INTERNATIONAL STANDARD

ISO 20534

First edition
2018-09

Industrial automation systems and integration — Formal semantic models for the configuration of global production networks

Systèmes d'automatisation industrielle et intégration — Modèles sémantiques formels pour la configuration des réseaux de production mondiaux

Reference number
ISO 20534:2018(E)

Contents

Page

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/TC 184, *Automation systems and integration*, Subcommittee SC 4, *Industrial data*.

Any feedback or questions on this document should be directed to the user’s national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Introduction

In reacting to change, competitive manufacturing industry aims to best understand the balance of possible options when making decisions on complex multi-faceted problems. Understanding how best to configure and re-configure a global production network, set against rapidly changing product-service requirements is one such complex problem area. Decisions consider multiple existing product and service variants, multiple new products and services to embrace the implications of new technological, economic, social, environmental and political requirements, and current production and service loads, as well as harmonizing and synchronizing production networks spread throughout the world, considering factors such as local supplier capabilities, transportation constraints, plant energy usage and production load forecasts.

While current Information and Communication Technology (ICT) tools play a significant role in support of such business development decisions, they need to do this in a well-integrated, trans-disciplinary way, with holistic solutions being critical to long-term competition (Huber 2014). Part of the solution to this problem lies in the exploitation of semantic technologies that provide a formal, logic base route to sharing meaning. This has been recognized as offering the potential to support interoperability across multiple related applications (Borgo et al. [23]; Chungoora et al. [24]). These provide formal, computer-based, methods of interpreting the meaning of concepts, their relationships to other concepts, and the constraints and rules that apply to their use.

The range of work in the use of formal semantics can be categorized into domain ontologies, foundation ontologies and reference ontologies. Domain ontologies tend to be limited to fairly narrow domains of applicability and so do not meet the holistic requirements mentioned above. Foundation ontologies are developed with a view to defining the semantics of everything but are too generic to offer positive constraints on any particular area of interest. The aim of reference ontologies for manufacture is to bridge the gap between these two and offer an effective support for interoperability, but in a targeted area or interest, i.e. in manufacturing. This document provides a contribution to such a manufacturing reference ontology by providing a formal semantic modelling approach to support the configuration of global production networks. It exploits the understanding gained from ISO 15531 and ISO 18629.

The manufacturing industry focus of this document is on how to design and configure a global production network (GPN) to produce and operate a new physical product or product-service. It does not address any operational aspects of production facilities, but rather models the flow relations between facilities in a production network. Future standards can build on this document to develop standard sematic models for in-factory systems.

The approach taken is based on exploiting the specialization capabilities of formal logic in order to progressively develop and constrain concepts and their relationships from foundation level descriptions to a level where the semantic models can be exploited by domain level software services and applications. The modelling approach starts from a systems functionality context as base from which to progressively represent global production systems, their relationships, constraints and related rules. The use of formal logic enables these semantic models to not only capture hierarchical relationships but also to capture and computationally exploit the constraints and rules that have been defined within these models.

The levels described in this document are different from the layered operational functionality of IEC 62264. The levels in this document are focused on levels of concept specialization. This starts from the key concepts from the context of any type of system through to specializations that are specific to manufacturing business systems. This is distinct from the layered operational functionality of IEC 62264, which is focused on the operational hierarchy of the business from higher-level production management down to shop floor.

This document uses some concepts that are common to supply chain models such as the Supply-Chain Operations Reference (SCOR) Model (Supply Chain Council 2014) and some concepts that follow from ISO 19440. SCOR is of limited relevance to this document as operational support is beyond its scope. Concepts from ISO 19440 have the greatest synergy with this document, which shares a business process oriented approach to enterprise modelling. While ISO 19440 provides model views to allow the identification of relevant object hierarchies and relationships between the different classes and subclasses, this document starts from a systems functionality context in order to build all subsequent hierarchies and relationships, including semantic constraints and rules.

This document is also distinct from the IEC/TR 62541 OPC Unified Architecture, although they both provide support to multi-vendor systems interoperability. IEC/TR 62541-1 provides a standard interface to facilitate the development of applications by multiple vendors that interoperate seamlessly together.

There are many semantic models that impinge on decision-making concerning the configuration and re-configuration of global production networks, as illustrated in Figure 1. This document provides the underpinning approach to the development of all of these models, but is necessarily focused on the production network concepts shown in Figure 1. Models of indicators, metrics, business, project, risk, location, scenario and product are therefore outside the scope of this document. The full value of this document can be gained through the development of future standards covering this full range of information, based on this document and drawing on existing standards in these areas wherever possible.

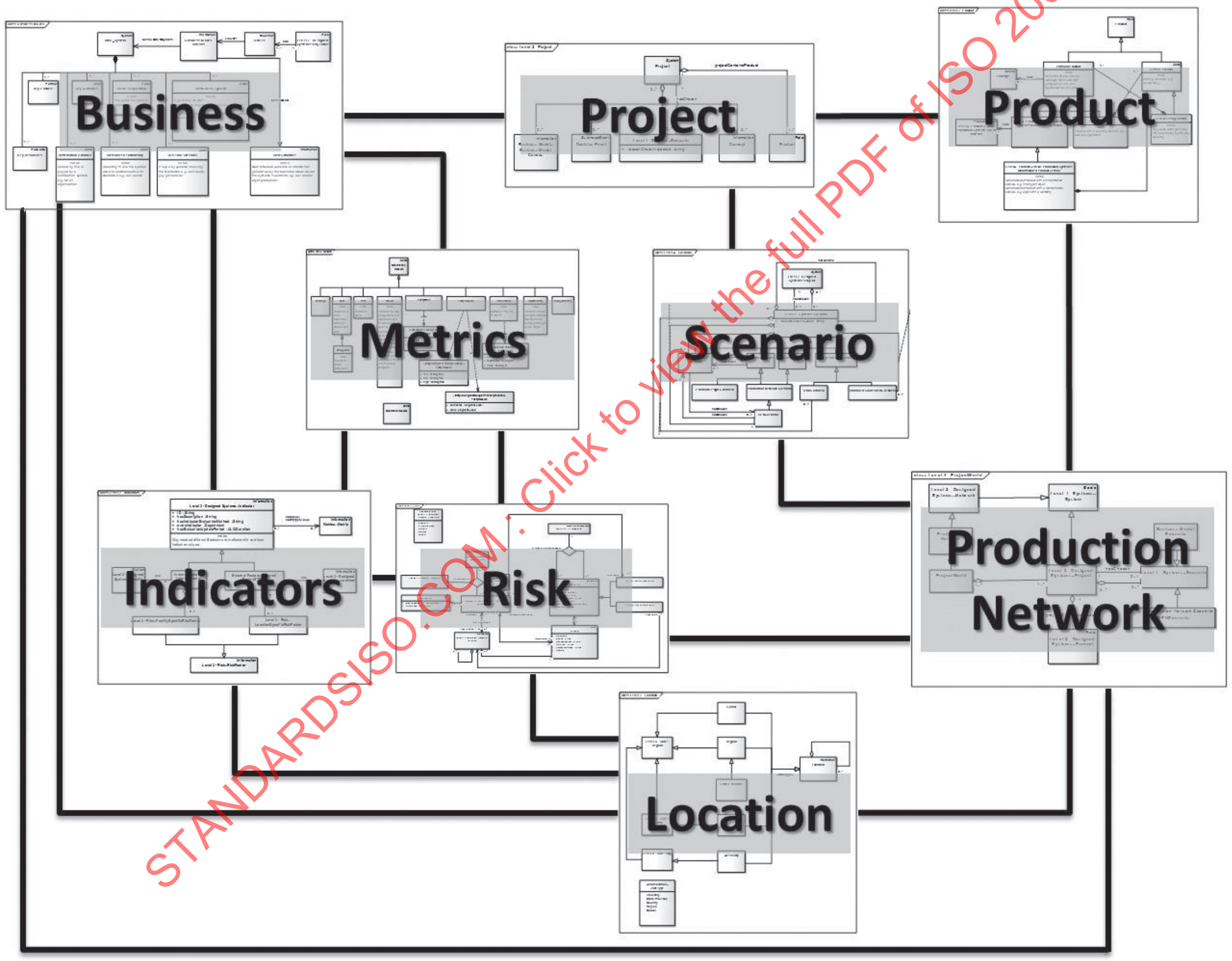


Figure 1 — Range of reference information needed to support production network configuration decisions

Industrial automation systems and integration — Formal semantic models for the configuration of global production networks

1 Scope

This document specifies a formal logic-based concept specialization approach to support the development of manufacturing reference models that underpin the necessary business specific knowledge models needed to support the configuration of global production networks.

This document specifies the following:

— the formal sematic model approach;

— hierarchical levels for property specialization;

— contexts for each level;

— key properties at each level;

— property relationships;

— property axioms;

— applicability rules.

The following are within the scope of this document:

— production networks for discrete product manufacture;

— formal semantics for the configuration of global production networks;

— system level formal semantics;

— designed system formal semantics;

— manufacturing business system formal semantics;

— global production systems network formal semantics.

The following are outside the scope of this document:

— in-factory formal semantics;

— formal semantics for the operation of global production networks.

2 Normative references

There are no normative references in this document.

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

3.1
activity
function
function that transforms *inputs* (3.26) to *outputs* (3.38)

[SOURCE: ISO/IEC/IEEE 31320-1:2012, 2.1.53, modified — Adapted from definition of “function”.]

3.2
actor
role (3.50) that is played by an *activity* (3.1)

Note 1 to entry: Actors are *processes* (3.41) that perform *functions* (3.1).

EXAMPLE A facility can act as a *supplier* (3.55), or as a *producer* (3.42), or as a *customer* (3.12).

3.3
axiom
well-formed formula in a formal *language* (3.28) that provides *constraints* (3.10) on the interpretation of symbols in the lexicon of a language

[SOURCE: ISO 18629-1:2004, 3.1.1]

3.4
basic
representation of an *entity* (3.19) or an *activity* (3.1)

Note 1 to entry: A basic requires no other concept to provide a *context* (3.11).

3.5
bill of materials
BOM
listing of all the subassemblies, parts and/or *materials* (3.30) that are used in the production of a *product* (3.43), including the quantity of each material required to make a product

Note 1 to entry: Adapted from IEC 62264-1:2013.

3.6
business event
event (3.20) that occurs in the business domain

3.7
business process
construct that represents a partially ordered set of business *processes* (3.41) or *enterprise* (3.18) *activities* (3.1), or both, that can be executed to realize one or more given objectives of an enterprise or a part of an enterprise to achieve some desired end-result

Note 1 to entry: Adapted from ISO 18629-1:2004.

3.8
cardinality
constraint (3.10) on the number of *entity* (3.19) *instances* (3.27) that are related to the subject entity through a *relationship* (3.47)

Note 1 to entry: Adapted from ISO/IEC 15474-1:2002, 4.2.

3.9
control
condition or set of conditions required for a *function* (3.1) to produce correct *output* (3.38)

[SOURCE: ISO/IEC/IEEE 31320-1:2012, 2.1.32]

3.10
constraint
rule that specifies a valid condition of *data* (3.13)

[SOURCE: ISO/IEC/IEEE 31320-2:2012, 3.1.41 (B)]

3.11
context
immediate environment in which a *function* (3.1) operates

[SOURCE: ISO/IEC/IEEE 31320-1:2012, 2.1.30]

3.12
customer
organization or person that receives a *product* (3.43)

[SOURCE: ISO/IEC/IEEE 26511:2011, 4.6, modified — The words "or service" have been deleted at the end of the definition and the note to entry has been deleted.]

3.13
data
character strings, words or numbers without any given *context* (3.11)

3.14
date
string that specifies a date

Note 1 to entry: Aligned with the Highfleet Ontology Library Reference definition of "date" (see Annex A).

3.15
decision event
event (3.20) where decisions are made or taken

3.16
end event
end of a *process* (3.41) sequence

Note 1 to entry: A specialized type of *basic* (3.4) which has an *input* (3.26) *role* (3.50) only.

3.17
energy
quantity characterizing the ability of a *system* (3.56) to do work

[SOURCE: ISO 772:2011, 1.134]

3.18
enterprise
group of organizations sharing a set of goals and objectives to offer *products* (3.43), *services* (3.53) or both

[SOURCE: ISO 14258:1998, 2.1.1]

3.19
entity
concrete or abstract thing in the domain under consideration

[SOURCE: ISO 19439:2006, 3.29]

3.20
event
construct that represents a solicited or unsolicited fact indicating a state change in the *enterprise* (3.18) or its environment

Note 1 to entry: Aligned with the Highfleet Ontology Library Reference definition of "event" (see Annex A).

[SOURCE: ISO 19440:2007, 3.1.33], modified — Original note to entry has been replaced by a new note to entry.

3.21
facility function
function (3.1) in an organization that performs one or more specific *activities* (3.1) by the provision of *resources* (3.49) in a given global *location* (3.29)

3.22
flow
relationship (3.47) from an *input* (3.26) to an *output* (3.38) or from an output to an input

Note 1 to entry: This is consistent with the Athena[28] definition: a flow is a relationship between two decision points: *process* (3.41) *roles* (3.50) [input, output, *control* (3.9), or *resource* (3.49)] or *gateways* (3.23) in a process.

3.23
gateway
element used to control how *process* (3.41) *flows* (3.22) interact as they converge and diverge within a *network* (3.35)Note 1 to entry: Adapted from Object Management Group[33] .

3.24
global production network
GPN
specialization of a *production network* (3.44), which contains *roles* (3.50) played by globally dispersed facilities

3.25
information
data (3.13) put in *context* (3.11)

EXAMPLE Facts, concepts, or instructions.

[SOURCE: ISO 10303-1:1994, 3.2.20, modified — The original definition is given as an example and a new definition has been provided.]

3.26
input
that which is transformed by an *activity* (3.1) into *output* (3.38)

[SOURCE: ISO/IEC/IEEE 31320-1:2012, 2.1.62, modified — The words "a function" have been replaced by "an activity".]

3.27
instance
individual occurrence of a *property* (3.46)

Note 1 to entry: This is distinct from the ":Inst" directive, which states what a property instantiates (see B.2.3).

Note 2 to entry: Adapted from ISO/IEC 15474-1:2002, 4.2.

3.28
language
combination of a lexicon and a grammar

[SOURCE: ISO 18629-1:2004, 3.1.12]

3.29
location
site or position

3.30
material
representation of an *entity* (3.19) that is comprised of physical materials

3.31
manufacturer
business *enterprise* (3.18) that is involved in the full *product* (3.43) life cycle

Note 1 to entry: The product life cycle includes the product design, production, operation and end of life phases.

3.32
manufacturing
function (3.1) or act of converting or transforming *material* (3.30) from raw material or semi-finished state to a state of further completion

[SOURCE: ISO 15531-1:2004, 3.6.22, modified — Note to entry has been deleted.]

3.33
manufacturing network
network (3.35) which is concerned with full life cycle of a *manufactured product* (3.34)

Note 1 to entry: The life cycle of a manufactured product includes the design, production, operation and end of life phases.

3.34
manufactured product
product (3.43) that exploits/consumes a raw *material* (3.30)

3.35
network
arrangement of nodes and interconnecting branches

[SOURCE: ISO/IEC 2382:2015, 2121314, modified — Notes to entry have been deleted.]

3.36
ontology
logical structure of the terms used to describe a domain of knowledge, including both the definitions of the applicable terms and their *relationships* (3.47)

[SOURCE: IEEE 1175-1:2002, 3.9]

3.37
organization function
key *function* (3.1) that an *enterprise* (3.18) provides

3.38
output
that which is produced by an *activity* (3.1)

[SOURCE: ISO/IEC/IEEE 31320-1:2012, 2.1.89, modified — The words “a function” have been replaced by “an activity”.]

3.39
physical product
entity (3.19) that plays the *role* (3.50) of a *product* (3.43)

3.40
plan
account of intended future course of action aimed at achieving specific goal(s) or objective(s) within a specific timeframe

3.41
process
structured set of *activities* (3.1), involving various *enterprise* (3.18) *entities* (3.19), that is designed and organized for a given purpose

[SOURCE: ISO 15531-1:2004, 3.6.29, modified — Note to entry has been deleted.]

3.42
producer
business *enterprise* (3.18) that produces goods or *services* (3.53) for sale

3.43
product
role (3.50) played by an *entity* (3.19) or *service* (3.53), or some combination of both, that is developed to be sold

3.44
production network
specialization of a *manufacturing network* (3.33) which is concerned with producing a *product* (3.43)

3.45
project
planned undertaking with specified targets

3.46
property
class
predicate that can apply to one *entity* (3.19) at a time and is used describe what the entity is

Note 1 to entry: Properties make up the taxonomic component of the *ontology* (3.36).

Note 2 to entry: The term "property" is used because the ontologies are semantic (modelling meaning) rather than set-theoretical (modelling categorizations).

3.47
relationship
real-world association among one or more *entities* (3.19)

Note 1 to entry: Adapted from ISO/IEC 15474-1:2002, 4.2.

3.48
requirement
condition or capability to be met or possessed by a *system* (3.56), *product* (3.43), *service* (3.53), result, or component to satisfy a contract, standard, specification, or other formally imposed document

Note 1 to entry: Adapted from Project Management Institute[34] .

3.49
resource
means used by an *activity* (3.1) to transform *input* (3.26) into *output* (3.38)

[SOURCE: ISO/IEC/IEEE 31320-1:2012, 2.1.71, modified — Adapted from definition of "mechanism", and the words "a function" have been replaced by "an activity".]

3.50
role
part played by a *basic* (3.4) in an *activity* (3.1)

Note 1 to entry: A role cannot exist without a *context* (3.11).

Note 2 to entry: A *system* (3.56) provides a context for the roles it contains.

Note 3 to entry: To aid efficiency all roles are declared as pairwise disjoint.

Note 4 to entry: A basic can play more than one role.

3.51
semantics
meaning of the syntactic components of a *language* (3.28)

[SOURCE: ISO/IEC/IEEE 31320-2:2012, 3.1.175]

3.52
scenario
representation of a specific potential or actual configuration to achieve the functionality of a *system* (3.56)

Note 1 to entry: Different scenarios allow for different what-if analyses to be performed.

3.53
service
system (3.56) *function* (3.1) that plays the *role* (3.50) of a *product* (3.43)

3.54
start event
start of a *process* (3.41) sequence

Note 1 to entry: A specialized type of *basic* (3.4) which has an *output* (3.38) *role* (3.50) only.

3.55
supplier
facility in an organization that enters into an agreement with the acquirer for the supply of a *product* (3.43)

[SOURCE: ISO/IEC/IEEE 15288:2015, 4.1.45, modified — The words "organization or an individual" have been replaced by "facility in an organization", the words "or service" have been deleted at the end of the definition and the notes to entry have been deleted.]

3.56
system
combination of interacting *resources* (3.49) organized to achieve one or more stated purposes

[SOURCE: ISO/IEC/IEEE 15288:2015, 4.1.46, modified — The word "elements" has been replaced by "resources" and the notes to entry have been deleted.]

3.57
system function
activity (3.1) in a *system* (3.56) that transforms an *input* (3.26) set of elements into a set of *output* (3.38) elements

3.58
timespan
period of time with a start date and an end date where *date* (3.14) can be specified down to fractions of second

4 Abbreviated terms

BOM	Bill of Materials
KFL	Knowledge Frame Language (Highfleet)
CLIF	Common Logic Interchange Format (ISO/IEC 24707)
ECLIF	Extended Common Logic Interchange Format (Highfleet)
GPN	Global Production Network
IC	Integrity Constraint
MLO	Middle Level Ontology (Highfleet)
ULO	Upper Level Ontology (Highfleet)
UML	Unified Modelling Language (ISO/IEC 19501).

5 Formal semantic models for the configuration of global production networks

5.1 Formal semantics

Reference models based on this document provide a formal semantic base with which application vendors shall conform if they are to maintain interoperability.

The concept specialization approach of this document, based on formal logic, provides a new dimension to the computational interpretation of models based on the document when compared to most existing production related standards. It ensures that users shall conform to the defined standardized constraints, through the use of its logic.

The Formal Semantic Models for the Configuration of Global Production Networks are expressed using the Knowledge Frame Language (KFL) and Extended Common Logic Interchange Format (ECLIF) from Highfleet Ontology Library Reference (see Annex A), both of which are based on Common Logic (ISO/IEC 24707). The KFL, ECLIF and the upper level concept definitions used in this document are provided in the annexes of this document.

NOTE Annex B provides a reference guide to the syntax and commonly used techniques for writing Knowledge Framework Language (KFL) files. Annex C contains information on ECLIF. Annex D outlines the FLEXINET approach.

5.2 Overview of the levels of specialization

The premise behind the formal semantic models for the configuration of global production networks is that for ease of construction, effective interoperability and flexible re-use enterprise ontologies shall be built from a common base that utilizes a common reference ontology wherever possible. A simple statement describes the basis of generalization: “A design of an ontology representing the core elements of a particular enterprise, will end up with a good number of elements that are not exclusive to this particular enterprise, but common to some other enterprises that operate in the same sector”.

Following this reasoning, it is inferred that a subset of the elements that are common to a particular sector shall be applicable or extrapolated to different sectors. In other words, some of the elements that are applicable to the Pumps Industry sector will also be applicable to the White Goods sector.

Both sectors are part of the manufacturing industry, so it is stated that the concepts that are widely applicable to different manufacturing sectors belong to the broader area of manufacturing industry, and not to a particular sector. In this area reside the elements that are specific to the manufacturing industry, but not necessarily for other industry sectors such as Finance, Assurance, Construction, Mining and Agriculture.

However, some of the concepts identified for manufacturing industry are applicable, in a generalized sense, to other industry sectors. In this case, they belong to the broader area of Designed Systems. This leads to a comprehensive set of general concepts and relations that shall be universally accepted and understood across the full range of business sectors.

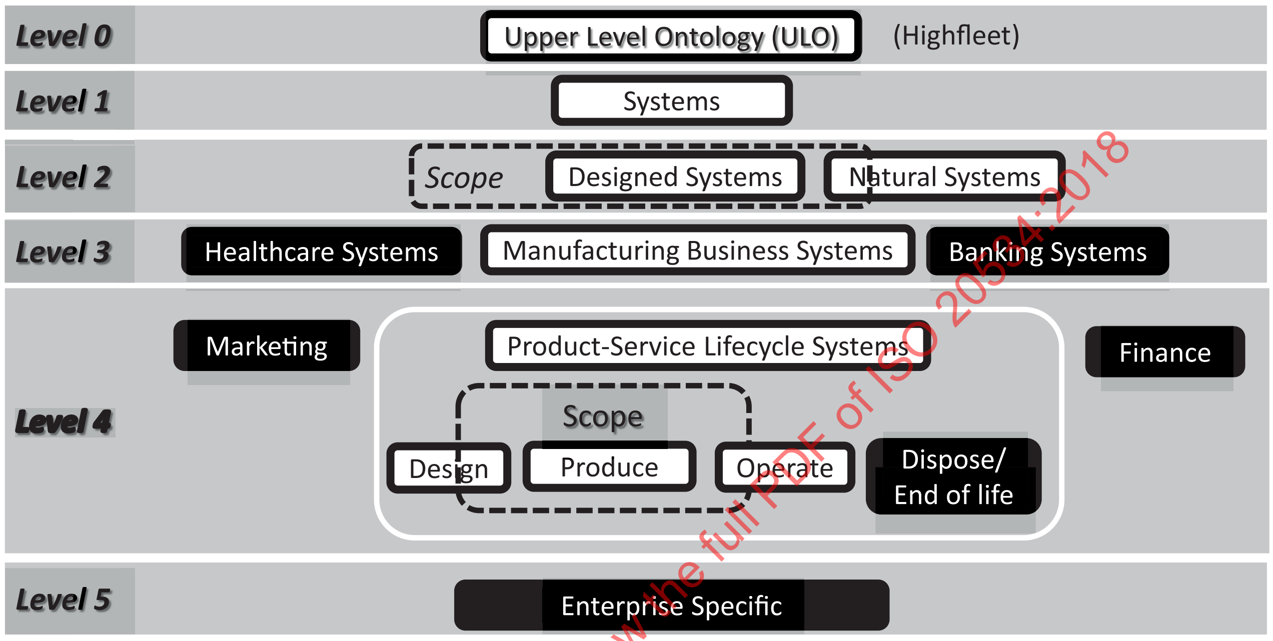


Figure 2 — Levels of the formal semantic models for the configuration of global production networks approach

To enable the management of complexity within the ontology and to facilitate re-use across enterprise specific domains the formal semantic models for the configuration of global production networks approach is organized into five levels, as illustrated in Figure 2. These levels are needed to specialize the concepts from the foundation level (level 0) to the enterprise specific level (level 5). Each level inherits concepts from the level above and provides a constraining influence on the subsequent level, the ontology becoming more domain specific (specialized) with each level. The contexts that are of relevance to this document are shown in Figure 2 in boxes with a pale background. The boxes with a dark background are illustrations of contexts that are outside the scope of this document.

The Level 0 Upper Level Ontology, which is fully generic, consists of foundation concepts applicable to all domains. The foundation concepts include time, events, aggregation and lists which, are derived from the Highfleet Upper Level Ontology (ULO) (see Annex A). The Highfleet ULO is based on OntoClean (Guarino and Welty, 2002[26]) and the subsequent Common Logic Standard (ISO/IEC 24707).

Level 1 starts from the premise that a systems context is fundamental to the definition of any systems based network. Level 1 therefore contains the few key generic concepts necessary to support the system functionality provision in any system.

Level 2 uses Banathy's classification to specialize systems into "Natural Systems" and "Designed Systems". Natural systems are living systems of all kinds, including the solar system and the Universe. Designed systems, are man-made creations, including fabricated physical systems, conceptual knowledge and purposeful creations.

Level 3 differentiates designed systems by specifying the business system context for subsequent applicable areas. A few are illustrated in Figure 2, but include manufacturing, healthcare, construction, tourism, grocery, farming, finance. Manufacturing Business Systems concepts are then further specialized within Level 4. Level 3 areas such as Healthcare Systems and Banking Systems would also possess areas providing relevant specializations at Level 4 and some of these areas can be similar to those within the Manufacturing Business Systems domain, however, the concepts contained would be specialized within the context of the parent area.

Level 4 contains concepts specifically relating to the Manufacturing Business Systems domain. The area considered at level 4 is Product-Service Lifecycle Systems, implemented in Global Production Networks. The lifecycle phases are denoted as design, produce, operate and end of life (including disposal, recycling and remanufacturing). The focus is how to design a GPN to produce and operate a product-service. The main area considered within the Product-Service Lifecycle is therefore “Produce” (producing the product-service) but the scope also overlaps into “Design” (of the network) and “Operate” as the operation of the product and the service needs to be considered in design.

Level 5 provides enterprise specific concepts for the product-service production domain for a specific enterprise. These shall be developed for each specific enterprise and be mapped to the reference ontology concepts in order to take advantage of the underlying structure that it provides.

6 System functionality formal semantics — Level 1

6.1 Overview

As stated in 5.2, Level 1 starts from the premise that a systems context is fundamental to the definition of any system based network. Level 1 therefore shall contain the few key concepts necessary to model any generic system’s functionality. In a typical systems engineering approach a system is considered to transform inputs into outputs and is defined as “a combination of interacting elements organized to achieve one or more stated purposes” (ISO/IEC/IEEE 15288:2015; Athena 2006. This document focuses on the constructs to model system functions where the “interacting elements” in a system are represented as “basics” that play the role of “resource” while the “achieve one of more stated purposes” is represented by the “output” concept which in turn is linked to the “activity” and “system function” concept.

Figure 3 provides a visual illustration of the level 1 concepts and relations using Unified Modelling Language (UML) class diagrams. The full formal ontology formal ontology is then provided in the subclauses below, in terms of properties, relations and axioms. This same form of explanation is provided subsequently for each level.

One key aspect to the level 1 is the representation of Activity and System Function through the use of formalized IDEF0 concepts (PUBs 1993). The Activity concept being equivalent to an IDEF0 activity and the Roles of Input, Output, Resource and Control playing the part of IDEF0 flows. An Input represents what is brought into and is transformed or consumed by the activity to produce Outputs. A Control is a specialized Input that provides a condition required to produce the correct activity output (PUBs 1993; Athena 2006). Resource represents the means by which an activity is performed.

Another key aspect is the definition of the two main parent classes at level 1; those of Basic and Role. A Basic concept is independent of the system or context so its definition does not depend on any other concept and an instance of a Basic always retains its identity. Role defines how Basics are used in any particular System.

Basics occurring at level 1 are classified as Activity and Entity, with an Entity being anything of interest. Subtypes of Entity at level 1 are Information, Material and Energy. Note that Material is used to denote any material thing and not raw materials. A Basic can be comprised of Basics.

EXAMPLE 1 “Bottled water” comprised of the Materials “bottle”, “cap” and “mineral water”.

This recursion can also be applied to Activity to represent sub-activities or System Functions to represent systems of systems functionality. An Activity is also a subtype of Basic and provides a context for the Roles that other Basics play in its performance.

Role depends on the Activity for its context and an instance of a Role cannot exist without such a context. Basics are used to play a Role within an Activity.

EXAMPLE 2 A person Joe may have a Role as a lecturer in a university i.e. he plays the Role of a Resource in the context of a university activity. If the university closes the lecturer role ceases to exist whereas the person Joe (an instance of a Basic) will still be present.

EXAMPLE 3 Consider two manufacturing companies X and Y, with X selling a machine to dry clothes and Y selling a clothes-drying service. In Company X a drying machine plays the Role of an output (a product). In company Y the drying machine plays the Role of a Resource to support the service activity whose output is the service. Hence the Role of the drying machine changes whereas the Basic "drying machine" is always a "drying machine".

Roles can be comprised of Roles,

EXAMPLE 4 A lecturer Role can be allowed to be comprised of administration, teaching and research Roles.

Additionally, there is no specific requirement for a Role to be played by a Basic, enabling empty Roles to be modelled.

EXAMPLE 5 If a person Joe left his Role as a lecturer the Role of lecturer would still exist but as a lecturer vacancy.

A Basic playing a Role for certain period of time is represented using the TimeSpan concept and modelled using the relation "playsRole".

EXAMPLE 6 In the context of a manufacturing organization system, the Basic "drying machine" can play the Role of a Product during the TimeSpan of that business system. If the company moves across fully to providing a drying service then the TimeSpan of the dryer as a product will come to an end.

Roles are also represented as being played within a Scenario, so the playsRole relation is a quaternary relation. The Scenario concept has been provided in order to represent alternative ways of meeting the same overall system functional requirements i.e. in production terms each scenario provides one way of providing the system output. Alternative scenarios offer the potential for companies to perform 'what-if" analysis on each alternative.

EXAMPLE 7 A manufacturer can use a supplier from Germany in one scenario and in another to use a supplier from Japan.

The concept of Role here is well aligned with that of ISO 19440, although at this level no distinction is drawn between people, machine or product roles, as these would be specializations at a lower level. The ideas of Roles also share views with those of Kozaki *et al*. (2006), Kozaki *et al*. (2008) and Mizoguchi *et al*. (2012). In common with those views, the concepts of Basic, Role and Role aggregation are captured. However in this approach Time and Role context are explicitly modelled.

A Basic can affect the state of a role.

EXAMPLE 8 The size of a Basic "bottled water" playing the Role of a product can influence the dimensions required for a packing resource Role.

Additionally a Role can affect the state of a Role.

EXAMPLE 9 Within the lecturer Role more duties allotted to the administration Role would cause duties to be removed from the teaching Role.

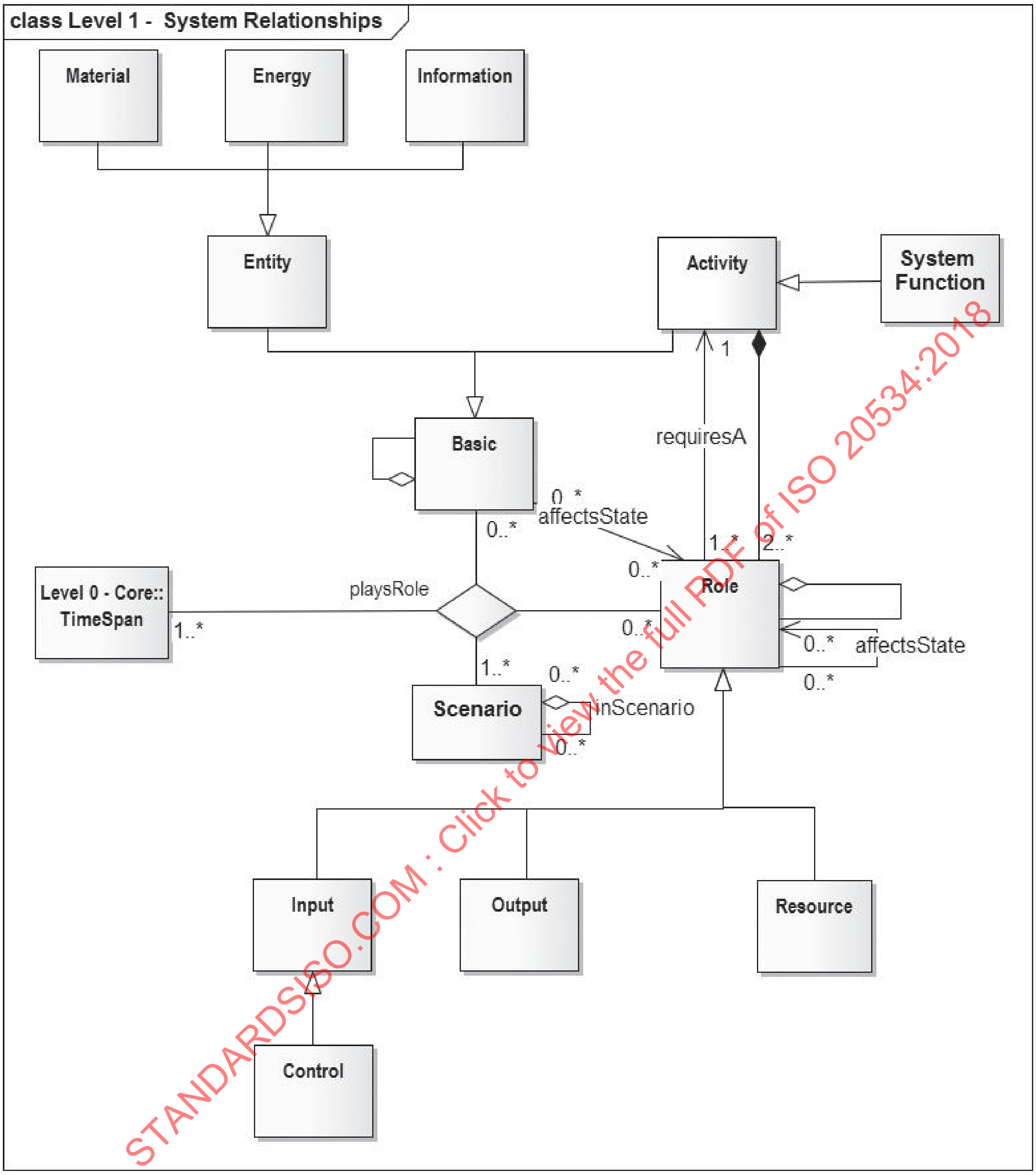


Figure 3 — Level 1 UML class diagram

6.2 Level 1 formal semantics — Context

6.2.1 Context

KFL notation for context:

:Use MLO

:Ctx 1SYSCtx

:Inst UserContext

:supCtx MLO

6.3 Level 1 formal semantics — Properties

6.3.1 Property — Basic

KFL notation for basic:

:Prop Basic

:Inst Type

:Inst NonLogicalFunctor

:sup Particular

:rem "represents an entity or an activity. A Basic concept is independent of the <sym>1SYSCtx.Activity</sym> or context, its definition does not depend on another concept. At the Systems level a Basic can be classified as <sym>1SYSCtx.Activity</sym> or <sym>1SYSCtx.Entity</sym>."

6.3.2 Property — Entity

KFL notation for entity:

:Prop Entity

:Inst Type

:Inst NonLogicalFunctor

:sup Basic

:partitionedBy (listof Energy Material Information)

:rem " any concrete or abstract thing in the domain under consideration. A <sym>1SYSCtx.Basic</sym> which is not a <sym>1SYSCtx.Activity</sym>."

6.3.3 Property — Activity

KFL notation for activity:

:Prop Activity

:Inst Type

:sup Basic

:sup MLO.Object

:rem " A function that transforms *inputs* to *outputs*."

(disjointSubProps Activity)

6.3.4 Property — System function

KFL notation for system:

:Prop SystemFunction

:Inst Type

:sup Activity

:sup MLO.Object

:rem “the activity in a system that transforms an input set of elements into a set of output elements that may be the same or measurably different from the input set"

(disjointSubProps SystemFunction)

6.3.5 Property — Energy

KFL notation for energy:

:Prop Energy

:Inst Type

:Inst NonLogicalFunctor

:sup Entity

:rem "The capacity of a body or <sym>1SYSCtx.System</sym> to do work. Can be an integer so modelled as a <sym>MLO.NonLogicalFunctor</sym>.

Properties subsumed by Energy should also be subsumed by <sym>MLO.ConcreteEntity</sym> or <sym>MLO.AbstractEntity</sym>"

(disjointSubProps Energy)

6.3.6 Property — Material

KFL notation for material:

:Prop Material

:Inst Type

:sup Entity

:sup MLO.Object

:rem "an entity that is comprised of physical materials. An <sym>MLO.Object</sym> is anything that has a location in time and space."

(disjointSubProps Material)

6.3.7 Property — Information

KFL notation for information:

:Prop Information

:Inst Type

:Inst NonLogicalFunctor

:sup Entity

:rem "Information is data put in context; it is related to other pieces of data. Can be a number so modelled as a <sym>MLO.NonLogicalFunctor</sym>. Properties subsumed by Information should also be subsumed by <sym>MLO.ConcreteEntity</sym> or <sym>MLO.AbstractEntity</sym>."

(disjointSubProps Information)

6.3.8 Property — Role

KFL notation for role:

:Prop Role

:Inst Type

:sup MLO.AbstractEntity

:disjointWith Basic

:rem "The part played by a basic in an activity. Role type includes all of the types of <sym>MLO.AbstractEntity</sym> terms that participate in activities. A Role cannot exist without a context. A <sym>1SYSCtx.SystemFunction</sym> provides a context for the Roles it contains. To aid efficiency all Roles are declared as pairwise disjoint. A Basic can play more than one role."

6.3.9 Property — Input

KFL notation for input:

:Prop Input

:Inst Type

:sup Role

:rem "that which is transformed by an activity into output."

6.3.10 Property — Output

KFL notation for output:

:Prop Output

:Inst Type

:sup Role

:rem "that which is produced by an activity."

6.3.11 Property — Resource

KFL notation for resource:

:Prop Resource

:Inst Type

:sup Role

:rem " the means used by an activity to transform *input* into *output* "

6.3.12 Property — Control

KFL notation for control:

:Prop Control

:Inst Type

:sup Input

:rem "condition or set of conditions required for a function to produce correct output. A control is also an input."

6.3.13 Property — Scenario

KFL notation for scenario:

:Prop Scenario

:Inst Type

:sup AbstractEntity

:rem " represents a specific potential or actual configuration to achieve a system's functionality."

:disjointWith Role

:disjointWith Basic

(disjointSubProps Scenario)

6.4 Level 1 formal semantics — Relationships

6.4.1 Relationship — Affects state

KFL notation for affects state:

:Rel affectsState

:Inst BinaryRel

:Inst RigidRel

:Inst AsymmetricBR

:Sig Particular Role

:Args "Affector" "Role"

:lex "?1 affects Role ?2"

:rem "affectsState is unidirectional so is an AsymmetricBR"

6.4.2 Relationship — Basic affects role

KFL notation for basic affects the state of a role:

:Rel basicAffectsState

:Inst BinaryRel

:Inst RigidRel

:supRel affectsState

:Sig Basic Role

:Args "Basic" "Role"

:lex "Basic entity ?1 affects Role ?2"

:rem "basicAffectsState holds between Basic individuals and Role individuals."

6.4.3 Relationship — Role affects the state of role

KFL notation for role affects the state of a role:

:Rel roleAffectsState

:Inst BinaryRel

:Inst RigidRel

:supRel affectsState

:Sig Role Role

:Args "Role" "Role"

:lex "Role ?1 affects Role ?2"

:rem "roleAffectsState holds between Role individuals and Role individuals."

6.4.4 Relationship — Plays role

KFL notation for plays role:

:Rel playsRole

:Inst TernaryRel

:Inst NonRigidRel

:Sig Basic Role Scenario

:Args "Basic" "Role" "Scenario"

:lex "Basic entity ?1 plays Role ?2 in Scenario ?3"

:rem "A basic plays a role in a scenario. To provide a <sym>RootCtx.TimeSpan</sym> use the ECLIF operator <sym>holdsIn</sym>."

6.4.5 Relationship — Role requires a context provided by an activity

KFL notation for a role requires a context provided by an activity:

:Rel requiresA

:Inst BinaryRel

:Inst RigidRel

:Sig Role Activity

:Args "Role" "Activity"

:lex "Role ?1 depends on Activity ?2"

:rem "A role requires a context provided by one activity."

(functionalArg requiresA 2)

6.4.6 Relationship — Basic composed of a basic

KFL notation for basic contains a basic:

:Rel basicContainsBasic

:Inst BinaryRel

:Inst RigidRel

:Inst AntisymmetricBR

:Sig Basic Basic

:Args "sup" "sub"

:lex "?1 is composedOf ?2"

:rem "sup contains sub. Given that sup and sub are not identical, then it is not the case that sub contains sup."

6.4.7 Relationship — Role composed of a role

KFL notation for Role contains a role:

:Rel roleContainsRole

:Inst BinaryRel

:Inst RigidRel

:Inst AntisymmetricBR

:Sig Role Role

:Args "sup" "sub"

:lex "?1 is composedOf ?2"

:rem "sup contains sub. Given that sup and sub are not identical, then it is not the case that sub contains sup."

6.4.8 Relationship — Activity contains a role

KFL notation for activity contains a role:

:Rel activityContainsRole

:Inst BinaryRel

:Inst RigidRel

:Sig Activity Role

:Args "Activity" "Role"

:lex "Activity ?1 contains Role ?1"

6.5 Level 1 formal semantics — Axioms

6.5.1 Axiom — Role requires an activity to provide a context

ECLIF notation for 'a role requires an activity to provide a context':

:Name "1SYS - Axioms"

:Description "Axioms pertaining to relations occurring at the Systems level."

:Use 1SYSCtx

```
(=> (Role ?r)
 (exists (?a)
      (and (Activity ?a)
           (requiresA ?r ?a))))
```

:IC hard "The Role ?r requiresA Activity to provide a context."

6.5.2 Axiom — An activity cannot contain a role and play the role

ECLIF notation for ‘an activity cannot contain a role and play the role’:

```
(=> (and (Activity ?activity)
    (playsRole ?activity ?role ?scenario)
    )
     (not(activityContainsRole ?activity ?role)))
```

:IC hard "The same Activity cannot contain a Role and play the Role. ?activity playsRole ?role in Scenario ?scenario. The relation activityContainsRole does not hold between ?activity and ?role."

6.6 Level 1 formal semantics — Rules

6.6.1 Rule — Role requires an activity

ECLIF notation for ‘role requiring an actvity:

```
:Name "1SYS - Rules"
:Use 1SYSCtx
(=> (requiresA ?x ?y)
    (activityContainsRole ?y ?x))
```

;;;A Role requiring an activity as a context implies that the Activity contains the Role

6.6.2 Rule — Activity containing a role

ECLIF notation for ‘activity containing a role’:

```
(=> (activityContainsRole ?x ?y)
    (requiresA ?y ?x))
```

;;;An Activity containing a Role implies that the Role requires the Activity as a context

7 Designed systems formal semantics — Level 2

7.1 Overview

As mentioned in the introduction and illustrated in Figure 1, there are many semantic models that impinge on decision-making concerning the configuration and re-configuration of global production networks. This document focuses on production network concepts, but includes some general definitions for related concepts that should ideally be developed in or drawn from other future standards. The concept for location is a good example of such a concept. This is needed to identify the global location of production facilities, but this is a narrow definition compared with the potential full range of location information requirements.

7.1.1 Level 2 Systems overview

There are many potential activities that can be described at level 2 as system functions that are relevant to designed systems. Figure 4 provides a UML class diagram that depicts a selection of these that have relevance to manufacturing system functions.

NOTE Of those that are shown, the System Functions that are directly relevant to this document are Organization Function, Facility Function and Project. The facilities in an organization provide the key system elements in configuring a production network to achieve some overall functionality. The Project concept represents the functions to be performed against some new task, in this case related to a production network configuration task. It also provides a concept that can be expanded to include product design and business modelling activities in future standards development.

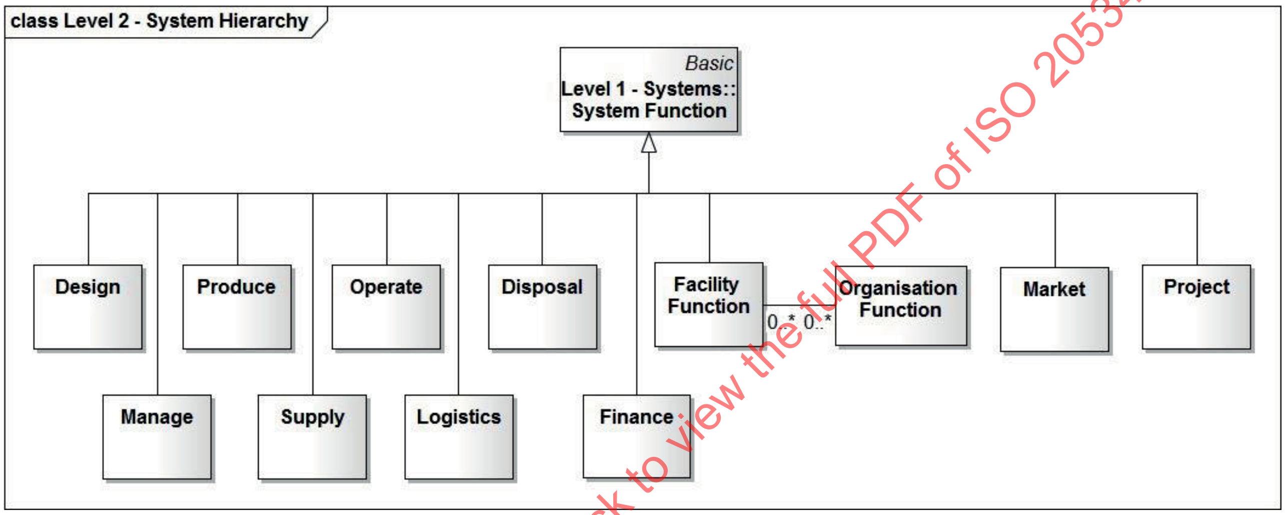


Figure 4 — UML class diagram depicting system functions at level 2

7.1.2 Level 2 Role overview

As explained in 6.1, the concept Role provides a view on how Basics are used in any particular System. The same "basic" business can be considered to be playing the role of

a) a manufacturer when considering how things are made;

b) a supplier when considering what it is going to offer to potential customers, or

c) a customer when considering what it needs to purchase from other suppliers.

The term Actor has been introduced to indicate roles that can be played by System Functions. A few examples of possible Actors are illustrated in Figure 5 with those that are of particular importance to this document being Supplier, Manufacturer, Producer, Customer and Service Provider.

Similarly Entities can also play roles in particular systems.

EXAMPLE 1 A machine tool (a Material Entity) can play the role of a Resource in a machine shop operation.

EXAMPLE 2 An order (an Information Entity) can play the role of an Input in a scheduling operation.

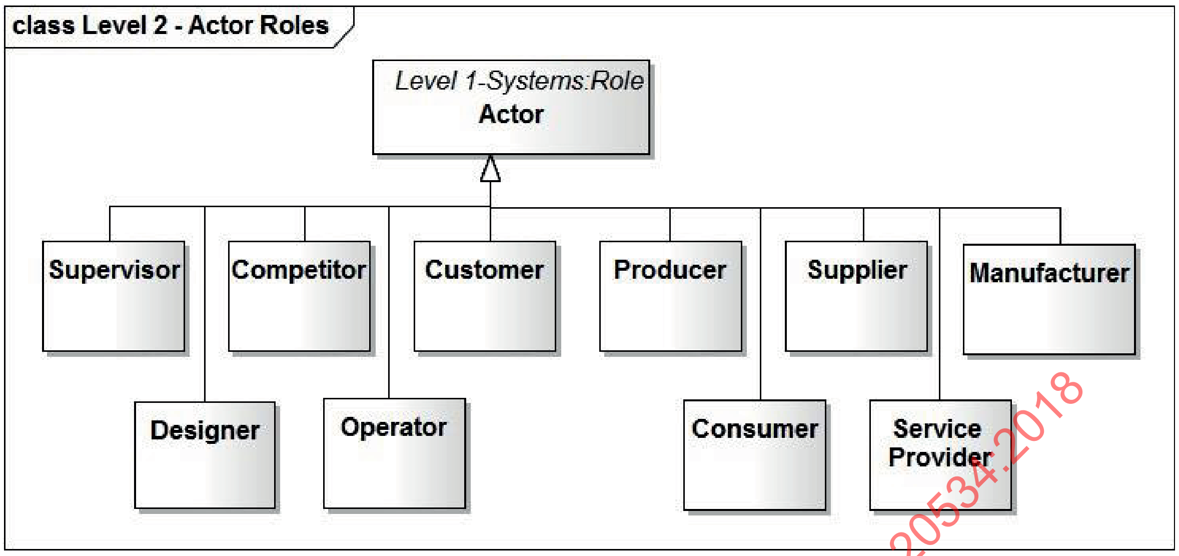


Figure 5 — UML class diagram depicting roles, especially actor roles

7.1.3 Level 2 Network overview

The UML diagram depicted in Figure 6 illustrates the concepts and relations for the level 2 representation of a network. This defines a network as a scenario that captures an arrangement of nodes as System Functions and Gateways while Flows between Inputs and Outputs provide the interconnecting branches. This model is in line with process modelling techniques in general use but also adds the potential for the logical constraints to be included. The concepts and relations are modelled in 7.2 and 7.3, while the logical constraints that are added in this document through the formal axioms are defined in 7.5.

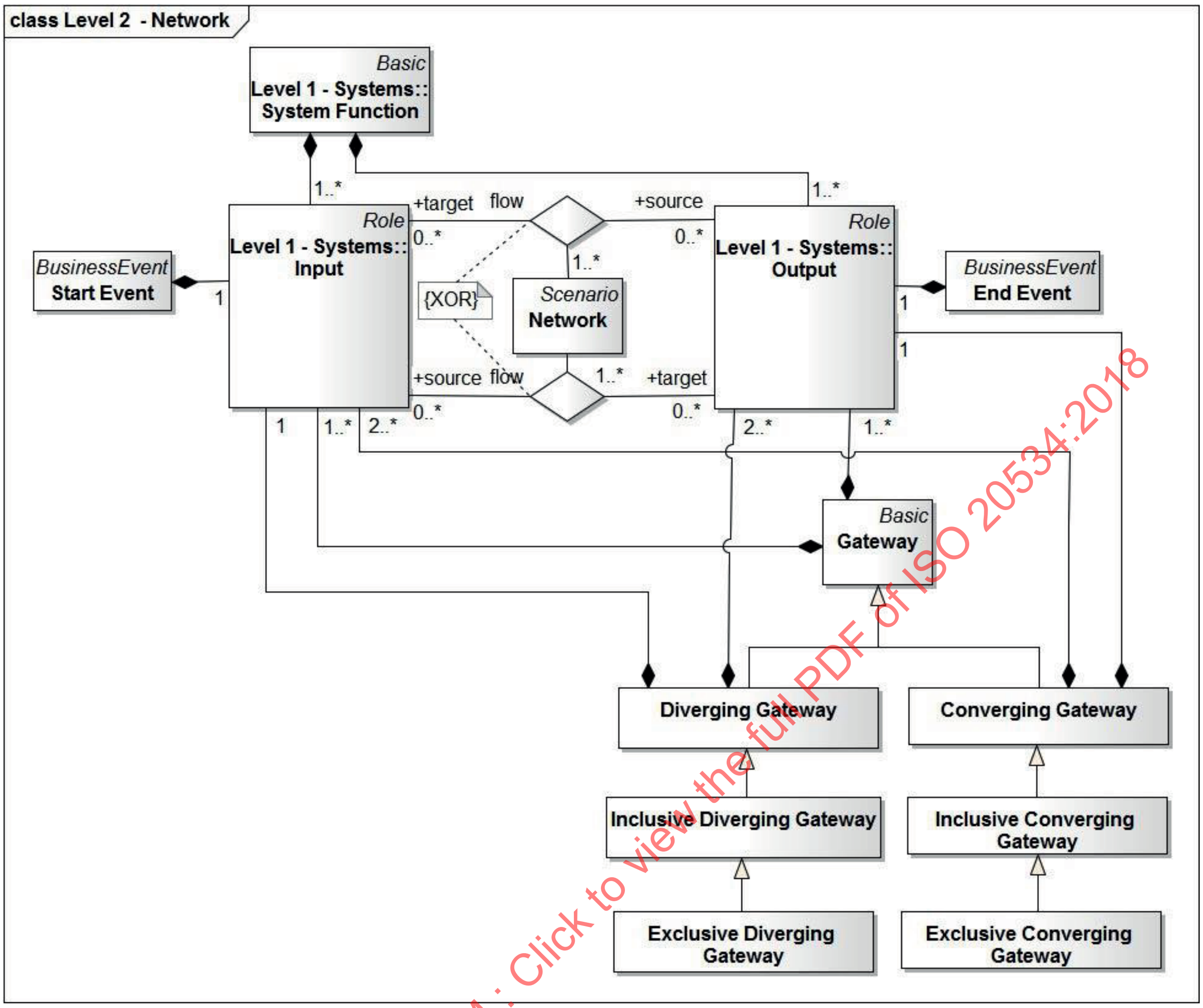


Figure 6 — UML class diagram depicting the representation of network at level 2

7.1.4 Product overview at levels 2 and 4

Figure 7 provides the UML class diagram of Product relevant to this document that seeks to model the key concepts and relationships that are significant to distinguish between physical products and services that are offered for sale. It is not intended to model the concept of product beyond this requirement, along with the necessary link from a physical manufactured product to its bill of materials, in order to provide a link to potential suppliers.

In this document a product is defined as an Entity or Service or some combination of both that is developed to be sold i.e. the Product is an Entity or Service that is playing the role of a Product. The term PhysicalProduct is used to denote an Entity playing the role of a Product. The term Service is used to denote a System Function that plays the role of a Service.

EXAMPLE 1 A machine tool being sold by a machine tool manufacturer is a Material Entity playing the role of a Product.

EXAMPLE 2 A will being sold by a solicitor is an Information Entity being sold as a Product.

EXAMPLE 3 A volume of gas being supplied to a consumer is an Energy Entity playing the role of a Product.

Service is defined as system function that is sold as a Product i.e. rather than a physical thing, a Service is a function that has some value that can therefore be sold as product.

EXAMPLE 4 Advice from a consultant can be sold as a Product.

Many Service products will also exploit a physical product and these are denoted as ServiceUsingPhysicalProduct.. In these cases the purchase is for the use of the physical product and not for the physical product itself.

EXAMPLE 5 Power by the Hour is a Service which requires an aero engine.

EXAMPLE 6 A lease car requires the use of a car.

Physical Products can also include a Service element which is part of the product and these are denoted by the term PysicalProductWithService.

EXAMPLE 7 A washing machine with a maintenance contract.

A "Manufactured Product" is a product that consumes raw materials in order to produce physical entities. A Bill of Material is a list of all parts required to manufacture and assemble the Manufactured Product.

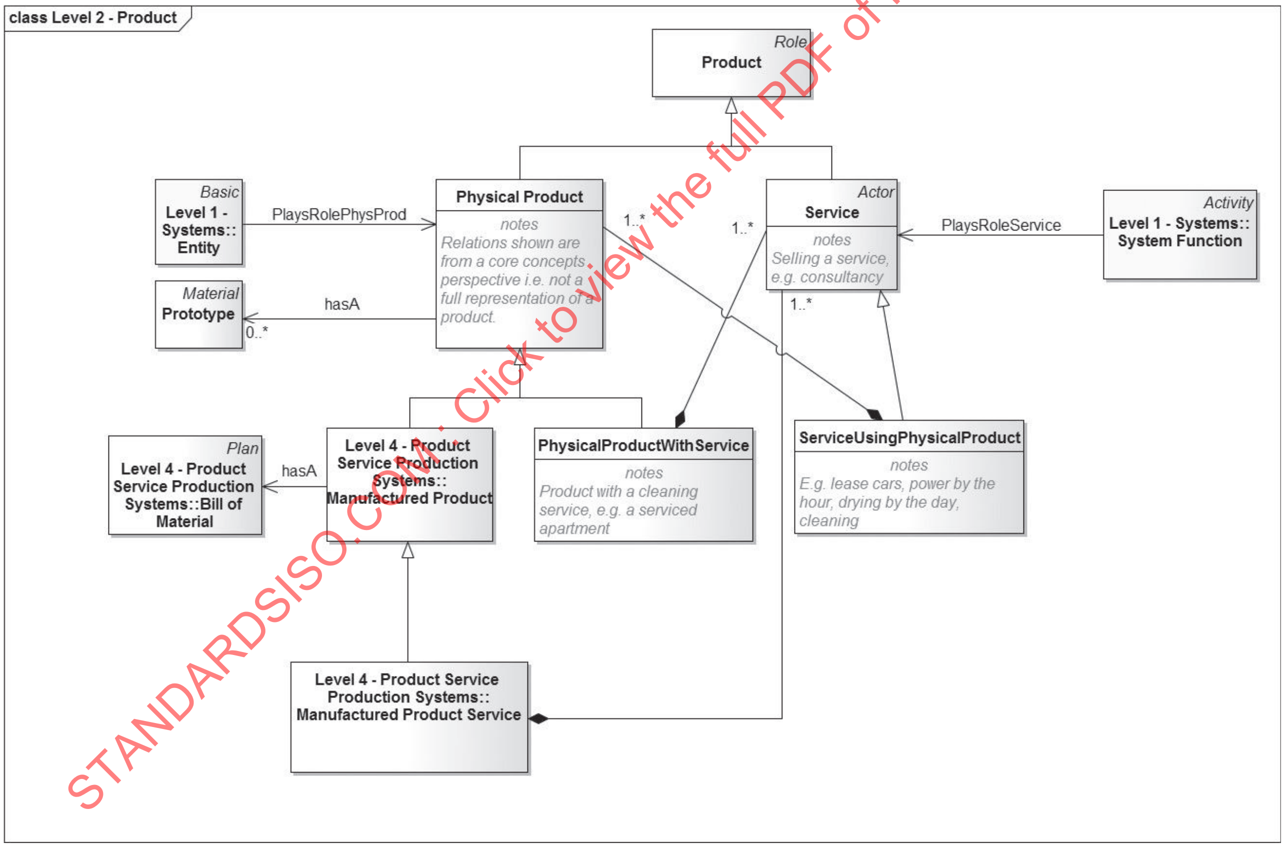

class Level 2 - Product
Role
Product
Basic
Level 1 - Systems:: Entity
PlaysRolePhysProd
Physical Product
notes
Relations shown are from a core concepts perspective i.e. not a full representation of a product.
1..*
1..*
Actor
Service
notes
Selling a service, e.g. consultancy
PlaysRoleService
Activity
Level 1 - Systems:: System Function
Material
Prototype
hasA
0..*
1..*
Plan
Level 4 - Product Service Production Systems::Bill of Material
hasA
Level 4 - Product Service Production Systems:: Manufactured Product
PhysicalProductWithService
notes
Product with a cleaning service, e.g. a serviced apartment
ServiceUsingPhysicalProduct
notes
E.g. lease cars, power by the hour, drying by the day, cleaning
Level 4 - Product Service Production Systems:: Manufactured Product Service

Figure 7 — UML class diagram depicting the representation of product at levels 2 and 4

7.1.5 Overview of Location

Location is an important concept for globally based decision making. The appropriate definitions for Location are dependent on use and can be difficult to define precisely as location boundaries are often determined by human demarcation (Smith & Varzi, 2000). In this document, to support decisions related to global production networks, the concepts and relations shown in Figure 8 have been used. As mentioned at the start of Clause 7, a future standard that defines an ontology for location that can be commonly exploited across industrial data standards would be of great benefit.

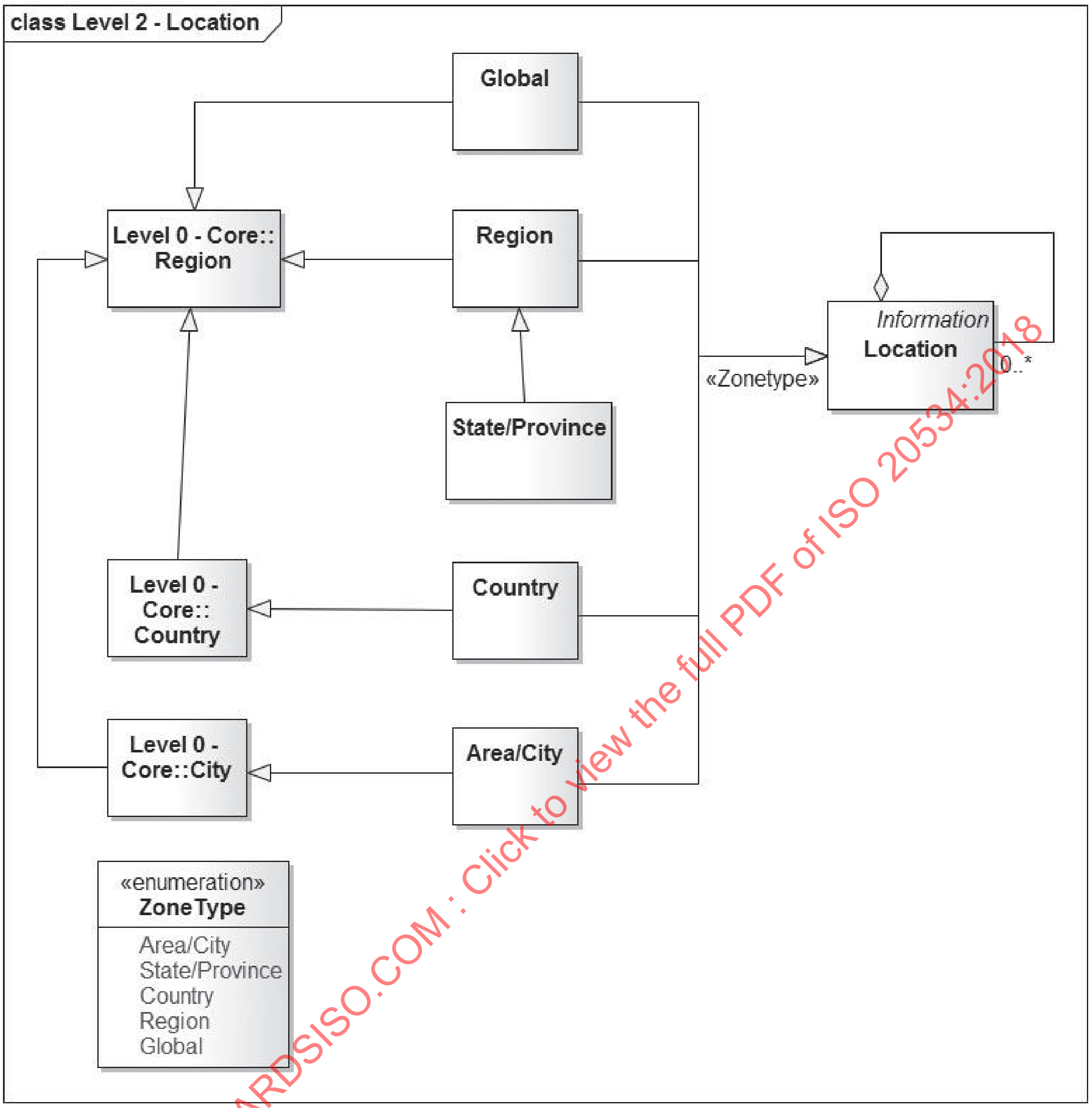


Figure 8 — UML class diagram depicting the representation of location

7.2 Level 2 formal semantics — Context

7.2.1 Context — Designed systems

KFL notation for the level 2 designed systems context:

:Ctx 2DSCtx

:Inst UserContext

:supCtx ../1SYSCtx

7.3 Level 2 formal semantics — Properties

7.3.1 Property — Network

KFL notation for network:

:Prop Network

:Inst Type

:sup Scenario

:rem "arrangement of nodes and interconnecting branches. The nodes are system functions and gateways. The branches are the flows of inputs and outputs from system functions and gateways”

7.3.2 Property — Product

KFL notation for product:

:Prop Product

:Inst Type

:sup Role

:rem “a role played by an entity (physical product) or service (actor) or some combination of both that is developed to be sold”

7.3.3 Property — PhysicalProduct

:Prop PhysicalProduct

:Inst Type

:sup 2DSCtx.Product

:rem "A Physical Product is an Entity that is developed to be sold.

7.3.4 Property — Physical product with service

KFL notation for product service:

:Prop PhysicalProductWithService

:Inst Type

:sup Role

:sup 2DSCtx.PhysicalProduct

: rem “a physical product that incorporates a service”

7.3.5 Property — Service

KFL notation for service product:

:Prop Service

:Inst Type

:sup Actor

:rem “activity sold as a product”

7.3.6 Property — Service using physical product

:Prop ServiceUsingPhysicalProduct

:Inst ActorType

:sup Actor

:rem "A service that utilizes a physical product."

7.3.7 Propery — Prototype

:Prop Prototype

:Inst Type

:sup Material

:name "Prototype"

:rem "Pre-production model of a product."

7.3.8 Property — Organization function

KFL notation for organization function:

:Prop OrganizationFunction

:Inst Type

:sup SystemFunction

:rem “The key function(s) that an enterprise provides”

(disjointSubProps OrganizationFunction)

7.3.9 Property — Facility function

KFL notation for facility function:

:Prop FacilityFunction

:Inst Type

:sup SystemFunction

:rem “a system that performs one or more specific activities by the provision of resources in a given global location”

(disjointSubProps FacilityFunction)

7.3.10 Property — Actor type

;;;metaproperty - needed to refer to types of actor in the ontology

:Prop ActorType

:Inst MetaProperty

:sup Type

:name "Type of the Actor"

:metaPropFor Actor

7.3.11 Property — Actor

KFL notation for actor:

:Prop Actor

:Inst Type

:sup Role

:rem “actor is a role which is played by a systemfunction”

(disjointSubProps Actor)

7.3.12 Property — Customer

KFL notation for customer:

:Prop Customer

:Inst ActorType

:sup Actor

:rem “organization or person that receives a product”

7.3.13 Property — Supplier

KFL notation for supplier:

:Prop Supplier

:Inst ActorType

:sup Actor

:rem “facility in an organization that enters into an agreement with the acquirer for the supply of a product“

7.3.14 Property — Gateway

KFL notation for gateway:

:Prop Gateway

:Inst Type

:sup Basic

:rem "Gateways are used to control how Sequence Flows interact as they converge and diverge within a network "

7.3.15 Property — Diverging gateway

KFL notation for diverging gateway:

:Prop DivergingGateway

:Inst Type

:sup Gateway

:rem " A diverging gateway has only one input and two or more outputs (describes an opening AND or a fork)"

7.3.16 Property — Converging gateway

KFL notation for converging gateway:

:Prop ConvergingGateway

:Inst Type

:sup Gateway

:rem "A converging gateway has two or more inputs and only one output (describes a closing AND or a join)"

7.3.17 Property — Inclusive diverging gateway

KFL notation for inclusive diverging gateway:

:Prop InclusiveDivergingGateway

:Inst Type

:sup DivergingGateway

:rem "A inclusive diverging gateway (opening OR) has one input and two or more outputs. An output shall have a condition (a Boolean)".

7.3.18 Property — Inclusive converging gateway

KFL notation for inclusive converging gateway:

:Prop InclusiveConvergingGateway

:Inst Type

:sup ConvergingGateway

:rem " An inclusive converging gateway ("closing OR") has one default output and two or more inputs. An input shall have a condition (a Boolean). "

7.3.19 Property — Exclusive diverging gateway

KFL notation for exclusive diverging gateway:

:Prop ExclusiveDivergingGateway

:Inst Type

:sup InclusiveDivergingGateway

:rem "An exclusive diverging gateway ("opening XOR", branch) inherits from an inclusive diverging gateway. The value of the condition of an output shall not be identical to the value of the condition of any of the other outputs of the XOR gateway (i.e. only one condition within the XOR gateway can be activated, so only one branch can be taken)."

7.3.20 Property — Exclusive converging gateway

KFL notation for exclusive converging gateway:

:Prop ExclusiveConvergingGateway

:Inst Type

:sup ExclusiveDivergingGateway

:rem "An exclusive converging gateway ("closing XOR", merge) inherits from an inclusive diverging gateway. The value of the condition of an input shall not be identical to the value of the condition of any of the other inputs of the XOR gateway (i.e. only one condition within the XOR gateway can be activated, so only one incoming flow is needed)."

7.3.21 Property — Condition

KFL notation for condition:

:Prop Condition

:Inst Type

:sup MLO.Object

:rem "Boolean value as a result of a condition. "

7.3.22 Property — Business event

KFL notation for business event:

:Prop BusinessEvent

:Inst Type

:sup Basic

:rem "Event that occurs in the business domain."

7.3.23 Property — Start event

KFL notation for start event:

:Prop StartEvent

:Inst Type

:sup BusinessEvent

:rem "Start of a process sequence."

7.3.24 Property — End event

KFL notation for end event:

:Prop EndEvent

:Inst Type

:sup BusinessEvent

:rem "End of a process sequence."

7.3.25 Property — Decision event

KFL notation for decision event:

:Prop DecisionEvent

:Inst Type

:sup BusinessEvent

:rem "An event where decisions are made or taken."

7.3.26 Property — Plan

KFL notation for project:

:Prop Plan

:Inst Type

:sup Information

:sup MLO.Object

:name "Plan"

:rem "Account of intended future course of action aimed at achieving specific goal(s) or objective(s) within a specific timeframe”

7.3.27 Property — Project

KFL notation for project:

:Prop Project

:Inst Type

:sup SystemFunction

:name "Project"

:rem “A planned undertaking with specified targets”

7.3.28 Property — Location

KFL notation for location:

:Prop Location

:Inst Type

:sup Information

:sup MLO.Object

:name "Location"

:rem "A site or position"

7.3.29 Property — Zonetype

KFL notation for zonetype:

:Prop ZoneType

:Inst MetaProperty

:sup Type

:name "Measure of Position in Geographical Hierarchy"

:metaPropFor Location

7.3.30 Property — Area/city

KFL notation for area/city:

:Prop {Area/City}

:Inst ZoneType

:sup Location

:sup MLO.City

:rem "urban areas"

7.3.31 Property — State/province

KFL notation for location:

:Prop {State/Province}

:Inst ZoneType

:sup Region

:rem "one of many areas into which some countries are divided"

7.3.32 Property — Country

KFL notation for country:

:Prop Country

:Inst ZoneType

:sup Location

:sup MLO.Country

:rem "The territory of a nation”

7.3.33 Property — Region

KFL notation for region:

:Prop Region

:Inst ZoneType

:sup Location

:sup MLO.Region

:rem "A subdivision of the earth"

7.3.34 Property — Global

KFL notation for global:

:Prop Global

:Inst ZoneType

:sup Location

:sup MLO.Region

:rem "Of, relating to, or involving the whole world, worldwide."

7.4 Level 2 formal semantics — Relationships

7.4.1 Relationship — Plays role actor

KFL notation for the relationship plays role actor:

:Rel playsRoleActor

:Inst TernaryRel

:Inst NonRigidRel

:supRel playsRole

:Sig SystemFunction Actor Scenario

:Args " SystemFunction " "Actor" "Scenario"

:lex "System entity ?1 plays Actor Role ?2 in Scenario ?3"

:rem "A SystemFunction plays the role of a Actor in a Scenario. An Actor is played by a <sym>1SYSCtx. SystemFunction</sym> which is a subtype of <sym>1SYSCtx.Basic </sym>."

7.4.2 Relationship — Plays role Service

KFL notation for the relationship plays role service:

:Rel playsRoleService

:Inst TernaryRel

:Inst NonRigidRel

:supRel playsRoleActor

:Sig SystemFunction Service Scenario

:Args " SystemFunction " "Service" "Scenario"

:lex "System entity ?1 plays Service Role ?2 in Scenario ?3"

:rem "A SystemFunction plays the role of a Service in a Scenario. A Service is played by a <sym>1SYSCtx. SystemFunction</sym> which is a subtype of <sym>1SYSCtx. Basic </sym>."

7.4.3 Relationship — Plays role physical product

KFL notation for the relationship plays role physical product:

:Rel playsRolePhysProd

:Inst TernaryRel

:Inst NonRigidRel

:supRel playsRole

:Sig Entity PhysicalProduct Scenario

:Args " Entitiy" "PhysicalProduct" "Scenario"

:lex "Entity ?1 plays Physical Product Role ?2 in Scenario ?3"

:rem "An Entity plays the role of a PhysicalProduct in a Scenario. An PhysicalProduct role is played by a <sym>1SYSCtx.Entity</sym> which is a subtype of <sym>1SYSCtx.Basic </sym>."

7.4.4 Relationship — Flow occurs from an output to an input

KFL notation for the relationship flow occurs from an output to an input:

:Rel flow

:Inst TernaryRel

:Inst RigidRel

:Sig Role Role Scenario

:Args "Source" "Target" "Scenario"

:lex "Flow from Source ?1 to Target ?2 in Scenario ?3"

:rem "A Flow occurs from an <sym>1SYSCtx.Input</sym> to an <sym>1SYSCtx.Output</sym> or from an <sym>1SYSCtx.Output</sym> to an <sym>1SYSCtx.Input</sym> within a given <sym>1SYSCtx.Scenario</sym>."

7.4.5 Relationship — Gateway contains role

KFL notation for the relationship gateway contains role:

:Rel gatewayContainsRole

:Inst BinaryRel

:Inst RigidRel

:Sig Gateway Role

:Args "Gateway" "Role"

:lex "Gateway ?1 contains Role ?1"

7.4.6 Relationship — Start event has an output role

KFL notation for the relationship start event has an output role:

:Rel startEventRel

:Inst BinaryRel

:Inst RigidRel

:Sig StartEvent Output

:Args "StartEvent" "Output"

:lex "Start Event ?1 has an output role ?2"

7.4.7 Relationship — End event has an input role

KFL notation for the relationship end event has an input role:

:Rel endEventRel

:Inst BinaryRel

:Inst RigidRel

:Sig EndEvent Input

:Args "EndEvent" "Input"

:lex "End Event ?1 has an input role ?2"

7.4.8 Relationship — Define a project

KFL notation for the relationship define a project:

:Rel defProject

:Inst BinaryRel

:Inst RigidRel

:Sig String Project

:Args "String" "Project"

:lex "String ?1 provides a definition for Project ?2"

7.4.9 Relationship — Project contains scenario

KFL notation for the relationship project contains scenario:

:Rel projectContainsScenario

:Inst BinaryRel

:Inst RigidRel

:Sig Project Scenario

:Args "project" "scenario"

:lex "Project ?1 contains scenario ?2"

7.4.10 Relationship — Project has chosen scenario

KFL notation for the relationship project has chosen scenario:

:Rel projecthasChosenScenario

:Inst BinaryRel

:Inst RigidRel

:Sig Project Scenario

:Args "project" "scenario"

:lex "Project ?1 hasChosen scenario ?2"

:rem "A <sym>2DSCtx.Project</sym> has one chosen <sym>1SYSCtx.Scenario</sym>"

(functionalArg projecthasChosenScenario 2)

7.4.11 Relationship — Physical product with service

KFL notation for the relationship physical product contains service:

:Rel pContainsService

:Inst BinaryRel

:Inst RigidRel

:supRel roleContainsRole

:Sig PhysicalProductWithService Service

:Args "Physical Product with Service" "Service"

:lex "Physical Product with Service ?1 contains a Service ?2"

:rem "A <sym>2DSCtx.PhysicalProductWithService</sym> contains at least one <sym>2DSCtx.Service</sym>."

(totalArg pContainsService 1)

7.4.12 Relationship — Service contains physical product

KFL notation for the relationship

:Rel spContainsProduct

:Inst BinaryRel

:Inst RigidRel

:supRel roleContainsRole

:Sig ServiceUsingPhysicalProduct PhysicalProduct

:Args "ServiceUsingPhysicalProduct" "PhysicalProduct"

:lex "Service ?1 contains a Physical Product ?2"

:rem "A <sym>2DSCtx.ServiceUsingPhysicalProduct</sym> contains at least one <sym>2DSCtx.PhysicalProduct</sym>."

(totalArg spContainsProduct 1)

7.4.13 Relationship — Physical product has a prototype

:Rel hasAPrototype

:Inst BinaryRel

:Inst RigidRel

:Sig PhysicalProduct Prototype

:Args "PhysicalProduct" "Prototype"

:lex "PhysicalProduct ?1 has a Prototype ?2"

:rem "A <sym>2DSCtx.PhysicalProduct</sym> has a <sym>2DSCtx.Prototype</sym>."

7.4.14 Relationship — Organization function is composed of facility functions

KFL notation for the relationship

:Rel organizationFnComposedOfFacilityFn

:supRel basicContainsBasic

:Inst BinaryRel

:Inst RigidRel

:Sig OrganizationFunction FacilityFunction

:Args "organizationFn" "facilityFn"

:lex "?1 is composedOf ?2"

:rem "An organization function is composed of facility functions."

7.4.15 Relationship — System function location

KFL notation for the relationship

:Rel systemfunctionLocation

:Inst RootCtx.QuaternaryRel

:Inst NonRigidRel

:Sig SystemFunction Coordinates {Area/City} 2DSCtx.Country

:Args "SystemFunction" "Location" "City" "Country"

:lex "?1 Location ?2"

:rem "Relation to describe that a SystemFunction is performed at a Location."

7.4.16 Relationship — Facility function location

KFL notation for the relationship

:Rel facilityfunctionLocation

:Inst RootCtx.QuaternaryRel

:Inst NonRigidRel

:supRel systemfunctionLocation

:Sig Facility Coordinates {Area/City} 2DSCtx.Country

:Args "Facility" "Location" "City" "Country"

:lex "?1 Location ?2"

:rem "Relation to describe that a FacilityFunction is performed at a Location."

7.5 Level 2 formal semantics — Axioms

7.5.1 Axiom — A system function cannot both play a role and contain the same role

ECLIF notation:

```
(=> (and (SystemFunction ?systemfunction)
 (playsRole ?systemfunction ?role ?scenario)
   )
   (not(activityContainsRole ?systemfunction ?role))
)
```

:IC hard "The same SystemFunction cannot contain a Role and play the Role”

7.5.2 Axiom — A network shall contain a flow between two system functions

ECLIF notation for a network shall contain a flow between two system functions:

```
(=> (Network ?net)
      (exists (?source ?target ?system1 ?system2)
                    (and
                           (SystemFunction ?system1)
                                 (SystemFunction ?system2)
                                 (activityContainsRole ?system1 ?source)
                                 (activityContainsRole?system2 ?target)
                                 (/= ?system1 ?system2)
                                 (flow ?source ?target ?net))
      )
)
```

:IC hard "A network shall contain a flow between two system functions"

7.5.3 Axiom — A flow can only exist between an input and an output or an output and an input

ECLIF notation for a flow can only exist between an input and an output or an output and an input:

```
(=> (flow ?source ?target ?scenario)
      (or (and (Input ?source)
            (Output ?target))
            (and (Output ?source)
                (Input ?target))
        )
)
```

:IC hard "A flow can only exist between an Input and an Output or an Output and an Input. This does not occur in the relation (flow ?source ?target ?scenario)."

7.5.4 Axiom — A flow can only exist from a target to a source or a source to a target but not in both directions

ECLIF notation for a flow can only exist from a target to a source or a source to a target but not in both directions:

```
(=> (flow ?source1 ?target1 ?scenario1)
        (exists (?source2 ?target2 ?scenario2)
        (and (= ?source1 ?source2)
              (= ?target1 ?target2)
              (= ?scenario1 ?scenario2)
              (not (flow ?target2 ?source2 ?scenario2) ) ) )
)
```

:IC hard "A flow can only exist from a target to a source or a source to a target but not in both directions (XOR). There is flow from ?source1 to ?target1 in Scenario ?scenario1 for which a reverse flow exists."

7.5.5 Axiom — In a flow relation the source basic shall flow to a target

ECLIF notation for in a flow relation the source basic shall flow to a target:

```
(=> (and (flow ?source ?target ?scenario)
        (playsRole ?basic ?source ?scenario))
    (playsRole ?basic ?target ?scenario)
)
```

:IC hard "In a flow relation the source basic shall flow to a target."

7.5.6 Axiom — A basic playing the role of an output in an network shall play the role of an input

ECLIF notation for a basic playing the role of an output in a network shall play the role of an input to another entity (This axiom is needed to define the network connectivity):

```
(=> (and (Basic ?basic)
                    (Output ?outputRole)
                    (SystemFunction ?sOutput)
                    (Network ?net)
                    (playsRole ?basic ?outputRole ?net)
```

```
                (activityContainsRole ?sOutput ?outputRole)

        )

            (exists (?inputRole ?sInput)

                (and (Input ?inputRole)

                    (SystemFunction ?sInput)

                                (/= ?sInput ?sOutput)

                                (playsRole ?basic ?inputRole ?net)

                                (activityContainsRole ?sInput ?inputRole)

                )))
```

;:IC hard "A Basic playing the role of an Output from a SystemFunction shall play the role of an Input to a different SystemFunction within the Network."

7.5.7 Axiom — A start event is a specialized type of basic which shall have an output role only

ECLIF notation for a start event is a specialized type of basic which shall have an output role only:

```
(=> (StartEvent ?startEvent)

    (and

        (Output ?outputRole)

        (count (startEventRel ?startEventRel ?outputRole) ?nRels)

        (= ?nRels 1)))
```

:IC hard "A Start Event is a specialized type of Basic which has an output role only (which is played by the trigger for the network)."

7.5.8 Axiom — An end event is a specialized type of basic which shall have an input role only

ECLIF notation for an end event is a specialized type of basic which shall have an input role only:

```
(=> (EndEvent ?endEvent)

     (and

        (Input ?inputRole)

        (count (endEventRel ?endEvent ?inputRole) ?nRels)

        (= ?nRels 1)))
```

:IC hard "An End Event is a specialized type of Basic which has an input role only."

7.5.9 Axiom — A gateway shall have at least one input and one output

ECLIF notation for a gateway shall have at least one input and one output:

```
(=> (Gateway ?g)

     (exists (?inputRole ?outputRole)

       (and
            (Input ?inputRole)
            (Output ?outputRole)
            (gatewayContainsRole ?g ?inputRole)
            (gatewayContainsRole ?g ?outputRole))))
:IC hard "A gateway shall have one input and one output"
```

7.5.10 Axiom — A diverging gateway shall have only one input and two or more outputs

ECLIF notation for a diverging gateway shall have only one input and two or more outputs:

```
(=> (DivergingGateway ?dg)
    ;(exists (?inputRole ?outputRole)
      (and
            (Input ?inputRole)
            (Output ?outputRole)
            (count (gatewayContainsRole ?dg ?inputRole) ?nin)
            (count (gatewayContainsRole ?dg ?outputRole) ?nout)
            (= ?nin 1)
            (RootCtx.lteNum 2 ?nout))
:IC hard "A diverging gateway has only one input and two or more outputs (describes an opening AND or a fork)"
```

7.5.11 Axiom — A converging gateway shall have has two or more inputs and only one output

ECLIF notation for a converging gateway shall have has two or more inputs and only one output:

```
(=>(ConvergingGateway ?dg)
    ;(exists (?inputRole ?outputRole)
      (and
            (Input ?in)
            (Output ?out)
            (count (gatewayContainsRole ?dg ?inputRole) ?nin)
            (count (gatewayContainsRole ?dg ?outputRole) ?nout)
            (= ?nout 1)
            (RootCtx.lteNum 2 ?nin))
:IC hard "A converging gateway has two or more inputs and only one output (describes an closing AND or a join)"
```

7.5.12 Axiom — An inclusive diverging gateway has one input and two or more outputs

ECLIF notation for an inclusive diverging gateway has one input and two or more outputs:

```
(=>(InclusiveDivergingGateway ?idg)
    (and
                (Condition ?outCondition)
                (OutputHasCondition ?outputRole ?outCondition)))
```

:IC hard "An inclusive diverging gateway (opening OR) has one input and two or more outputs. An output shall have a condition (a Boolean)".

7.5.13 Axiom — An inclusive converging gateway has one default output and two or more inputs

ECLIF notation for an inclusive converging gateway has one default output and two or more inputs:

```
(=>(InclusiveConvergingGateway ?icg)
    (and
                (Condition ?inCondition)
                (InputHasCondition ?inputRole ?inCondition)))
```

:IC hard "An inclusive converging gateway (“closing OR”) has one default output and two or more inputs. An input shall have a condition (a Boolean). "

7.5.14 Axiom — An exclusive diverging gateway inherits from an inclusive diverging gateway

ECLIF notation for an exclusive diverging gateway inherits from an inclusive diverging gateway:

```
(=> (and (ExclusiveDivergingGateway ?edg)
          (Output ?outputRole)
          (Output ?outputRole1)
          (/= ?outputRole ?outputRole1)
          (gatewayContainsRole ?edg ?outputRole)
          (gatewayContainsRole ?edg ?outputRole1))
    (not (exists (?outCondition)
           (and (Condition ?outCondition)
                 (OutputHasCondition ?outputRole ?outCondition)
                 (OutputHasCondition ?outputRole1 ?outCondition)))))
```

:IC hard "An exclusive diverging gateway (“opening XOR”, branch) inherits from an inclusive diverging gateway. The value of the condition of an output shall not be identical to the value of the condition of any of the other outputs of the XOR gateway (i.e. only one condition within the XOR gateway can activated, so only one branch can be taken). "

7.5.15 Axiom — An exclusive converging gateway inherits from an inclusive diverging gateway

ECLIF notation for an exclusive converging gateway inherits from an inclusive diverging gateway:

```
(=> (and (ExclusiveConvergingGateway ?ecg)
        (Input ?inputRole)
        (Input ?inputRole1)
        (/= ?inputRole ?inputRole1)
        (gatewayContainsRole ?ecg ?inputRole)
        (gatewayContainsRole ?ecg ?inputRole1))
   (not (exists (?inputRole1)
            (and
                 (Condition ?inCondition)
                 (InputHasCondition ?inputRole ?inCondition)
                 (InputHasCondition ?inputRole1 ?inCondition)))))
```

:IC hard "An exclusive converging gateway ("closing XOR", merge) inherits from an inclusive diverging gateway. The value of the condition of an input shall not be identical to the value of the condition of any of the other inputs of the XOR gateway (i.e. only one condition within the XOR gateway can activated, so only one incoming flow is needed)."

7.5.16 Level 2 formal semantics — Rules

ECLIF notation for if a systemfunction contains another systemfunction, then the child systemfucntion belongs to the same Network as the parent system.

```
(=> (and (SystemFunction ?x)
   (SystemFunction ?z)
      (basicContainsBasic ?z ?x)
      (Network ?net)
      (basicContainsBasic ?net ?z))
      (basicContainsBasic ?net ?x))
```

8 Manufacturing business systems formal semantics — Level 3

8.1 Overview

Level 3 of the reference ontology is different to levels 1, 2, and 4. It is the level used to define the context for the particular business specific concept specializations that follow at level 4. The context specified here makes it possible to distinguish between the set of concepts related to manufacturing businesses as opposed to those used in healthcare or banking systems. Many other business areas can also have a context included at this level, such as shipping, oil and gas, retail and farming. The sort of concepts that best illustrate the context distinction for manufacturing businesses are possibly the operational process related concepts such as machining, casting, moulding, welding, assembly and their related resources, which would not be important concepts for other areas of business. Nonetheless these operational processes are outside the scope of this document.

8.2 Level 3 formal semantics — Context

KFL notation for the level 3 manufacturing business systems context:

:Ctx 3MBSCtx

:Inst UserContext

:supCtx ../2DSCtx

9 Global production systems network formal semantics — Level 4

9.1 Overview

Level 4 of the ontology utilizes the concepts developed at level 2 and further specializes them within the context of manufacturing business systems at level 3. The specialization of Scenario down to the definition of Global Production Network (GPN) is illustrated in Figure 9.

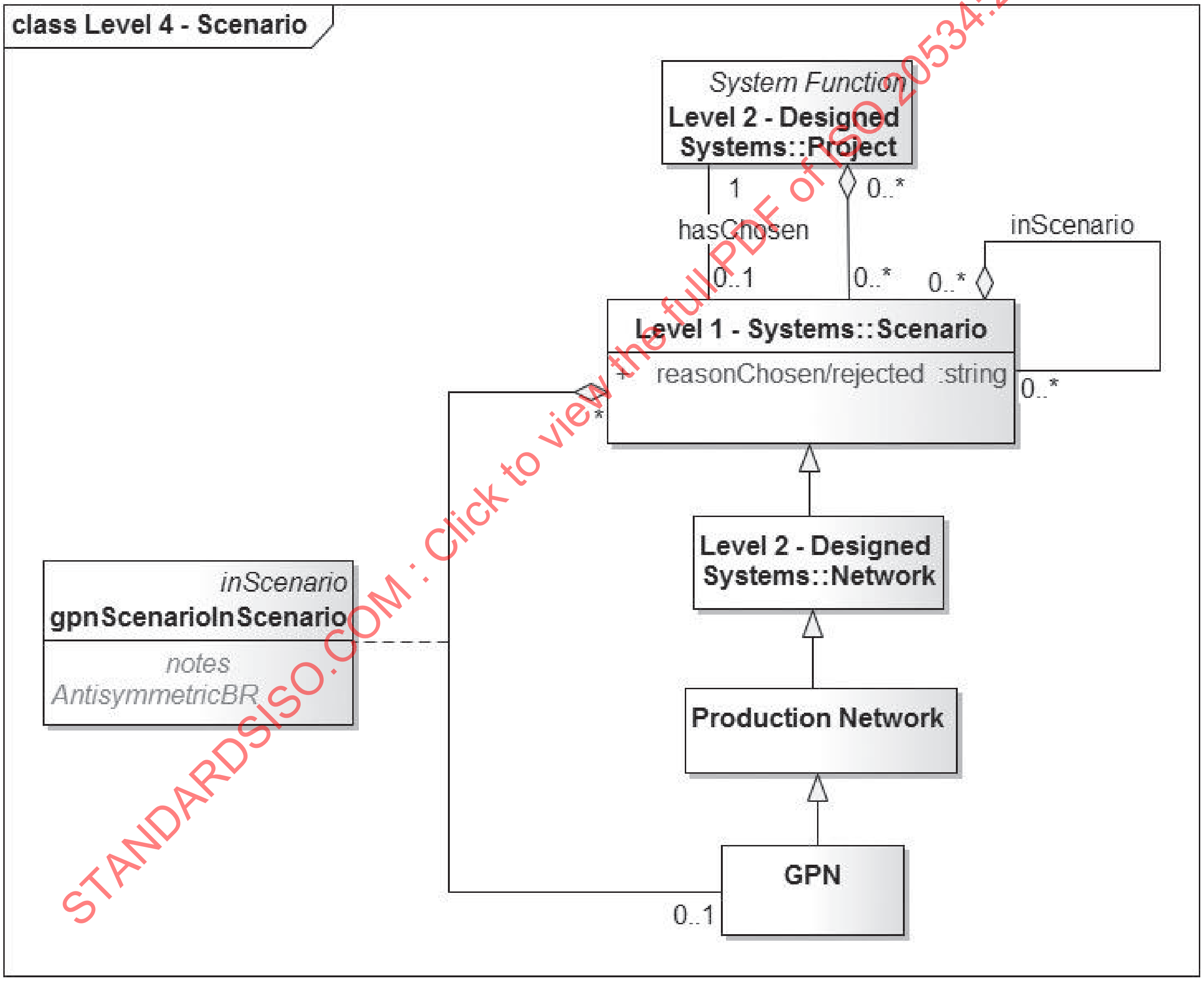


Figure 9 — UML class diagram depicting the representation of scenario at level 4

At Level 4, Scenario is specialized, into those concepts that are relevant to production. These being Production Network, GPN, and GPNScenarioinScenario. The Production Network concept provides a view of any production network, inheriting from level 2 Network. GPN is then a global specialization of a Production Network that requires its related facilities to be drawn from more than one country.

In addition the “Project” concept is included here as a way of capturing scenarios that relate to a specific planned undertaking. In this document, the focus is on Global Production Networks but Project can also potentially encompass Scenarios related to, for example, the product under development, the business model to be chosen and the risk involved in alternative scenario possibilities.

9.2 Level 4 formal semantics — Context

KFL notation for the level 4 product service production context:

:Ctx 4PSPCtx

:Inst UserContext

:supCtx ../4PSLSCtx

9.3 Level 4 formal semantics — Properties

9.3.1 Property — Production network

KFL notation for production network:

:Prop ProductionNetwork

:Inst Type

:sup 2DSCtx.Network

:name "Production Network"

9.3.2 Property — GPN

KFL notation for gpn:

:Prop GPN

:Inst Type

:sup ProductionNetwork

:rem "A GPN is a global specialization of a </sym>ProductionNetwork</sym>."

9.3.3 Property — Producer

KFL notation for producer:

:Prop Producer

:Inst ActorType

:sup Actor

:rem "a business enterprise that produces goods or services for sale "

9.3.4 Property — Manufacturer

KFL notation for manufacturer:

:Prop Manufacturer

:Inst ActorType

:sup Actor

:rem "a business enterprise that is involved in the full product life cycle i.e. product design, production, operation and end-of-life."

9.3.5 Property — Manufactured product

KFL notation for manufactured product:

:Prop ManufacturedProduct

:Inst Type

:sup PhysicalProduct

:rem "A <sym>2DSCtx.Product</sym> that exploits/consumes a raw material."

9.3.6 Property — Manufactured product service

KFL notation for manufactured product service

:Prop ManufacturedProductService

:Inst Type

:sup ManufacturedProduct

:rem "A specialization of <sym>ManufacturedProduct</sym> within the Level 4 Product Service Production domain."

9.3.7 Property — BOM

:Prop BOM

:Inst Type

:sup Plan

:name "Bill of Materials"

:rem "BOM lists the components required to build a <sym>2DSCtx.Product</sym> as well as information related to these components."

9.4 Level 4 formal semantics — Relationships

9.4.1 Relationship — gpn in scenario

KFL notation for the relationship gpn in scenario. Only one GPN can be contained within a Scenario:

:Rel gpnScenarioInScenario

:Inst BinaryRel

:Inst RigidRel

:supRel inScenario

:Inst AntisymmetricBR

:Sig GPN Scenario

:Args "component GPN" "compound Scenario"

:lex "GPN ?1 is contained within Scenario ?2"

:rem "Only one GPN can be contained within a Scenario."

(functionalArg gpnScenarioInScenario 1)

9.4.2 Relationship — Manufacturer produces manufactured product

KFL notation for the relationship manufacturer produces manufactured product.

:Rel ManufacturerProduceManufacturedProduct

:Inst TernaryRel

:Inst NonRigidRel

:Sig Manufacturer ManufacturedProduct Scenario

:Args "Manufacturer" "ManufacturedProduct" "Scenario"

:lex "Manufacturer ?1 produce ManufacturedProduct?2 in Scenario ?3"

:rem "A Manufacturer produces a ManufacturedProduct."

9.4.3 Relationship — Manufactured product has a bill of materials

:Rel hasABOM

:Inst BinaryRel

:Sig ManufacturedProduct BOM

:Args "ManufacturedProduct" "Bill of Material"

:lex "ManufacturedProduct ?1 has a BOM ?2"

:rem "A <sym>4PSPCtx.ManufacturedProduct</sym> has at most one <sym>4PSPCtx.BOM</sym>."

9.4.4 Relationship — Manufactured product service contains service

KFL notation for the relationship manufactured product service contains service.

:Rel mpContainsService

:Inst BinaryRel

:Inst RigidRel

:supRel 2DSCtx.pContainsService

:Sig ManufacturedProductService Service

:Args "ManufacturedProductService" "Service"

:lex "A Manufactured Product Service ?1 contains a Service ?2"

:rem "A <sym>4PSPCtx.ManufacturedProductService</sym> contains at least one <sym>2DSCtx.Service</sym>."

;;;inherits cardinality constraints from level 2 relation

9.5 Level 4 formal semantics — Axioms

9.5.1 Axiom — The role producer shall be played in a production network

ECLIF notation for the role producer shall be played in a production network:

```
(=> (ProductionNetwork ?pNetwork)
    (exists (?producerRole ?player)
        (and (4PSPCtx.Producer ?producerRole)
            (playsRole ?player ?producerRole ?pNetwork)
        )
    )
)
;IC hard "The Role Producer shall be played in a Production Network."
```

9.5.2 Axiom — The role product shall be played in production network

ECLIF notation for the role product shall be played in a production network:

```
(=> (ProductionNetwork ?pNetwork)
    (exists (?productRole ?player)
        (and (2DSCtx.Product ?productRole)
            (playsRole ?player ?productRole ? pNetwork)
        )
    )
)
;IC hard "The Role Product shall be played in GPN Scenario but in ?gpn this is not the case."
```

9.5.3 Axiom — The role supplier should be played in a production network

ECLIF notation for the role supplier should be played in a production network scenario:

```
(=> (ProductionNetwork ?pNetwork)
    (exists (?supplierRole ?player)
        (and (2DSCtx.Supplier ?supplierRole)
                playsRole ?player ?supplierRole ?pNetwork)
        )
    )
)
:IC soft "The role Supplier should be played in a Production Network."
```

9.5.4 Axiom — A supplier shall have an output

ECLIF notation for a supplier shall have an output:

```
(=> (and (2DSCtx.FacilityFunction ?facility)
    (2DSCtx.Supplier ?supplierRole)
    (playsRole ?facility ?supplierRole ?pNetwork)
    )
    (exists (?output ?basic)
            (and (Output ?output)
              (activityContainsRole ?facility ?output)
              (playsRole ?basic ?output ?pNetwork)
      )
   )
)
```

;:IC hard "A Supplier shall have an output. Facility ?facility plays the role of a Supplier ?supplierRole but does not have an output Role which is played in ProductionNetwork Scenario ?pNetworkS."

9.5.5 Axiom — A producer shall have an output

ECLIF notation for a producer shall have an output:

```
(=> (and (2DSCtx.FacilityFunction ? facility)
    (4PSPCtx.Producer ?producerRole)
    (playsRole ?facility ?producerRole ?pNetwork)
   )
    (exists (?output ?basic)
                (and (Output ?output)
                    (activityContainsRole ?facility ?output)
                    (playsRole ?basic ?output ?pNetwork)
                )
   )
)
```

;:IC hard "A Producer shall have an output."

9.5.6 Axiom — A customer shall have an input

ECLIF notation for a customer shall have an input:

```
(=> (and (2DSCtx.FacilityFunction ?facility)
    (2DSCtx.Customer ?customerRole)
    (playsRole ?facility ?customerRole ?pNetwork)
    )
    (exists (?input ?basic)
              (and (Input ?input)
                     (activityContainsRole ?facility ?input)
                     (playsRole ?basic ?input ?pNetwork)
              )
   )
)
;:IC hard "A Customer shall have an input."
```

9.5.7 Axiom — A facility in a GPN shall have a location/city/country

ECLIF notation for a facility in a GPN shall have a location/city/country:

```
(=> (and (FacilityFunction ?facility)
(GPN ?gpn)
(playsRole ?facility ?role ?gpn)
)
(exists (?location ?city ?country)
(facilityfunctionLocation ?facility ?location ?city ?country)
)
)
:IC hard "A facility located in a GPN shall have a location/city/country but Facility ?facility does not."
```

9.5.8 Axiom — A GPN shall have facilities located in more than one country

ECLIF notation for a GPN shall have facilities located in more than one country:

```
(=> (and (FacilityFunction ?facility1)
(GPN ?gpn1)
(playsRole ?facility1 ?role ?gpn1)
(facilityfunctionLocation ?facility1 ?location ?city ?country1)
)
(exists (?gpn2 ?facility2 ?location ?city ?country2)
(and (GPN ?gpn2)
(playsRole ?facility2 ?role ?gpn2)
          (facilityfunctionLocation ?facility2 ?location ?city ?country2)
(= ?gpn1 ?gpn2)
(/=?facility1 ?facility2)
(/= ?country1 ?country2)
)
    )
)
:IC hard "a GPN shall have Facility functions located in more than one country but GPN ?gpn1 does not."
```

Annex A
(informative)

Highfleet Ontology Library Reference

A.1 General

This annex contains information about the capabilities of the Highfleet system for ontology design, including taxonomy, time, events, aggregation, and lists, as well as an explanation of the ontological commitments made in the Highfleet Upper Level Ontology (ULO) and ontology modules.

A.2 Upper Level Ontology (ULO)

A.2.1 General

The Highfleet Upper Level Ontology (ULO) defines a small number of properties (classes), relations, and axioms. The ULO is intended to be a small ontology expanded by ontologists on a per-project basis, as opposed to being a single large upper level intended for re-use on every project.

The ULO distinguishes abstract and concrete entities on a spatial basis. ***Concrete entities*** are entities capable of being located as well as being locations. ***Abstract entities*** are entities that can neither be located nor be locations. The *locatedIn* relation provides the primary means for this distinction.

Concrete entities are either ***objects*** or ***events***, events being things that occur, or perdure, through time, and objects being those things that do not occur or perdure. ***Groups*** are special objects that can have members. The *memberOf* relation is used to provide the semantics for groups.

The theory of events in the ULO is drawn from Donald Davidson's token-theory of events. Events are the only things in the ULO that can have durations (*hasDuration*) and participants (*participant*), and have a suite of *event relations* to help define their behaviour. Participation can mean different things for different kinds of events, so part of defining a type of event is defining what various event relations mean for that type.

Finally, a mereology is defined in the ULO, centring on the *partOf* relation. This mereology defines a partial ordering on *particulars*. A related strict partial ordering (*properPartOf*) is defined as well.

A.2.2 Top

Those things which exist are instances of Top, be it in an abstract, spatial, fictional, or other way.

:Prop Top

:Inst Property

A.2.3 Particular

Particulars are those things that are unique insofar as nothing else is the same thing as they are - particulars are only identical with themselves.

:Prop Particular

:Inst Category

:Inst IntensionalRel

:Inst NonLogicalFunctor

:Mode g

:name "Particular"

:lex "?1 is a particular"

:sup Top

A.2.4 Concrete Entity

A Particular that can be *locatedIn* some place.

:Prop ConcreteEntity

:Inst Type

:sup Particular

:name "Concrete Entity"

:lex "?1 is a concrete entity"

A.2.5 Abstract Entity

A Particular that cannot be *locatedIn* some place.

:Prop AbstractEntity

:Inst Type

:Inst NonLogicalFunctor

:sup Particular

:name "Abstract Entity"

:lex "?1 is an abstract entity"

A.2.6 Location

Concrete entities are particulars that can be given a location, as opposed to abstract entities, which cannot. A concrete entity is *locatedIn* another when the boundaries of the first are fully contained in (thus not identical to) the boundaries of the second. Definitions of boundaries and boundary-containment are undefined primitives that may be defined by extensions to the ULO. This definition of location is meant to cover cases such as a person's heart being located in their body, Philadelphia being located in Pennsylvania, and John being located in the 4th floor of a building at a particular time.

A Concrete entity X, is located in Y when X's boundaries are fully contained within Y's. Because this is a basic relation, the upper level ontology does not define boundaries and containedness.

A.2.7 Occurrence

Events are particulars that have a duration; at no particular instant is the entirety of the event fully realized. For example, at any point during a football game the entirety of the game is not at-hand; there are still minutes on the clock and parts of the game have already occurred in the past. In other words, the football game is constantly unfolding over time.

In contrast, objects are fully realized at every moment they exist. Unlike a football game, a football field is fully present at points in time that it exists. It is possible to take a picture of a whole football field, but it is not possible take a picture of a whole football game: it is only possible to capture a small part, a snapshot, of it.

This parts-over-time distinction between objects and events is a standard definition amongst ontologists and many people share intuitions about which things are objects and which are events. However, there are cases when these intuitions are not shared, for example when a strong four-dimensionalist representation is proposed, the above definitions still hold up. If everything unfolds over time, then there are simply no objects around. This does not mean that there is no Object *type*, but rather that all locatable things are events. As such, the ULO is a bi-categorialist ontology that can support both 3-D and 4-D ontologies.

A.2.8 Participation

The ***event relations*** are relations defined in the ULO to help ontologists define events and their classes. The primary event relation is *participant*, which relates an event to a particular that plays some role in the event. Specialized participant relations, like *agent*, *instrument*, *object*, and *patient* help the ontologist define different ways something can participate.

The meanings of these participation relations are left intentionally ambiguous; they are intended to be re-defined for each *EventType*. For example, the agent of a sale can be defined as the seller, while the agent of an explosion can be the explosive device.

Often it is the case that only certain types of entities can participate in particular roles in a particular event type. The ***participantKind relations*** add this information to the ontology. For example, if it is decided that the *agents* of *Sale* events are the sellers, who shall necessarily be people, in this case, it can be stated that the *agentKind* of the *Sale EventType* is *Person*.

When the particular participant of a relation is not known, but its type is, a generic event relation can be used. For example, in the case of a sale of phones, where it is not known which phones were sold, only that the things that were sold were of the type *Phone*, it would be stated that the *genericObject* of the event is *Phone*.

A.2.9 Groups

The two primary relations for working with groups are *memberOf* and *subGroupOf*, which are defined to reflect the membership and subset relations of standard set theory. The *hasMemberKind* relation restricts members of a group to a particular class, just like the *participantKind* relations defined for events. A potentially unfamiliar relation is *hasIntension*, which relates a group to a class when the group includes precisely all the members of this class. Thus, if a group has an intension of *Object*, then every instance (including the group itself) is a member of the group. It follows, then, that a group can have at most one intension.

A.2.10 Identity and Equality

To express the fact that two objects are in fact the same, use the = relation. To say that two objects are different, use the /= relation. If these objects are numbers, then numerical equality (instead of object identity) is probably required.

See Number Relations.

Entity inequality can never be asserted, but can be queried: (/= John_2 John_3)

Entity equality may be asserted, but may negatively impact query performance, since assertions and rules on one entity will then have to be checked against the other: (= MarkTwain SamClemens) This can nevertheless be useful when attempting to posit a single identity for two entities previously thought to be distinct.

Equality using eqNum may not be asserted, but only inferred or queried: (eqNum 5 (numPlus 2 3))

A.3 Description of Individual Ontology Modules

A.3.1 Theory of Business Rules

Defines a theory of commercial events based on the REA ontology developed by William E. McCarthy, Department of Accounting and Information Systems, Michigan State University. It is especially concerned with those events that involve business transactions.

A.3.2 Theory of Genealogy

Defines a theory of spatial relations including support for GIS data sources. Includes the Region Connection Calculus, the 9-Intersection Calculus, and ontology to bridge between the two models.

A.3.3 Theory of Measurement

Defines a theory of measurable qualities and the appropriate mathematics for units of measure.

A.3.4 Theory of Spatial Relations

Defines a theory of spatial relations including support for GIS data sources. Includes the Region Connection Calculus, the 9-Intersection Calculus, and ontology to bridge between the two models.

A.3.5 Theory of Temporal Relations

Defines a theory of time based on the Allen Calculus for Interval-Temporal relations.

A.3.6 Theory of WordNet Relations

Defines a theory of word relations according to the WordNet English lexical database.

A.4 Built-in Data Types

A.4.1 Times

A.4.1.1 TimeInstant and TimeSpan

The ULO uses two properties to describe Time: *TimeInstant* and *TimeSpan*. TimeInstants are the smallest intervals of time and are really only used to talk about the boundaries of a time span, i.e., *startingInstant* and *endingInstant*, or potentially as a marker for the identity of an event. More commonly, *time* in the general usage, such as a date or a clock time, refers to a time span.

TimeSpans are not asserted directly, but are constructed using the functions described below.

A.4.1.2 Date and Span

There are two functions used to construct instances of *TimeSpan*: *Date* and *Span*. *Date* takes a string that specifies some date and returns a *TimeSpan* including the first and last instants of that date and all instances in between. For example, (Date "1997") denotes the *TimeSpan* for the year 1997.

The full format of a date string is

“YYYY MMM DD HH:mm:SS.sss”

where

YYYY	denotes the Gregorian calendar year;
MMM	denotes the three-letter abbreviation of the month;
DD	denotes the day of the month;
HH	denotes the (24-hour clock) hour;
mm	denotes the minute;
SS	denotes the seconds;
sss	denotes thousandths of seconds.

A date string may omit any part of format, so long as it omits all of the parts following; the result is a time interval as long as the shortest part specified.

The *Span* function takes two *TimeSpans* and returns the *TimeSpan* from the start of the first to the end of the second.

For example:

(Span (Date "1997") (Date "2000"))

denotes the span that has the first instant of (Date "1997") as its starting instant and the last instant of (Date "2000") for its ending instant.

There are two distinguished time instants representing the beginning of all time and the end of all time: *Start* and *Now*. All time instants come after *Start* (except *Start* itself) and all instances come before *Now* (except *Now* itself). Neither *Start* or *Now* are properties expected to return a date, but are instead function terms relative to all time instants instantiated in the database.

A.4.1.3 Time Zones

All specified times are interpreted as times in Greenwich Mean Time (GMT). To ensure compatibility with a particular domain, all times specified in that domain should be converted to GMT for use in an ontology if they are not already.

A.4.1.4 *holdsin*

Time intervals are related to propositions using the *holdsIn* binary relation. The first argument is a *TimeSpan*; the second argument is a clause denoting the proposition. For example:

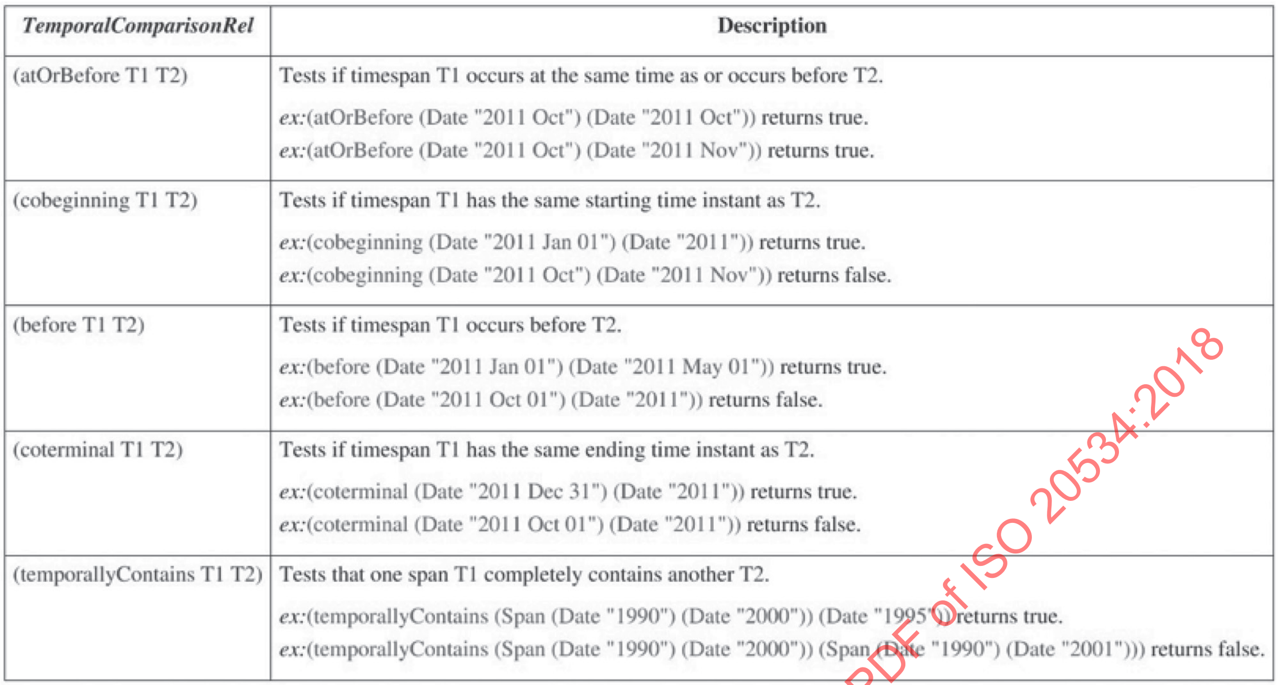
(holdsIn (Date "2002 Jan 25 15:30") (visited John Larry))

indicates that it is true on January 25, 2002 at 3:30 PM that John visited Larry.

A.4.1.5 Comparing Time Spans

The relations given in Table A.1 (instances of *TemporalComparisonRel*) are available for reasoning over time spans. For each, usage is supplied, followed by a definition and one or more examples.

Table A.1 — Temporal comparison relations

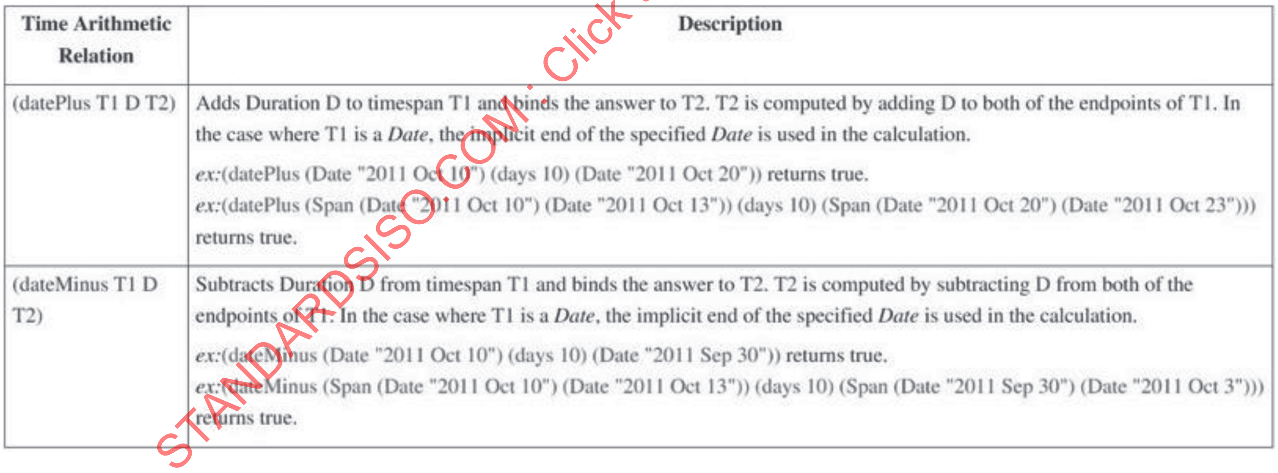


TemporalComparisonRel	Description
(atOrBefore T1 T2)	Tests if timespan T1 occurs at the same time as or occurs before T2. *ex:*(atOrBefore (Date "2011 Oct") (Date "2011 Oct")) returns true. *ex:*(atOrBefore (Date "2011 Oct") (Date "2011 Nov")) returns true.
(cobeginning T1 T2)	Tests if timespan T1 has the same starting time instant as T2. *ex:*(cobeginning (Date "2011 Jan 01") (Date "2011")) returns true. *ex:*(cobeginning (Date "2011 Oct") (Date "2011 Nov")) returns false.
(before T1 T2)	Tests if timespan T1 occurs before T2. *ex:*(before (Date "2011 Jan 01") (Date "2011 May 01")) returns true. *ex:*(before (Date "2011 Oct 01") (Date "2011")) returns false.
(coterminal T1 T2)	Tests if timespan T1 has the same ending time instant as T2. *ex:*(coterminal (Date "2011 Dec 31") (Date "2011")) returns true. *ex:*(coterminal (Date "2011 Oct 01") (Date "2011")) returns false.
(temporallyContains T1 T2)	Tests that one span T1 completely contains another T2. *ex:*(temporallyContains (Span (Date "1990") (Date "2000")) (Date "1995")) returns true. *ex:*(temporallyContains (Span (Date "1990") (Date "2000")) (Span (Date "1990") (Date "2001"))) returns false.

A.4.1.6 Span Arithmetic

The relations given in Table A.2 are available for adding or subtracting durations (*Duration*) to instances of *Time*.

Table A.2 — Time arithmetic relations



Time Arithmetic Relation	Description
(datePlus T1 D T2)	Adds Duration D to timespan T1 and binds the answer to T2. T2 is computed by adding D to both of the endpoints of T1. In the case where T1 is a *Date*, the implicit end of the specified *Date* is used in the calculation. *ex:*(datePlus (Date "2011 Oct 10") (days 10) (Date "2011 Oct 20")) returns true. *ex:*(datePlus (Span (Date "2011 Oct 10") (Date "2011 Oct 13")) (days 10) (Span (Date "2011 Oct 20") (Date "2011 Oct 23"))) returns true.
(dateMinus T1 D T2)	Subtracts Duration D from timespan T1 and binds the answer to T2. T2 is computed by subtracting D from both of the endpoints of T1. In the case where T1 is a *Date*, the implicit end of the specified *Date* is used in the calculation. *ex:*(dateMinus (Date "2011 Oct 10") (days 10) (Date "2011 Sep 30")) returns true. *ex:*(dateMinus (Span (Date "2011 Oct 10") (Date "2011 Oct 13")) (days 10) (Span (Date "2011 Sep 30") (Date "2011 Oct 3"))) returns true.

A.5 Strings

A.5.1 General

The property *String* is used to store strings. A string is a sequence of characters delimited with double-quotes. The double-quotes acts as a kind of quoting operator that denote these quoted-strings.

A.5.2 String Operations

The relations given in Table A.3 (*StringOperation*) are available for reasoning over strings. For each, usage is supplied, followed by a definition and one or more examples.

Table A.3 — String operations

StringOperation	Description
(lengthOfString Str Num1)	Relates string Str to its length in characters Num1. *ex:* (lengthOfString "string" 6) returns true. *ex:* (lengthOfString "another" ?x) binds ?x to 7.
(concatenateStrings Str1 Str2)	Concatenate two strings together. *ex:* (concatenateStrings "ab" "cd" ?str) binds ?str to "abcd".
(concatenateAllStrings StrList Str)	Concatenate a list of strings together to form one string. *ex:* (concatenateAllStrings (listof "a " "b " "c") ?str) binds ?str to "a b c".
(substring Sub Str)	Tests if string Sub is a substring of string Str. *ex:* (substring "abc" "123abcd") returns true.
(equalsIgnoreCase Str1 Str2)	Is true if Str1 is identical to Str2 independent of case. *ex:* (equalsIgnoreCase "symbol" "SymBoL") returns true.
(toUpperCase Str1 Str2)	Converts a string to upper case. *ex:* (toUpperCase "usa" ?x) binds ?x to "USA".
(toLowerCase Str1 Str2)	Converts a string to lower case. *ex:*(toLowerCase "JOKE" ?x) binds ?x to "joke".
(getSubstring Str St En SubStr)	Binds SubStr to the segment of Str from position St to position En (inclusive). Positions are 1-count; 1 is the position of the first character in the string. *ex:*(getSubstring "abcdef" 2 4 ?Sub) binds ?Sub to "bcd".
(matchesPattern Pat Str)	Tests is a string Str matches a given pattern Pat. Allows for the wild-card character * which matches zero or more characters. *ex:* (matchesPattern "foo*" "football") returns true. *ex:* (matchesPattern "foo*" "baseball") is false.
(matchesPatternIgnoreCase Pat Str)	Behaves as *matchesPattern* but ignores case. *ex:* (matchesPatternIgnoreCase "FOO*" "football") returns true.
(matchesPatternToSymbol Pat Sym)	Tests that symbol Sym matches the pattern Pat. *ex:*(matchesPatternToSymbol "RootCtx.Bi*yR*" RootCtx.BinaryRel) returns true.
(matchesPatternToSymbolIgnoreCase Pat Sym)	Behaves as *matchesPatternToSymbol* but ignores case. *ex:*(matchesPatternToSymbolIgnoreCase "rootctx.bi*yr*" RootCtx.BinaryRel) returns true.
(lexSubstitution Form List Result)	Binds Result to the result of substituting the members of List into the string Form, where "?*k*" in Form refers to the *k*th member of List (counting from zero), and "@*k*" refers to the *k*th member of List and every member following it. A backslash before '?' or '@' avoids the substitution. *ex:*(lexSubstitution "?1 ?2 ?0" (listof "2014" "Nov" "12") ?R) binds R to "Nov 12 2014". *ex:*(lexSubstitution "Set \?1 to ?0" (listof "40" "FF" "3E") ?R) binds R to "Set ?1 to 40".

A.6 Numbers

A.6.1 General

Numbers are similar to Times and Strings in that their class extent is infinite. This characteristic lends itself to a few capabilities and restrictions, i.e. questions can be asked about the properties, relationships and rules that govern numbers without having to declare numbers as instances. All numbers are already inherent in the system and ready to use. However, because there are infinitely many numbers, queries shall be posed where the arguments of a numerical Relation are bound, for example it is impossible to ask for all numbers greater than 5, as the set would be infinitely large.

A.6.2 Number Properties

There are several types of number properties available for defining new relations, properties, or functions. When defining a relation involving numbers, as with all relations, when choose the most specific signature properties possible.

ComplexNumber

all complex numbers

RealNumber

all real numbers

NonNegReal

nonnegative real numbers

PosReal

real numbers greater than zero

NegReal

real numbers less than zero

IntegerNumber

all integers

NonNegInt

nonnegative integers

PosInt

integers greater than zero

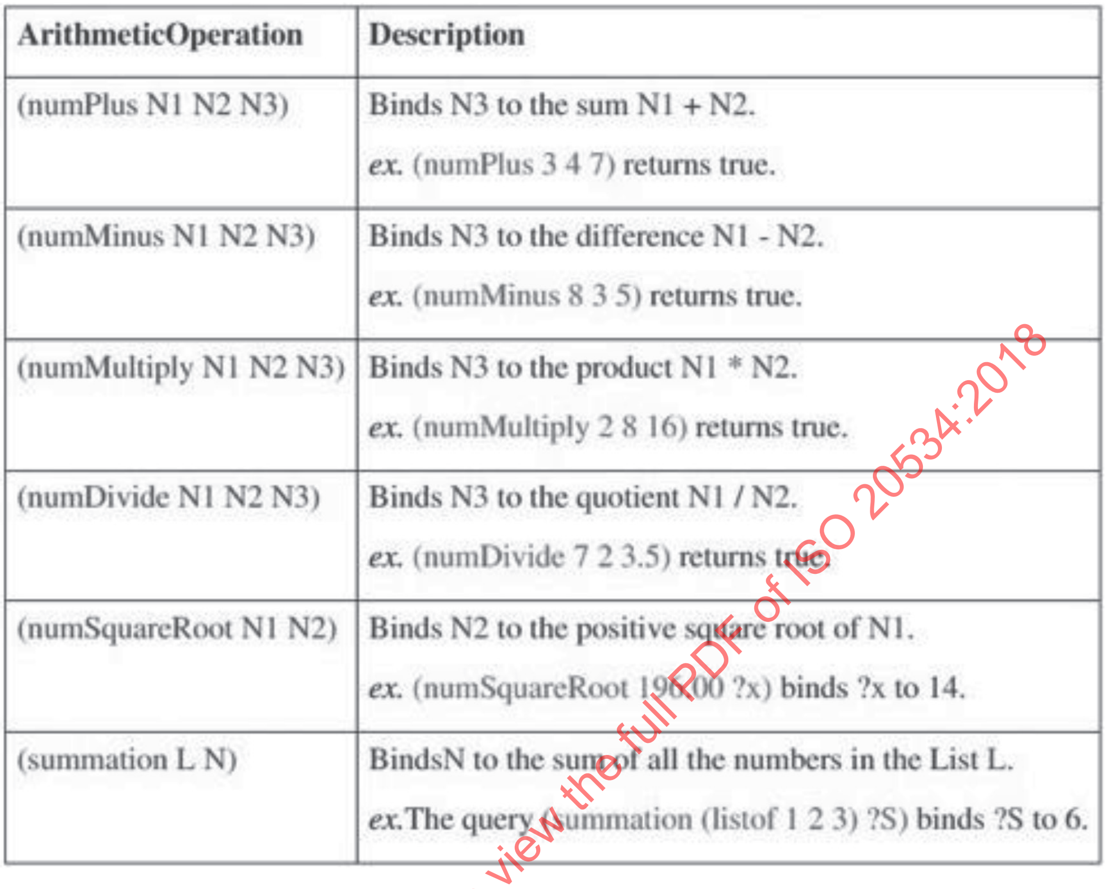
NegInt

integers less than zero

A.6.3 Number Relations

The ternary relations given in Table A.4 (instances of *ArithmeticOperation*) define the four arithmetic operations.

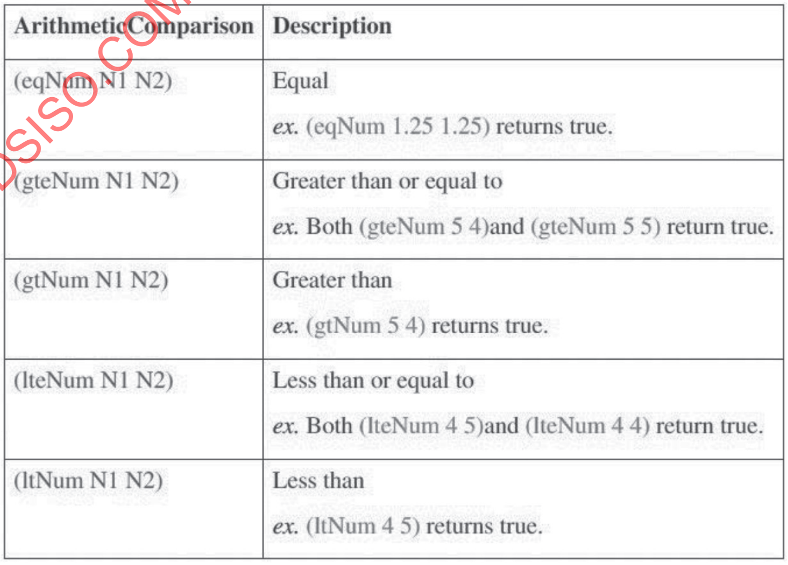
Table A.4 — Arithmetic operations



ArithmeticOperation	Description
(numPlus N1 N2 N3)	Binds N3 to the sum N1 + N2. *ex.* (numPlus 3 4 7) returns true.
(numMinus N1 N2 N3)	Binds N3 to the difference N1 - N2. *ex.* (numMinus 8 3 5) returns true.
(numMultiply N1 N2 N3)	Binds N3 to the product N1 * N2. *ex.* (numMultiply 2 8 16) returns true.
(numDivide N1 N2 N3)	Binds N3 to the quotient N1 / N2. *ex.* (numDivide 7 2 3.5) returns true.
(numSquareRoot N1 N2)	Binds N2 to the positive square root of N1. *ex.* (numSquareRoot 196.00 ?x) binds ?x to 14.
(summation L N)	BindsN to the sum of all the numbers in the List L. *ex.*The query (summation (listof 1 2 3) ?S) binds ?S to 6.

The binary relations given in Table A.5 define the five numerical comparison (*ArithmeticComparison*) operations.

Table A.5 — Arithmetic comparisons



ArithmeticComparison	Description
(eqNum N1 N2)	Equal *ex.* (eqNum 1.25 1.25) returns true.
(gteNum N1 N2)	Greater than or equal to *ex.* Both (gteNum 5 4)and (gteNum 5 5) return true.
(gtNum N1 N2)	Greater than *ex.* (gtNum 5 4) returns true.
(lteNum N1 N2)	Less than or equal to *ex.* Both (lteNum 4 5)and (lteNum 4 4) return true.
(ltNum N1 N2)	Less than *ex.* (ltNum 4 5) returns true.

A.7 Intervals

A.7.1 Defining Intervals

An ordered set of numbers can be defined with the interval function. With this function, the boundary conditions can be specified at both ends. These conditions are either:

— *in* – inclusive;

— *ex* – exclusive.

The numbers between 5 and 10, including both 5 and 10:

(interval in 5 10 in)

The numbers between 5 and 10, including neither 5 nor 10:

(interval ex 5 10 ex)

The numbers between 5 and 10, including 5 but not 10:

(interval in 5 10 ex)

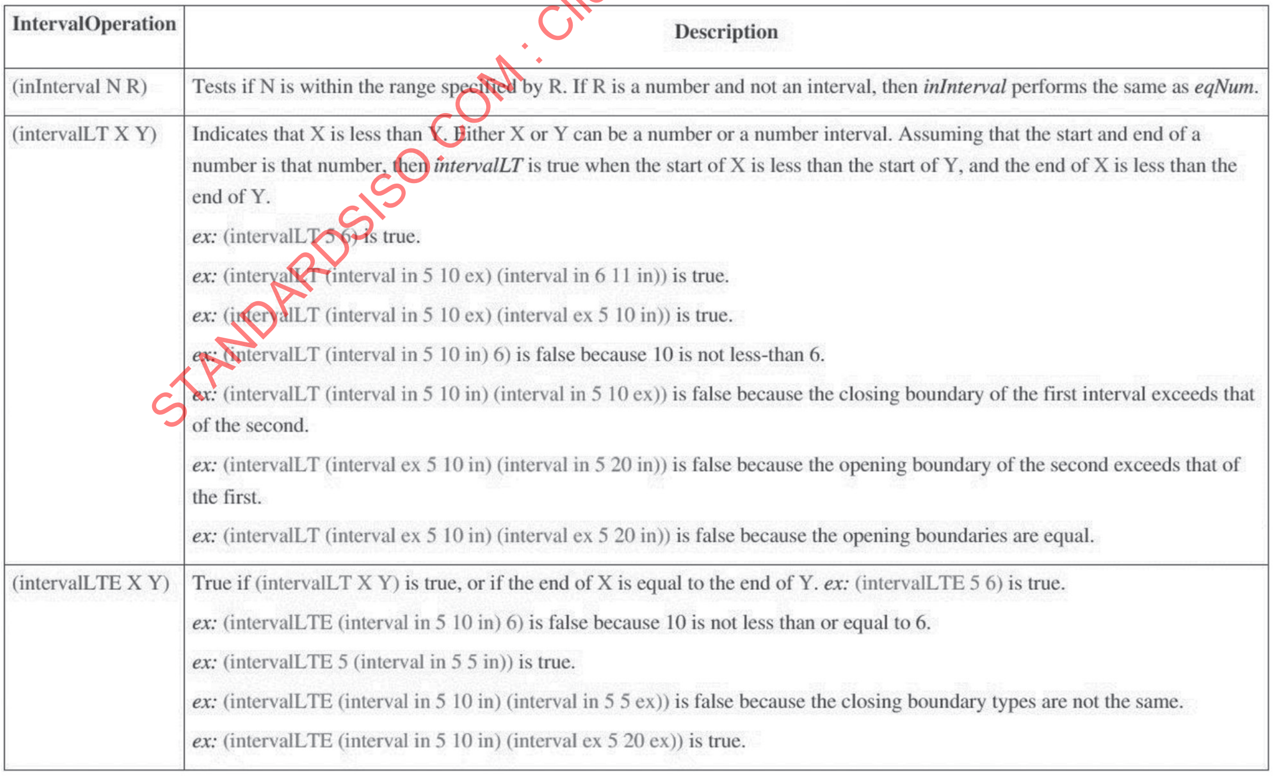
Intervals are instances of the Property NumberInterval. For instance, the following is true:

(NumberInterval (interval in 5 10 ex))

A.7.2 Interval Operations

The operations given in Table A.6 (IntervalOperation) are available for reasoning over intervals. For each, usage is supplied, followed by a definition and one or more examples.

Table A.6 — Interval operations



IntervalOperation	**Description**
(inInterval N R)	Tests if N is within the range specified by R. If R is a number and not an interval, then ***inInterval*** performs the same as ***eqNum***.
(intervalLT X Y)	Indicates that X is less than Y. Either X or Y can be a number or a number interval. Assuming that the start and end of a number is that number, then ***intervalLT*** is true when the start of X is less than the start of Y, and the end of X is less than the end of Y. *ex:* (intervalLT 5 6) is true. *ex:* (intervalLT (interval in 5 10 ex) (interval in 6 11 in)) is true. *ex:* (intervalLT (interval in 5 10 ex) (interval ex 5 10 in)) is true. *ex:* (intervalLT (interval in 5 10 in) 6) is false because 10 is not less-than 6. *ex:* (intervalLT (interval in 5 10 in) (interval in 5 10 ex)) is false because the closing boundary of the first interval exceeds that of the second. *ex:* (intervalLT (interval ex 5 10 in) (interval in 5 20 in)) is false because the opening boundary of the second exceeds that of the first. *ex:* (intervalLT (interval ex 5 10 in) (interval ex 5 20 in)) is false because the opening boundaries are equal.
(intervalLTE X Y)	True if (intervalLT X Y) is true, or if the end of X is equal to the end of Y. *ex:* (intervalLTE 5 6) is true. *ex:* (intervalLTE (interval in 5 10 in) 6) is false because 10 is not less than or equal to 6. *ex:* (intervalLTE 5 (interval in 5 5 in)) is true. *ex:* (intervalLTE (interval in 5 10 in) (interval in 5 5 ex)) is false because the closing boundary types are not the same. *ex:* (intervalLTE (interval in 5 10 in) (interval ex 5 20 ex)) is true.

A.8 Lists

A.8.1 General

In many situations it is helpful to not only operate on single pieces of data, but also on lists of data. In Highfleet systems, a List is simply a collection of elements with some implicit order. For example (listof a d c) is the collection of the elements “a”, “c” and “d”, which is ordered in the sense that “a” appears in the first position, “d” appears in the second position, etc. Lists are thus distinguishable from sets in that ordering among items matters and duplication is acceptable, for example (listof a a a) is a valid List. Lists are useful for any situation that involves passing around collections of objects.

A list takes the form of a function term; the functor is the special listof, and there can be any number of arguments. The following demonstrates a use of the listof function.

:Rel midgeCounts

:Inst BinaryRel

:Sig Location List

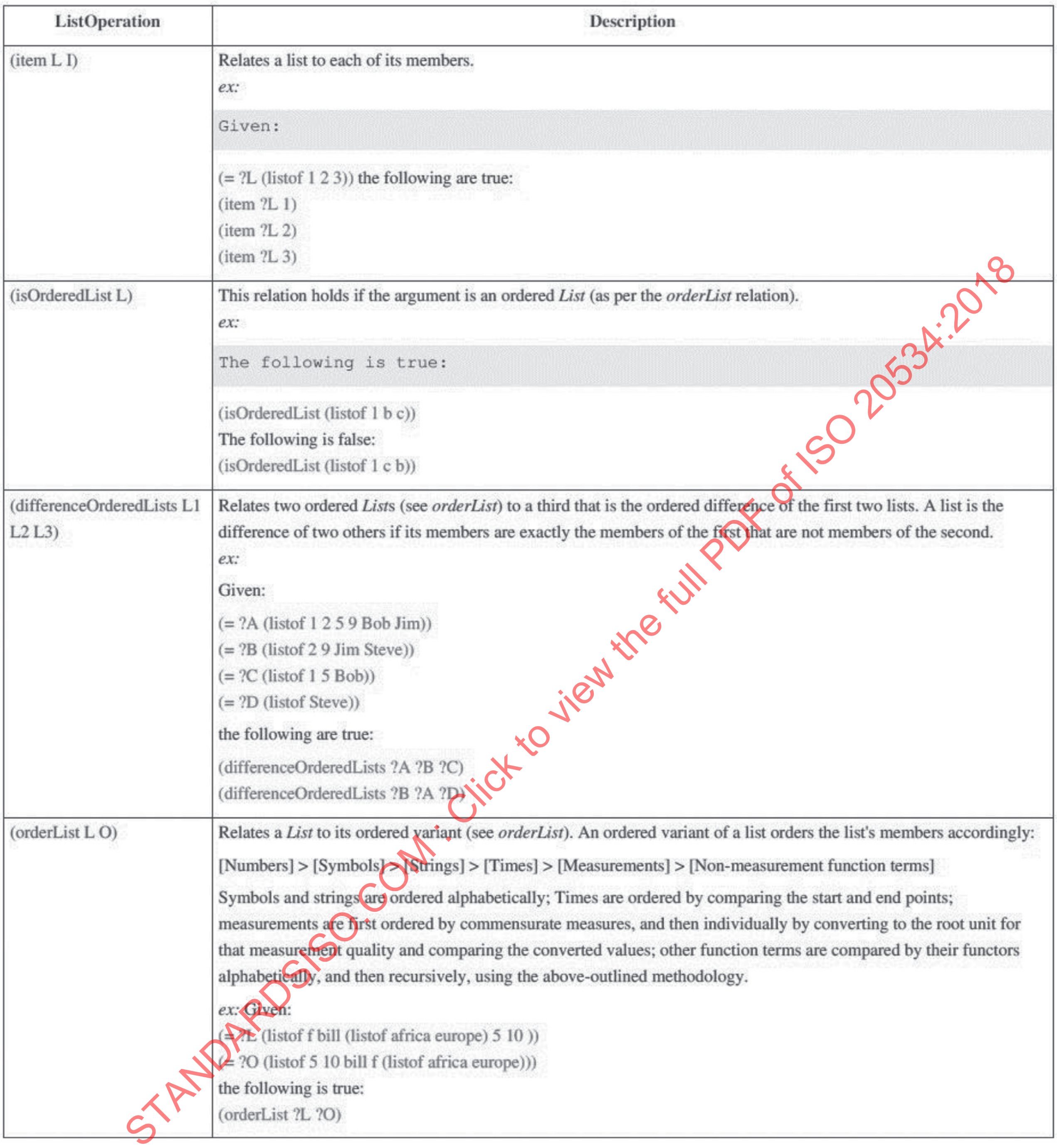
:rem "Relates a list of midge counts to the location where the counts were taken."

(midgeCounts HerringRunPark (listof 34 55 12 19 26))

A.8.2 List Operations

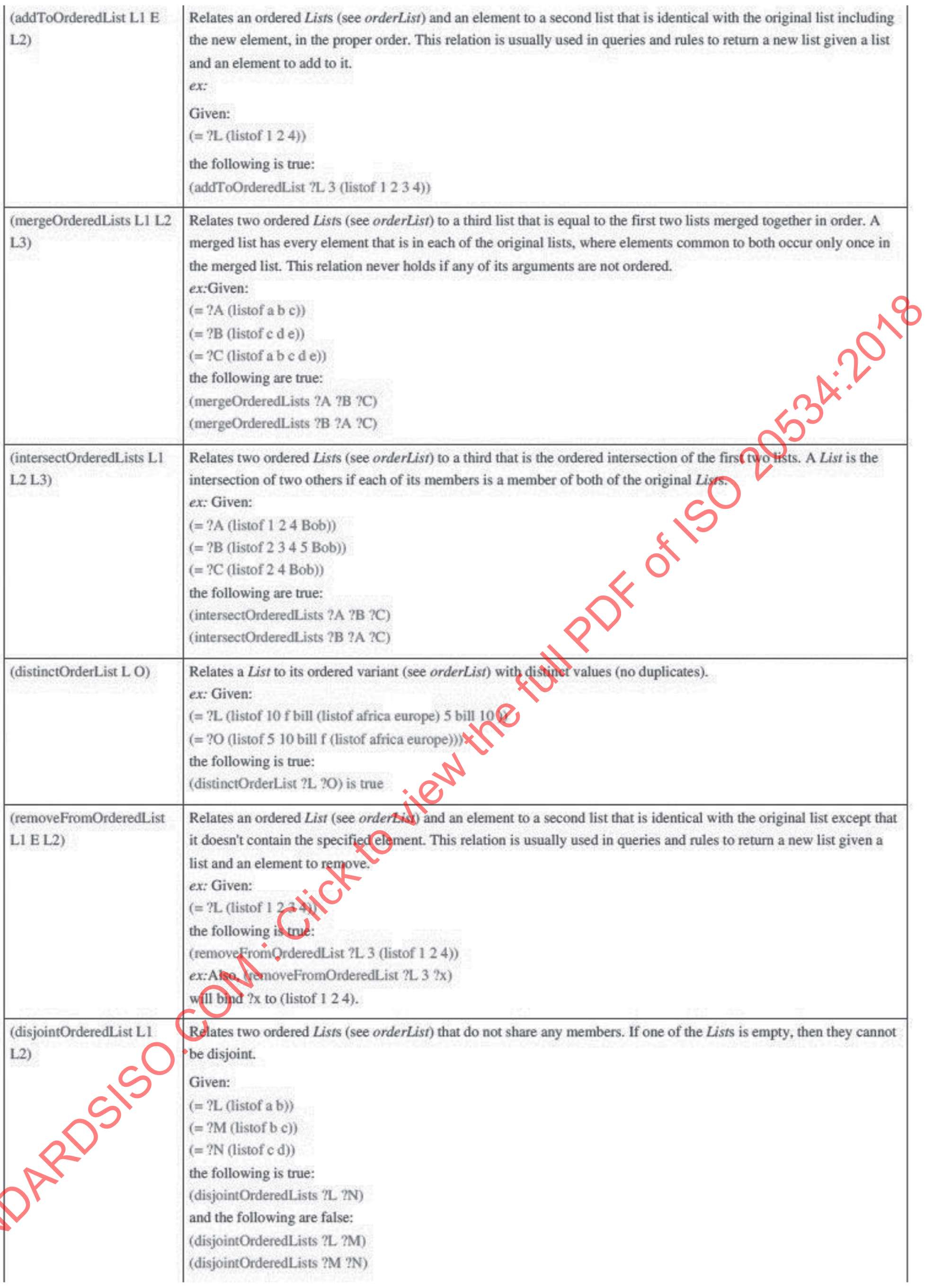
The operations given in Table A.7 (ListOperation) are available for reasoning over Lists. For each, usage is supplied, followed by a definition and one or more examples.

Table A.7 — List operations



ListOperation	Description
(item L I)	Relates a list to each of its members. *ex:* `Given:` (= ?L (listof 1 2 3)) the following are true: (item ?L 1) (item ?L 2) (item ?L 3)
(isOrderedList L)	This relation holds if the argument is an ordered *List* (as per the *orderList* relation). *ex:* `The following is true:` (isOrderedList (listof 1 b c)) The following is false: (isOrderedList (listof 1 c b))
(differenceOrderedLists L1 L2 L3)	Relates two ordered *Lists* (see *orderList*) to a third that is the ordered difference of the first two lists. A list is the difference of two others if its members are exactly the members of the first that are not members of the second. *ex:* Given: (= ?A (listof 1 2 5 9 Bob Jim)) (= ?B (listof 2 9 Jim Steve)) (= ?C (listof 1 5 Bob)) (= ?D (listof Steve)) the following are true: (differenceOrderedLists ?A ?B ?C) (differenceOrderedLists ?B ?A ?D)
(orderList L O)	Relates a *List* to its ordered variant (see *orderList*). An ordered variant of a list orders the list's members accordingly: [Numbers] > [Symbols] > [Strings] > [Times] > [Measurements] > [Non-measurement function terms] Symbols and strings are ordered alphabetically; Times are ordered by comparing the start and end points; measurements are first ordered by commensurate measures, and then individually by converting to the root unit for that measurement quality and comparing the converted values; other function terms are compared by their functors alphabetically, and then recursively, using the above-outlined methodology. *ex:* Given: (= ?L (listof f bill (listof africa europe) 5 10)) (= ?O (listof 5 10 bill f (listof africa europe))) the following is true: (orderList ?L ?O)

Table A.7 *(continued)*



(addToOrderedList L1 E L2)	Relates an ordered *Lists* (see *orderList*) and an element to a second list that is identical with the original list including the new element, in the proper order. This relation is usually used in queries and rules to return a new list given a list and an element to add to it. *ex:* Given: (= ?L (listof 1 2 4)) the following is true: (addToOrderedList ?L 3 (listof 1 2 3 4))
(mergeOrderedLists L1 L2 L3)	Relates two ordered *Lists* (see *orderList*) to a third list that is equal to the first two lists merged together in order. A merged list has every element that is in each of the original lists, where elements common to both occur only once in the merged list. This relation never holds if any of its arguments are not ordered. *ex:*Given: (= ?A (listof a b c)) (= ?B (listof c d e)) (= ?C (listof a b c d e)) the following are true: (mergeOrderedLists ?A ?B ?C) (mergeOrderedLists ?B ?A ?C)
(intersectOrderedLists L1 L2 L3)	Relates two ordered *Lists* (see *orderList*) to a third that is the ordered intersection of the first two lists. A *List* is the intersection of two others if each of its members is a member of both of the original *Lists*. *ex:* Given: (= ?A (listof 1 2 4 Bob)) (= ?B (listof 2 3 4 5 Bob)) (= ?C (listof 2 4 Bob)) the following are true: (intersectOrderedLists ?A ?B ?C) (intersectOrderedLists ?B ?A ?C)
(distinctOrderList L O)	Relates a *List* to its ordered variant (see *orderList*) with distinct values (no duplicates). *ex:* Given: (= ?L (listof 10 f bill (listof africa europe) 5 bill 10)) (= ?O (listof 5 10 bill f (listof africa europe))) the following is true: (distinctOrderList ?L ?O) is true
(removeFromOrderedList L1 E L2)	Relates an ordered *List* (see *orderList*) and an element to a second list that is identical with the original list except that it doesn't contain the specified element. This relation is usually used in queries and rules to return a new list given a list and an element to remove. *ex:* Given: (= ?L (listof 1 2 3 4)) the following is true: (removeFromOrderedList ?L 3 (listof 1 2 4)) *ex:*Also, (removeFromOrderedList ?L 3 ?x) will bind ?x to (listof 1 2 4).
(disjointOrderedList L1 L2)	Relates two ordered *Lists* (see *orderList*) that do not share any members. If one of the *Lists* is empty, then they cannot be disjoint. Given: (= ?L (listof a b)) (= ?M (listof b c)) (= ?N (listof c d)) the following is true: (disjointOrderedLists ?L ?N) and the following are false: (disjointOrderedLists ?L ?M) (disjointOrderedLists ?M ?N)

Table A.7 *(continued)*

(listChoose L1 L2)	Relates two *Lists* such that the second is a sublist of the first. Unlike the *isSublist* relation, *listChoose* allows you to put variables into the list positions of the chosen list. This relation is usually used to quickly narrow down the sublists of a list. *ex:* Given: (= ?L (listof a b c d)) the query (listChoose ?L (listof ?x c)) binds ?x to a and b.
(emptyList L)	Returns true if L has no members, i.e., if L is the empty list (listof).
(length L N)	Binds to N the length of the *List* L, i.e. ?x is the number of objects in L. *ex:* (length (listof a b c) ?x) binds ?x to 3.
(first L I)	binds to I the first element of L. *ex:*(first (listof a b c) ?x) binds ?x to a.
(last L I)	Binds to I the last element in *L*. *ex:*(last (listof a b c) ?x) binds ?x to c.
(nth L N I)	Binds to I the element of *L* at position N. *ex:*(nth (listof a b c) 2 ?x) binds ?x to b.
(isSublist L1 L2)	Returns true if L2 is a (contiguous) sublist of L1. *ex:*(isSublist (listof a b c) (listof b c)) is true.
(isPrefix L1 L2)	Returns true if L2 is the first n elements of L1, where 1 < n < length of L1 *ex:*(isPrefix (listof a b c) (listof a b)) is true.
(listInsert L1 I N L2)	Binds L2 to a List containing all the elements with L1 with entity I inserted at position N (where 1 < N < length(L1) + 1) *ex:*(listInsert (listof a b c) d 2 ?newL) binds ?newL to (listof a d b c).
(append L1 L2 L3)	Binds ?newL to a *List* containing all of the elements of L1 followed by all of the elements of L2. *ex:* (append (listof a b c) (listof d e f) ?newL) binds ?newL to (listof a b c d e f).
(rest L1 L2)	Binds to L2 the *List* containing all of the elements of L1 between positions 2 and length(L1). *ex:*(rest (listof a b c) ?newL) binds ?newL to (listof b c).
(butLast L1 L2)	Binds L2 to L1 with the element at position N omitted, where N is the length of L1. *ex:*(butLast (listof a b c) ?newL) binds ?newL to (listof a b).
(butNth L1 N L2)	Binds L2 to a list with all the elements of L1 except for the the element at position N. *ex:*(butNth (listof a b c) 2 ?newL) binds ?newL to (listof a c).
(reverse L1 L2)	Binds to L2 a *List* containing all of the elements of L1 in reverse order. *ex:*(reverse (listof a b c) ?newL) binds ?newL to (listof c b a).
(replace L1 I N L2)	Binds L2 to L with the element at position N replaced with I (N must be a position in the list that already contains an element). *ex:*(replace (listof a b c) q 2 ?newL) binds ?newL to (listof a q c).

Table A.7 *(continued)*

(move L1 L2 L3 L4)	Binds to L4 a *List* containing elements of L1 with the elements at the positions specified in L2 moved to those positions specified in L3. *ex:*(move (listof a b c) (listof 1 2) (listof 3 1) ?newL) binds ?newL to (listof b b a).
(listProj L1 L2 L3)	Binds to L3 the *List* containing those elements of L3 specified by the positions in the *List* L1. *ex:*(listProj (listof 3 2) (listof a b c) ?newL) binds ?newL to (listof c b).
(enumerate N1 N2 L2)	Binds to L2 the *List* [N1..N2], enumerating from N1 to N2. *ex:*(enumerate 1 5 ?newL) binds ?newL to (listof 1 2 3 4 5)
(permutations L1 L2)	Binds to L2 all of the possible permutations of L1. *ex:*(permutations (listof a b c) ?x) binds ?x to (listof a b c), (listof a c b), (listof b a c), (listof b c a), (listof c b a), and (listof c a b).

A.9 Aggregation

A.9.1 General

The aggregation operators support collecting the answers to a query and processing those answers to determine a single value (with the exception of 'equivalenceClasses', which produces multiple answers - one answer for each equivalence class found). The supported operators are count, avg, min, max, minTerm, maxTerm, sum, setof, and equivalenceClasses.

Every aggregation statement contains a subquery, called the ***template query***, that retrieves the results to be aggregated, and a ***target***, a variable that will be bound to the aggregated value. For example, to calculate the average salary of all employees, the following can be queried:

(avg ?s

(and

(Employee ?e)

(salary ?e ?s))

?avg)

The template query is

(and

(Employee ?e)

(salary ?e ?s))

and the target is ?avg.

A.9.2 Aggregation and Variable Binding Lists

Aggregation operators pose special considerations for variable binding. Every variable that appears in an template query shall be bound, either within the template query, outside of the aggregation, or by the aggregation operator itself.

Like existential quantification (exists) and universal quantification (forall), aggregation operators can bind variables in the clauses they operate over (the template query).

A.9.3 Full Form, Brief Form, and Distinguishing Variables

Each aggregation operator has two forms, brief and full. The full form aggregation operators (countf, avgf, minf, maxf, minTermf, maxTermf, sumf, setofx, and equivalenceClassesf) take an argument that specifies which unbound variables in the template query to bind with the aggregation literal. The short form aggregation operators (count, avg, min, max, minTerm, maxtTerm, sum, setof, and equivalenceClasses) bind *all* of the variables in the template query. For instance, the full form of the query above is:

```
(avgf ?s (?e)

(and

(Employee ?e)

(salary ?e ?s))

?avg)
```

The second argument (?e) specifies the variable bound by the aggregation operator. Variables bound this way are often called distinguishing variables because it is their bindings that distinguish the values to be aggregated from one another. Consider the following two queries:

```
(countf (?x ?y)

(and

(Person ?x)

(Automobile ?y)

(owns ?x ?y))

?total)

(countf (?x)

(exists (?y)

(and

(Person ?x)

(Automobile ?y)

(owns ?x ?y)))

?total)
```

The first query lists both ?x and ?y as distinguishing variables and counts the number of person-automobile pairs. If one person owns three automobiles, that person will contribute three results to the total count.

The second query lists only ?x as a distinguishing variable, and binds ?y within the template query with existential quantification. This query counts the number of people who own one or more cars. If one person owns three automobiles, that person will contribute one result to the total count. The second query will return a count that is necessarily less than or equal to the first.

A.9.4 Binding *Across* Aggregation Operators

The full form aggregation operators allow the variables to be bound within the template query with other clauses outside of the aggregation statement. For instance, to query how many automobiles of each make people own:

(and

(VehicleMake ?z)

(countf (?x ?y)

(and

(Person ?x)

(Automobile ?y)

(owns ?x ?y)

(hasVehicleMake ?y ?z))

?total)

)

This query binds ?z outside the aggregation statement with the (VehicleMake ?z) clause. If there are 10 different vehicle makes, the query will return 10 counts, one for each vehicle make.

NOTE The brief aggregation operators do not allow variables to be bound across the aggregation this way.

A.9.5 Aggregation Syntax

The countf aggregation operator has the following syntax:

(countf (?DistinguishingVariables) ?Sentence ?Count)

Queries of this form count the number of distinct bindings corresponding to the distinct bindings of ?DistinguishingVariables by ?Sentence.

The short form *count* aggregation operator has a similar syntax with the removal of the distinguishing variables argument:

(count ?Sentence ?Count)

Queries of this form count the number of distinct bindings corresponding to the distinct bindings of all the variables in ?Sentence.

The non-count aggregation operators are similar in syntax and semantics to the *countf* and *count* aggregation operators with the addition of a value variable. For instance, consider the following example to determine the average salary:

(avgf ?sal (?p) (salaryOf ?p ?sal) ?avg)

In this example, the *salaryOf* relation relates a person ?p to a salary ?sal. It is assumed that each person has a single salary. The arguments to avgf are the same as those for countf with the addition of the value variable ?sal. This value variable specifies the variable the bindings of which are to be gathered into a set over which the average is to be calculated.

The brief form of the avg operator has the same syntax and semantics as the avgf operator with the removal of the distinguishing variables argument. Modifying the previous example:

(avg ?sal (salaryOf ?p ?sal) ?avg)

This has the same meaning as the example avgf query. The brief form assumes that the distinguishing variables are

all of the variables free in the aggregation literal argument, just as was described for the count aggregation operator.

The syntax for each *of sum, min, max, minTerm, maxTerm, equivalenceClasses* is identical.

(sum ?Value ?Sentence ?Sum)

(sumf ?Value (?DistinguishingVariables) ?Sentence ?Sum)

The sum of the set of ?Value bindings corresponding to the distinct bindings of ?DistinguishingVariables by

?Sentence.

(min ?Value ?Sentence ?Min)

(minf ?Value (?DistinguishingVariables) ?Sentence ?Min)

The minimum number of the set of ?Value bindings corresponding to the distinct bindings of

?DistinguishingVariables by ?Sentence.

(max ?Value ?Sentence ?Max)

(maxf ?Value (?DistinguishingVariables) ?Sentence ?Max)

The maximum number of the set of ?Value bindings corresponding to the distinct bindings of ?DistinguishingVariables by ?Sentence.

(minTerm ?Value ?Sentence ?Min)

(minTermf ?Value (?DistinguishingVariables) ?Sentence ?Min)

The minimum term (ordered according to the ECLIF ordering of all terms) of the set of ?Value bindings corresponding to the distinct bindings of ?DistinguishingVariables by ?Sentence. The minTerm aggregation result may be any ECLIF term, for example a list, a date, a number, a string.

(maxTerm ?Value ?Sentence ?Max)

(maxTermf ?Value (?DistinguishingVariables) ?Sentence ?Max)

The maximum term (ordered according to the ECLIF ordering of all terms) of the set of ?Value bindings corresponding to the distinct bindings of ?DistinguishingVariables by ?Sentence. The maxTerm aggregation result may be any ECLIF term, for example a list, a date, a number, a string.

(setof ?Value ?Sentence ?Set)

(setofx ?Value (?DistinguishingVariables) ?Sentence ?Set)

The set of ?Value bindings corresponding to the distinct bindings of ?DistinguishingVariables by ?Sentence. The setof aggregation result is an ordered list, unique with respect to the ?DistinguishingVariables which allows for the values in the list to be duplicates.

(equivalenceClasses ?Value ?Sentence ?Class)

(equivalenceClassesf ?Value (?DistinguishingVariables) ?Sentence ?Class)

The aggregation variable values (bound to ?Value) shall be of the form (Functor ?a ?b). This is interpreted as '?a is equivalent to ?b'. This operator finds the equivalence classes of the interpreted-as-equivalent values in the set of all bindings of answer variables in solutions of ?Sentence. The answer variables are all of the free variables in ?Sentence.

The equivalenceClasses aggregation result is an ordered list for each class in the set of equivalence classes.

A.9.6 Multiple Aggregation Operations

The aggregation operators can be used together in a query. For instance, consider a query to determine how many people have above average salaries:

```
(and
   (avg ?sal
      (salaryOf ?p ?sal)
   ?avg)
    (countf (?p ?sal)
      (and
           (salaryOf ?p ?sal)
           (ltNum ?avg ?sal))
?cnt))
```

In this example ?avg is the average of all ?sal for all (salaryOf ?p ?sal) answers and ?cnt is the number of people with salaries greater than ?avg.

A.10 Materialization

Materialization is persistent caching of answers for an intensional relation. The answers to an unbound literal for that relation are inserted as extensional facts in the backend store, and a subsequent query that references that relation gets answers solely from these extensional facts.

A relation shall be defined as a *MaterializeableRel* to apply materialization to it. The MaterializeableRel property is used instead of either *IntensionalRel* or *ExtensionalRel*. For example:

```
:Use TestCtx
:Rel foo
:Inst BinaryRel
:Inst MaterializeableRel
:Sig String String
(<= (foo ?x ?y)
(bar ?x ?y)
)
```

There are commands available in the IODE to materialize and dematerialize relations. These commands are options

on the IODE Database menu:

— Materializable relations > Materialize into Database...

— Materializable relations >Dematerialize from Database...

— Materializable relations >Extract materializable relations from Database...

These commands support 'multi-select': multiple relations can be materialized or dematerialized at the same time. The materialization and dematerialization operations can also be invoked using queries.

(owi.MaterializeFacts (RootCtx.listof A B C) ?relation)

Requests materialization of relations A, B, and C. An answer is returned for each of these relations that is successfully materialized.

(owi.DematerializeFacts (RootCtx.listof A B C) ?relation)

Requests dematerialization of relations A, B, and C. An answer is returned for each of these relations that is successfully dematerialized.

A.11 Ontological Introspection

A.11.1 General

It is often useful to use an ontology to examine itself, i.e. to write queries or rules about the structure of an ontology. The following subclauses list of some of the tools available to do so.

A.11.2 Taxonomic Structure

A.11.2.1 General

Two relations describe the taxonomic (hierarchical) structure of ontologies: sup and supRel.

A.11.2.2 sup

Defines the hierarchy of properties; relates a property to (one of) its super-properties. Every instance of the sub-property is an instance of the super-property.

(sup Dog Mammal) means that *Dog* is a sub-property of *Mammal*, and every instance of *Dog* is an instance of *Mammal*.

A.11.2.3 supRel

Defines the hierarchy of relations; relates a relation to (one of) its super-relations. *supRel* is a generalization of sup that allows for multiple arguments. Every fact in the extent of a sub-relation is in the extent of the super-relation (supRel hasBrother hasSibling) means that there is a *hasSibling* fact for every *hasBrother* fact. For example, if (hasBrother Tom Steven) is a fact in the knowledge server, then (hasSibling Tom Steven) will be also.

A.11.3 Relation Structure

A.11.3.1 General

Two relations describe the way relations structure the ontology: *argProp* and *arity*.

Relations hold between entities, and every entity is an instance of some class. The ***signature*** of a relation is a sequence of properties that specify what kind of entities the relation relates.

For example, *hasMother* can be defined as relating a *Person* to a *FemalePerson*, i.e. *Person* would be the first argument in the signature of *hasMother*, and *FemalePerson* would be the second argument.

A.11.3.2 argProp

argProp is a 3-place relation. Each instance of argProp describes one argument in a relation's signature. A relation itself, argProp has a signature: the first argument shall be a Relation, the second argument shall be a positive integer (PposInt), and the third argument shall be a Property. The first argument gives the relation to be described, the second argument gives an argument position in that relation, and the third argument gives kind of entity in that argument of the signature of the relation.

hasMother is a two-place relation relating a Person to a FemalePerson. Thus, every hasMother statement has to have an instance of Person as the first argument and an instance of FemalePerson as a second argument. Thus, if

(hasMother John Mary)

is true, then

(and (Person John) (FemalePerson Mary))

shall also be true, or an IC violation will be triggered. To describe the signature of hasMother, the following two facts are used:

(argProp hasMother 1 Person)

(argProp hasMother 2 FemalePerson)

The following query would find all relations that relate the Person property to any other entity:

(argProp ?x ?y Person)

A.11.3.3 arity

The arity of a relation is the number of arguments. *arity* is a two-place relation holding between a *Functor* (a super-property of *Relation*) and a non-negative integer.

hasMother is a two-place relation holding between a *Person* and a *MalePerson*. Thus,

(arity hasMother 2)

is true in the system. A 3-place relation, like *isBetween*, which can be defined to hold between *Objects*:

(argProp isBetween 1 Object)

(argProp isBetween 2 Object)

(argProp isBetween 3 Object)

would have an arity of 3:

(arity isBetween 3)

The following query would find all relations that have an arity of 3:

(arity ?x 3)

A.11.4 Property Instance Structure

A.11.4.1 General

These relations represent what it is to be a certain kind of thing. When an object is an instance of a property, it is also an instance of every super-property (*sup*) of that property.

There are three 'inst' relations in the ontology: *inst*, *instDirect*, and *instAsserted*.

A.11.4.2 inst

inst is a *BinaryRel* that relates instances to the properties of which those instances are members.

inst relates instances to *all* of the properties they belong to, including those that they inherit from the property hierarchy. For example, if it is true that P1 is a *MalePerson* and that all *MalePersons* are *Persons*, as the following

ECLIF states:

(MalePerson P1)

(sup MalePerson Person)

then the following are true:

(inst P1 MalePerson)

(inst P1 Person)

Logical Definition

(<=>

(inst ?x ?c)

(and (Property ?c) (?c ?x)))

where at least one of ?x and ?c shall be specified. If ?c is a non-logical functor (i.e., it has infinite extent, like *PosInt*), then ?x shall also be bound.

A.11.4.3 instDirect

instDirect is a subrelation of *inst* that relates instances to the properties of which they are direct members.

instDirect does not relate instances to properties inherited from the property hierarchy.

For example, if it is true that P1 is a *MalePerson* and that all *MalePersons* are *Persons*, as the following ECLIF states:

(MalePerson P1)

(sup MalePerson Person)

then the following are true

(instDirect P1 MalePerson)

(inst P1 Person)

(Person P1)

but the following is false:

(instDirect P1 Person) ;; false

Logical Definition

(<=>

(instDirect ?x ?c)

(and (inst ?x ?c) (not (exists (?y) (and (inst ?x ?y) (supTC ?y ?c))))))

A.11.4.4 instAsserted

instAsserted is a subrelation of *instDirect* that relates instances to the properties of which they are *asserted* to be members.

instAsserted does not relate instances to properties derived from rules. It only returns instance information that was explicitly asserted either as ground ECLIF statements or through KFL directives (for example :Inst Type).

For example, if it is true that all entities with a *MaleName*are instances of *MalePerson*, and P1 is a Person with the *MaleName* "John", as the following ECLIF states:

(Person P1)

(name P1 "John")

(MaleName "John")

(=>

(and

(name ?x ?y)

(MaleName ?y)

)

(MalePerson ?x)

)

then the following are true

(instAsserted P1 Person)

(instDirect P1 MalePerson)

(inst P1 MalePerson)

(MalePerson P1)

but the following is false:

(instAsserted P1 MalePerson) ;; false

Logical Definition

(<=> (instAsserted ?x ?c) (and (asserted ?P) (quotedBy (?c ?x) ?P)))

Multiple Aggregation Operations

A.12 Administrative: pseudoPredicate

A *pseudo-predicate* is a predicate that is not modelled in the ontology, but that may be invoked as part of an ECLIF query as if it were. Pseudo-predicates usually retrieve information about the XKS, or cause certain operations to be performed, such as setting various query planning properties or toggling statistics collection. *pseudoPredicate* shows information about the pseudo-predicates currently available in the XKS. It cannot be asserted.

(pseudoPredicate Pred Arity Mode Desc) means that Pred is a pseudo-predicate that takes Arity arguments, has usage Mode ("relation" if it retrieves information, "assertion" if it causes something to happen), and description Desc.

Annex B
(informative)

Ontologies in Knowledge Framework Language (KFL)

B.1 Overview

B.1.1 General

This annex provides a reference guide to the syntax and commonly used techniques for writing Knowledge Framework Language (KFL) files. The KFL ontology modelling language includes properties, relations, rules and integrity constraints, functions, and constants.

B.1.2 What is KFL?

Knowledge Frame Language (or KFL) is a convenience layer of syntactic sugar that sits on top of a base layer of logical syntax called ECLIF (Extended Common Logic Interchange Format). For convenience, the whole language is referred to as (ECLIF + the syntactic sugar) *KFL*. A parenthesis-heavy syntax like ECLIF is confusing for some users. So, KFL is designed according to a “90 % rule” – most of what an ontologist needs to do can be done without raw ECLIF.

Most of KFL takes the form of *directives*. Directives are expressed as a colon at the start of a line, followed by a keyword and some arguments. Some arguments (like labels) are simply strings, but most are elements of the ontology already present on the server.

This means that the tools used to create the ontology are themselves part of the ontology. The KFL that is created builds on Upper Level Ontology (ULO) content preloaded on each new knowledge server. Writing effective KFL requires familiarity with the ULO, so this manual will introduce some of the ULO content alongside KFL. For more detail, refer to the Ontology Library Reference.

B.1.3 Contexts

The example at the beginning of this document demonstrates how to use a pre-existing context. Custom contexts can also be created to stake out a new namespace or make modules that make a large ontology more manageable.

The following KFL creates a new context *Chemistry* whose super-context (:supCtx) is *MLO* (short for Mid-Level Ontology, a predefined context). *Chemistry* is an instance of *UserContext*, a special type of context. (Any context that is defined should be an instance of *UserContext*.) All context declarations shall include a :supCtx directive specifying their parent in the context hierarchy. This declaration also supplies an optional descriptive remark (:rem) and name (:name).

```
:Ctx Chemistry

:Inst UserContext

:supCtx MLO

:name "Chemistry Context"

:rem "basic chemistry"
```

Once a context has been defined, it can subsequently be used with a :Use directive:

```
:Use Chemistry
```

:Rel molecularWeight

:Inst BinaryRel

:Sig MolecularSpecies MassQuality

:name "has molecular weight"

:rem "the mass of one molecule of a molecular species"

Contexts dictate how a symbol is referenced later. The full name of the relation introduced above is Chemistry.molecularWeight. Since the :Use Chemistry directive is in effect, symbols appearing without context are understood to have the *Chemistry* context by default.

Symbols are only qualified by the context they are a part of, not their parent contexts. For example, reference is made to *Chemistry.molecularWeight*, not to *MLO.Chemistry.molecularWeight*.

B.2 Properties

B.2.1 General

Properties make up the taxonomic component of any KFL ontology. A property is a predicate that can apply to one entity at a time and is used describe what the entity is. Examples of a property include Person, Vehicle, and City. Because ontologies are semantic (modelling meaning), rather than set-theoretical (modelling categorizations), the term *property* is used rather than *class* or *category*.

B.2.2 Useful property kinds

In practice, two kinds of properties pre-defined in the ULO are generally used: *Type* and *MaterialRole*. Types are properties that “stick” to their instances permanently. An instance of a type cannot cease to be that type while it exists. Material roles, in contrast, can come and go. Below are examples of declaring a type and a material role:

:Prop Person

:Inst Type

:sup ConcreteObject

:name "person"

:rem "A human being, living or dead."

:Prop Waiter

:Inst MaterialRole

:sup Person

:name "waiter"

:rem "A person hired to wait tables at a restaurant."

The :Prop, :Inst, and :sup directives are required for all properties. The :name and :rem directives are optional, though strongly recommended.

The :Prop directive introduces the property. The :Inst directive states what the property instantiates (usually *Type* or *MaterialRole*).Hierarchies of properties are defined via the super-property relation, referred to as *sup* in KFL. Property A is a super-property of property B if every instance of property B is necessarily an instance of property A.

The :sup directive has some special properties:

— its argument shall be an instance of Property;

— a property cannot be a sub-property of itself;

— a property may have more than one direct super-property. In KFL, this appears as multiple :sup directives.

Types cannot be subsumed by material roles.

B.2.3 Disjointness, Partitions and Covers

In addition to specifying that one property subsumes another, it can also be stated that two properties have no instances in common; in other words, they are *disjoint*. Disjointness assertions improve the integrity of the model and can improve query times.

For example, this code:

:Prop Car

:Inst Type

:sup ConcreteObject

:disjointWith Bus

means that no instance of *Car* may also be an instance of *Bus*, and vice versa. Any data that implies that a single entity is both a bus and a car will trigger an integrity constraint violation in the server.

Parts of taxonomies often form systems of related properties. For example, if it is necessary to define a group of types that are all special kinds of vehicle, but each needs to be disjoint from the others, by declaring a partition this can be done in one sentence:

(partitionedBy Vehicle (listof Car Train Plane))

(The *listof* operator above is simply an ECLIF operator for denoting lists. For more detail on *listof*, see the Ontology Library Reference.)

The consequences of declaring a partition are that:

— each member of the partition list shall be directly subsumed by the property being partitioned;

— the members of the partition list are pairwise disjoint;

— any instance of the property being partitioned shall be an instance of one of the partitioning properties.

A similar mechanism is a covering. A covering works like a partition, except that it does not require mutual disjointness, e.g. as follows:

(coveredBy RealNumber (listof NonzeroNumber IntegerNumber RationalNumber IrrationalNumber))

NOTE Every partition is a covering, but not every covering is a partition.

There is also an easy way to say “Every property subsumed by P in this taxonomy is (pairwise) disjoint” – just write:

(disjointSubProps P)

Thus, the following definitions of *Vehicle* are equivalent:

:Prop Vehicle

:Inst Type

:sup ConcreteObject

:coveredBy (listof Car Train Plane)

(disjointSubProps Vehicle)

:Prop Vehicle

:Inst Type

:sup ConcreteObject

:partitionedBy (listof Car Train Plane)

B.3 Relations

B.3.1 General

Relations are the glue that holds objects together. Properties by themselves do not convey much. Properties are given meaning by the relationships between their instances. For example, the sup relation previously discussed defines a hierarchy of properties.

Unlike UML and other modelling paradigms, KFL does not have "attributes"; entities are described by relating them to each other and to primitive data values.

A relation declaration in KFL is as follows:

:Rel brotherOf

:Inst BinaryRel

:Sig Person MalePerson

:Args "sibling" "brother"

:name "is a brother of"

Like properties, relation declarations have three required fields:

:Rel

:Inst

:Sig

Just as the :Inst line in a property declaration gave the kind of property, the :Inst line in a relation declaration gives the kind of relation. Usually the type of relation specifies the number of arguments the relations takes, also known as the relation's arity. In this case, the new relation is an instance of the type *BinaryRel*, meaning that it will relate two properties and have two argument places. Five properties classify relations by number of arguments:

— *UnaryRel*: one argument;

— *BinaryRel*: two arguments;

— *TernaryRel*: three arguments;

— *QuaternaryRel*: four arguments;

— *QuinaryRel*: five arguments.

It is valid to declare a relation with more than five arguments, although there is no special property to describe relations with arity greater than five. When none of the above applies, use :Inst Relation. Arity is not the only characteristic that can be used to select what to supply to an :Inst directive, but it is the most important, and convenient as well. Furthermore, more than one :Inst directive can be supplied if the relation fits more than one kind. If one kind is implied by another, the :Inst can be omitted for the first one, as it is redundant information otherwise.

The :Sig line shall have a property for each argument position. It restricts the arguments of that relation to objects that are instances of those properties.

B.3.2 Special Kinds of Binary Relations

If :Inst BinaryRel is used to declare a binary relation, this indicates that the relation has two argument places. Binary relations often have more complex logical behaviors, such as being symmetric or transitive. Each of the specialized binary relations below can be declared with an :Inst directive.

— *ReflexiveBR*: for all *x*, *(p x x)*

— *SymmetricBR*: if *(p x y)* then *(p y x)*

— *TransitiveBR*: if *(p x z)* and *(p z y)* then *(p x y)*

— *IrreflexiveBR*: for all *x*, *(not (p x x))*

— *AntisymmetricBR*: if *(p x y)* and *(p y x)*, then *x=y*

— *AsymmetricBR*: antisymmetric and irreflexive; i.e. if *(p x y)*, then *(not (p y x))*

— *EquivalenceBR*: reflexive, symmetric, and transitive; for example the relation =

— *PartialOrderBR*: reflexive, antisymmetric, and transitive

— *StrictPartialOrderBR*: irreflexive, asymmetric, and transitive

— *TotalOrderBR*: reflexive, antisymmetric, transitive, and every pair of entities in the domain is related: for every *x* and *y*, either *(p x y)* or *(p y x)*; for example using *lessThan* to compare integers

— *StrictTotalOrderBR*: irreflexive, asymmetric, transitive, and every pair of different entities in the domain is related: for every *x* and *y*, either *(p x y)*, *(p y x)*, or *(= x y))*

B.3.3 MetaProperty

Properties whose instances may only be properties are known as *meta-properties*. Meta-properties come in handy for a large number of modelling tasks. For example, *Airplane* can be declared to be a super-property of *Boeing747* and *Boeing747* to be an instance of *AirplaneModel*. In this case, *AirplaneModel* would be a meta-property, which would be declared as follows:

:Prop AirplaneModel

:Inst MetaProperty

:sup Type

:name "airplane model"

It is common for a meta-property *M* to relate to a (non-meta-) property *P* in a special way. In the above example, all instances of *AirplaneModel* should be sub-properties of *Airplane*. This can be enforced by adding the line:

:metaPropFor Airplane

to the above example. This means that any instance of *AirplaneModel* shall either be the property *Airplane* or a sub-property of *Airplane*.

B.3.4 Functional Relations and Cardinality

It is sometimes necessary to define a relation such that an object can only relate to a single other object. This can be done with the *functionalArg* property, e.g. in order to define the *hasFather* relation so that a person can only have one father, as follows:

:Rel hasFather

:Inst BinaryRel

:Sig Person MalePerson

:functionalArg 2

:name “has father”

There can be more than one functional argument on a relation. For example, a vehicle can have only one VIN, and a VIN can match only one vehicle:

:Rel hasVIN

:Inst BinaryRel

:Sig Vehicle VehicleID

:functionalArg 1

:functionalArg 2

:name “has vehicle identification number”

In the case of a binary relation, this is analogous to a one-to-one mapping. Relations with arity greater than 2 can also be functional. For example, *argProp* – the relation that captures the signature properties of a relation – is functional on position 3, since a relation has one property at a given position:

:Rel argProp

:Inst TernaryRel

:Sig Relation PosInt Property

:functionalArg 3

B.3.5 Rigid and Nonrigid Relations

Some relations are true either all of the time for given arguments, or true none of the time. These are called *rigid relations*. For example, *biologicalParentOf* would be a RigidRel:

:Rel biologicalParentOf

:Inst BinaryRel

:Inst RigidRel

:Sig Person Person

Other relations may be true some of the time. These are *nonrigid relations*. For example, a person may be an employee of multiple companies over his or her lifespan:

:Rel employeeOf

:Inst BinaryRel

:Inst NonRigidRel

:Sig Person Company

The XKS interprets all relations as nonrigid by default. For the best performance, it is recommended that all rigid relations are identified as instances of such.

B.3.6 Intensional and Extensional Relations

Normally all relations in an ontology may have assertions both concluded from rules, and asserted directly. (Rules are explained below.) For example, it can be asserted that Jerry and Beth are cousins by inserting (hasCousin Jerry Beth) to the database; meanwhile, the database can also conclude (hasCousin Mary Allen) from assertions that Mary's and Allen's parents are siblings and a rule that concludes cousinhood from those statements.

In limited cases, the manner in which certain relations may acquire assertions can be restricted. (Such restrictions are used to optimize queries involving these relations and provide data integrity controls.) A elation populated only by explicit assertions is an *ExtensionalRel*. A relation populated only by statements concluded from rules is an *IntensionalRel*.

:Rel canCommunicateWith

:Inst BinaryRel

:Inst IntensionalRel

:Sig CommsPlatform CommsPlatform

:Rel friendsWith

:Inst SymmetricBR

:Inst ExtensionalRel

:Sig Person Person

B.3.7 Forming the Relations Heirarchy

If it is necessary to define a specialization of a relation, the relation *supRel* (super-relation) can be used, e.g. *hasSibling* is already defined and a specialization, *hasBrother*: needs to be defined:

:Rel hasBrother

:Inst BinaryRel

:Sig Person MalePerson

:name “has brother”

:supRel hasSibling

B.4 Logic, Rules and Integrity Constraints

B.4.1 General

Beyond relations and properties, ontologies gain a lot of value from rules and integrity constraints. Rules allow new information to be deduced from existing statements. Integrity constraints prevent inconsistent statements.

B.4.2 Variables

Every nontrivial rule and integrity constraint uses variables in its expression. Variables look like ECLIF symbols that begin with a question mark. They work the same way in rules and integrity constraints as they do in ECLIF queries: each variable is treated as a blank in which to substitute a value from the assertions in the ontology, and if the same variable appears twice within a given rule or constraint, each appearance shall use the same value substitution.

B.4.3 Logical Operators

B.4.3.1 General

There are six primary logical operators in ECLIF used to write rules and integrity constraints: and (conjunction), or (disjunction), not (negation), exists (existential quantification), forall (universal quantification), and => (implication).

Conjunction, disjunction, negation, existential quantification, universal quantification, and the special operators

ifThenElse and ifExistsThenElse are used in rules and integrity constraints exactly as they are used in queries. Some examples using each are given below; see Statements and Queries in ECLIF for more information on them. Implication is of special importance for rules and constraints, and so is discussed in detail.

B.4.3.2 Implication

Implication is the basic building block for rules. The implication operator is an arrow made up of an equals sign followed by a greater-than sign. This operator always takes two arguments. The first argument is called the *antecedent*. The second argument is called the *consequent*.

Antecedents and consequents are *clauses*; they look like ECLIF queries. If the antecedent is true for a given variable assignment, or binding, then the conclusion is also true for that binding. For example:

; A father of someone is also that person's parent.

(=> (fatherOf ?x ?y)

(parentOf ?x ?y))

With the rule above, for every *fatherOf* statement the server finds with some value assignment to ?x and ?y, a

parentOf statement is also true for that same ?x and ?y. If the model included a statement (fatherOf Allen Brian), a query looking for the parent of Allen would return Brian.

Implications can also be written in reverse. The reverse implication operator is <=. It takes the conclusion first and the antecedent second. The meaning does not change, just as “if P is true, then Q is true” means the same as “Q is true if P is true”.

(<= (parentOf ?x ?y)

(fatherOf ?x ?y))

(Both of the above examples can of course be replaced by a :supRel directive in KFL, making *parentOf* the super-relation of *fatherOf*. This is what *supRel* means. Simple, frequently occurring rule patterns often have such replacements, as they have been included in the ULO.)

B.4.3.3 Chains of Implication

When finding the set of bindings for the antecedent of an implication, the XKS does not distinguish between extensional and intensional statements. In the example above, it is possible that the statement (fatherOf Allen Brian) was loaded extensionally through the Asserter or concluded by another implication. This allows the modeller to build complex chains of reasoning.

B.4.4 Rules

B.4.4.1 General

Rules deduce new information from existing statements. Rules are almost always implications, with an antecedent and a consequent. For every binding of the variables in the antecedent of an implication, a new statement formed by substituting those values into the consequent is added to the ontology.

B.4.4.2 Example Rules

```
; A parent of a parent is a grandparent. (conjunction)

(=>

(and

(parentOf ?y ?x)

(parentOf ?z ?y))

(grandparentOf ?z ?x))

; Children of sibling parents are cousins. (conjunction)

(=>

(and

(parentOf ?x1 ?y1)

(parentOf ?x2 ?y2)

(hasSibling ?x1 ?x2))

(hasCousin ?y1 ?y2))

; Bordering or overlapping regions are connected. (disjunction)

(<=

(connectedTo ?x ?y)

(or

(bordering ?x ?y)

(overlapping ?x ?y)))

; Projects that are not active are inactive. (negation)

(<=

(InactiveProject ?x)

(and

(Project ?x)

(not (ActiveProject ?x))))

; If there is someone you are a parent of, you are a parent. (existential quantification)

(=>
(exists (?y)
(parentOf ?y ?x))
(Parent ?x))
; Someone whom everyone knows is a completely popular person. (universal quantification)
(=>
(forall (?y)
(=> (Person ?y)
(knowsAbout ?y ?x)))
(CompletelyPopular ?x))
; This recursive rule calls expands every item in a list. (ifThenElse)
(=>
(and
(first ?incoming ?item)
(rest ?incoming ?new)
(expandItem ?item ?exItem)
(ifThenElse
(emptyList ?new)
(= ?new ?remainder)
(expandList ?new ?remainder))
(append ?remainder (listof ?exItem) ?expanded))
(expandList ?incoming ?expanded))
; This rule derives an employee's work address from an optional department address, or chooses the address of headquarters if there is no department. (ifExistsThenElse)
(=>
(and
(Employee ?x)
(ifExistsThenElse (?y)
(department ?x ?y)
(address ?y ?z)
(address Headquarters ?z)))
(workAddress ?x ?z))
```

B.4.5 Integrity Constraints

Integrity constraints (ICs) prevent inconsistent statements in the ontology. Like rules, they are typically implications, but they are interpreted differently. Every valid binding of the variables in the antecedent of an integrity constraint is substituted into the consequent. If the consequent holds for all of the bindings, the integrity check succeeds. If the consequent does not hold for one or more of the bindings, the integrity check fails, triggering an integrity violation message.

Integrity constraints look like rules except that they are followed by an :IC directive. This directive indicates how strong the constraint shoul d be and what error message should be displayed when the constraint is violated.

IC strength is one of four values.

— *weak* – A violation should indicate an irregularity, but not necessarily a problem.

— *soft* – A violation should not prevent a transaction commit. This is stronger than a weak constraint.

— *hard* – A violation should rollback a transaction.

— *adamant* – A violation indicates assertions that can harm the integrity of the logic engine.

(=> (and (hasParent ?x ?y)

(hasParent ?x ?z)

(/= ?y ?z))

(knowsWell ?y ?z))

:IC soft "The parents of a child should know each other well, but ?y and ?z do not."

(=> (and (disconnectedFrom ?x ?y)

(hasPart ?x ?a)

(hasPart ?y ?b))

(disconnectedFrom ?a ?b))

:IC hard "Parts of disconnected regions should be disconnected, but ?a and ?b are not."

Note that the error messages for each of the ICs above describes the failure case. Integrity constraints expresses what should be true about the data and are triggered only when the implication does *not* hold. For example, the first IC above requires that if any ?x in the model has two parents recorded, ?y and ?z, who are not equal, then it should be possible to prove (knowsWell ?y ?z) is true. If, during an IC check, a counterexample to this constraint is found, then the IC fires and displays an error message.

ICs become active as soon as the KFL file containing them is completely loaded. They are checked against the content of the model up to that point.

B.5 Functions

Functions are used to produce additional entities from one or more parameters. For instance, reference is often made to masses, volumes, and speeds with measures. Reference can be made to places by their latitude and longitude, and to some companies by their NASDAQ symbol. Entities like *ninePointTwoGrams* and *fortyDegreesNByTwelveDegreesE* would quickly clutter the model, as well as obscure the parameters. Functions semantically distinguish between a description and what is described and permit parameters to be used in reasoning.

The following is an example of a function definition for grams:

:Fun gram

:Inst UnaryFun

:Sig RealNumber -> MassQuality

This allows expressions to be written that denote masses, e.g. (gram 3.25). In general, a function term is written as the function name, followed by its arguments in order, with parentheses enclosing the entire term.

The following is a latitude/longitude function mapping two real numbers (presumably denoting degrees) to a region:

:Fun latlong

:Inst BinaryFun

:Sig RealNumber RealNumber -> Region

Like relations and properties, functions have three mandatory fields:

:Fun

:Inst

:Sig ->

Other than the primary directive, the biggest difference between a function declaration and a relation declaration is the arrow in the :Sig directive. Values on the left of the arrow indicate the arguments to the function; the value on the right of the arrow is the property instantiated by the entire function term. For example, the sole argument to the *gram* function above is required to be an instance of *RealNumber*, and the whole term, *(gram 3.25)*, is an instance of *MassQuality*.

The return property of a function is true for every term that can be constructed using the properties defined in the signature of the function. As a consequence, if any of arguments of a function are non-logical (have infinitely many values, such as *IntegerNumber*), then the function implies that the return property will also have infinitely many values. This is the case in the *MassQuality* example above. With this definition, the query (MassQuality ?x) will have infinitely many results (one for every *IntegerNumber*) and therefore return an invalid binding pattern exception.

Like relations, functions may instantiate properties which classify them by arity:

— *UnaryFun*: one argument;

— *BinaryFun*: two arguments;

— *TernaryFun*: three arguments;

— *QuaternaryFun*: four arguments;

— *QuinaryFun*: five arguments.

Functions with an arity greater than five are possible; they are declared as instances of *Function* without further classification.

B.6 Constants

B.6.1 General

In addition to properties (which classify entities), relations (which tie entities together) and functions (which enable more entities to be constructed from other entities), there are also entities which are not properties, relations, or functions, but are still important parts of the ontology. They may instantiate properties, relate to other entities, serve as arguments to functions, or simply exist. These entities are called ***constants***.

B.6.2 Choosing to declare a constant in KFL versus ECLIF

In general, constants are not part of the ontology, but are declared in ECLIF data files. Separating data elements, which typically describe entities in the world, from the abstractions of the model gives the modeller flexibility to add and remove data from the XKS without updating the ontological schema. The XKS is optimized to apply a relatively small number of statements defining the ontology over a relatively large number of statements describing the data.

For example, the property *Person*, describes the class of persons and would most likely be declared in a KFL file as part of the ontology. However the real-world entity *JohnDoe*, as an instance of *Person*, would most likely be declared in an ECLIF data file as a data element that populates the ontology.

The distinction between descriptive entities and the entities described is context-sensitive: what is part of the model of one ontology can be a data element in another. For example, although individual geographic regions are typically handled as data elements, an ontology capturing local building codes across several counties can include each county as part of the model. On the other hand, although classes typically map to ontology properties, an ontology describing several controlled vocabularies can choose to treat the individual vocabulary terms as data elements, even though they are represent classes.

Constants are only declared in KFL when they denote especially important entities in a given model; they will exist as long as the model does. A good rule of thumb is that only constants that participate in rules or integrity constraints in the ontology should be included as KFL constants.

B.6.3 Declaring Constants in KFL

If it is determined that a constant should be declared in KFL rather than ECLIF, use the :Const primary directive.

:Const Hawaii

:Inst USState

:name “Hawaii”

:rem “The US State of Hawaii.”

Like properties, relations, and functions, constants shall instantiate a property via the :Inst directive. A name is recommended, as is a remark.

Note that simply adding an ECLIF statement to the KFL file, such as

(USState Hawaii)

is not sufficient to declare an otherwise undeclared constant in KFL. The :Const directive shall be used.

B.7 Documentation

Certain relations are used primarily to provide documentation to the reader, rather than facilitate reasoning within the model. Two of them (*name* and *rem*) are the most commonly used, and have been mentioned previously. The following list of relations can be use to document entities in KFL.

— *name*: a string representing an entity, as it is likely to appear in English text.

— *Args* (shorthand for *argName*): one or more strings giving names to the arguments positions of the relations

— *abbrevString*: like a name, but abbreviated. Not every entity requires an *abbrevString*.

— *copyright*: denotes the copyright holder of a concept.

— *lex*: an English rephrasing of a term built on a property, relation or function, incorporating the arguments supplied to it. An argument is denoted by a question mark followed by its numerical position.

— *rem*: short for “remark”; a comment summarizing the meaning of an entity in readable text. Rems may contain fairly large amounts of text. They can span multiple lines, and may contain double quotes if they are escaped with a backslash (\”).

— *exampleRem*: a phrase intended to denote an example of an entity.

— *limitationRem*: a remark intended to explain constraints on the meaning of an entity, or in other words, what an entity does *not* mean.

— *referenceRem*: a remark citing the external reference material from which an entity was derived.

The following relation declaration provides an example of each.

:Rel tangentialProperPartOf

:Inst BinaryRel

:Sig Region Region

:name "tangential proper part"

:lex "?1 is a tangential proper part of ?2"

:Args "part" "whole"

:abbrevString "tpp"

:copyright company-42

:rem "A spatial part that is not identical with the whole region, yet shares a boundary with the whole."

:exampleRem "Texas is a tangential proper part of the US"

:limitationRem "Tangential proper parthood is only defined for regions; a person's skin is not a tangential proper partof his body."

:referenceRem "1996 Ooley and Tooley, 'Spatial Parts', Journal of Geospatial Ontology"

B.8 More on Directives

B.8.1 Declaration and Support Directives

KFL directives are divided into two main types, with the integrity constraint directive being a distinct third type.

Directives such as :Ctx, :Use, and :Rel are *declaration directives*; they are standalone, requiring no directive immediately preceding them. They are typically preceded by a blank line for readability. (The KFL parser does not require one.) Other declaration directives include :Prop for properties, :Fun for functions, and :Const for constants; these are explained later.

The "support directives" are either per-entity-being-declared or per-argument. The per-argument directives are for specifying the signature properties (:Sig), the argument names (:Args), and the argument binding modes (:Mode).

These directives use the entity named by the most recent primary directive and combine it with the per-argument information provided to the support directive. The :Sig Region Region directive above is equivalent to two ECLIF facts:

— (argProp tangentialProperPartOf 1 Region);

— (argProp tangentialProperPartOf 2 Region).

The meaning of all other support directives are dependent on the most recent primary directive. For example, the :Inst BinaryRel directive above specifies that *molecularWeight* is a binary relation, and relates to no other directives elsewhere in the KFL file.

B.8.2 Relations as Directives

The documentation support directives described above are not special KFL syntax at all; they are actually derived from the relations *RootCtx.name* and *RootCtx.rem*, respectively. Any relation can be used as a per-entity-being-declared directive. The directive :rel b, included in a KFL block declaring entity a is interpreted as the statement(rel a b). So the following are equivalent:

:Rel exampleRelation

:Inst BinaryRel

:rem “An example relation.”

:Rel exampleRelation

:Inst BinaryRel

(rem exampleRelation “An example relation.”)

The :rem directive above simply allows the writer to avoid having to write *exampleRelation* more than once.

Annex C
(informative)

ECLIF language reference

C.1 General

This annex contains information on ECLIF, the language used to author Highfleet ontologies. It is subdivided into subclauses covering Motivations and Goals, Concepts and Syntax, and a presentation of the formal grammar for ECLIF.

C.2 ECLIF Motivations and Goals

ECLIF is a language designed for ontology authoring and database query in Highfleet products. It is a variant of the ISO Common Logic CLIF concrete syntax, with constructs added to support ontology management, database applications and query processing.

This document is intended as an informal specification of ECLIF syntax aimed primarily at ontology authors and developers working with Highfleet products. For more information on the semantics of ECLIF expressions, see the CLIF ISO spec.

C.3 ECLIF Concepts

C.3.1 ECLIF Concepts Overview

ECLIF uses the following key concepts:

— First-order logic;

— Polish Notation;

— Ontologies and Entailment;

— Assertions and Queries;

— Terms.

C.3.2 First-order Logic

C.3.2.1 Statements

ECLIF, like all logical languages, describes the application domain in terms of ***sentences***. Sentences are intended to be true statements about the application domain. To start with an example, consider the simple sentence "(The) Cat is on (the) Mat". What is important is that, whatever the grammar of the language is, it will have to make reference to "Cat", "Mat" and the one being on the other. In ECLIF, this can be done in the following way:

(on Cat Mat)

The words are arranged between the parentheses in a particular way. First comes "on". This is the ***predicate***. A predicate is a word that is used to relate words to state that the objects those other words refer to in the application domain are in fact related by what the predicate refers to. In the above example, the word "Cat" refers to some cat in the domain, "Mat" refers to some mat in the domain, and "on" refers to the situation that objects find themselves in when one is on the other in space.

NOTE *Sentences* is defined as true statements about the application domain.

ECLIF is a predicate calculus. ECLIF provides for expression of statements about arbitrary subjects; i.e. assertions of named predicates about specific named things. An ECLIF identifier can be constructed for anything that can be named, so ECLIF statements can be about any such things.

A set of ECLIF statements is called an ECLIF ontology (defined more formally in ECLIF Ontologies).

C.3.2.2 Simple Statements

A simple statement represents a single relationship between the things denoted by its arguments. Predicates loosely correlate with verbs and arguments with nouns. For example:

"Alfred Nobel is a person."

(Person AlfredNobel)

— Predicate = "is a person"

— Argument = "Alfred Nobel"

The order of the arguments is significant, as it affects the meaning of the statement. For example:

"Stockholm is located in Sweden."

(locatedIn Stockholm Sweden)

— Predicate = "located in"

— Argument 1 = "Stockholm"

— Argument 2 = "Sweden"

An simple statement may have n-many arguments, as needed to represent the relationship.

"Sweden is located between Norway and Finland."

(locatedBetween Sweden Norway Finland)

Statement 2:

— Predicate = "located between"

— Argument 1 = "Sweden"

— Argument 2 = "Norway"

— Argument 3 = "Finland"

A familiar representation of a simple statement can be as a row in a table in a relational database. The table has an arbitrary number of columns, corresponding to the arguments of the statement. The name of the table corresponds to the predicate of the statement.

A single simple statement typically corresponds to a single sentence in English, although this is not necessarily the case. As in any language, there are many ways to record the same information in ECLIF, and some ways are clearer or more concise than others. This is an issue on ontology design: for more information, see one of the many good references on the subject.

C.3.2.3 Complex Statements

Simple statements in ECLIF are combined into complex statements with the standard first-order logic operators: conjunction (logical-AND), disjunction (OR), negation (NOT), quantification (FORALL, EXISTS), implication (IF-THEN).

Complex statements can also be formed using aggregation operators (MIN, MAX, SUM, AVG, COUNT, SET OF) or by temporal quantification (HOLDS IN). See ECLIF Complex Statements for more information on the construction of complex statements.

C.3.2.4 Polish Notation

ECLIF is a derivative of the CLIF (Common Logic Interchange Format) concrete syntax for ISO/IEC 24707 Common Logic (CL), which had its genesis in the KIF (Knowledge Interchange Format) language developed at Stanford University in the 1990s. Like CLIF and KIF it uses a polish notation (predicates are placed to the left of their arguments) with parentheses as punctuation, making ECLIF expressions look a bit like expressions of the Lisp programming language. See Figure C.1.

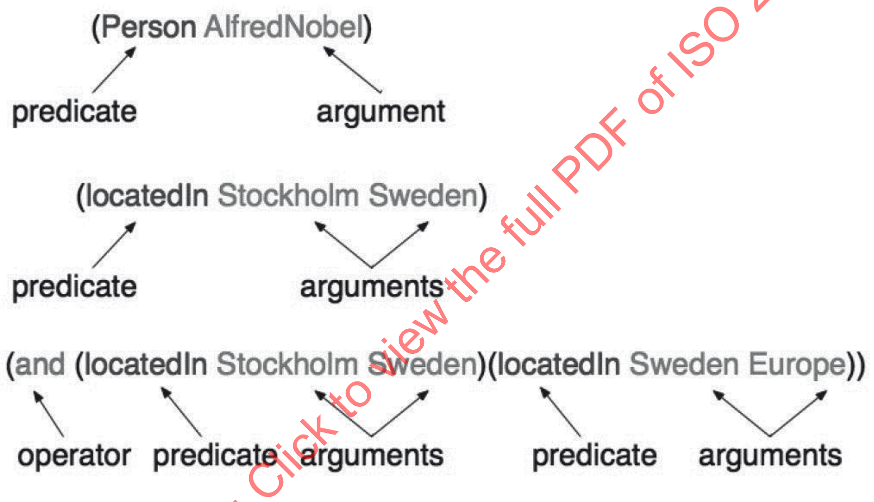


Figure C.1 — Polish notation examples

C.3.3 ECLIF Ontologies and Entailment

The assertion of an ECLIF statement says that the relationship indicated by the predicate holds between the things denoted by the arguments.

ECLIF statements entail other statements via the inference rules of Well-Founded Semantics. The most common mechanism of entailment is the if-then statement. For example:

"If A is located in B and B is located in C, then A is located in C."

(=>

(and

(locatedIn ?A ?B)

(locatedIn ?B ?C))

(locatedIn ?A ?C))

The assertion of an ECLIF ontology amounts to asserting all the statements in it, so the meaning of an ECLIF ontology graph is the conjunction (logical AND) of the statements it contains and all the statements entailed by that set. For example, if a user creates an ontology by recording the following three statements.

— *"If A is located in B and B is located in C, then A is located in C."*

— *"Stockholm is located in Sweden."*

— *"Sweden is located in Europe."*

(=>

(and

(locatedIn ?A ?B)

(locatedIn ?B ?C))

(locatedIn ?A ?C))

ECLIF Language Reference 4

(locatedIn Stockholm Sweden)

(locatedIn Sweden Europe)

a fourth statement is entailed, or *inferred*:

— "*Stockholm is located in Europe.*"

(locatedIn Stockholm Europe)

and the ontology is comprised of four statements:

— "*If A is located in B and B is located in C, then A is located in C.*"

— "*Stockholm is located in Sweden.*"

— "*Sweden is located in Europe.*"

— "*Stockholm is located in Europe.*"

(=>

(and

(locatedIn ?A ?B)

(locatedIn ?B ?C))

(locatedIn ?A ?C))

(locatedIn Stockholm Sweden)

(locatedIn Sweden Europe)

(locatedIn Stockholm Europe)

C.3.4 Built-in predicates

Some predicates (called ***built-in predicates***) reference plugins or other executable code; their semantics are not completely entailed by the set of statements.

C.4 ECLIF Assertions and ECLIF Queries

ECLIF statements are either assertions or queries. Assertions and queries are syntactically identical, differing only in interpretation.

Assertions are statements held to be true that comprise an ontology. Assertions that contain variables are called ***rules***.

Queries are statements of unknown truth value that are used to elicit information about the ontology. Queries that contain no variables are evaluated as either true or false depending on whether the statement is part of the queried ontology. Queries that do contain variables are evaluated as the set of all substitutions (or bindings) for those variables for which the query statement is true in the queried ontology.

C.5 ECLIF Terms

C.5.1 General

Arguments in ECLIF statements are ECLIF terms. ECLIF terms are used to express "data" objects. As opposed to statements that represent things that are *true* or *false*, terms are used to name things in the world of the application. In the example above, for example, the term AlfredNobel is used to name an object (it is intended to name a Swedish chemist).

Terms are divided roughly into three categories:

— *scalar terms* that name objects;

— *variables* that act as placeholders to be filled in by objects;

— *function terms* that act as complex composite names for objects.

C.5.2 Scalar terms

There are three kinds of scalar terms: names, numerals, and quoted strings.

Names like AlfredNobel are unquoted strings that are not numerals and contain only valid ECLIF name characters.

(See ECLIF Identifiers for more detail on valid ECLIF names.)

Numerals and quoted strings are also valid ECLIF terms. For example,

1.345

"hello"

C.5.3 Variables

Variables are placeholders for all values for which the statement containing them is true. Variables are preceded by a question mark, like this: ?x. For example, if an ontology contained these two statements

"The Nobel Prize in Physics was awarded to John Bardeen"

"The Nobel Prize in Physics was awarded to Serge Haroche."

(awarded NobelPhysicsPrize JohnBardeen)

(awarded NobelPhysicsPrize SergeHaroche)

then in the query

"The Nobel Prize in Physics was awarded to x."

(awarded NobelPhysicsPrize ?x)

?x would take the values, or ***bind*** to *JohnBardeen* and *SergeHaroche*.

C.5.4 Function terms

Functions are used to produce additional terms from one or more parameters. For instance, reference is often made to masses, volumes, and speeds with measures. Entities like *ninePointTwoGrams* would quickly clutter the model, as well as obscure the parameters.

Function terms consist of a function followed by one or more parameters, such as

9.2 grams

(grams 9.2)

Functions semantically distinguish between a description and what is described, and permit parameters to be used in reasoning. They can be used to name entities whose individual identity is not known, but are known to stand in relation to a known entity, as in:

Rita's boss

(bossOf Rita)

See ECLIF Function Terms for more information.

C.6 ECLIF Abstract Syntax

C.6.1 General

The subclauses below provide a complete account of the abstract syntax constructs of ECLIF, and explain the following:

— terms, i.e. constructs that name things;

— statements, i.e. constructs that express information about things.

Some static portions of the ontology are repetitive to write in ECLIF and are instead authored in KFL. See the Ontologies in KFL for more information

C.6.2 ECLIF Terms

C.6.2.1 General

There are four kinds of ECLIF terms: identifiers, literals, variables, and function terms.

C.6.2.2 ECLIF Identifiers

ECLIF identifiers are either enclosed or unenclosed.

a) **Unenclosed ECLIF Identifiers**

Unenclosed identifiers are composed of an optional context and a name:

context.name

The context is separated from the name by a '.' and is itself a name.

ECLIF names are composed of the following characters. ECLIF names cannot begin with a numeral.

'a'..'z'

'A'..'Z'

'\u00C0' .. '\u00D6'

'\u00D8' .. '\u00F6'

'\u00F8' .. '\u00FF'

'0'..'9'

'~' | '!' | '#' | '$' | '%' | '^' | '&' | '*' | '_' | '+' | '|' | '<' | '>' | '-' | '=' | '[' | ']' | '/'

b) **Enclosed ECLIF Identifiers**

Enclosed identifiers are enclosed in curly braces ('{' and '}') and may contain any UNICODE character. The characters preceding the first '.', with the exception of the '{', are interpreted as the context of the identifier, while the text following the first '.', with the exception of the '}', is interpreted as the name. If the identifier does not contain a '.', it is interpreted as having no context.

C.6.2.3 ECLIF Literals

ECLIF recognizes two kinds of literals: numerals and quoted strings. Quoted strings are enclosed in double quotes (").

C.6.2.4 ECLIF Variables

There are three types of ECLIF variables. Individual variables are most commonly used, while sequence variables are rarely used.

a) **Individual variables**

ECLIF individual variables are ECLIF names preceded by a question mark like this: ?var. They bind to a single value.

b) **Anonymous variables**

Anonymous variables are represented by a single question mark like this: ?. The values they bind to do not appear in the bindings of the clause that contains them.

c) **Sequence variables**

Sequence variables are precedes by the @ symbol like this: @seq. They bind to a sequence of values of variable length.

C.6.2.5 ECLIF Function Terms

A function term is any expression (f t1 ... tn) of n arguments (where each argument ti is a term and f is a function symbol of arity n). Function symbols are valid ECLIF identifiers.

NOTE In first-order logic, terms are defined as comprised solely of variables and function terms. Identifiers are 0-ary function symbols, and are thus terms.

C.6.3 ECLIF Statements

C.6.3.1 General

ECLIF statements are either simple (atomic) or complex. ECLIF statements are enclosed in parentheses.

C.6.3.2 ECLIF Simple Statement

A simple ECLIF statement consists of two parts:

— the predicate, which is an ECLIF identifier;

— one or more arguments, which are ECLIF identifiers, literals, variables, or function terms.

ECLIF Language Reference 8

C.6.3.3 ECLIF Complex Statements

C.6.3.3.1 General

A complex ECLIF statement consists of one or simple ECLIF statements combined with one or more operators:

- Boolean Operators:
  - Conjunction (*and*);
  - Disjunction (*or*);
  - Negation (*not*);
  - Implication (=>, <=, and <=>);
- Quantification Operators:
  - Universal Quantification (*forall*);
  - Existential Quantification (*exists*);
- Aggregation Operators:
  - *min* and *minf*;
  - *max* and *maxf*;
  - *minTerm* and *minTermf*;
  - *maxTerm* and *maxTermf*;
  - *sum* and *sumf*;
  - *avg* and *avgf*;
  - *count* and *countf*;
  - *setof* and *setofx*;
  - *equivalenceClasses* and *equivalenceClassesf*;
- Temporal Operators:
  - *holdsIn*;
- Special Operators:
  - *ifThenElse*;
  - *ifExistsThenElse*.

In the following, 'statement' refers to a simple or complex statement.