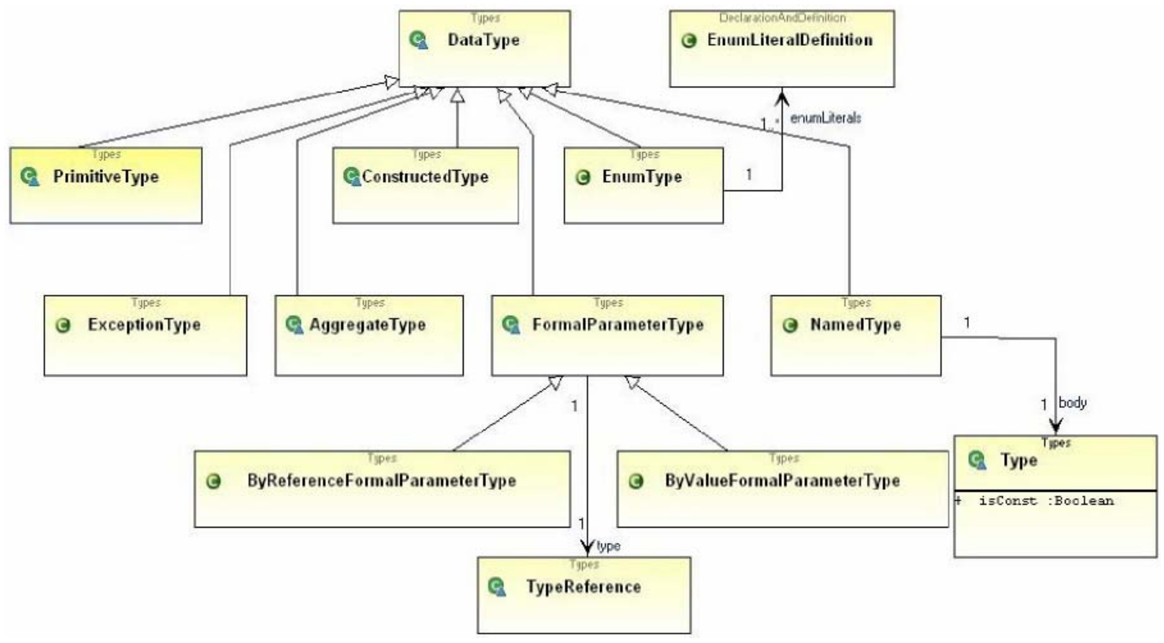


Figure 8.14 - DataType and its Subclasses

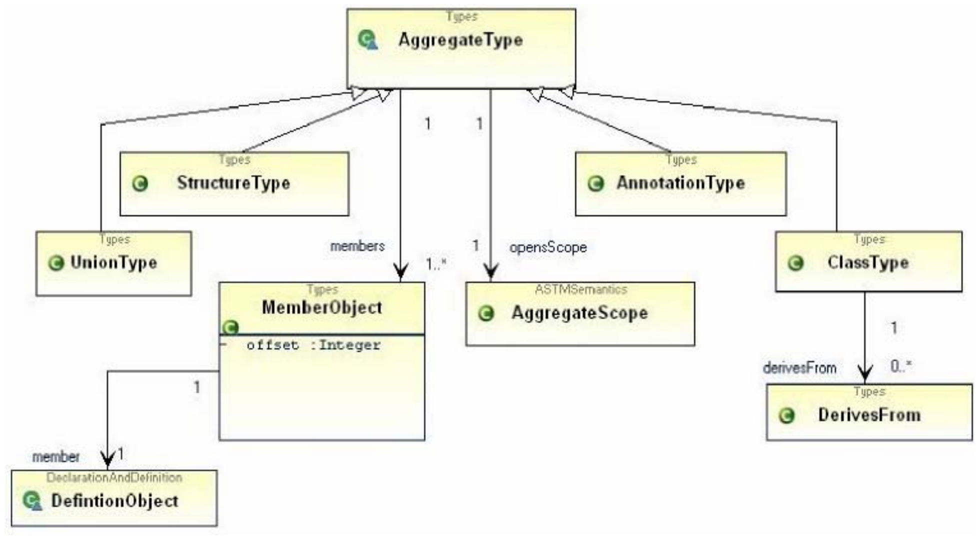


Figure 8.15 - AggregateType

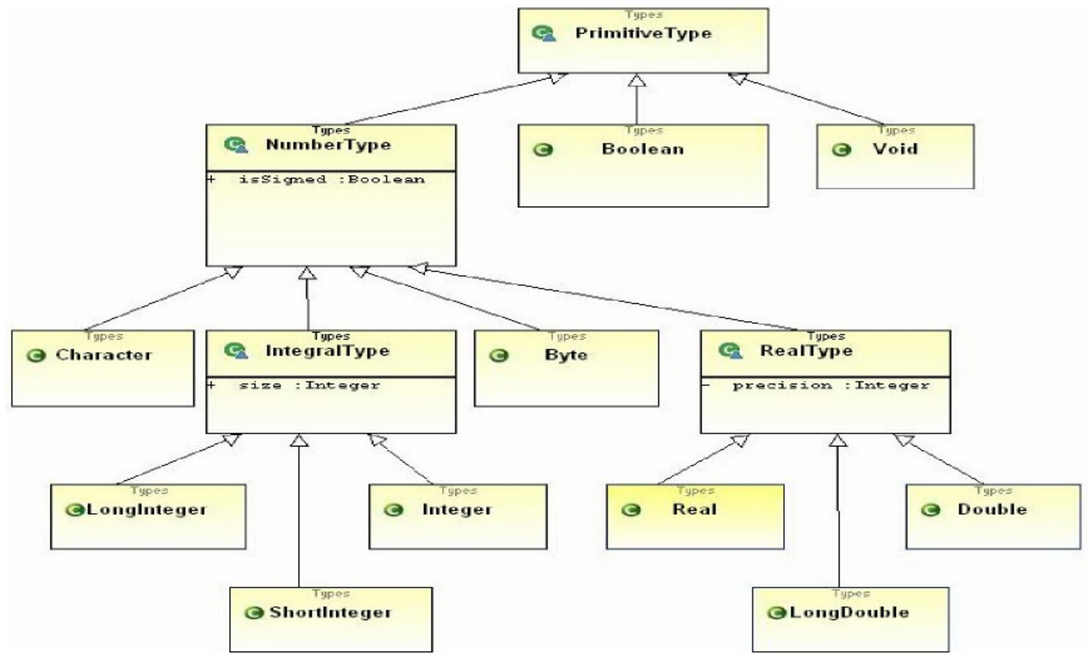


Figure 8.16 - PrimitiveType

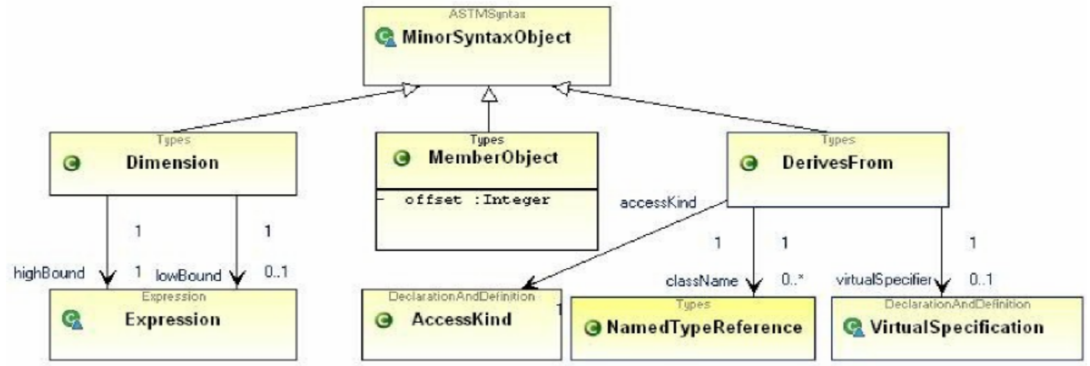


Figure 8.17 - MinorSyntaxObject (subclasses belonging to Types)

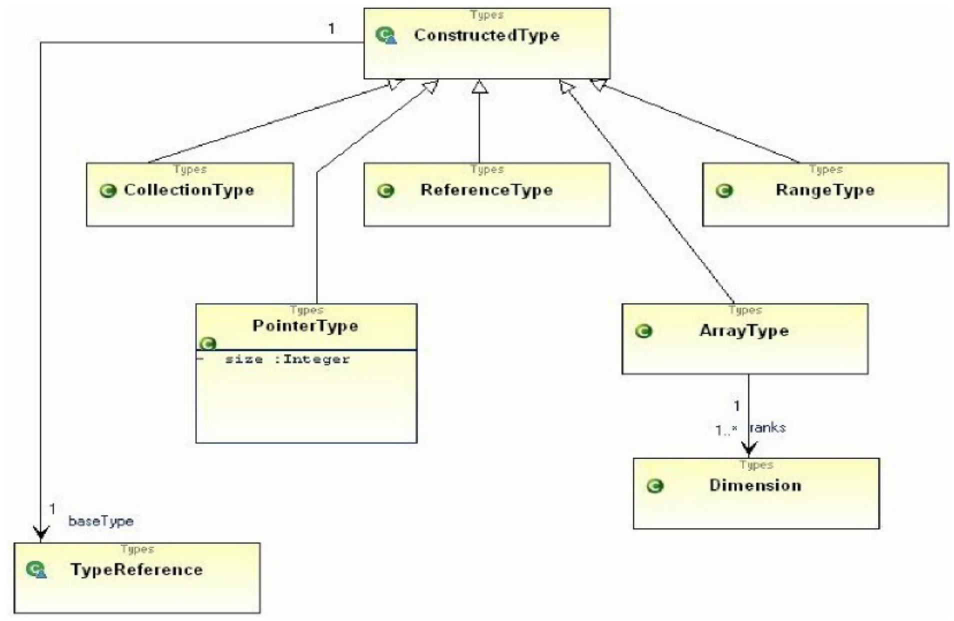


Figure 8.18 - ConstructedType

8.1.8 ASTM Core Statement

This section provides the High-Level Composite UML Class Diagrams of Statement, SwitchCase, CatchBlock, IfStatement, Return, Expression Statement and their subclasses.

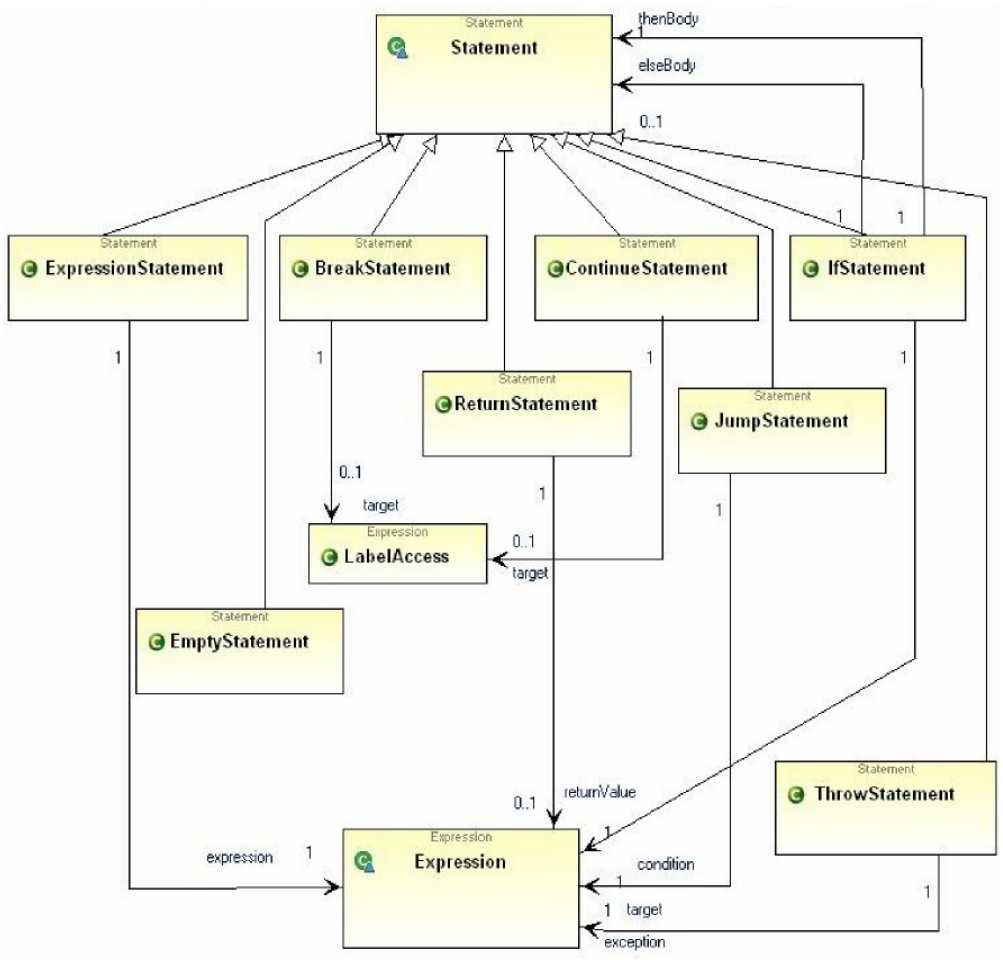


Figure 8.19 - Statement, Return, If and their Subclasses

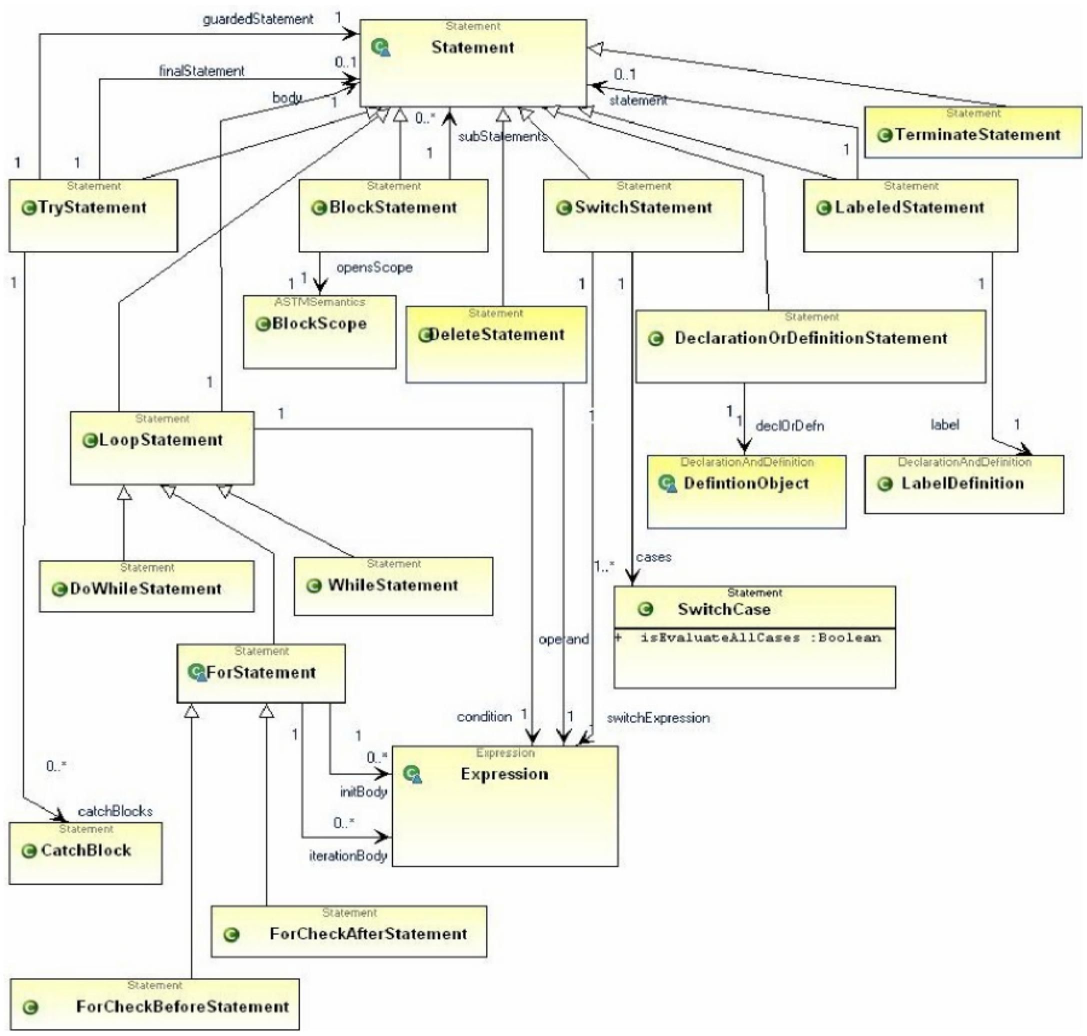


Figure 8.20 - Statement, SwitchCase, CatchBlock, and their Subclasses

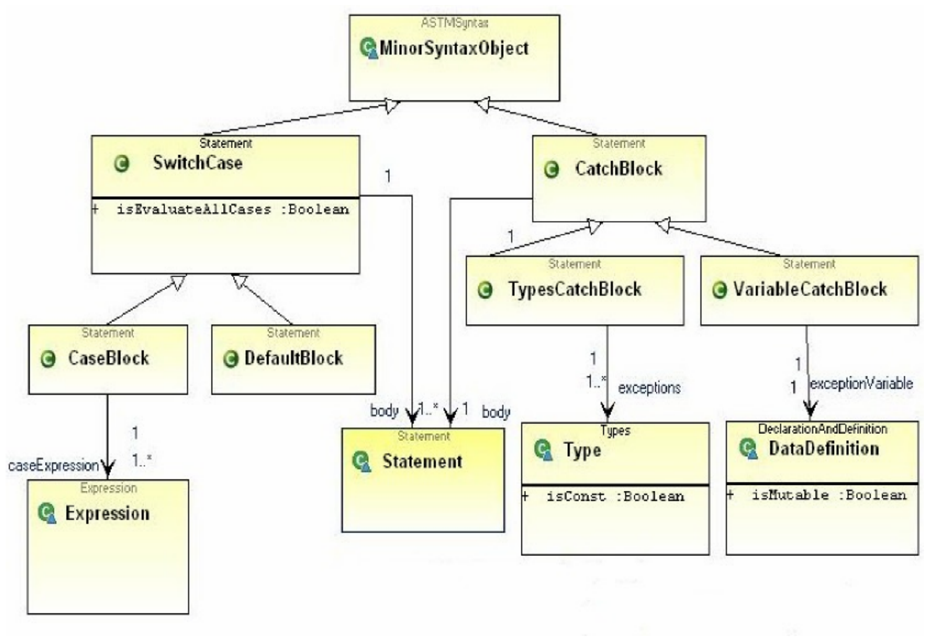


Figure 8.21 - MinorSyntaxObject (subclasses belonging to Statement)

8.1.9 ASTM Core Expression

This section describes the High-Level Composite UML Class Diagrams of Unary Operator, BinaryOperator, ActualParameter, Expression, Literal, NameReference, and their subclasses.

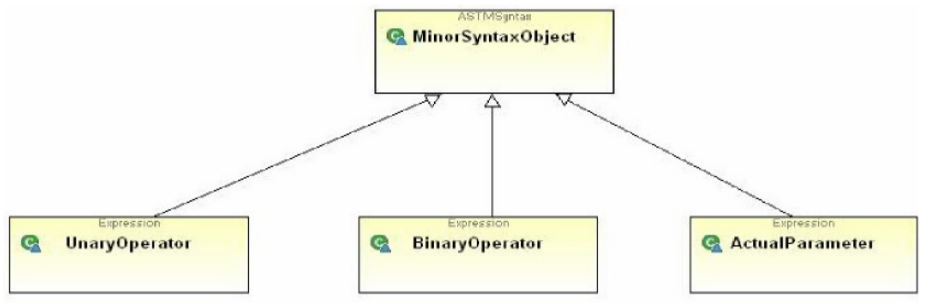


Figure 8.22 - MinorSyntaxObject (subclasses belonging to Expression)

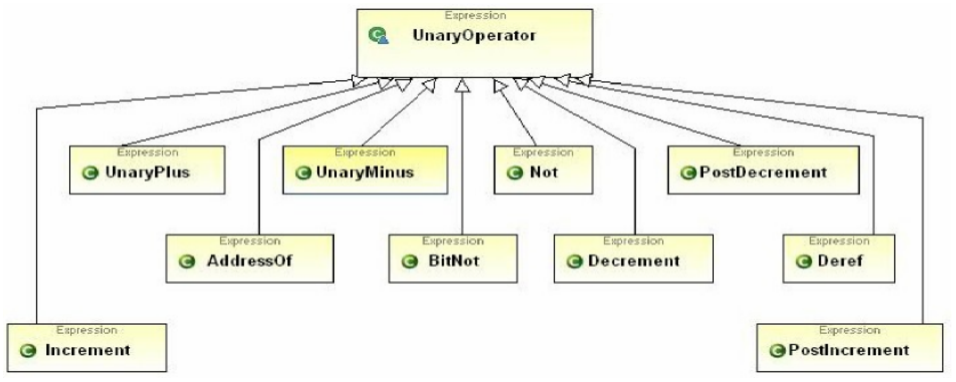


Figure 8.23 - UnaryOperator

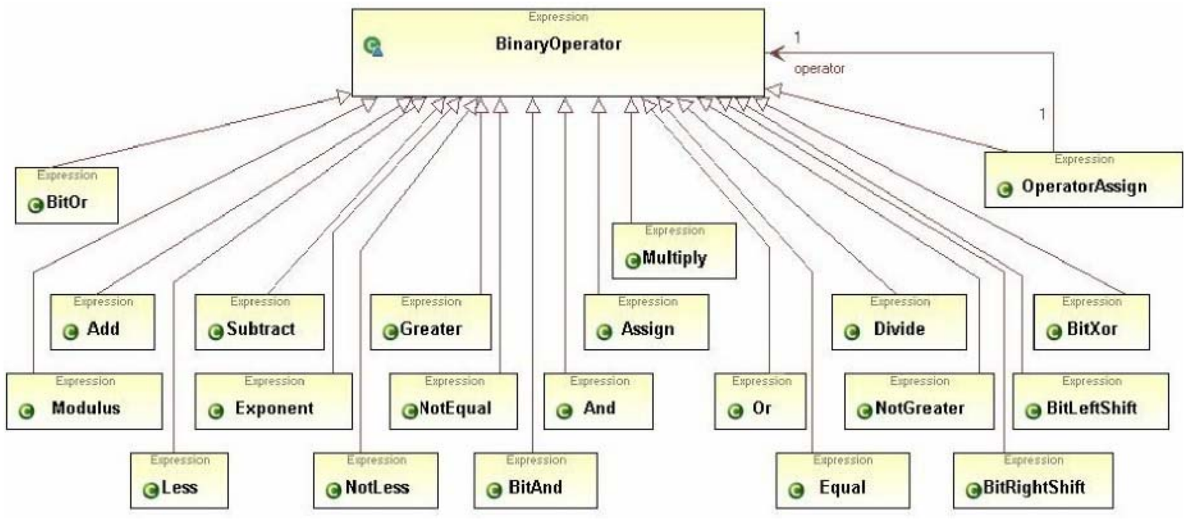


Figure 8.24 - BinaryOperator

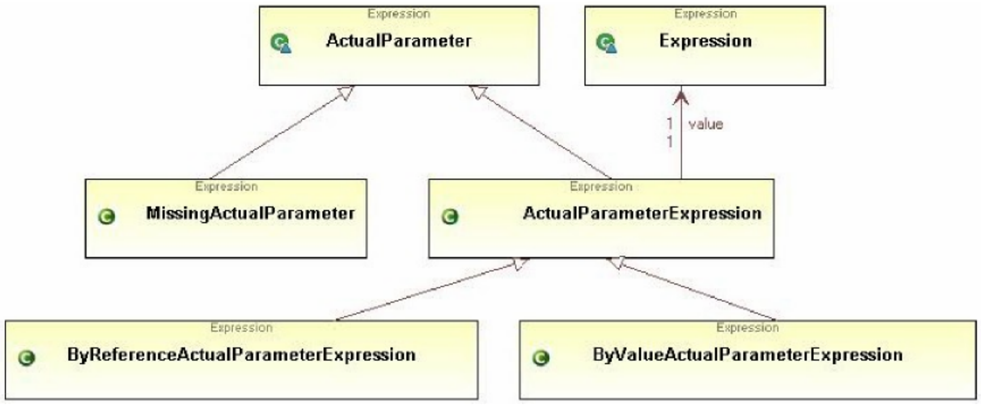


Figure 8.25 - ActualParameter

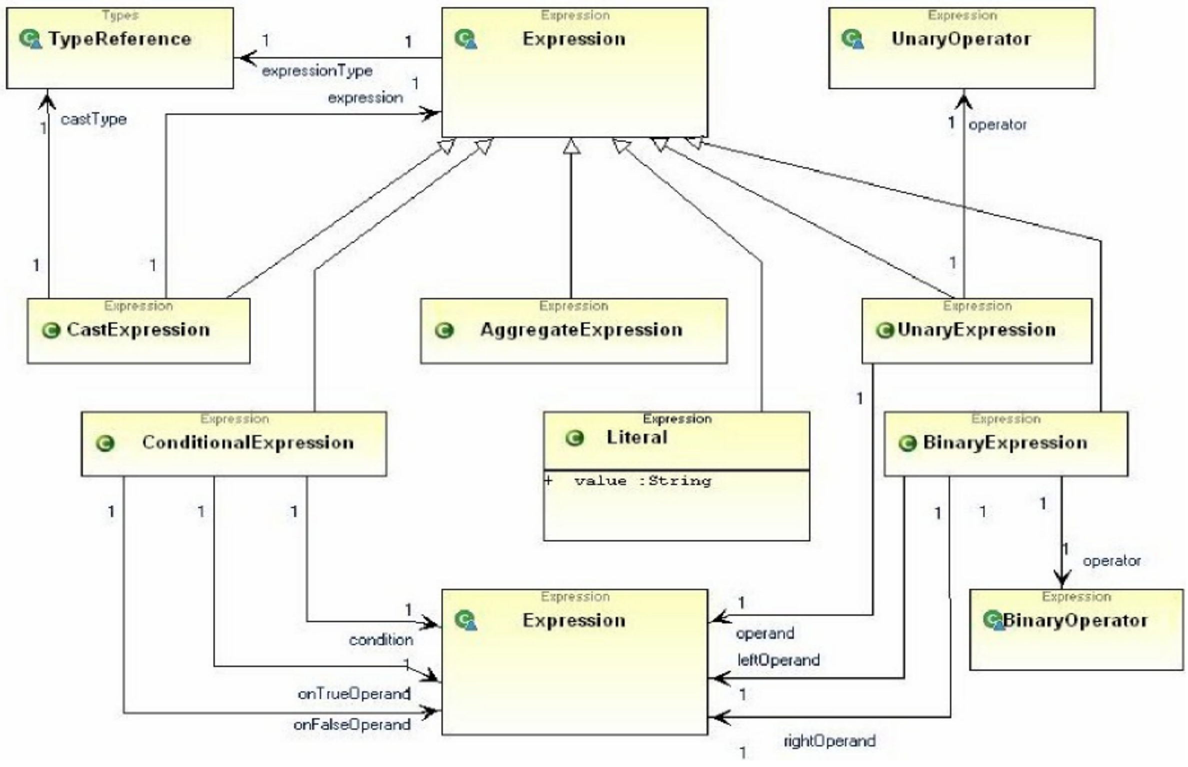


Figure 8.26 - Expression

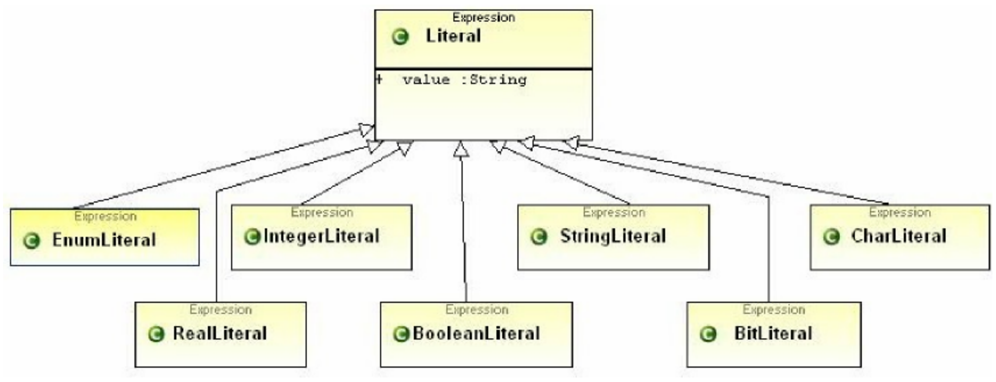


Figure 8.27 - Literal

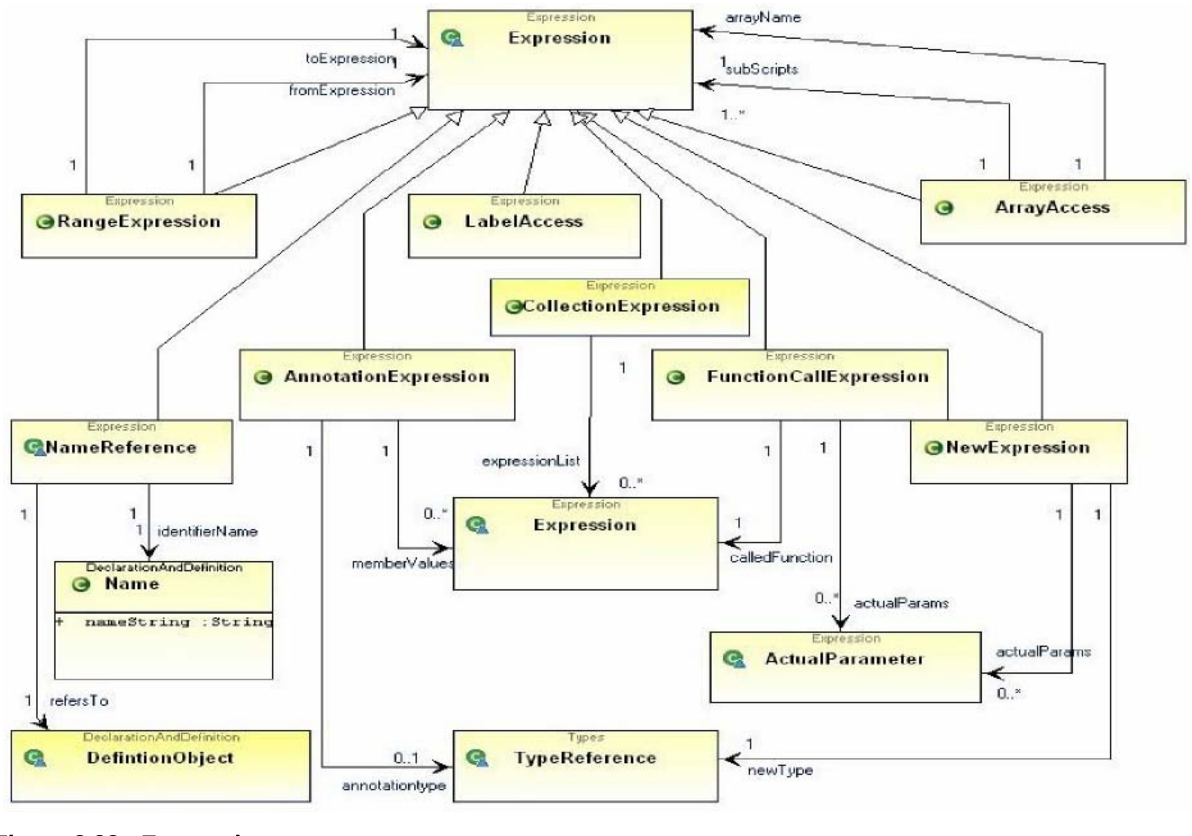


Figure 8.28 - Expression

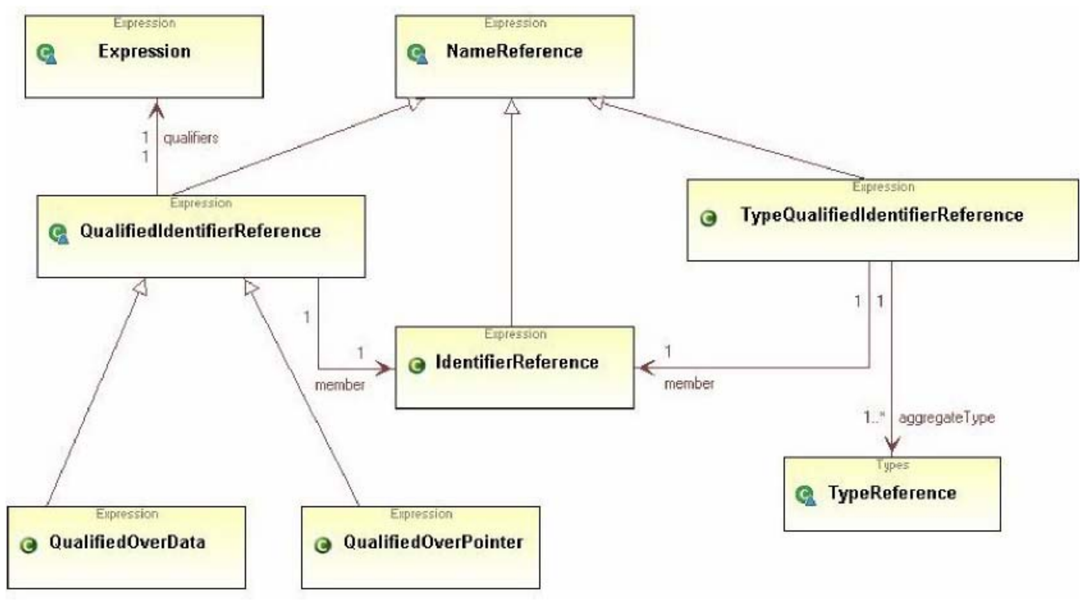


Figure 8.29 - NameReference

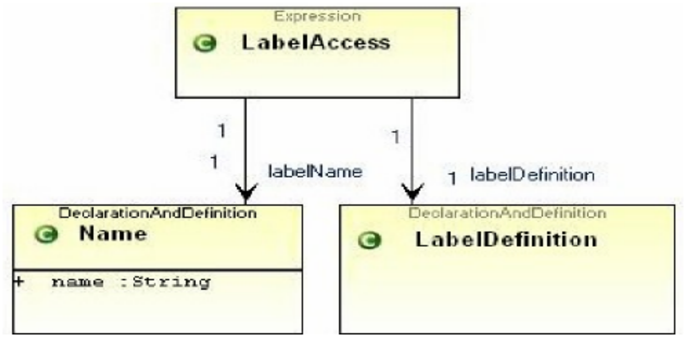


Figure 8.30 - LabelAccess

8.2 Low-Level (Detailed) GASTM Class Hierarchy

8.2.1 GASTMObject

The core of the GASTM consists of three abstract classes (denoted with a ! prefix) for depicting the syntactic, semantic, and source properties of programming language elements. The GASTMObject has abstract subclasses GASTMSourceObject, GASTMSemanticObject, and GASTMSyntaxObject.

Hierarchy Specification:

```
GASTMObject            =>     ! GASTMSyntaxObject
                       =>     ! GASTMSemanticObject
                       =>     ! GASTMSourceObject
;
```

Definition: The root of the GASTM class hierarchy.

8.2.1.1 GASTMSourceObject

GASTMSourceObject has subclasses SourceLocation and SourceFile.

Hierarchy Specification:

```
GASTMSourceObject      =>     SourceFile
                       =>     SourceLocation
;
```

Definition: Objects related to specifying locations within source files

8.2.1.1.1 SourceLocation

SourceLocation has unary Integer valued properties startLine, endPosition, endLine, and startPosition. The SourceLocation has unary association inSourceFile to SourceFile.

Property Specification:

```
SourceLocation         ->     < startLine : Integer >
                              < startPosition : Integer >
                              < endLine : Integer >
                              < endPosition : Integer > inSourceFile : SourceFile
;
```

Definition: Start/end line/column position information, part of a source location specification.

8.2.1.1.2 SourceFile

SourceFile is a subclass of GASTMSourceObject and has unary property path to String.

Property Specification:

```
SourceFile             ->     < path : String > ;
```

Definition: The source file part of a source location specification

8.2.1.2 GASTMSemanticObject

GASTMSemanticObject is a subclass of GASTMObject and has subclasses Project and Scope.

Hierarchy Specification:

```
GASTMSemanticObject
        =>     Project
        =>     Scope
;
```

Definition: Objects related to semantic artifacts of the modeled/analyzed system

8.2.1.2.1 Project

Project is a subclass of GASTMSemanticObject and has optional unary semantic association outerScope to GlobalScope. The Project one or more associations files to CompilationUnit.

Property Specification:

```
Project        ->     files : CompilationUnit +
                      [ outerScope : GlobalScope ? ]
;
```

Definition: The collection of compilation units to be modeled/analyzed as a whole

8.2.1.2.2 Scope

Scope is a subclass of GASTMSemanticObject and has 0 to any number associations childScope to Scope. Scope has 0 to any number association definitionObject to DefinitionObject. Scope has subclasses ProgramScope, AggregateScope, FunctionScope, GlobalScope, and BlockScope, which inherit the DeclDefn and ChildScope associations from Scope.

Scope and its subclasses are optional derivable semantic annotations.

Property Specification:

```
Scope          ->     definitionObject : DefinitionObject *
                      [ childScope : Scope * ];
                      !! Two way semantic association
;
```

Hierarchy Specification:

```
Scope          =>     FunctionScope
               =>     AggregateScope
               =>     BlockScope
               =>     ProgramScope
               =>     GlobalScope
;
```

Definition: Declaration context in which names declared must be unique

8.2.1.2.2.1 Function Scope

FunctionScope inherits the definitionObject and childScope associations. FunctionScope is an optional derivable semantic annotation.

Property Specification:

Hierarchy Specification:

Definition: The scope introduced by a function definition

8.2.1.2.2.2 Aggregate Scope

AggregateScope inherits the definitionObject and ChildScope associations. AggregateScope is an optional derivable semantic annotation.

Definition: The scope introduced by an aggregate type

8.2.1.2.2.3 Block Scope

BlockScope inherits the definitionObject and ChildS cope associations.

BlockScope is an option derivable semantic annotation.

Definition: The scope introduced by a block statement

8.2.1.2.2.4 Program Scope

ProgramScope inherits the definitionObject and ChildScope associations.

ProgramScope Scope is an optional derivable semantic annotation.

Definition: The scope introduced by a compilation unit

8.2.1.2.2.5 Global Scope

GlobalScope inherits the definitionObject and ChildScope associations. ProgramScope Scope is an optional derivable semantic annotation.

Definition: The outermost scope, surrounding all compilation units of a project

8.2.1.3 GASTMSyntaxObject

GASTMSyntaxObject is a subclass of GASTMObject and has immediate subclasses PreprocessorElement, DefinitionObject, Type, Expression, Statement, and MinorSyntaxObject. The GASTMSyntaxObject has zero to any number association preProcessorElements to PreprocessorElement. The GASTMSyntaxObject has zero to any number association annotations to AnnotationExpression. The GASTMSyntaxObject has optional unary association locationInfo to SourceLocation (a subclass of GASTMSourceObject).

Property Specification:

```
GASTMSyntaxObject
             ->     locationInfo : SourceLocation
                    preProcessorElements : PreprocessorElement * annotations :
                    AnnotationExpression *
;
```

Hierarchy Specification:

```
GASTMSyntaxObject
             =>     ! PreprocessorElement
             =>     ! DefinitionObject
             =>     ! Type
             =>     ! Expression
             =>     ! Statement
             =>     ! MinorSyntaxObject
;
```

Definition: All syntactic constructs

8.2.1.3.1PreprocessorEelement

PreprocessorElement is a subclass of GASTMSyntaxObject and has subclasses IncludeUnit, MacroCall, MacroDefinition and Comment.

Hierarchy Specification:

```
PreprocessorElement
```

```
                 =>     IncludeUnit
                 =>     MacroCall
                 =>     MacroDefinition
                 =>     Comment
        ;
```

Definition: Constructs involved in preprocessing

8.2.1.3.1.1 IncludeUnit

IncludeUnit is a subclass of PreprocessorElement and has unary association file to SourceFile.

Property Specification:

```
        IncludeUnit   ->     file : SourceFileReference
        ;
```

Definition: Inclusion of a file during preprocessing

8.2.1.3.1.2 MacroCall

A MacroCall is a subclass of PreprocesorElement and has unary association RefersTo to a MacroDefinition.

Property Specification:

```
        MacroCall     ->     refersTo : MacroDefinition
        ;
```

Definition: Invocation of a preprocessor macro

8.2.1.3.1.3 MacroDefinition

MacroDefinition is a subclass of PreprocessorElement and has unary property macroName to String and unary property body to String.

Property Specification:

```
        MacroDefinition     ->     < macroName : String >
                                   < body : String >
        ;
```

Definition: Definition of a preprocessor macro

8.2.1.3.1.4 Comment

Comment is a subclass of PreprocessorElement and has unary property body to String.

Property Specification:

```
        Comment             ->     < body : String >
```

Definition: Comments appearing in source files

8.2.1.3.2 DefinitionObject

DefinitionObject is a subclass of GASTMSyntaxObject and has immediate subclasses DeclarationOrDefinition. TypeDefinition, NamespaceDefinition, and LabelDefinition.

Hierarchy Specification:

```
        DefinitionObject    =>     ! DeclarationOrDefinition
                            =>     TypeDefinition
```

```
                        =>     NameSpaceDefinition
                        =>     LabelDefinition
                        =>     TypeDeclaration
        ;
```

Definition: Constructs that define entities

8.2.1.3.3 DeclarationOrDefinition

DeclarationOrDefinition is a subclass of GASTMSyntaxObject and has immediate subclasses Declaration, Definition. DeclarationOrDefinition has unary association storageSpecifiers to class StorageSpecification, unary association accessKind to class AccessKind, unary property linkageSpecifier to primitive String.

Property Specification:

```
        DeclarationOrDefinition
                        ->     storageSpecifiers : StorageSpecification
                               accessKind : AccessKind
                               < linkageSpecifier : String >
        ;
```

Hierarchy Specification:

```
        DeclarationOrDefinition
                        =>     ! Declaration
                        =>     ! Definition
        ;
```

Definition: Declarations and definitions

8.2.1.3.3.1 Declaration

Declaration is a subclass of DeclarationOrDefinition and has immediate subclasses VariableDeclaration and FunctionDeclaration and FormalParameterDeclaration. Declaration has unary semantic association defRef to Definition (This association is hidden below, but is shown above). Declaration has optional unary association identifierName to class Name.

Property Specification:

```
        Declaration     ->     [ defRef : Definition ]
                               identifierName : Name ?
        ;
```

Hierarchy Specification:

```
        Declaration     =>     FunctionDeclaration
                        =>     VariableDeclaration
                        =>     FormalParameterDeclaration
        ;
```

Definition: Constructs that declare entities without defining them

8.2.1.3.3.1.1 FunctionDeclaration

FunctionDeclaration is a subclass of Declaration, and has zero to any number of association formalParameters to class FormalParameterDeclaration, optional unary association functionMemberAttributes to class FunctionMemberAttributes.

Property Specification:

```
        FunctionDeclaration
```

```
                ->     formalParameters :
                       FormalParameterDeclaration * functionMemberAttributes :
                       FunctionMemberAttributes?
        ;
```

Definition: Function declarations

8.2.1.3.3.1.2 VariableDeclaration

VariableDeclaration is a subclass of Declaration and has unary property isMutable to Boolean.

Property Specification:

```
        VariableDeclaration
                ->     < isMutable : Boolean >
        ;
```

Definition: Variable declarations

8.2.1.3.3.1.3 FormalParameterDeclaration

FormalParameterDeclaration is a subclass of Declaration.

Definition: Formal Parameter Declarations, appearing in function declarations

8.2.1.3.3.2 Definition

Definition is a subclass of DeclarationOrDefinition, and has immediate subclasses FunctionDefinition, EntryDefinition, DataDefinition, EnumLiteralDefinition. Definition has unary association unary association identifierName to class Name, and optional unary association definitionType to TypeReference.

Property Specification:

```
        Definition          ->     identifierName : Name
                                   definitionType : TypeReference ?
                                   !! To allow K&R C formal parameter defn
        ;
```

Hierarchy Specification:

```
        Definition          =>     FunctionDefinition
                            =>     EntryDefinition
                            =>     ! DataDefinition
                            =>     EnumLiteralDefinition
        ;
```

Definition: Constructs that declare entities as they also define them

8.2.1.3.3.2.1 FunctionDefinition

FunctionDefinition is a subclass of Definition, and has unary association Body to class Statement, any number association FormalParameters to class FormalParameter, unary association FunctionMemberAttributes to class FunctionMemberAttributes, and optional unary semantic association OpensScope to the semantic class FunctionScope.

Property Specification:

```
        FunctionDefinition
                            ->     returnType : TypeReference ?
                                   formalParameters : FormalParameterDefinition *
```

```
                                        body : Statement*
                                        functionMemberAttributes : FunctionMemberAttributes?
                                        [ opensScope : FunctionScope ]
                                  ;
```

Definition: Subprogram definitions

8.2.1.3.3.2.2 EntryDefinition

EntryDefinition is a subclass of Definition, and has unary association Body to class Statement. EntryDefinition has zero to any number association FormalParameters to class FormalParameter.

Property Specification:

```
EntryDefinition      ->     formalParameters : FormalParameterDefinition*
                            body: Statement*
;
```

Definition: Subprogram entry definitions

8.2.1.3.3.2.3 DataDefinition

DataDefinition is a subclass of Definition, and has subclasses VariableDefinition, FormalParameter, and BitFieldDefinition. DataDefinition has unary property isMutable to primitive Boolean, and unary association initial Value to Expression.

Property Specification:

```
DataDefinition       ->     initialValue : Expression?
                            < isMutable : Boolean >
;
```

Definition: Definitions involving data

8.2.1.3.3.2.4 VariableDefinition

VariableDefinition is a subclass of DataDefinition, and has no subclasses, no immediate associations and no immediate properties.

Definition: Variable definitions

8.2.1.3.3.2.5 FormalParameterDefinition

FormalParameterDefinition is a subclass of DataDefinition, and has no subclasses, no immediate associations and no immediate properties.

Definition: Formal parameter definitions, appearing in function definitions

8.2.1.3.3.2.6 BitFieldDefinition

Bitfield is a subclass of DataDefinition, and unary association bitfieldSize to Expression.

Property Specification:

```
BitFieldDefinition
             ->     bitFieldSize : Expression
;
```

Definition: Definitions of bit-field data

8.2.1.3.3.2.7 EnumLiteralDefinition

EnumLiteralDefinition is a subclass of Definition, and has unary association value to class Expression.

Property Specification:

```
EnumLiteralDefinition
                    ->     value : Expression?
;
```

Definition: Definitions of enumerals (members of enumerated types)

8.2.1.3.3.3 TypeDefinition

TypeDefinition is a subclass of DefinitionObject and has unary association name to class typeName, and subclasses NamedTypeDefinition, AggregateTypeDefinition, and EnumTypeDefinition.

Property Specification:

```
TypeDefinition          ->     typeName : Name
;
```

Hierarchy Specification:

```
TypeDefinition          =>     NamedTypeDefinition
                        =>     AggregateTypeDefinition
                        =>     EnumTypeDefinition
;
```

Definition: Definitions of types

8.2.1.3.3.3.1 NamedTypeDefinition

NamedTypeDefinition is a subclass TypeDefinition and has unary definitionType to NamedType.

Property Specification:

```
NamedTypeDefinition
                        ->     definitionType : NamedType
;
```

Definition: Definitions of types to be referred to by a specified name

8.2.1.3.3.3.2 AggregateTypeDefinition

AggregateTypeDefinition is a subclass TypeDefinition and has unary aggregateType to AggregateType.

Property Specification:

```
AggregateTypeDefinition
                    ->     aggregateType : AggregateType
;
```

Definition: Definitions of aggregate types

8.2.1.3.3.3.3 EnumTypeDefinition

EnumTypeDefinition is a subclass TypeDefinition and has unary association definitionType to EnumType.

Property Specification:

```
        EnumTypeDefinition
                        ->     definitionType : EnumType
        ;
```

Definition: Definitions of enumeration types

8.2.1.3.3.4 NamespaceDefinition

NamespaceDefinition is a subclass of DeclarationOrDefinition, and one or more association body to class DeclarationOrDefinition, unary association nameSpaceType to NamespaceType, and unary association nameSpace to class Name.

```
        NameSpaceDefinition
                        ->     nameSpace : Name
                               body : DefinitionObject+
                               nameSpaceType : NameSpaceType
        ;
```

Property Specification:

Hierarchy Specification:

Definition: Definitions of namespaces

8.2.1.3.3.5 LabelDefinition

LabelDefinition is a subclass of DeclarationOrDefinition, and unary association labelName to Name, and unary association labelType to LabelType.

Property Specification:

```
        LabelDefinition     ->     labelName : Name
                                   labelType : LabelType
        ;
```

Definition: Definitions of labels

8.2.1.3.3.6 TypeDeclaration

TypeDeclaration is a subclass of DefinitionObject and has unary association typeReference to class TypeReference, and subclasses AggregateTypeDeclaration, and EnumTypeDeclaration.

Property Specification:

```
        TypeDeclaration     ->     typeReference : TypeReference
        ;
```

Hierarchy Specification:

```
        TypeDeclaration     =>     AggregateTypeDeclaration
                            =>     EnumTypeDeclaration
        ;
```

Definition: Forward Declaration of user-defined types (aggregates and enumerations)

8.2.1.3.3.6.1 AggregateTypeDeclaration

AggregateTypeDeclaration is a subclass TypeDeclaration.

Definition: Forward declaration of AggregateType

8.2.1.3.3.6.2 EnumTypeDeclaration

EnumTypeDeclaration is a subclass TypeDeclaration.

Definition: Forward declaration of EnumerationType

8.2.1.3.4 Type

Type is a subclass of GASTMSyntaxObject, and has subclasses DataType, FunctionType, LabelType and NamespaceType and TypeReference that are used for depicting categories of Type, and unary property isConst to the primitive Boolean.

Property Specification:

```
Type                ->      < isConst : Boolean >
;
```

Hierarchy Specification:

```
Type                =>      FunctionType
                    =>      ! DataType
                    =>      LabelType
                    =>      Name spaceType
                    =>      ! TypeReference
;
```

Definition: All types

8.2.1.3.4.1 FunctionType

FunctionType is a subclass of Type, and has zero to any association parameterTypes to class FormalParameterType and unary association returnType to class Type.

Property Specification:

```
FunctionType        ->      returnType : TypeReference?
                            parameterTypes : FormalParameterType*
;
```

Definition: Function types

8.2.1.3.4.2 DataType

DataType is a subclass of Type, and has subclasses PrimitiveType, EnumType, ConstructedType, AggregateType, ExceptionType, FormalParameterType, NamedType that are used for depicting kinds of datatypes.

```
DataType            =>      ! PrimitiveType
                    =>      EnumType
                    =>      ! ConstructedType
                    =>      ! AggregateType
                    =>      ExceptionType
                    =>      ! FormalParameterType
                    =>      NamedType ;
```

Definition: Types involving data

8.2.1.3.4.2.1 PrimitiveType

PrimitiveType is a subclass of DataType, and has terminal subclasses Boolean and Void, and subclass NumberType that is used to represent all signed types.

Property Specification:

Hierarchy Specification:

```
PrimitiveType        =>     ! NumberType
                     =>     Void,
                     =>     Boolean
;
```

Definition: Void and Boolean are primitive types (not further decomposable), Represents all other signed and unsigned primitive types also.

`Void`

Definition: Void type

`Boolean`

Definition: Boolean type

8.2.1.3.4.2.2 NumberType

NumberType is a subclass of PrimitiveType, and has terminal subclasses Byte, and Character, and subclasses IntegralType and RealType that is used to represent numeral types, and has property isSigned to primitive Boolean.

Property Specification:

```
NumberType           ->     < isSigned : Boolean >
;
```

Hierarchy Specification:

```
NumberType           =>     ! IntegralType
                     =>     ! RealType
                     =>     Byte
                     =>     Character
;
```

Definition: Byte and Character are primitive types (not further decomposable), Represents all other signed and unsigned numeral types also.

`Byte`

Definition: Byte type

`Character`

Definition: Character type

Sematics: Character is denoted by the variable length character encoding for Unicode (UTF16).

8.2.1.3.4.2.3 IntegralType

IntegralType is a subclass of NumberType, and has terminal subclasses ShortInteger, Integer and LongInteger, and has optional semantic property size of type Integer.

Property Specification:

```
IntegralType                ->     [ < size : Integer > ]
;
```

Hierarchy Specification:

```
    IntegralType                  =>     ShortInteger
                                  =>     Integer
                                  =>     LongInteger
    ;
```

Definition: ShortInteger, Integer, LongInteger are primitive types (not further decomposable). Enables to specify their sizes.

`ShortInteger`

Definition: Short integer type

`Integer`

Definition: Integer type

`LongInteger`

Definition: Long integer type

8.2.1.3.4.2.4 RealType

RealType is a subclass of NumberType, has terminal subclasses Real, Double and LongDouble, and has an optional semantic attribute precision of type Integer.

Property Specification:

```
    RealType                      ->     [ < precision : Integer > ]
    ;
```

Hierarchy Specification:

```
    RealType                      =>     Real
                                  =>     Double
                                  =>     LongDouble
    ;
```

Definition: Real, Double, LongDouble are primitive types (not further decomposable). Optionally, precision can be specified.

`Real`

Definition: Short floating-point type

`Double`

Definition: Floating-point type

`LongDouble`

Definition: Long floating-point type

8.2.1.3.4.2.5 EnumType

EnumType is a subclass of DataType, and has one to many association enumLiterals to EnumLiteralDefinition.

Property Specification:

```
    EnumType              ->     enumLiterals : EnumLiteralDefinition+
    ;
```

Definition: Enumerated types

8.2.1.3.4.2.6 ConstructedType

ConstructedType is a subclass of DataType, has unary property BaseType to class TypeReference, and subclasses PointerType, ArrayType, ReferenceType, CollectionType and RangeType for depicting types of these respective kinds.

Property Specification:

```
ConstructedType      ->     baseType : TypeReference
;
```

Hierarchy Specification:

```
ConstructedType      =>     CollectionType
                     =>     PointerType
                     =>     ReferenceType
                     =>     RangeType
                     =>     ArrayType ;
```

Definition: Types constructed from a specified base type

Collection Type

Definition: Types characterized as collections (lists, sets, bags, ...)

PointerType

The class PointerType is a subclass of ConstructedType and has an optional semantic attribute size of type Integer.

Property Specification:

```
PointerType          ->     [ < size : Integer > ]
```

Definition: Types whose values are pointers. Optionally, the size of the pointer can be specified.

Reference Type

Definition: Types whose values are references

Range Type

Definition: Types whose values are ranges

ArrayType

Definition: Array types

8.2.1.3.4.2.7 AggregateType

The class AggregateType is a subclass of DataType and has one or more association members to class MemberObject and the optional unary semantic property opensScope to the semantic class AggregateScope. The AggregateType has subclasses StructureType, UnionType, ClassType, AnnotationType.

Property Specification:

```
AggregateType        ->     members : MemberObject+
                            [ opensScope : AggregateScope ]
;
```

Hierarchy Specification:

```
AggregateType        =>     StructureType
                     =>     UnionType
                     =>     ClassType
                     =>     AnnotationType ;
```

Definition: Types composed of heterogenous subtypes

Structure Type

StructureType is a subclass of AggregateType and is used for denoting structured types.

Definition: Simple structure types (no inheritance or function members)

Union Type

UnionType is a subclass of AggregateType and is used for denoting aggregate types.

Definition: Union types (like structures but each data member occupies the same location)

Class Type

ClassType is a subclass of AggregateType, and has unary association derivesFrom to class DerivesFrom, and is used for denoting class types.

```
ClassType            ->     derivesFrom : DerivesFrom
;
```

Property Specification:

Hierarchy Specification:

Definition: Class types

Annotation Type

AnnotationType is a subclass of AggregateType, and used for denoting annotation types.

Definition: Denotations that complete or extend the definitions of other types

8.2.1.3.4.2.8 ExceptionType

ExceptionType is a subclass of DataType used for denoting exception types.

Definition: Types used in the context of exception generation/handling

8.2.1.3.4.2.9 FormalParameterType

FormalParameterType is a subclass of DataType has unary association type to interior class Type and has subclasses ByReferenceFormalParameterType and ByValueParameterType. These two subclasses have no immediate properties or associations and are used for depicting the kind of FormalParameterType.

Property Specification:

```
FormalParameterType
             ->     type : TypeReference
;
```

Hierarchy Specification:

```
FormalParameterType
             =>     ByValueFormalParameterType
             =>     ByReferenceFormalParameterType ;
```

Definition: Specifies the by-value/reference nature of formal parameters

```
ByValueFormalParameterType
```

Definition: Specifies that a formal parameter is to be passed by value

```
ByReferenceFormalParameterType
```

Definition: Specifies that a formal parameter is to be passed by reference

```
Named Type
```

NamedType is a subclass of DataType, and has zero to any number association body to class Type. [1]

Property Specification:

```
NamedType     ->     body : Type
```

Definition: Uses of named types

8.2.1.3.4.3 LabelType

LabelType is a subclass of Type, used for depicting that the type of an element is a label.Definition: The type of a label.

8.2.1.3.4.4 NamespaceType

NamespaceType is a subclass of Type used for depicting that the type of an element is a namespace.Definition: The type of a namespace.

8.2.1.3.4.5 TypeReference

TypeReference is a subclass of Type with subclasses UnnamedTypeReference and NamedTypeReference.

Hierarchy Specification:

```
TypeReference          =>     UnnamedTypeReference
                       =>     NamedTypeReference
;
```

Definition: References to types

8.2.1.3.4.5.1 UnnamedTypeReference

UnnamedTypeReference is a subclass of TypeReference with unary association type to Type.

Property Specification:

```
UnnamedTypeReference
                       ->     type : Type
```

Definition: References to types without the use of a name for the referenced type

8.2.1.3.4.5.2 NamedTypeReference

NamedTypeReference is a subclass of TypeReference with unary association type to Type and unary association typeName to Name.

1. the typeDef body

Property Specification:

```
        NamedTypeReference
                        ->      typeName : Name
                                type : TypeDefinition
```

Definition: References to types via the name of the referenced type

8.2.1.4 Expression

The class Expression has unary association expressionType to a TypeReference, and subclasses Literal, CastExpression, AggregateExpression, UnaryExpression, Binary Expression, ConditionalExpression, RangeExpression, FunctionCallExpression, NewExpression, NameReference, LabelAccess, ArrayAccess, AnnotationExpression, and CollectionExpression.

Property Specification:

```
        Expression          ->      [ expressionType : TypeReference ] ;
```

Hierarchy Specification:

```
        Expression          =>      Literal
                            =>      CastExpression
                    =>      AggregateExpression
                    =>      UnaryExpression
                    =>      BinaryExpression
                    =>      ConditionalExpression
                    =>      RangeExpression
                    =>      FunctionCallExpression
                    =>      NewExpression
                    =>      ! NameReference
                    =>      LabelAccess
                    =>      ArrayAccess
                    =>      AnnotationExpression
                    =>      CollectionExpression
        ;
```

Definition: All expressions

8.2.1.4.1 Literal

The inner class Literal is a subclass of Expression, and has unary association value to String and subclasses IntegerLiteral, StringLiteral, CharLiteral, RealLiteral, BooleanLiteral, BitLiteral, and EnumLiteral.

Property Specification:

```
        Literal     ->      < value : String >
        ;
```

Hierarchy Specification:

```
        Literal     =>      IntegerLiteral
                    =>      StringLiteral
                    =>      CharLiteral
                    =>      RealLiteral
                    =>      BooleanLiteral
                    =>     BitLiteral
                    =>     EnumLiteral
        ;
```

Definition: Literal expressions

8.2.1.4.1.1 IntegerLiteral

The class IntegerLiteral is a subclass of Literal.

Definition: Integer literals

8.2.1.4.1.2 StringLiteral

The class IntegerLiteral is a subclass of Literal.

Definition: String literals

8.2.1.4.1.3 CharLiteral

The class IntegerLiteral is a subclass of Literal.

Definition: Character literals

8.2.1.4.1.4 RealLiteral

The class IntegerLiteral is a subclass of Literal.

Definition: Floating-point Literals

8.2.1.4.1.5 BooleanLiteral

The class IntegerLiteral is a subclass of Literal.

Definition: Boolean literals

8.2.1.4.1.6 BitLiteral

The class IntegerLiteral is a subclass of Literal.

Definition: Binary literals

8.2.1.4.1.7 EnumLiteral

The class EnumLiteral is a subclass of Literal.

Definition: Enumeration literals

8.2.1.4.2 CastExpression

The class CastExpression is a subclass of Expression, and has unary association castType to TypeReference, and unary association expression to the class Expression.

Property Specification:

```
        CastExpression      ->     castType : TypeReference
                                   expression : Expression
        ;
```

Definition: Expressions that are cast to a specified type

8.2.1.4.3 AggregateExpression

The AggregateExpression is a subclass of Expression, has no associations, properties or subclasses.

Definition: Expressions consisting of a list of subexpressions

8.2.1.4.4 UnaryExpression

The interior class UnaryExpression is a subclass of Expression, and has unary association operand to the class Expression and unary association operator to class UnaryOperatory.

Property Specification:

```
UnaryExpression      ->     operator : UnaryOperator
                            operand : Expression
;
```

Definition: Expressions involving unary operators

8.2.1.4.5 BinaryExpression

The interior class BinaryExpression is a subclass of Expression, and has unary association leftOperand and unary association rightOperand to the class Expression and unary association operator to the terminal primitive class BinaryOperatory.

Property Specification:

```
BinaryExpression     ->operator : BinaryOperator
                     leftOperand : Expression rightOperand : Expression
;
```

Definition: Expressions involving binary operators

8.2.1.4.6 Conditional Expression

The class ConditionalExpression is a subclass of Expression, and has unary association condition, unary onFalseOperand and unary association onTrueOperand to the class Expression.

Property Specification:

```
ConditionalExpression
              ->     condition : Expression
                     onTrueOperand : Expression
                     onFalseOperand : Expression
;
```

Definition: Ternary conditional expressions

Semantics: ConditionalExpression has short-circuit semantics. This implies that the onTrueOperand is evaluated only if the condition is TRUE, and onFalseOperand is evaluated only if the condition is FALSE.

8.2.1.4.7RangeExpression

The interior class RangeExpression, is a subclass of Expression, and has unary association fromExpression and the unary association toExpression to the interior class Expression.

Property Specification:

```
RangeExpression      ->     fromExpression : Expression
                            toExpression : Expression
;
```

Definition: Expressions consisting of a range of values

8.2.1.4.8 FunctionCallExpression

The interior class FunctionCallExpression is a subclass of Expression, and has any number of associations of actualParams to the class ActualParameter, and unary association calledFunction to the class Expression.

Property Specification:

```
FunctionCallExpression
                 ->     calledFunction : Expression
                        actualParams : ActualParameter*
;
```

Definition: Function calls

`NewExpression`

The class NewExpression has unary association newType to the class TypeReference, and zero to any number association actualParams to ActualParameter.

Property Specification:

```
NewExpression        ->     newType : TypeReference
                            actualParams : ActualParameter *
;
```

Definition: Instance creation expressions

8.2.1.4.9 NameReference

The class NameReference is a subclass of Expression, and unary semantic association RefersTo to the inner class DeclarationOrDefinition, unary association identifierName to class Name, and subclasses IdentifierReference, QualifiedIdentifierReference, and TypeQualifiedIdentifierReference.

Property Specification:

```
NameReference        ->     identifierName : Name
                            refersTo : DefinitionObject
;
```

Hierarchy Specification:

```
NameReference        =>     IdentifierReference
                     =>     ! QualifiedIdentifierReference
                     =>     TypeQualifiedIdentifierReference ;
```

Definition: References to named entities

`ArrayReference`

The class ArrayReference has unary association arrayName to inner class Expression and one or more subscripts to the inner class Expression.

Property Specification:

```
ArrayAccess          ->     arrayName : Expression
                            subscripts : Expression+
;
```

Definition: References to individual array elements

8.2.1.4.10 AnnotationExpression

AnnotationExpression is a subclass of Expression and has unary association annotationType to TypeReference and zero to any association memberValues to Expression.

Property Specification:

```
AnnotationExpression[2]
       ->     annotationType : TypeReference ?[3]
              MemberValues : Expression *
;
```

Definition: Expressions that supply annotations for other elements

8.2.1.4.11 CollectionExpression

CollectionExpression is a subclass of Expression and has one to many association expressionList to Expression.

Property Specification:

```
CollectionExpression
       ->     expressionList : Expression *
;
```

Definition: Expressions that are collections of other expressions

8.2.1.4.11.1 IdentifierReference

The interior class IdentifierReference is a subclass of NameReference, and has any number of association Qualifiers to interior class NamedType and unary semantic association RefersTo to the inner class DeclarationOrDefinition.

Definition: References to simply-named (unqualified) entities

8.2.1.4.11.2 QualifiedIdentifierReference

The class QualifiedIdentifierReference is a subclass of NameReference, and has unary association qualifiers to class Expression and unary association member to the class IdentifierReference and subclasses QualifiedOverData and QualifiedOverPtrs.

Property Specification:

```
QualifiedIdentifierReference
       ->     qualifiers : Expression
              member : IdentifierReference
;
```

Hierarchy Specification:

```
QualifiedIdentifierReference
       =>     QualifiedOverPtr
```

2. The AnnotationExpression is used for depicting EGL-style annotations
3. Annotation Type is optional. This is to to allow attribute-value pairs (i.e., Member Values) to allow default values for members.

```
                => QualifiedOverData
        ;
```

Definition: References to entities with qualified names

8.2.1.4.11.2.1 QualifiedOverPointer

QualifiedOverPtr is a subclass of the class QualifiedIdentifierReference.

Definition: References to entities with qualified names where the qualifying portion of the name is a pointer value

8.2.1.4.11.2.2 QualifiedOverData

QualifiedOverData is a subclass of the class QualifiedIdentifierReference.

Definition: References to entities with qualified names where the qualifying portion of the name is not a pointer value

8.2.1.4.12 LabelAccess

LabelAccess is a subclass of Expression with unary association labelDefinition to class LabelDefinition and unary association labelName to class Name.

Property Specification:

```
        LabelAccess       ->    labelName : Name
                                labelDefinition : LabelDefinition
        ;
```

Definition: Reference to a label

8.2.1.4.13 ArrayAccess

ArrayAccess is a subclass of Expression with unary association ArrayName to class Expression and one to many association Subscripts to Expression.

```
        ArrayAccess       ->    arrayName : Expression
                                subscripts : Expression+
        ;
```

Property Specification:

Hierarchy Specification:

Definition:

8.2.1.4.14 Statement

The inner class Statement is a subclass of GASTMSyntaticObject, and has interior subclasses ExpressionStatement, JumpStatement, BreakStatement, ContinueStatement, LabeledStatement, BlockStatement, EmptyStatement, IfStatement, SwitchStatement, ReturnStatement, TryStatement, ThrowStatement, DeleteStatement, TerminateStatement and inner class LoopStatement further classified into interior subclasses WhileStatement, DoWhileStatement, and ForStatement.

Hierarchy Specification:

```
        Statement   =>  ExpressionStatement
                    =>  JumpStatement
                    =>  BreakStatement
                    =>  ContinueStatement
                    =>  LabeledStatement
                    =>      BlockStatement
                    =>      EmptyStatement
                    =>      IfStatement
                    =>      SwitchStatement
                    =>      ReturnStatement
                    =>      LoopStatement
                    =>      TryStatement
                    =>      DeclarationOrDefinitionStatement
                    =>      ThrowStatement
                    =>      DeleteStatement
                    =>      TerminateStatement
        ;
```

Definition: All statements

8.2.1.4.14.1 ExpressionStatement

The class ExpressionStatement is a subclass of Statement, and has unary association expression with the interior inner class Expression.

Property Specification:

```
        ExpressionStatement        ->     expression : Expression
        ;
```

Definition: Statements comprised of just an expression

8.2.1.4.14.2 JumpStatement

The interior class JumpStatement is a subclass of Statement, and has unary association target with the interior inner class Expression.

Property Specification:

```
        JumpStatement        ->     target : Expression
        ;
```

Definition: Statements that branch to a label

8.2.1.4.14.3 BreakStatement

The interior class BreakStatement is a subclass of Statement, and is a subclass of Statement, and has unary association target with the interior class IdentifierReference.

Property Specification:

```
        BreakStatement       ->     target : LabelAccess?
        ;
```

Definition: Statements that exit a loop or a switch

8.2.1.4.14.4 ContinueStatement

The interior class ContinueStatement is a subclass of Statement, and has unary association target with the interior class IdentifierReference.

Property Specification:

```
        ContinueStatement    ->     target : LabelAccess?
        ;
```

Definition: Statements that branch to the top of a loop

8.2.1.4.14.5 LabeledStatement

The interior class LabeledStatement is a subclass of Statement, and has unary association label with the interior class LabelDefinition.

Property Specification:

```
        LabeledStatement     ->     label : LabelDefinition
                                    statement : Statement?
        ;
```

Definition: Statements that are associated with a label definition

8.2.1.4.14.6 BlockStatement

The interior class BlockStatement is a subclass of Statement, and has unary association subStatements with the interior inner class Statement and unary semantic association opensScope with the semantic class BlockScope.

Property Specification:

```
        BlockStatement       ->     subStatements : Statement*
                                    [ opensScope : BlockScope ]
        ;
```

Definition: Statements consisting of a series of substatements

8.2.1.4.14.7 EmptyStatement

The terminal class EmptyStatement is a subclass of Statement and has no associations, no properties, and no subclasses.

Property Specification:

```
        EmptyStatement       ->     ;
```

Definition: Statement that does nothing

8.2.1.4.14.8 IfStatement

The interior class IfStatement is a subclass of Statement, and has unary association condition to the interior inner class Expression and the unary association thenBody to the interior inner class Statement and the unary association elseBody to the interior inner class Statement.

Property Specification:

```
        IfStatement          ->     condition : Expression
                                    thenBody : Statement
                                    elseBody : Statement?
        ;
```

Definition: Statements that conditionally execute one of two substatements

8.2.1.4.14.9 SwitchStatement

The interior class SwitchStatement is a subclass of Statement, and has unary association cases to the interior class SwitchCase and the unary association switchExpression to the interior inner class Expression.

Property Specification:

```
SwitchStatement      ->     switchExpression : Expression
                            cases : SwitchCase
;
```

Definition: Statements that conditionally execute one of many substatements

8.2.1.4.14.10 ReturnStatement

The interior class ReturnStatement is a subclass of Statement, and has unary association returnValue with the interior inner class Expression.

Property Specification:

```
ReturnStatement      ->     returnValue : Expression?
;
```

Definition: Statements that cause return from a function, possibly with a return value

8.2.1.4.14.11 LoopStatement

The interior inner class LoopStatement is a subclass of Statement, and has unary association body to the interior inner class Statement and the unary association condition to the interior inner class Expression. The inner LoopStatement is further classified into interior subclasses WhileStatement, DoWhileStatement, and ForStatement.

Property Specification:

```
LoopStatement        ->     condition : Expression
                            body : Statement
;
```

Hierarchy Specification:

```
LoopStatement        =>     WhileStatement
                     =>     DoWhileStatement
                     =>     ! ForStatement
;
```

Definition: Statements with a substatement (body) that is potentially repeatedly executed

8.2.1.4.14.11.1 WhileStatement

The WhileStatement is a subclass of LoopStatement, and is the variation of the LoopStatement for which the Condition is tested before the Body is executed.

Definition: Loop statement whose body is repeatedly executed while a specified condition, tested before each execution, is true

8.2.1.4.14.11.2 DoWhileStatement

The DoWhileStatement is a subclass of LoopStatement, and is the variation of the LoopStatement for which the Condition is tested after the Body is executed.

Definition: Loop statement whose body is repeatedly executed while a specified condition, tested after each execution, is true

8.2.1.4.14.11.3 ForStatement

The ForStatement is a subclass of LoopStatement, and is the variation of the LoopStatement for which the Condition is tested before the Body is executed and any number of associations of initBody to interior inner class Expression and any number of associations of iterationBody to interior inner class Expression.

Property Specification:

```
        ForStatement         ->     initBody : Expression*
                                    iterationBody : Expression*
        ;
```

Hierarchy Specification:

```
        ForStatement         =>     ForCheckBeforeStatement
                             =>     ForCheckAfterStatement
        ;
```

Definition: Loop statement with initializing and incrementing parts

8.2.1.4.14.11.4 ForCheckBeforeStatement

The ForCheckBeforeStatement is a subclass of ForStatement, and is the variation of the LoopStatement for which the Condition is tested before the Body is executed

Definition: For statement with test before each iteration

8.2.1.4.14.11.5 ForCheckAfterStatement

The ForCheckAfterStatement is a subclass of ForStatement, and is the variation of the LoopStatement for which the Condition is tested after the Body is executed.

Definition: For statement with test after each iteration

8.2.1.4.14.12 TryStatement

The class TryStatement is a subclass of Statement, and has any number of association catchBlocks to to interior class CatchBlock, unary association of finalStatement to interior inner class Statement and unary association of guardedStatement to interior inner class Statement.

Property Specification:

```
        TryStatement         ->     guardedStatement : Statement
                                    catchBlocks : CatchBlock*
                                    finalStatement : Statement?
        ;
```

Definition: Exception-handling statements, consisting of a substatement that may throw exceptions and catch blocks to handle them

8.2.1.4.14.13 ThrowStatement

The class ThrowStatement is a subclass of Statement, and has unary association exception to interior inner class Expression.

Property Specification:

```
        ThrowStatement       ->     exception : Expression
```

Definition: Statements that cause an exception to be thrown

8.2.1.4.14.14 DeleteStatement

DeleteStatement is a subclass of Statement, and has unary association operand to class Expression, and is used for depicting deallocation of storage.

Property Specification:

```
        DeleteStatement      ->     operand: Expression
        ;
```

Definition: Statements that deallocate storage

```
TerminateStatement
```

TerminateStatement has no immediate properties, associations, or subclasses it is used for depicting the termination of execution.

Definition: Statement that terminates execution

8.2.1.5 MinorSyntaxObject

Hierarchy Specification:

```
MinorSyntaxObject
        =>     Dimension
        =>     CompilationUnit
        =>     Name
        =>     SwitchCase
        =>     CatchBlock
        =>     ! UnaryOperator
        =>     ! BinaryOperator
        =>     ! StorageSpecification
        =>     ! VirtualSpecification
        =>     AccessKind
        =>     ! ActualParameter
        =>     FunctionMemberAttributes
        =>     DerivesFrom
        =>     MemberObject
;
```

Definition: Various syntactic entities not otherwise categorized

8.2.1.5.1 Dimension

The interior class Dimension is a subclass of OtherSytnaxObject, and has unary associations highBound and lowBound to interior class Expression.

Property Specification:

```
Dimension           ->     lowBound : Expression?
                           highBound : Expression
;
```

Definition: Range of subscript values for one dimension of an array type

8.2.1.5.2 CompilationUnit

The interior class CompilationUnit is a subclass of OtherSytnaxObject, and has any number of associations Fragments to interior class ProgramCodeFragment, unary semantic property Language to the terminal primitive class String, and unary semantic property OpensScope to the semantic class ProgramScope.

Property Specification:

```
CompilationUnit     ->     < language : String >
                           fragments : DefinitionObject*
```

```
                                        [ opensScope : ProgramScope? ]
        ;
```

Definition: Unit of compilation; typically corresponding to a source file

8.2.1.5.3 Name

Name is a subclass of OtherSytnaxObject, and has Unary Association nameString to Primitive String.

Property Specification:

```
        Name          ->     < nameString : String >
        ;
```

Definition: Names that may appear in declarations and definitions

8.2.1.5.4 SwitchCase

The class SwitchCase is a subclass of OtherSytnaxObject, has a boolean attribute isEvaluateAllCases, has unary association body to interior inner class Statement, and subclasses CaseBlock and DefaultBlock.

Property Specification:

```
        SwitchCase    ->     < isEvaluateAllCases : Boolean >
                             body : Statement+
        ;
```

Hierarchy Specification:

```
        SwitchCase           =>     CaseBlock
                             =>     DefaultBlock
        ;
```

Definition: Parts of a switch statement that are conditionally executed

8.2.1.5.5 CaseBlock

The class CaseBlock is a subclass of OtherSytnaxObject, and has any number of associations of caseExpression to interior inner class Expression.

Property Specification:

```
        CaseBlock            ->     caseExpressions : Expression+
        ;
```

Definition: Switch cases that are executed when one of their values matches that of the enclosing switch statement

8.2.1.5.6 DefaultBlock

The DefaultBlock is a subclass of OtherSytnaxObject, and depict the fall through CaseBlock.

Definition: Switch cases that are executed when no other switch case in the enclosing switch statement is executed

8.2.1.5.7 CatchBlock

The class CatchBlock is a subclass of OtherSyntaxObject, and has interior subclasses TypesCatchBlock and VariableCatchBlock. The CatchBlock has unary association Body to Statement.

Property Specification:

```
        CatchBlock    ->     body : Statement
```

```
;
```

Hierarchy Specification:

```
CatchBlock    =>     TypesCatchBlock
              =>     VariableCatchBlock
;
```

Definition: Parts of a try statement that specify a statement to execute under specified exception conditions

8.2.1.5.7.1 TypesCatchBlock

The class TypesCatchBlock is a subclass of CatchBlock, and has any number association exceptions class Type.

Property Specification:

```
TypesCatchBlock      ->     exceptions : Type+
;
```

Definition:

8.2.1.5.7.2 VariablesCatchBlock

The class VariablesCatchBlock is a subclass of CatchBlockObject, and has a unary association exceptionVariable to the interior inner class DataDefinition.

Property Specification:

```
VariableCatchBlock
      ->     exceptionVariable : DataDefinition
;
```

Definition:

8.2.1.5.8 UnaryOperator

The inner class UnaryOperator is a subclass of OtherSyntaxObject, and has primitive terminal subclasses UnaryPlus, UnaryMinus, Not, BitNot, AddressOf, Deref, Increment, Decrement, PostIncrement, PostDecrement.

Hierarchy Specification:

```
UnaryOperator        =>     UnaryPlus
                     =>     UnaryMinus
                     =>     Not
                     =>     BitNot
                     =>     AddressOf[4]
                     =>     Deref[5]
                     =>     Increment
                     =>     Decrement
                     =>     PostIncrement
                     =>     PostDecrement
;
```

Definition: Operators taking a single operand

4. e.g., *& va rname*
5. e.g., **ptrto*

`UnaryPlus`

Definition: Unary plus operator

`UnaryMinus`

Definition: Negation operator

`Not`

Definition: Logical complement operator

`BitNot`

Definition: Bitwise complement operator

`AddressOf`

Definition: Operator which results in the address of its operand

`Deref`

Definition: Operator which results in the value of which its operand is the address

`Increment`

Definition: Operator which increments its operand and results in the incremented value

`Decrement`

Definition: Operator which decrements its operand and results in the decremented value

`PostIncrement`

Definition: Operator which results in the value of its operand before it is incremented

`PostDecrement`

Definition: Operator which results in the value of its operand before it is decremented

8.2.1.5.9 BinaryOperator

The inner class BinaryOperator is a subclass of OtherSyntaxObject, and has primitive terminal subclasses Add, Subtract, Multiply, Divide, Modulus, Exponent, And, Or , Equal, NotEqual, Greater, NotGreater, Less, NotLess , BitAnd , BitOr, BitXor, BitLeftShift, BitRightShift, Assign.

Hierarchy Specification:

```
BinaryOperator       =>    Add
                     =>    Subtract
                     =>    Multiply
                     =>    Divide
                     =>    Modulus
                     =>    Exponent
                     =>    And
                     =>    Or
                     =>    Equal
                     =>    NotEqual
                     =>    Greater
```

```
                                    =>     NotGreater
                                    =>     Less
                                    =>     NotLess
                                    =>     BitAnd
                                    =>     BitOr
                                    =>     BitXor
                                    =>     BitLeftShift
                                    =>     BitRightShift
                                    =>     Assign
                                    =>     OperatorAssign
        ;
```

Definition: Operators taking two operands

Semantics: Operators And, Or, BitAnd, BitOr, BitXor have short-circuit semantics. This implies that the second operand is evaluated only if the first operand does not suffice to determine the value of the expression. For example:

- a AND b => if (a) then if (b) then TRUE else FALSE else FALSE
- a OR b => if (a) then TRUE else if (b) then TRUEelse FALSE

`Add`

Definition: Addition operator

`Subtract`

Definition: Subtraction operator

`Multiply`

Definition: Multiplication operator

`Divide`

Definition: Division operator

`Modulus`

Definition: Modulo operator

Semantics: Default behavior of Modulus operator: Integer.mod(i: Integer): Integer post: result = self - (self.div(i) * i)

`Exponent`

Definition: Exponentiation operator

`And`

Definition: Logical conjunction operator

`Or`

Definition: Logical disjunction operator

`Equal`

Definition: Equality operator

`NotEqual`

Definition: Inequality operator

`Greater`

Definition: Relational operator in which the result is true iff the left operand is greater than the right operand

`NotGreater`

Definition: Relational operator in which the result is true iff the left operand is not greater than the right operand

`Less`

Definition: Relational operator in which the result is true iff the left operand is less than the right operand

`NotLess`

Definition: Relational operator in which the result is true iff the left operand is not less than the right operand

`BitAnd`

Definition: Bitwise conjunction operator

`BitOr`

Definition: Bitwise disjunction operator

`BitXor`

Definition: Bitwise exclusive-or operator

`BitLeftShift`

Definition: Bitwise left-shift operator

`BitRightShift`

Definition: Bitwise right-shift operator

Semantics: Operator BitRightShift applied to unsigned integer implies that a shift by n bits of a two's complement value is equivalent to dividing by 2^n .

`Assign`

Definition: Assignment operator

8.2.1.5.9.1 OperatorAssign

The interior class OperatorAssign is a subclass of BinaryOperator, and has unary association operator to inner class BinaryOperator.

Property Specification:

```
    OperatorAssign       ->     operator : BinaryOperator[6] ;
```

Definition: Assignment operators compounded with a binary operator

6. e.g., +=, *= , *etc.*

8.2.1.5.9.2 StorageSpecification

StorageSpecification is a subclass of OtherSytnaxObject, and has subclasses External, FunctionPersistent, FileLocal, PerClassMember, NoDef.

Hierarchy Specification:

```
StorageSpecification
       =>     External
       =>     FunctionPersistent[7]
       =>     FileLocal
       =>     PerClassMember[8]
       =>     Nodef[9];
```

Definition: a property of data that depicts how it is is allocated

8.2.1.5.9.2.1 External

External is a subclass of Storage Specification and depicts storage that is external.

Definition: depicts storage that is external

8.2.1.5.9.2.2 FunctionPersistent

FunctionPersistent is a subclass of Storage Specification and depicts storage that is allocated and persists within a function.

Definition: depicts storage that is allocated and persists within a function.

8.2.1.5.9.2.3 FileLocal

FileLocal is a subclass of Storage Specification and depicts storage that is allocated and local within a file.

Definition: depicts storage that is allocated and local within a file.

8.2.1.5.9.2.4 PerClassMember

PerClassMember is a subclass of Storage Specification and depicts storage that is allocated for each class.

Definition: depicts storage that is allocated for each class

8.2.1.5.9.2.5 NoDef

NoDef is a subclass of Storage Specification and depicts storage for which the allocator is not defined.

Definition: depicts storage for which the allocator is not defined.

8.2.1.5.9.3 VirtualSpecification

VirtualSpecification is subclass of MinorSyntaxObject that is used for specifying if a class member is virtual.

Hierarchy Specification:

```
VirtualSpecification
       =>     Virtual
```

7. e.g., C's 'static':
8. e.g., C++ static member
9. e.g., default attributes

```
        ;
```

Definition: Specifications of the virtual characteristics of a function member

8.2.1.5.9.3.1 Virtual

Virtual is subclass of VirtualSpecification used for specifying that class member is virtual.

Definition: Specifies that the associated function member is virtual

8.2.1.5.9.4 AccessKind

AccessKind is subclass of MinorSyntaxObject used for specifying that class member is Public, Protected of Private and has subclasses Public, Protected, and Private for those denotations.

Hierarchy Specification:

```
        AccessKind          =>  Public
                            => Protected => Private
        ;
```

Definition: Specifications of the kind of access provided by a member or base class

8.2.1.5.9.4.1 Public

Public is subclass of AccessKind used for specifying that class member is Public.

Definition: Specifies that the associated member or base class provides public access

8.2.1.5.9.4.2 Protected

Protected is subclass of AccessKind used for specifying that class member is Protected.

Definition: Specifies that the associated member or base class provides protected access

8.2.1.5.9.4.3 Private

A private is subclass of AccessKind used for specifying that class member is Private.

Definition: Specifies that the associated member or base class provides private access

8.2.1.5.9.5 ActualParameter

ActualParameter is subclass of MinorSyntaxObject used for denoting actual parameters, and has two subclasses ActualParameterExpression and MissingActualParameter.

Property Specification:

```
        ActualParameter     =>     ActualParameterExpression
                            =>     MissingActualParameter
        ;
```

Definition: Actual parameters

8.2.1.5.9.5.1 ActualParameterExpression

ActualParameterExpression is subclass of ActualParameter and has two subclasses ByValueActualParameterExpression and ByReferenceActualParameterExpression that are used for denoting parameters passed by value and reference and unary association value to Expression.

Property Specification:

```
        ActualParameterExpression
                -> value : Expression
        ;
```

Hierarchy Specification:

```
        ActualParameterExpression
                => ByValueActualParameterExpression
                => ByReferenceActualParameterExpression
        ;
```

Definition: Actual parameters involving expressions (as opposed to missing)

`ByValueActualParameterExpression`

The ByValueActualParameterExpression is a subclass of ActualParameterExpression used for denoting parameters passed by value.

Definition: Actual Parameters passed by value

`ByReferenceActualParameterExpression`

The ByReferenceActualParameterExpression is a subclass of ActualParameterExpression used for denoting parameters passed by reference.

Definition: Actual Parameters passed by reference

`MissingActualParameter`

The MissingActualParameter is a subclass of ActualParameter used for denoting that the actual parameters are not present.

Definition: Missing actual parameter

8.2.1.5.9.6 FunctionMemberAttributes

The FunctionMemberAttributes is a subclass of MinorSyntaxObject used for attributing members of classes. FunctionMemberAttributes has Boolean properties isFriend, isInLine and isThisConst to depict the corresponding member properties and the association virtualSpecifier to the class Virtual Specification to depict whether the member is virtual.

Property Specification:

```
        FunctionMemberAttributes
                ->      < isFriend : Boolean >
                        < isInline : Boolean >
                        < isThisConst : Boolean > virtualSpecifier : VirtualSpecification[10]
```

Definition: Specifies various properties of function members

10. e.g., !! default

8.2.1.5.9.7 DerivesFrom

DerivesFrom is a subclass of MinorSyntaxObject used for depicting the derived properties of a class. Derives from has optional association virtualSpecifier to class VirtualSpecification, association AccessKind to class accessKind, and association className to class NamedType.

Property Specification:

```
DerivesFrom   ->     virtualSpecifier : VirtualSpecification ?
                     accessKind : AccessKind
                     className : NamedTypeReference
;
```

Definition: Specifies relationships between class types and the types from which they are derived

8.2.1.5.9.8 MemberObject

MemberObject is a subclass of MinorSyntaxObject used for depicting the members of an aggregate type. MemberObject has optional Integer property offset., and an assocation member to DefinitionObject.

Property Specification:

```
MemberObject  ->     [ < offset : Integer > ]
                     member : DefinitionObject
;
```

Definition: Specifies members of an aggregate type. Optionally, allows specification of offset for each member.

8.3 Specialized Abstract Syntax Tree Specifications

The SASTM extensions are outside the scope of this document. The process for adding SASTM extensions to the GASTM is established within this document. Future SASTM will be sought through the OMG RFP process, proposed by submission teams, reviewed and recommended by the ADMTF and adopted OMG adoption process.

8.3.1 SASTM Extension for RDBMS Languages

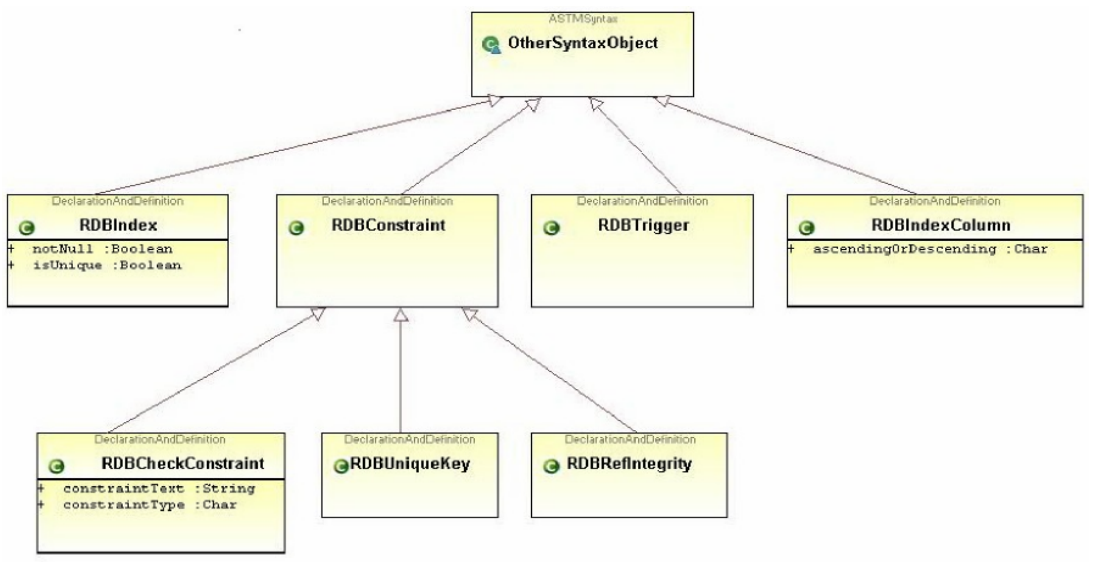
An example proposed SASTM extension for Relational Data Base Management System Languages is included as Annex A of this document.

Annex A - RDBMS Extensions

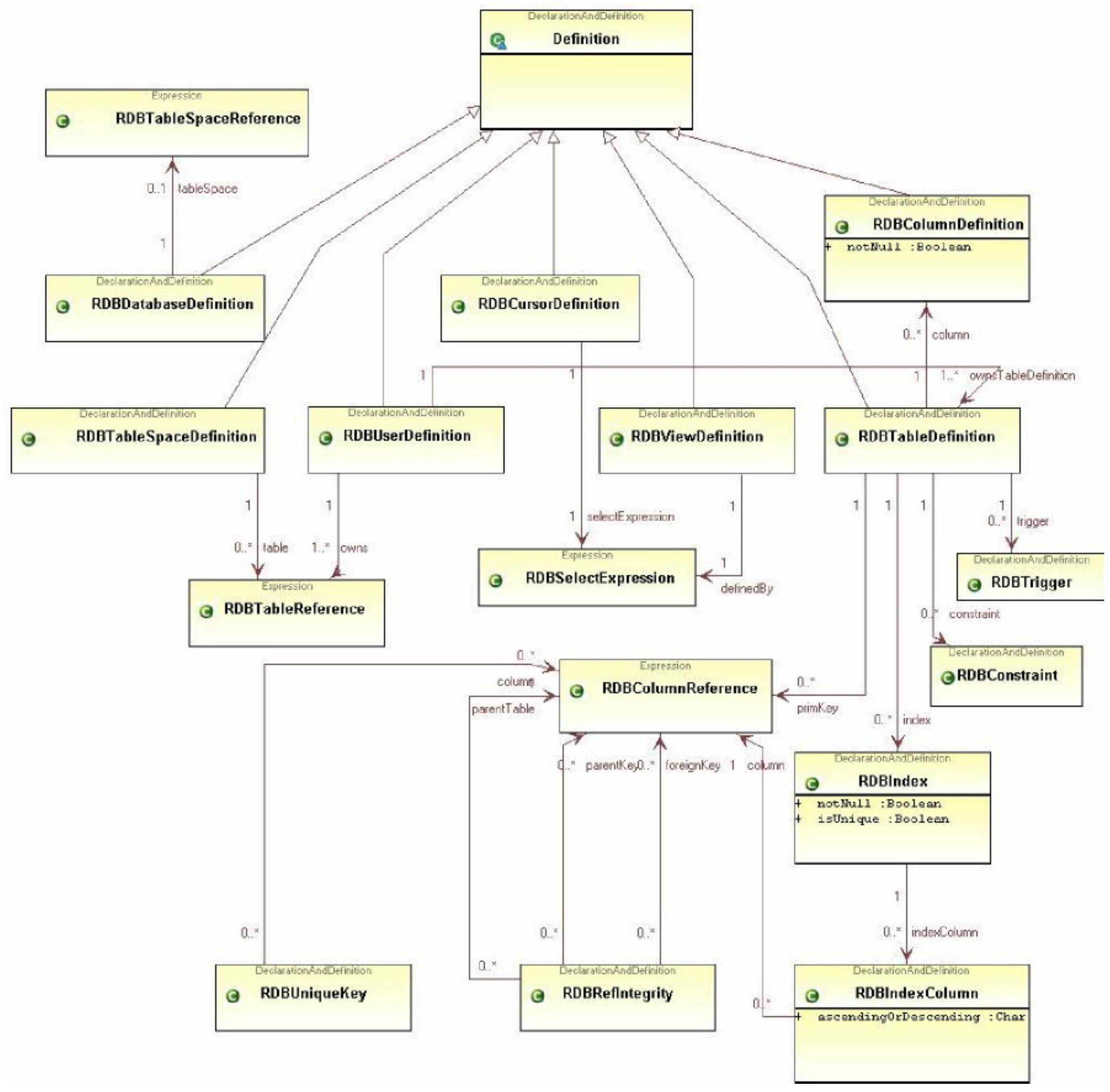
This annex defines extensions for modeling relational database constructs, including Data Definitions and Embedded SQLconstructs, which are common to many programming languages. The vendor specific clauses like Triggers, Constraints, and others are not included in the GASTM, and can be made part of the SASTM for the specific vendors.

A.1 Class Diagrams

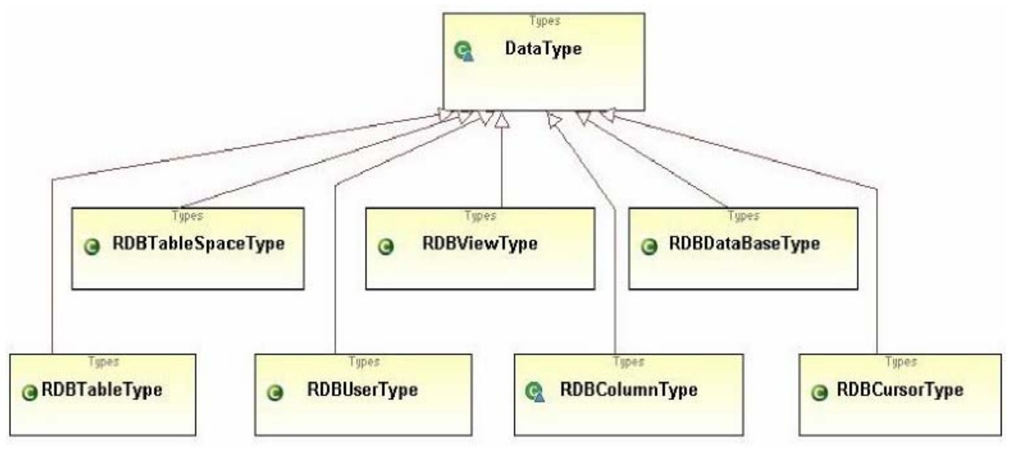
A.1.1 RDBMS Other Syntax Object Extensions Class Diagrams

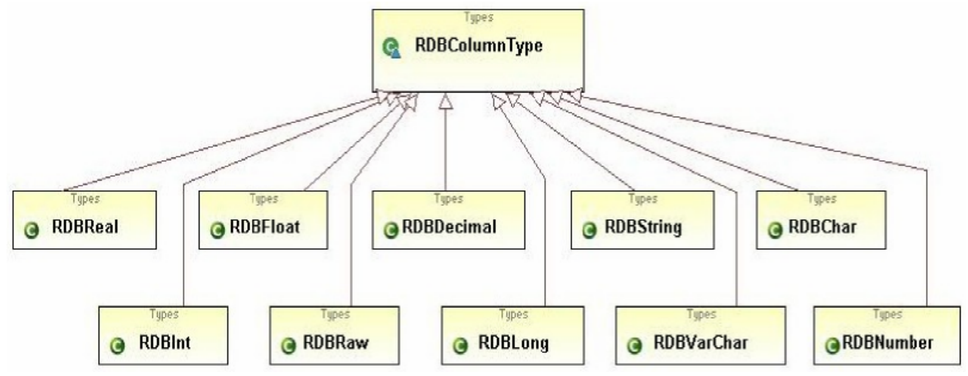


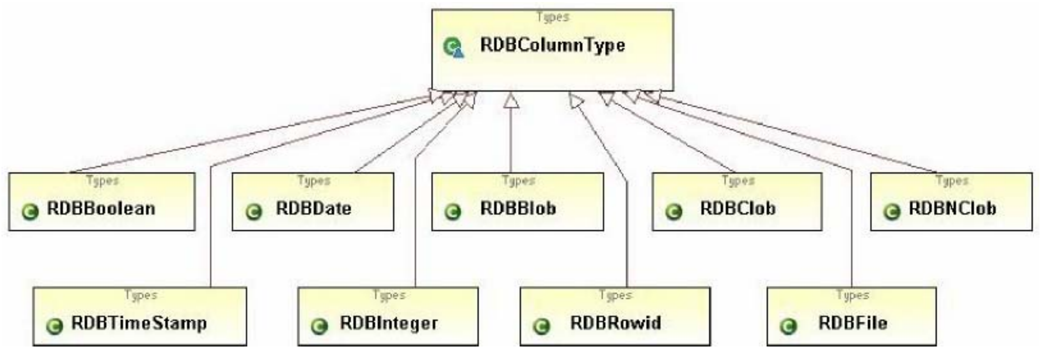
A.1.2 RDBMS Definitions Class Diagrams



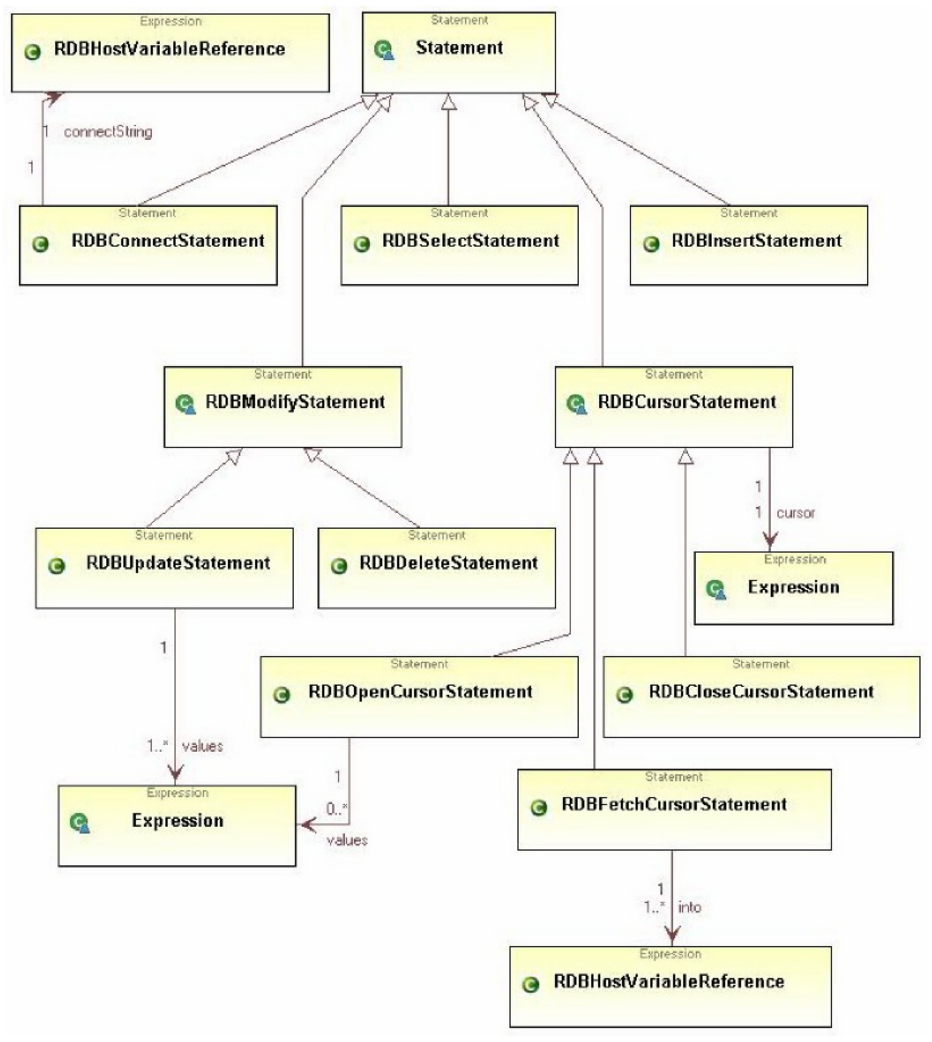
A.1.3 RDBMS Types Class Diagrams



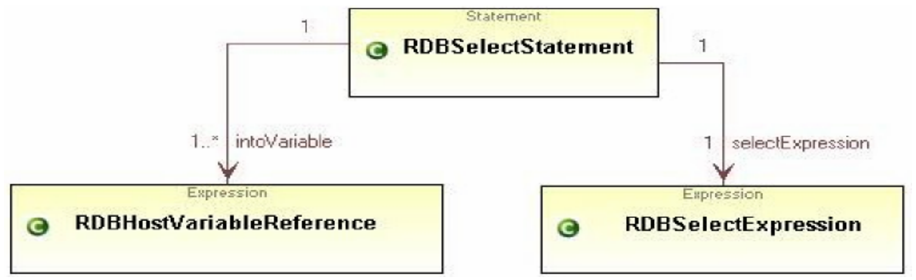


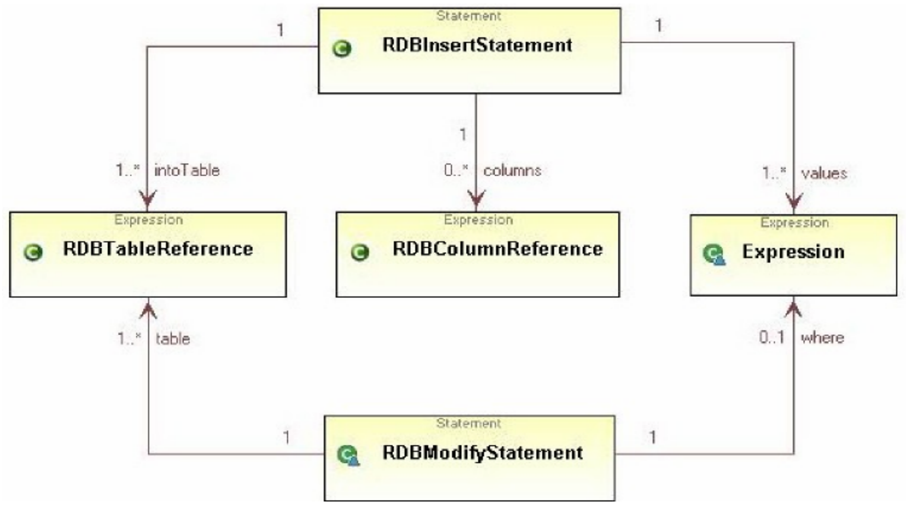


A.1.4 RDBMS Statements Class Diagrams

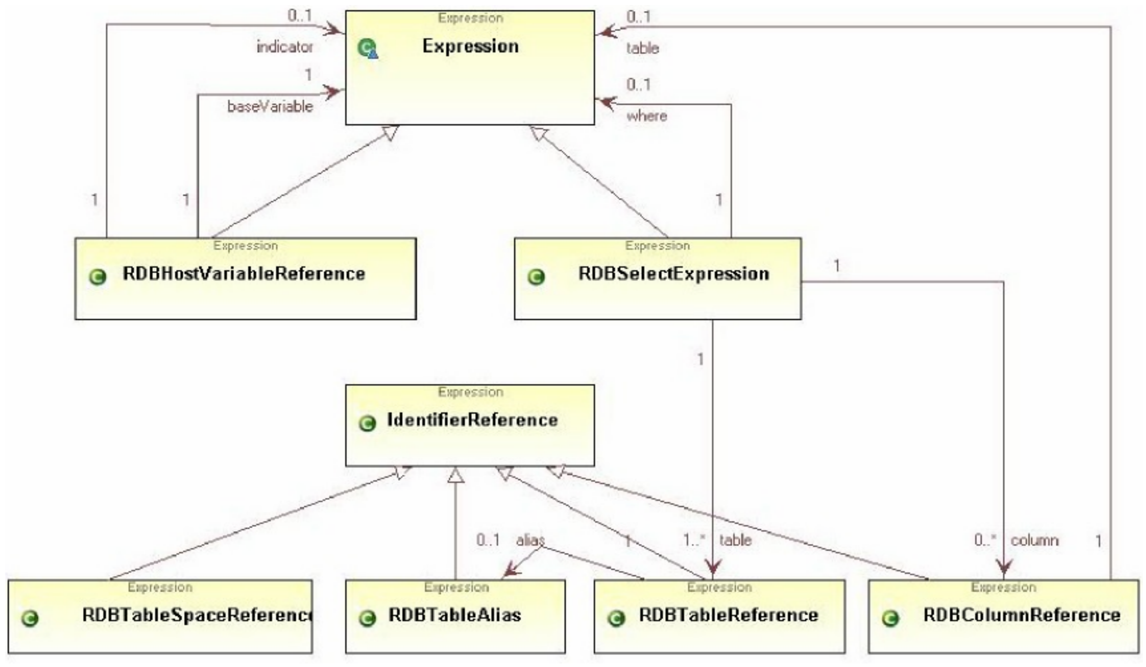


A.1.5 RDBMS Select Statements Class Diagrams





A.1.6 RDBMS Expression Class Diagrams



A.2 RDBMS Extended BNF

A.2.1 Extensions to Core

```
MinorSyntaxObject   =>     RDBIndex
                    =>     RDBIndexColumn
                    =>     RDBTrigger
                    =>     RDBConstraint
;
```

A.2.2 Extensions to Declarations and Definitions

```
Definition          =>     RDBDatabaseDefinition
                    =>     RDBUserDefinition
                    =>     RDBTableSpaceDefinition
                    =>     RDBTableDefinition
                    =>     RDBColumnDefinition
                    =>     RDBViewDefinition
                    =>     RDBCursorDefinition
;
```

```
                                  !! Default TableSpace is always present
RDBDatabaseDefinition      ->     TableSpace : RDBTableSpaceReference +
      ;
RDBUserDefinition          ->     Owns : RDBTableReference + ;
RDBTableSpaceDefinition    ->     Table : RDBTableReference *;
RDBTableDefinition         ->     PrimKey : RDBColumnReference *
                                  Column : RDBColumnDefinition *
                                  Constraint : RDBConstraint *
                                  Index : RDBIndex *
                                  Trigger : RDBTrigger *
      ;
RDBColumnDefinition        ->     < NotNull : Boolean > ;

RDBViewDefinition          ->     DefinedBy : RDBSelectExpression
      ;
RDBCursorDefinition        ->     SelectExpression : RDBSelectExpression ;

RDBIndex                   ->     IndexColumn : RDBIndexColumn*
                                  < NotNull : Boolean>
                                  < IsUnique : Boolean >
      ;

RDBIndexColumn             ->     Column : RDBColumnReference
                                  !!    cardinality M:1
                                  < AscendingOrDescending : Char >
                                  !! Ascending or Descending
      ;

RDBTrigger                 ->     ; ! ! Details of trigger are vendorspecific
RDBConstraint              =>     RDBCheckConstraint
                           =>     RDBRefIntegrity
                           =>     RDBUniqueKey
      ;
RDBCheckConstraint         ->     < RDBConstraintText : String >
                                  < RDBConstraintType : Char >
      ;
RDBRefIntegrity            ->     ForeignKey : RDBColumnReference *
                                  ! !   cardinality M:M
                                  ParentKey : RDBColumnReference *
                                  !!    cardinality M:M
                                  ParentTable : RDBColumnReference
                                  !!    cardinality M:1
      ;
RDBUniqueKey               ->     Column : RDBColumnReference*;
                                  !!    cardinality M:M
```

A.2.3 Extensions to Data Types

```
DataType              =>    RDBDataBaseType
                      =>    RDBUserType
                      =>    RDBTableSpaceType
                      =>    RDBTableType
                      =>    RDBViewType
                      =>    ! RDBColumnType
                      =>    RDBCursorType;
RDBColumnType         =>    RDBInteger,RDBInt,
                            RDBReal,
                            RDBFloat,
                            RDBDecimal,
                            RDBNumber,
                            RDBLong,
                            RDBChar,
                            RDBVarchar,
                            RDBString,
                            RDBRaw,
                            RDBDate,
                            RDBTime stamp,
                            RDBRowid,
                            RDBBoolean,
                            RDBBlob,
                            RDBClob,
                            RDBNClob,
                            RDBBFile
```

A.2.4 Extensions to Statements

```
Statement             =>    RDBConnectStatement RDBSelectStatement
                      =>    RDBInsertStatement
                      =>    ! RDBModifyStatement
                      =>    ! RDBCursorStatement
                      ;
RDBConnectStatement         ->    ConnectString : RDBHostVariableReference;
RDBSelectStatement          ->    SelectExpression : RDBSelectExpression
                                  IntoVariable : RDBHostVariableReference+
                      ;
RDBInsertStatement          ->    IntoTable : RDBTableReference +
                                  Columns : RDBColumnReference *
                                  Values : Expression +
                            ;
RDBModifyStatement          =>    RDBUpdateStatement
                            =>    RDBDeleteStatement
                      ;
```

```
RDBModifyStatement          ->    Table : RDBTableReference +
                                  Where : Expression?
                   ;
RDBUpdateStatement          ->    Values : Expression + ;
RDBCursorStatement          =>    RDBOpenCursorStatement
                            =>    RDBFetchCursorStatement
                            =>    RDBCloseCursorStatement
                   ;
RDBCursorStatement          ->    Cursor : Expression ;
RDBOpenCursorStatement      ->    Values : Expression *;
RDBFetchCursorStatement     ->    Into : HostVariableReference + ;
```

A.2.5 Extensions to Expressions

```
Expression                  =>    RDBHostVariableExpres sion
                            =>    RDBSelectExpression
                            ;
RDBHostVariableReference
                            ->    BaseVariable : Expression
                                  Indicator : Expression ?
                            ;
RDBSelectExpression         ->    Table : RDBTableReference +
                                  Column : RDBColumnReference *
                                  Where : Expression?
                                  ! !   Clauses like ConnectBy,StartWith,
                                  GroupBy,!! Having, OrderBy, ForUpdateOf are
                                  vendor!!specific;

IdentifierReference         =>    RDBTableReference
                            =>    RDBTableAlias
                            =>    RDBColumnReference
                            ;
RDBTableReference           ->    Alias  : RDBTableAlias ?;
RDBTableAlias               ->    ;!! Leaf level class to represent alias of a
                                  ! !   table
RDBColumnReference          ->    Table : Expression?
                                  !! RDBTableReference or RDBTableAlias
```

Annex B - Glossary

The external industry and published source definitions that are the basis for the GASTM core definitions are to be found in the table below. The industry sources for the ASTM core concept definitions are listed in the ASTM Core Terminology Bibliograpy (Annex C).

NOTE: NSD refers to: No standard definition found

B.1 Generic Abstract Syntax Tree Glossary

Term	**Definition**
4GL Statement	An informal (or popular) form of reference to a statement within a programming paradigm. See Programming Paradigm below.
Abstract Syntax Tree	In computer science, an abstract syntax tree (AST) is a *finite*, labeled, *directed tree*, where the *internal nodes* are labeled by operators, and the *leaf nodes* represent the operands of the node operators. Thus, the leaves have nullary operators, i.e., variables or constants. In computing, it is used in a *parser* as an intermediate between a *parse tree* and a *data structure*, the latter which is often used as a *compiler* or *interpreter'*s internal *representation* of a *computer program* while it is being *optimized* and from which code generation is performed. The range of all possible such structures is described by the *abstract syntax*. An AST differs from a parse tree by omitting nodes and edges for syntax rules that do not affect the semantics of the program. The classic example of such an omission is grouping parentheses, since in an AST the grouping of operands is explicit in the tree structure.
Abstract Semantic Graph	In *computer science*, an abstract semantic graph (ASG) is a *data structure* used in representing or deriving the *semantics* of an expression a formal language (for example, a *programming language*). An abstract semantic graph is a higher level abstraction than an abstract syntax tree (or AST), which is used to express the syntactic structure of an expression or program. An abstract semantic graph is typically constructed from an abstract syntax tree by a process of enrichment and abstraction. The enrichment can, for example, be the addition of back-pointers, *edges* from an identifier node (where a variable is being used) to a node representing the *declaration* of that variable. The abstraction can entail the removal of details that are relevant only in parsing, not for semantics.
Actual Parameter	An actual parameter is the particular entity associated with the corresponding formal parameter in a subprogram call, entry call, or generic instantiation.
Address	A location in physical memory.
AddressOf	A reference to an address.
Array	1. A composite type that consists of homogeneous components, indexed by discrete value. 2. An array is a collection of elements of some fixed type, laid out in a k-dimensional rectangular structure.

Term	**Definition**
Array Reference	A reference to a composite type that consists of homogeneous components, indexed by discrete value.
Base Type	The C++ term for any type from which another type is derived via subtyping.
Binary relationship	Any relationship between two things.
Binary Expression	An expression that computes a relationship between two things.
Binding	The act of associating attributes to a name is often referred to a binding. Most languages allow only static binding (compile time). Some languages, such as SNOBOL, allow dynamic binding, or binding of attributes while the program is running (at run time).
Block	A block introduces a (possibly named) sequence of statements, optionally preceded by a declarative part.
Block Statement	A statement that introduces a (possibly named) sequence of statements, optionally preceded by a declarative part.
Branch Statement	A statement that defines a program point at which the control flow has two or more alternatives.
Break Statement	The break statement causes program control to proceed with the first statement after the switch structure.
Call	1. to send a message, 2. to evaluate a post fix expression identifying an object and associated function followed by parentheses containing a possibly empty, comma-separated list of expressions, which constitute the actual arguments to the function (C++), 3. to invoke the method function of a method object, 4. to apply a certain feature to a certain object, possibly with arguments. A call has three components: • the target of the call, an expression whose value is attached to the object • the feature of the call, which must be a feature of the base class of the object's type • an actual argument list.
Call-by-address	See Call-by-reference
Call-by-reference	1. When an argument is passed by reference, the caller gives the called method the ability to access the caller's data directly and to modify that data if the called method so chooses. Pass-by-reference improves performance because it eliminates the overhead of copying large amounts of data, 2. any message passing in which a reference to (e.g., the address of) each argument is passed rather than its value.
Call-by-value	1. When an argument is passed by value, a copy of the argument's value is made and passed to the called method, 2. any message passing in which a copy of the value of each argument is passed rather than a reference (e.g., its address).
Case Statement	One of a series of case labels in a switch statement concluded by an optional default case.
Cast Expression	A special operation that handles type conversion.
Class	A set of objects in the object-oriented model that contain the same types of values and the same methods; also, a type definition for objects.
Collection	The entire set of allocated objects of an access type.

Term	Definition
Collection Expression	A comma-separated list of expressions with left-to-right associativity.
Collection Type	1. Any type of collection objects. 2. any instantiation of a collection type generator.
Compilation Unit	A program unit presented for compilation as an independent text. It is preceded by a context specification, naming the other compilation units on which it depends. A compilation unit may be the specification or body of a subprogram or package, including generic units or subunits.
Continue Statement	The continue statement, when executed in a ‘while,’ ‘for,’ or ‘do/while’ structure skips the remaining statements in the loop body and proceeds with the next iteration of the loop.
Condition	1. (a) Any boolean (or enumeration-valued) expression involving the values of one or more properties. (b) Any Boolean function of object values that is valid over an interval of time. 2. Any logical statement about the current state of an object, the current state of the system environment, the existence or absence of an object, or the existence or absence of relationships among objects.
Conditional Expression	A conditional expression is an operation of one or more operands that evaluates to true or false.
Definition	The specification of the implementation of something.
Default Statement	In a switch/case statement, the default case is the one that will be chosen whenever the value does not match any of the other cases. See Switch.
Declaration	1. Any line of code that introduces one or more names into a program and specifies the types of the names. 2. Any language construct that associates a name with a view of an entity. 3. Associates and identifies with a declared entity, including objects, types, subprograms, tasks, renamed entities, numbers, subtypes, packages, exceptions, and generic units.
Dimension	The number of independent interpreted input variables over which a domain is defined.
Enumeration Type	A discrete type whose values are given explicitly in the type declaration. These values may be either identifiers or character literals, which are considered enumeration literals.
Enumeration literal	Instances of an enumeration type that instances are named literal objects.
Enumeration (type)	Any developer-defined type whose instances are named literal objects.
Enumeration Reference	A reference to an enumeration type or an enumeration literal.
Entry	Used for communication between tasks. Externally, an entry is called just as a subprogram is called. Its internal behavior is determined by one or more accept statements that specify the actions to be performed when the entry is called.
Exception	An event that causes suspension of normal program execution. Bringing an exception to attention is called raising an exception. An exception handler is a piece of program text specifying a response to the exception. Execution of such a program text is called handling the exception.
Exception type	NSD
Expression	Part of a program that computes a value.
Expression Statement	A statement that computes a value.

Term	**Definition**
Extensible	Adj. Describing software that is easy to modify to implement new or changed requirements, especially without modification to existing modules. Examples: Classes may easily be extended via subclasses.
Extensible Expression	A class denoting an expression that may be extended via subclassing to denote specialized forms of expressions.
Extensible Statement	A class denoting a statement that may be extended via subclassing to denote specialized forms of statements.
Formal Declaration	See Formal parameter or arguments
Formal parameter or arguments	A parameter name that appears in the declaration or header of a procedure or function.
Global	Data available to more than one unit.
Global Declaration	A declaration available to more than one unit. See Global.
Guard (condition)	Any condition that must be true (or have the proper enumeration value) for a trigger to cause the associated transition to fire.
Guarded Transition	Any statement transition that occurs only if the trigger fires while its associated guard condition evaluates to true (or to the enumeration value associated with the transition).
Guarded Statement	A statement in a language that introduces a guard condition. For example, the ‘Try’ statement in java, C++, and C#. The On statement in Visual Basic.
Identifier	One of the basic lexical elements of the language. An identifier is used as the name of an entity or as a reserved word.
Identifier Reference	A reference to an identifier.
Include Statement	A statement in a language (i.e., such as C or C++), which lexically introduces one program unit into the body of another program unit.
Inherits	To obtain the declarations and definition of features via inheritance. Example: Child classes inherits features from their parent classes.
Iteration	Iteration is a repetition structure (such as ‘for,’ ‘while,’ or ‘do/while’) that terminates when the loop-continuation condition fails. Iteration modifies a counter until the counter assumes a value that makes the loop-continuation condition fail. An infinite loop occurs with iteration if the loop continuation step never becomes false.
Jump Statement	1. Jumps are changes in the flow of control typically invoked by ‘goto’ commands of the form ‘goto1.’ 2. When a jump occurs the normal continuation is ignored and control passes to a continuation corresponding to the rest of the program following the label jumped to.
Label	A unique identifier assigned to a statement or program location.
Label Statement	A statement denoting a location to which flow of control can pass to.
Label Reference	A reference to a label.
Label Type	NSD
Literal	Denotes an explicit value of a given type, for example, a number, an enumeration value, a character, or a string.

Term	**Definition**
Local	Available only to the procedure within which it is declared.
Local Declaration	A declaration available only to the procedure within which it is declared.
Loop Statement	See Iteration. Definite loop: a loop whose iteration count is known at entry. Indefinite loop: a loop which iteration count is unknown at entry time (e.g., based on values calculated within the loop).
Macro	Many assembly (and programming) languages provide a “macro” facility whereby a macro statement will translate into a sequence of assembly (or high-level) language statements and perhaps other macro statements before being translated into machine code. There are two aspects to macros: definition and use.
Macro Definition	The definition of a macro.
Macro Call	The use of a macro.
Member definition	Any specification of the implementation of something defined as part of the definition of a class.
Member function (or method)	A procedure or function that is defined as part of a class and is invoked in a message passing style. Every instance of a class exhibits the behavior described by a member function/method of the class.
Operator	Any one of the special symbols (such as ‘*’ or ‘mod’) that perform some logical or mathematical operation upon objects and literals.
Parent	The Parent attribute is an attribute that links each object in an AST to its parent object, if any. The parent-child relationship is defined as follows. If object X has an AST (i.e., non-semantic) attribute A, and the value of attribute A for object X is, or contains, object Y, then the parent of Y is X. The Parent attribute is a universal converse relationship that is applicable to all syntactic attributes. It is a derivable relationship and is therefore treated as an optional universal converse relationship for all syntactic attributes.
Primitive type	See basic type
Parameter	One of the named entities associated with a subprogram, entry, or generic program unit.
Parameters	Data objects passed between the caller and the called procedural abstraction.
Pointer	An attribute of one object that contains an explicit reference to another object.
Pointer Expression	An expression that resolves to an attribute of one object that contains an explicit reference to another object.
Procedure	1. Any operation that does not return a significant value. 2. Any operation that may perform multiple actions and modify the instance to which it is applied, but does not return a value.
Program	Any static object-based application consisting of a set of types and classes interrelated specific to a particular (end-use) objective [OMG].
Programming Language	A language for programming, such as COBOL, C, C++, Java, etc. consisting of syntax and semantics for programming. The syntax of the language specifies those combinations of symbols that are in the language. The semantics specifies the “meaning” of syntactically correct constructs in the language.
Procedure Call	A call to a procedure.

Term	Definition
Qualified expression	An expression qualified by the name of a type or a subtype. It can be used to state the type or subtype of an expression, such as an overloaded literal.
Qualified Identifier Reference	A reference to a qualified name.
Qualified name	The name of any feature that has been qualified by the name of its enclosing class. Qualified names are used to prevent overriding in order to explicitly select the correct associated implementation.
Range	A continuous set of values of a scalar type. A range is specified by giving the lower and upper bounds for the values. A range may be used in a membership test.
Range Expression	An expression that evaluates to a range.
Range Type	A type that characterizes a range.
Return [Statement]	A keyword that causes a method to return to its caller. This is normally done as the last statement in a method, but return can appear anywhere within the body of a method. If a method has been declared as void, there is no return value and the return statement will not accept an argument. If the method has been declared as rerunning any data type other than void, a return statement is required, and the return statement must be followed by an expression of the correct type.
Reference	A reference to a value is just a location holding it.
Reference Expression	An expression that evaluates to a reference.
Scope	The region of program text over which a declaration has an effect (is in existence), also Program scope, procedure scope, block scope, type scope.
Sequence	X
Source	1. The thing operated upon or used as input to an operation or a complex process (i.e., a code generator, translator, or transformer). 2. See Source code.
Source Location	A location within a source file.
Source File	1. A file in which code in a programming language is located. 2. A file from which source is taken.
Source Program	A program defined in the programming.
Specialization	1. (a) the process of creating a specialization from one or more generalizations (b) the creation of a subclass via extension, refinement, or restriction. (c) An extension of the behavior of ta type of object. 2. (a) the result of using the specialization process (b) any derived class. 3. (a) any relationship from a generalization to one or more of its specializations (b) the relationship between a class and its parents, (c) the relationship between a parent and a descendant that has been modified by refinement or deletion so that the descendant is no longer behaviorally compatible with its parent.
Symbol	NSD
Target	The produced by or output by an operation.
Target File	A file (typically code) produced by an operation.

Term	**Definition**
Structure	NSD
Switch Statement	A statement used to compare an expression to a series of possible cases. If the expression matches the case, the code below the case statement is executed. A break statement is used to exit the switch block once a case has been matched. If the expression matches none of the case statements, the code below the default statement is run. The break and default statements are not mandatory.
Template	The C++ term for any parameterized metaclass or any parameterized function. Commentary: A class template specifies how a family of individual classes can be constructed much as a class declaration specifies how individual objects can be constructed.
Template Definition	The definition of a template.
Template Type	The definition of a template type.
Throw Statement	A throw statement is executed to indicate that an exception has occurred (i.e., a method could not complete successfully). This 1 called throwing an exception. A throw statement specifies an object to be thrown.
Type	Characterizes a set of values and a set of operations applicable to those values.
Type Definition	A type definition is a language construct introducing a new, unique type, whereas a subtype creates a compatible (possibly) constrained definition of the base type.
Type Declaration	A type declaration associates a name with a type introduced by a type definition.
Unary expression	An expression that computes a relation on one thing.
Variable	An abstraction used in imperative languages for memory location of cell.

Annex C - ASTM Core Concept Bibliography

The ASTM Core Terminology Bibliograpy provides a list of the published sources and references for the terminology and definitions provided in Annex B.

C.1 ASTM Core Terminology Bibliography

1 Data Models (pg. 991, The Computer Science and Engineering Handbook, CRC Press, 1997

2 Imperative Programming Paradigm (pg. 2004, The Science and Engineering Handbook, CRC Press, 1997)

3 Dictionary of Object Technology, SIGS Books, 1995

4 Booch, G., Software Engineering with Ada, Benjamin Cummings Publishing Company, 1983

5 Run Time Environments and Memory Management (pgs. 2188, The Science and Engineering Handbook, CRC Press, 1997)

6 The Object-Oriented Language Paradigm (pg. 2063, The Science and Engineering Handbook, CRC Press, 1997)

7 Beizer, Boris, Software Testing Techniques, Van Nostrand Reinhold, 1990

8 Deitel, Harvey M, Java, How to Program, Prentice Hall, 4th Edition, 2001

9 What is an Operating System? (pgs. 1662-1663, The Science and Engineering Handbook, CRC Press, 1997)

10 Palmer, G, Java Programmer's Reference, Wrox Press, Ltd., 2000.

11 Backhouse, R.C., Syntax of Programming Languages Theory and Practice, Prentice-Hall, 1979.

12 Griffith, A, Java Master Reference The Definitive Java Language Reference!, IDG Books, 1998.

13 Aho, A, Ullman, J., Principles of Compiler Design, Addisen-Wesley, 1977.

14 Gordon, M.J.C., The Denotational Description of Programming Languages An Introduction, SpringerVerlag, 1970.

15 The Computer Science and Engineering Handbook, CRC Press, 1997

16 Wikipedia The Free Encyclopedia. (http://en.wikipedia.org/wiki/Main_Page)

INDEX