Behavior Language for Embedded Systems with Software Annex Sublanguage for AADL

DRAFT Version v0.28

Brian R Larson[1]

December 2, 2016

[1]brl@multitude.net brl@ksu.edu brianrlarson@comcast.net

Author's note:

> "It is not only that this proceeding can never lay claim to the very rare merit of a true philosophical popularity since there is no art in being intelligible if one renounces all thoroughness of insight; but also it produces a disgusting medley of complied observations and half-reasoned principles."

By "this proceeding" Kant means *Fundamental Principles of the Metaphysics of Morals*, from witch the quote was taken. By "disgusting medley of complied observations and half-reasoned principles" Kant means something like SysML.

Please forgive the brevity of mathematical definition from first principles, and the verbosity of the simplest examples. If my attempts to be clear, concise, and complete–*simultaneously*–are lacking, suggestions for improvement will be much appreciated.

Brian R Larson, December 2, 2016

Contents

List of Figures

List of Tables

Part I

BLESS Language Reference Manual

Chapter I 1

Introduction

I 1.1 Scope

(1) This language reference manual (LRM) defines three languages used as Architecture Analysis and Design Language (AADL) annex sublanguages, one for assertions ("Assertion"), one for subprograms ("subBLESS"), and one for threads ("BLESS"). *Assertion* declares properties of system-visible values over time. *subBLESS* defines behavior of subprograms through which proof outlines using Assertions may be interleaved. *BLESS* defines behavior threads.

(2) The goal for the three annex sublanguages is automatically-checked correctness proofs of AADL models of embedded electronic systems with software. Therefore all specifications, programs, and executions must be precise, mathematical objects, about which, true statements can be proved correct.

I 1.2 Overview

(1) BLESS specifications and behaviors can be attached to AADL models using an *annex subclause*. If applied to component type specifications, an annex subclause applies to all the associated implementations. If a component is extended, annex subclauses defined in an ancestor are applied to its descendants except when the later defines its own annex subclause of the same kind.

(2) An annex subclause can be specified for a specific *mode* by appending an `in modes` clause.[1] If the annex subclause is not mode specific, then it must be unique and it applies for all modes. If no mode-specific annex subclauses apply to a mode, then the behavior is undefined.[2] The foregoing applies to all AADL annex sublanguages, not just the three defined by this document.

[1] AS5506A §12 Modes and Mode Transitions

[2] Best never to have behavior undefined in possible modes; explicitly state is does nothing in modes when not active.

(3) The *Assertion annex sublanguage* defines temporal logic formulas over the behavior of systems over time. A particular temporal logic formula applied to an instance of behavior is like a boolean function that returns *true* if the behavior satisfies the formula, and *false* otherwise. Because Assertions are declarative rather than imperative, an Assertion may be applied to many different behaviors. Every behavior that satisfies an assertion is deemed to be correct, that is, the formula in the assertion, applied to the behavior, returned true. AADL defines both annex subclauses attached to particular components or features, and annex libraries which are not. The Assertion annex sublanguage defines grammar for both; Assertion annex libraries may have more than one Assertion.

(4) The *BLESS annex sublanguage* defines operational state-transition systems with guards and actions, augmented with Assertions. A state-transition system consists of a set of states, and a set of transitions from one state to another, and a set of local variables whose state may persist. Transitions may have a guard–a boolean expression that must be true for the transition to occur. Transitions may also include actions that may manipulate the values of variables, or send events and values on output ports.

(5) The *subBLESS annex sublanguage* defines the variable value manipulation of subprograms. SubBLESS is `action` from BLESS transitions, without port references. None of the local variables may have persistent values. All of the in parameters (including in out) must have valid values, frozen until completion of the subprogram. All of the out parameters (including in out) that may have had values assigned, become valid and available to the caller at completion of the subprogram.

(6) The subprogram behavior defined by subBLESS is considerably weaker than semantics for subprograms in the core language.[3] By eschewing persistent values, events on ports, timing, and data component access, subprograms devolve to value manipulation, completely specified by pre- and post-conditions. Assertions used for subBLESS specifications and proof outlines are just first-order predicates.

(7) BLESS does not redefine behavior of mode transitions from the core language.

(8) Default behavior of subprogram calls in AADL is synchronous. This means that a client remains blocked until completion of the remote subprogram. Because BLESS subprogram behavior annex subclauses cannot initiate or respond to events, and are time-free, and BLESS thread behavior annex subclauses never provide subprogram access, no other synchronization protocol for subprogram calls is necessary.

(9) Grammar productions follow AS5506B with the exception of literal symbols.[4] The standard way of writing literals in **`bold`** works fine for reserved words, but can be hard to see for symbols. To make literal symbols in grammar productions easier to see, they have been colored **`purple`**.

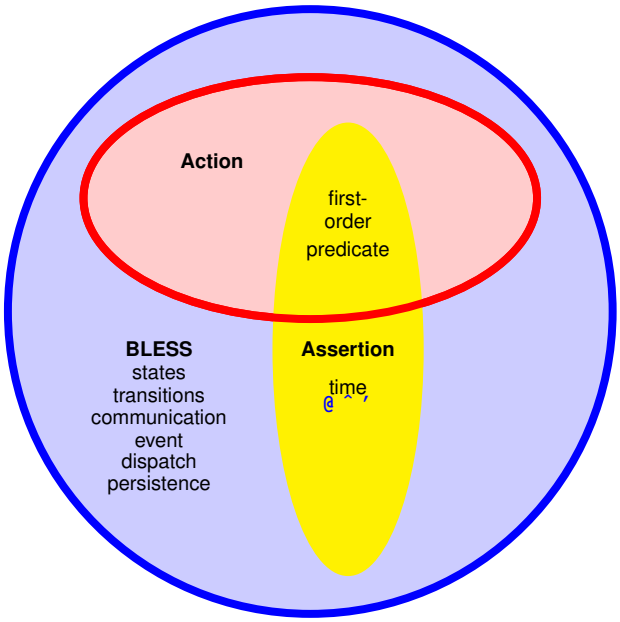


Figure I 1.1: Language Inclusion Relation Between BLESS, subBLESS, and Assertion

[3] AS5506A §5.2 Subprograms and Subprogam Calls
[4] AS5506A §1.5 Method of Description and Syntax Notation

Listings of AADL/BLESS examples have reserved words
from AADL in **red**, and those new to BLESS in **blue**.
There are a few exceptions, and some of the special symbols in BLESS are also **blue**.

(10) BLESS was created concurrently with standardization of the Behavior Annex sublanguage to AADL.[5] BLESS was created to be a similar as possible to BA to allow migration from BA. Differences between BLESS and BA are noted by footnotesand the index.

[5] SAE International document AS5506/2

Chapter I 2

Mathematics

(1) To prove correctness, a programming language must be mathematically defined. Therefore, the foundational mathematics must be derived from First Principles.

The foundational mathematics was deliberately selected to be as simple as possible, using only a few fundamental concepts from which all else flows. The following sections tersely declare these fundamental concepts, and is not meant as a textbook or tutorial. Many pieces of standard mathematics, like numbers and arithmetic are just assumed.[1]

Later, computation will be defines as satisfaction of interval-temporal logic formulas with lattices of states–not as a sequence of imperative commands. A lattice is a relation with some special properties. Thinking about programs as logic formulas, instead of traditional, imperative, sequential control flow, takes some getting used to.

Still, this document attempts to be self-contained, explicitly built on simple math defined herein, starting with sets.

I 2.1 Sets

(1) A *set* is a collection of elements. Finite sets may be specified by enumerating their elements between curly braces. For example, $\{true, false\}$ denotes the set consisting of the Boolean constants *true* and *false*. When enumerating elements of a set, "..." is used to denote repetition. For example, $\{1, \ldots, n\}$ denotes the set of natural numbers from 1 to n where the upper bound, n, is a natural number that is not further specified.

(2) More generally, sets are specified by referring to some property of their elements. $\{x \mid P\}$ denotes the set of all elements x that satisfy the property P. The bar, $\mid$, can be read as "such that". For example, $\{x \mid x$ is an integer and x is divisible by 2$\}$ denotes the infinite set of all even integers.

[1] as defined by *CRC Concise Encyclopedia of Mathematics*, Eric W Weisstein, editor, second edition, Chapman & Hall/CRC, 2003.

(3) For membership, $a \in A$ denotes that a is an element of the set A, and $b \notin A$ to denote that b is not an element of the set A.

(4) Some sets have customary symbols:

$\emptyset$ denotes the empty set;

$\mathbb{N}_0$ denotes the set of all natural numbers, including 0;

$\mathbb{Z}$ denotes the set of all integers;

$\mathbb{Q}$ denotes the set of rational numbers;

$\mathbb{R}$ denotes the set of real numbers;

$\mathbb{C}$ denotes the set of complex numbers.

$\mathbb{B}$ denotes the set $\{true, false\}$.

Fixed-point numbers are rational numbers with fixed divisor.

(5) In a set, one does not distinguish repetitions of elements. Thus $\{T, F\}$ and $\{T, T, F\}$ are the same set. Often it is convenient to refer to a given set when defining a new set. $\{x \in A \mid P\}$ is an abbreviation for $\{x \mid x \in A \text{ and } P\}$. Similarly, the order of elements is irrelevant. Two sets A and B are equal, $B = A$, if-and-only-if they have the same elements.

(6) Let A and B be sets. Then $A \subseteq B$ denotes that A is a *subset* of B; $A \cap B$ denotes the *intersection* of A and B; $A \cup B$ denotes the *union* of A and B; and, $A - B$ denotes the *difference* of A and B. The symbol $\equiv$ is used to define equivalence.

$A \subseteq B \equiv a \in B$ for every $a \in A$
$A \cap B \equiv \{a \mid a \in A \text{ and } a \in B\}$
$A \cup B \equiv \{a \mid a \in A \text{ or } a \in B\}$
$A - B \equiv \{a \mid a \in A \text{ and } a \notin B\}$

(7) Sets A and B are *disjoint* if they have no element in common, $A \cap B = \emptyset$.

(8) The definitions of intersection and union can be generalized to more than two sets. Let A_k be a set for every element k of some other set J in $\bigcap_{k \in J} \equiv \{a \mid a \in A_k \text{ for all } k \in J\}$ $\bigcup_{k \in J} \equiv \{a \mid a \in A_k \text{ for some } k \in J\}$

(9) For a finite set A, $\|A\|$ denotes the *cardinality*, or number of elements in A. For a non-empty, finite set $B \subset \mathbb{Z}$, $\min(B)$ denotes the *minimum* of all integers in B.

(10) $\mathbb{D}$ is the set of all possible constructed values, including strings, records, and arrays, formally defined in DI 4 Type.

I 2.2 Tuples

(1) For sets, the repetition of elements and their order is irrelevant. When ordering matters, ordered pairs and tuples are used. For elements a and b, not necessarily distinct, $\langle a, b \rangle$ is an *ordered pair* or simply pair. Then a and b are called *components* of $\langle a, b \rangle$. Two pairs $\langle a, b \rangle$ and $\langle b, c \rangle$ are equal, $\langle a, b \rangle = \langle c, d \rangle$ if-and-only-if $a = c$ and $b = d$.

(2) More generally, let n be any natural number, $n \in \mathbb{N}_0$. Then if $a_1, \ldots, a_n$ are any n elements, then $\langle a_1, \ldots, a_n \rangle$ is an *n-tuple*. The element a_k where $k \in \{1, \ldots, n\}$ is called the k-th element of $\langle a_1, \ldots, a_n \rangle$. An n-tuple $\langle a_1, \ldots, a_n \rangle$ is equal to an m-tuple $\langle b_1, \ldots, b_m \rangle$, $\langle a_1, \ldots, a_n \rangle = \langle b_1, \ldots, b_m \rangle$, if-and-only-if $m = n$ and $a_k = b_k$ for all $k \in \{1, \ldots, n\}$. Note that 2-tuples are pairs. Additionally, a 0-tuple is written as $\langle\rangle$, and a 1-tuple as $\langle a \rangle$ for any element a.

(3) The *Cartesian product*, $A \times B$ of sets A and B consists of all pairs, $\langle a, b \rangle$ with $a \in A$ and $b \in B$. The n-fold Cartesian product, $A_1 \times \ldots \times A_n$ of sets $A_1, \ldots, A_n$ consists of all n-tuples, $\langle a_1, \ldots, a_n \rangle$ with $a_k \in A_k$ for $k \in \{1, \ldots, n\}$. If all A_k are the same set A, then the n-fold Cartesian product, $A \times \ldots \times A$ is also written A^n.

I 2.3 Relations

(1) A binary *relation* R between sets A and B is a subset of their Cartesian product, $A \times B$, that is, $R \subseteq A \times B$. If $A = B$, then R is called a relation on A. For example, $\{\langle a, 1\rangle, \langle b, 2\rangle, \langle c, 2\rangle\}$ is a binary relation between $\{a, b, c\}$ and $\{1, 2\}$. More generally, for any natural number n, and n-ary relation R between sets $A_1, \ldots, A_n$ is a subset of the n-fold Cartesian product $A_1 \times \ldots \times A_n$, that is, $R \subseteq A_1 \times \ldots \times A_n$. Note that 1-ary relations are called unary relations, 2-ary relations are called binary relations, and 3-ary relations are called ternary relations.

(2) Consider a binary relation R on a set A. R is called *reflexive* if $\langle a, a \rangle \in R$ for every $a \in A$; it is called *irreflexive* if $\langle a, a \rangle \notin R$ for every $a \in A$. R is called *symmetric* if for all $a, b \in R$, whenever $\langle a, b \rangle \in R$ the also $\langle b, a \rangle \in R$; it is called *antisymmetric* if for all $a, b \in A$, whenever $\langle a, b \rangle \in R$ and $\langle b, a \rangle \in R$ then $b = a$. R is called *transitive* if for all $a, b, c \in A$ whenever $\langle a, b \rangle \in R$ and $\langle b, c \rangle \in R$ then also $\langle a, c \rangle \in R$.

(3) The transitive, reflexive *closure*, $R^\star$, of a binary relation R over a set A, is the smallest, transitive and reflexive, binary relation on A that contains R as a subset. The transitive, irreflexive closure, R^+, of a binary relation R over a set A, is the smallest, transitive and irreflexive binary relation that contains R as a subset.

$$R^\star \equiv \; R \subseteq R^\star \text{ and for all } a, b, c \in A \mid \begin{array}{c} \langle a, b \rangle \in R^\star \wedge \langle b, c \rangle \in R^\star \rightarrow \langle a, c \rangle \in R^\star \\ \langle a, a \rangle \in R^\star \end{array}$$

$$R^+ \equiv \; R \subseteq R^+ \text{ and for all } a, b, c \in A \mid \begin{array}{c} \langle a, b \rangle \in R^+ \wedge \langle b, c \rangle \in R^+ \rightarrow \langle a, c \rangle \in R^+ \\ \langle a, a \rangle \notin R^+ \end{array}$$

(4) The *relational composition*, $R_1 \circ R_2$, of relations R_1 and R_2 on a set A creates a new relation by combining them:

$R_1 \circ R_2 \equiv \{\langle a, c \rangle \mid \text{ there exists } b \in A \text{ with } \langle a, b \rangle \in R_1 \text{ and } \langle b, c \rangle \in R_2\}$

(5) For any natural number n, the n-fold relational composition, R^n, of a relation R on a set A is defined inductively:

$R^0 \equiv \{\langle a, a \rangle \mid a \in A\}$

$R^n \equiv R^{n-1} \circ R$ for $n > 0$.

$R^\star \equiv \bigcup_{n \in \mathbb{N}_0} R^n$

$R^+ \equiv R^\star - R^0$

(6) Membership of pairs in a binary relation is usually written in infix notation; instead of $\langle a, b\rangle \in R$, use aRb. Any binary relation $R \subseteq A \times B$ has an inverse $R^{-1} \subseteq B \times A$ such that $bR^{-1}a$ if-and-only-if aRb.

I 2.4 Functions

(1) Let A and B be sets. A *function* or mapping from A to B is a binary relation f between A and B with the following special property: for each element $a \in A$ there is exactly one element $b \in B$ such that afb. Usually functions use prefix notation for function application writing $f(a) = b$ instead of afb. For some functions postfix notation is used to write $af = b$. To indicate that f is a function from A to B write $f : A \to B$. The set A is called the *domain* of f and the set B is called the *range* or *co-domain* of f.

(2) Consider a function $f : A \to B$ and some set $X \subseteq A$. The *restriction* of f to X is denoted by $f[X]$ and defined as the intersection of f (which is a subset of $A \times B$) with $X \times B$: $f[X] \equiv f \cap (X \times B)$. Functions may have special properties. A function $f : A \to B$ is called *one-to-one* or *injective* if $f(a_1) \neq f(a_2)$ for any two distinct elements $a_1, a_2 \in A$. It is called *onto* or *subjective* if for every element $b \in B$ there exists an element $a \in A$ with $f(a) = b$. It is called *bijective* if it is both injective and subjective.

(3) Consider an n-ary function whose domain is a Cartesian product, $f : A_1 \times \ldots \times A_n \to B$. It is customary to drop tuple brackets when applying f to a tuple $\langle a_1, \ldots, a_n\rangle \in A_1 \times \ldots \times A_n$ writing $f(2, 3)$ instead of $f(\langle 2, 3\rangle)$.

(4) Consider a binary function whose domain and co-domain coincide, $f : A \to A$. An element $a \in A$ is called a *fixed point* of f if $f(a) = a$.

(5) Boolean logic can also be considered to be functions on $\{true, false\}$.

conjunction of a and b is $a \wedge b$;

disjunction is $a \vee b$;

implication is $a \to b$;

if-and-only-if is $a \leftrightarrow b$;

exclusive disjunction is $a \oplus b$;

complement is $\neg a$.

Table I 2.1: Boolean Function Truth Table

a	b	$a \wedge b$	$a \vee b$	$a \to b$	$a \leftrightarrow b$	$a \oplus b$	$\neg a$
false	*false*	*false*	*false*	*true*	*true*	*false*	*true*
true	*false*	*false*	*true*	*false*	*false*	*true*	*false*
false	*true*	*false*	*true*	*true*	*false*	*true*	*true*
true	*true*	*true*	*true*	*true*	*true*	*false*	*false*

I 2.5 Sequences

(1) Sequences are ordered sets. In the following, let A be a set. A *sequence* of elements of A of length $n > 0$ is a function $f : \{1, \ldots, n\} \to A$. A sequence is denoted by listing its values in order $a_1, \ldots, a_n$ where $a_1 = f(1), \ \ldots \ , a_n = f(n)$. Then the k-th element of the sequence $a_1, \ldots, a_n$ is a_k when $k \in \{1, \ldots, n\}$. A finite sequence is a sequence of any length $n \geq 0$. A sequence of length 0 is called the *empty sequence* and denoted ϵ. A countably-infinite sequence of elements from A is a function $\xi : \mathbb{N}_0 \to A$. To exhibit the general form of a countably-infinite sequence ξ is written $\xi : a_0 \ a_1 \ a_2 \ \ldots$ when $a_k = \xi(k)$ for all $k \in \mathbb{N}_0$. Then k is called the *index* of element a_k.

(2) Consider now a set of relations, $R = \{R_1, R_2, \ldots, R_{n-1}\}$, on a set A. For any finite sequence of elements of A, $a_1 \ \ldots \ a_n$, such that each element is related to the next by a relation in R, $a_1 R_1 a_2, \ a_2 R_2 a_3, \ \ldots, \ a_{n-1} R_{n-1} a_n$ can be written as a finite chain, $a_1 R_1 a_2 R_2 a_3 \ldots R_{n-1} a_n$ For example, using the relations = and < over $\mathbb{Z}$, a finite chain may be written $a_0 < a_1 = a_2 < a_3 < a_4$. Similarly for infinite sequences and infinite chains.

(3) A *permutation* of a sequence has the same elements in different order. In the following, let f and g be sequences of distinct[2] elements $f : \{1, \ldots, n\} \to A$ and $g : \{1, \ldots, m\} \to A$. The sequences are permutations of each other, $f \leftrightarrows g$, when they are the same length and have the same elements $f \leftrightarrows g \ \equiv \ n = m \wedge \{f(1), f(2), \ldots, f(n)\} = \{g(1), g(2), \ldots, g(m)\}$

I 2.6 Strings

(1) A set of symbols is often called an *alphabet*. A *string* over an alphabet A is a finite sequence of symbols from A. For example, 1 + 2 is a string over the alphabet $\{1, 2, +\}$. Syntactic objects like AADL annex subclauses are strings.

(2) The *concatenation* of two strings s_1 and s_2 yields the string $s_1 s_2$ formed by first writing s_1 and then s_2. A string t is called a *substring* of a string s if there exist strings s_1 and s_2 such that $s = s_1 t s_2$. Because s_1 and s_2 may be empty, every string is a substring of itself.

I 2.7 Partial Orders

(1) A *partial order* is a pair $(A, \sqsubset)$ consisting of a set A and a irreflexive, antisymmetric, and transitive relation $\sqsubset$ on A. The reflexive partial order is denoted $\sqsubseteq$. If $x \sqsubset y$ for some $x, y \in A$, then x is called *less than* y, or y is *greater than* x. Consider an element $a \in A$ and a subset $X \subseteq A$. When $a \in X$ and $a \sqsubset x$ for all $x \in X - \{a\}$, the a is called the *least element* of X. When $x \sqsubset b$ for all $x \in X - \{b\}$, then b is called an *upper bound* of X. Upper bounds of X need not be elements of X. Let U be the set of all upper bounds of X. Then a is called the *least upper bound* of X if a is the least element of U.

(2) A partial order $(A, \sqsubset)$ is called *complete* if A contains a least element, and for every ascending chain $a_0 \sqsubset a_1 \sqsubset a_2 \cdots$ of elements from A, the set $\{a_0, a_1, a_2, \ldots\}$ has a least upper bound.

[2] may not need restriction on repeated elements; that the sets are the same, repeated elements and all, may be enough; but then they have equal bags, not sets and that way madness lies

I 2.8 Graphs

(1) A *graph* is a pair $\langle V, E \rangle$ where V is a finite set of vertices $\{v_1, v_2, \ldots, v_n\}$ and E is a finite set of edges where each edge is a pair of vertices in V, $\{\langle v_m, v_l \rangle, \ldots, \langle v_j, v_k \rangle\}$. All graphs considered here are *directed* in the the order of vertices within the pair describing the edge is significant. The set of edges forms a relation on the set of vertices $E \subseteq V \times V$. The transitive, irreflexive closure of E, called E^+ is especially important.

I 2.9 Lattices

(1) A graph $\langle V, E \rangle$ is a *lattice* if the transitive, irreflexive closure of E, E^+, is an irreflexive partial order $\sqsubset$, it has a least element $\ell \in V$, and an upper bound $u \in V$. Executions of BA2015 actions create lattices.

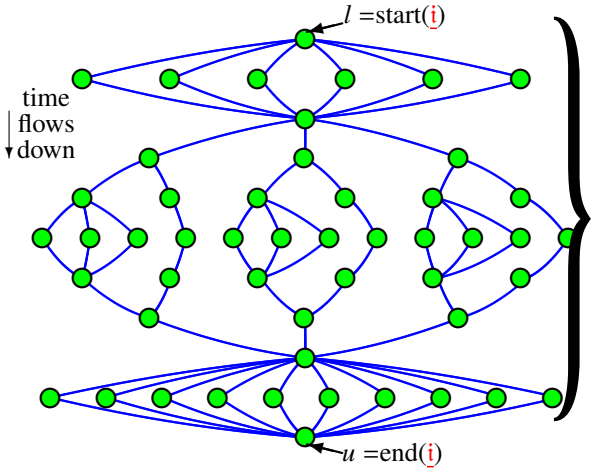


Figure I 2.1: Generic Lattice

(2) Depictions of lattices place the least element (a.k.a. "start") at the top, and the greatest element (a.k.a. "end") at the bottom. Directed edges use no arrowheads; instead, the edges are presumed to flow from the higher vertex to the lower vertex.

(3) Because lattices will be used to define intervals of time, when an unspecified-further lattice needs a name it is often called $\underline{i}$. Interval $\underline{i} = \langle V_{\underline{i}}, E_{\underline{i}} \rangle$ has a least element at the top called start($\underline{i}$) $\in V_{\underline{i}}$, and an upper bound at the bottom called end($\underline{i}$) $\in V_{\underline{i}}$. Like trees and conventional current, representations are reflected; least is top (because it's first) and upper most is bottom (because it's last). To define a notion of "before" is why all that stuff about irreflexive partial orders, least elements and upper bounds was needed.

(4) Every edge and vertex in the lattice is reachable from the least element; every edge and vertex in the lattice can reach the upper bound.[3] $\forall v \in V_{\underline{i}} - \ell \mid \ell \sqsubset v \ \wedge \ \forall v \in V_{\underline{i}} - u \mid v \sqsubset u$

(5) If there is a path between v_1 and v_2, then $v_1 \sqsubset v_2$, which means v_1 occurs *before* v_2. If there is no path between v_1 and v_2, $v_1 \not\sqsubset v_2 \wedge v_2 \not\sqsubset v_1 \rightarrow v_1 \parallel v_2$ then v_1 and v_2 may occur in either order, or concurrently.

[3] $\forall$ means "for all"

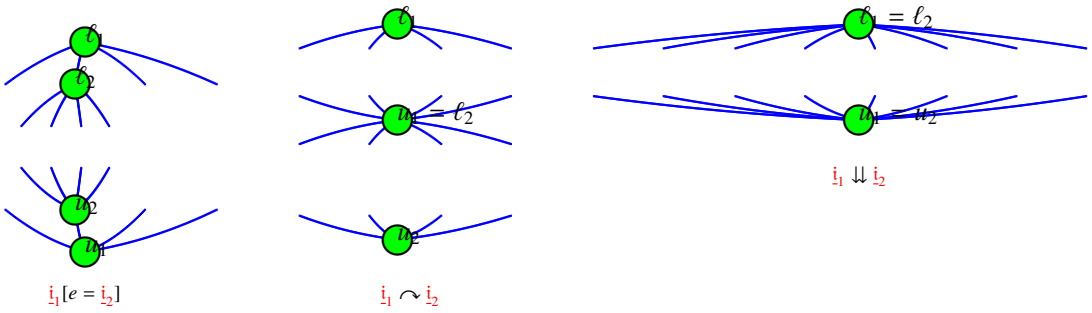


Figure I 2.3: Lattice Combinations

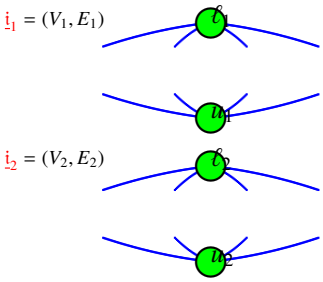


Figure I 2.2: Two Lattices

(6) Lattices may combined into new lattices in three ways: sequential, concurrent, and insertion. Consider two lattices $\underline{i}_1 = \langle V_1, E_1 \rangle$ and $\underline{i}_2 = \langle V_2, E_2 \rangle$ that have no vertices in common, $V_1 \cap V_2 = \emptyset$, least elements $\ell_1 \in V_1$ and $\ell_2 \in V_2$, and upper bounds $u_1 \in V_1$ and $u_2 \in V_2$.

(7) Their *sequential lattice combination*, $\underline{i}_1 \rightsquigarrow \underline{i}_2$, may be performed as follows: substitute u_1 for ℓ_2 in V_2 and E_2, then form the union of the vertices and edges, $\underline{i}_{sc} = \langle V_1 \cup V_2, E_1 \cup E_2 \rangle$.

(8) Their *concurrent lattice combination*, $\underline{i}_1 \Downarrow\!\Downarrow \underline{i}_2$, may be performed as follows: substitute u_1 for u_2, and ℓ_1 for ℓ_2 in V_2 and E_2, then form the union of the vertices and edges, $\underline{i}_{cc} = \langle V_1 \cup V_2, E_1 \cup E_2 \rangle$.

(9) Their *insertion combination*, $\underline{i}_1[e = \underline{i}_2]$, may be performed as follows: choose an edge $e \in E_1$, $e = \langle v_j, v_k \rangle$, remove it from from E_1, substitute v_j for ℓ_2, and v_k for u_2 in V_2 and E_2, then form the union of the vertices and edges, $\underline{i}_{ic} = \langle V_1 \cup V_2, E_1 \cup E_2 \rangle$.

I 2.10 Meaning

(1) The *meaning* of BA2015 language constructs is defined by giving an *interpretation* within a *context* for a *subject*:
$\mathfrak{M}_{context}[\![\text{subject}]\!] \equiv$ interpretation
where

context if given, is usually a state or set of states

subject is some construct in BA2015

interpretation is the defining formula for that subject, in that context

I 2.11 Time

(1) It is important to distinguish *model time* from *real time*. Model time is a mathematical abstraction useful for defining what a system is supposed to do. Real time is where the actual systems will operate. Model time is the same everywhere in the system, and is non-negative and real, $t \in \mathbb{R} \wedge t \geq 0$. Real time is different everywhere; synchronous temporal domains are limited in volume by the speed of light. Great care must be used for information that flows across temporal domain boundaries. Defining such temporal domains in AADL using precise, model time is means to make them nicely work together in real time when integrated into an operational system.

(2) Periods discretize time exactly in BA2015 by choosing a countably-infinite subset of $\mathbb{R}$,

$$P_d = \{p_j \mid p_j = d\,j \text{ for all } j \in \mathbb{N}_0\}$$

where d is the period's duration, and $d\,j$ is multiplication of d by j.

(3) Defining the system's *hyperperiod* is naturally the product of all of the different durations:[4]

$$h \equiv \prod_{d \in D} d$$

where D is the set of all different durations in the system.[5]

(4) The present instant is called *now*.

(5) Many entities in BA2015 have sensible time of occurrence, T, such as events.

$$T[\![\text{e}]\!] \equiv t \mid t \in \mathbb{R} \wedge t \geq 0 \wedge \text{ e occurs at } t$$

(6) Durations are continuous sets of non-negative real numbers. Commas denote open ranges that do not include the upper and/or lower bound: ".."=closed, includes both endpoints; ",,"=open both, neither endpoint included; ",."=open left, lower bound not included; ".,"=open right, upper bound not included.

$$\{l..u\} \equiv \{\, m \mid m \in \mathbb{R} \wedge m \geq l \wedge m \leq u \,\}$$
$$\{l,,u\} \equiv \{\, m \mid m \in \mathbb{R} \wedge m > l \wedge m < u \,\}$$
$$\{l,.u\} \equiv \{\, m \mid m \in \mathbb{R} \wedge m < l \wedge m \leq u \,\}$$
$$\{l.,u\} \equiv \{\, m \mid m \in \mathbb{R} \wedge m \geq l \wedge m < u \,\}$$

(7) Frequently, time will be used to define the context of meaning. Subscripted time as context notation $\mathfrak{M}_t[\![X]\!] \equiv \cdots$ is used to define the meaning for whatever X is, at a given time t. This notation is used in DI 5.3.2 and DI 5.4.1 to define temporal meaning for Assertions.

I 2.12 Values

(1) A *value* is a mathematical object. A *type* is a set of values (see DI 4 Type). Values used in BA2015 are the same as the AADL Data Modeling Annex and AADL property values (which have records, but not arrays).

[4] In cases where every system-level clock is a multiple of the same reference clock, then least-common multiple of different durations can suffice.

[5] $\prod$ means "product of", usually defined over all of the numbers in a given set

(2) Usually, values are singular: numbers in $\mathbb{N}_0$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, or $\mathbb{C}$; boolean in $\mathbb{B}$; a character (enclosed in apostrophes); a string (enclosed in quotation marks); or an enumeration literal (sequence of alphanumeric characters starting with a letter).

(3) More complex values are constructed from sets of pairs. Records are sets of pairs in which the first element is a record field identifier, and the second is the value of that field. Arrays are sets of pairs in which the first element is an integer index, and the second is the value of that index. Record and array values are functions in which the first element of the pair is unique. Array values generally constrain the second element of pairs to the same type. Of course, array element and record field values can be themselves be arrays or records, making arbitrarily complex values.

(4) The bottom sign, $\perp$, represents the absence of a value at a given time. The absence of value is also called *null*.

(5) The *clock operator* ^ determines when something has value. At time t,

$$\mathfrak{M}_t[\![\hat{p}]\!] \equiv \begin{array}{l} false \text{ when } \mathfrak{M}_t[\![\mathrm{p}]\!] = \perp \\ true \text{ otherwise} \end{array}$$

I 2.13 States

(1) BA2015 uses two kinds of states:

lattice states variable-value bindings during actions

machine states source and destination of transitions

I 2.13.1 Lattice States

(1) A *lattice state* is a set of pairs of variable names with values, with perhaps a time of the moment of occurrence.

$$L = (\{s_1, s_2, \ldots, s_m\}, t_S) \;\; s_k = \langle \mathrm{n}_k, \mathrm{v}_k \rangle \;\; t_S \in \mathbb{R} \wedge t_S \geq 0$$

Two states are equal if-and-only-if they have the same variables and those variables have the same values, but not necessarily the same time of occurrence. For states V and U,

$$V = (\{v_1, v_2, \ldots, v_m\}, t_V) \text{ and } U = (\{u_1, u_2, \ldots, u_m\}, t_U)$$

$$V = U \equiv \{v_1, v_2, \ldots, v_m\} = \{u_1, u_2, \ldots, u_m\}$$

Execution lattices satisfying temporal logic formulas have states as vertices (nodes).

I 2.13.2 Behavior States

(1) A *behavior state* is declared in the `states` section of thread behaviors (DI 6.2), and may be used as sources or destinations of transitions.

$$Q = (s, L)$$

Where s is a behavior state label, and L is a lattice state defining values of variables and time of occurrence.

I 2.14 Arithmetic

(1) Axiomatic definitions of arithmetic have been of interest to mathematicians for centuries. For BA2015, Peano arithmetic will be assumed for natural numbers, $\mathbb{N}_0$, extended appropriately for $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$.

I 2.15 Logic

(1) A *logic* is formal mathematical system for reasoning about a domain of interest. A logic consists of rules defined in terms of

symbols a set of graphical characters

formulas a set of sequences of symbols

axioms a set of distinguished formulas known to be true

rules a set of inferences to prove additional formulas from axioms, given formulas, and previously proved formulas

Not all sequences of symbols are formulas. Formulas are *well-formed* sequences of symbols. Formulas must be grammatically-correct to have meaning. A logic usually defines which sequences of symbols are formulas with a grammar.

To *satisfy* a formula means choosing values for its symbols such that the formula is true. A formula for which no choice of values for symbols is true is *unsatisfiable*. A formula always true regardless of chosen values for symbols is *tautology*.

The following formulas are assumed as axiomatic (eg. tautologies), with b, c, and d representing boolean-valued predcates, r being a bounded range, and j being an element in that range:[6]

Axiom 1 (*Complement*)**.** $b \equiv \neg(\neg b)$
Axiom 2 (*Excluded Middle*)**.** $b \vee \neg b$
Axiom 3 (*Contradiction*)**.** $\neg(b \wedge \neg b)$
Axiom 4 (*Implication*)**.** $a \rightarrow b \equiv \neg a \vee b$
Axiom 5 (*Equality*)**.** $a = b \equiv a \rightarrow b \vee b \rightarrow a$

[6] as in U.S. Pat. No. 5,867,649, col. 40, lines 40-70

Axiom 6 (*Disjunction*). $c \vee c \equiv c$
Axiom 7 (*Disjunction*). $c \vee true \equiv true$
Axiom 8 (*Disjunction*). $c \vee false \equiv c$
Axiom 9 (*Disjunction*). $c \vee (c \wedge b) \equiv c$
Axiom 10 (*Disjunction*). $b \vee c \equiv c \vee b$
Axiom 11 (*Disjunction*). $b \rightarrow (b \vee c)$
Axiom 12 (*Conjunction*). $b \wedge b \equiv b$
Axiom 13 (*Conjunction*). $b \wedge true \equiv b$
Axiom 14 (*Conjunction*). $b \wedge false \equiv false$
Axiom 15 (*Conjunction*). $b \wedge (c \vee b) \equiv b$
Axiom 16 (*Conjunction*). $b \wedge c \equiv c \wedge b$
Axiom 17 (*Conjunction*). $(b \wedge c) \rightarrow b$
Axiom 18 (*Distribution*). $b \vee (c \wedge d) \equiv (b \vee c) \wedge (b \vee d)$
Axiom 19 (*Distribution*). $b \wedge (c \vee d) \equiv (b \wedge c) \vee (b \wedge d)$
Axiom 20 (*Universal Quantification*). $\forall j \in r \mid (b \wedge c) \equiv (\forall j \in r \mid b) \wedge (\forall j \in r \mid c)$
Axiom 21 (*Existential Quantification*). $\exists j \in r \mid (b \vee c) \equiv (\exists j \in r \mid b) \vee (\exists j \in r \mid c)$

I 2.16 Computation ≡ Satisfaction

(1) BA2015 defines computation as satisfaction of interval temporal logic formulas by lattices of states.

$\mathfrak{M}_{\mathrm{i}}[\![w]\!] = true$ *(construct an interval* i *such that the formula w is true)*

(2) Each BA2015 program may be satisfied by a huge number of different lattices, all of which arrive at the same result.[7] The set of satisfying lattices is so large, it is effectively countably infinite. However, it suffices to consider the canonical member of the set of satisfying lattices–the shortest and bushiest lattice. Maximizing opportunities for concurrent execution is paramount for supercomputing, but embedded systems with multi-core systems-on-chip may benefit from rich opportunities for concurrent execution.

(3) For just the Action annex sublanguage, the states defined in I 2.13 suffice and need no more reference in time than its position in the lattice. For satisfying lattices of states for the BA2015 annex sublanguage need time-stamps. Therefore, the set of variable-value pairs comprising a state is augmented with a real-valued time-stamp.

$$L = (\{s_1, s_2, \ldots, s_m\}, t_s) \quad s_k = \langle \mathrm{n}_k, \mathrm{v}_k \rangle$$

where t_s the time lattice state L is created. Lattice state L says nothing about the values of variables at any time other than t_s. Other lattice states could have occurred infinitesimally earlier or later. Usually, only the time-stamps of least elements and upper bounds of lattices matter, and will be ignored when they don't.

I 2.17 Clock

(1) A *clock* is a boolean-valued operator over machine states, S, variables, V, and ports, P which is true only when its subject has a (non-null) value: $\hat{x} \equiv \mathfrak{M}_{now}[\![x \neq \perp]\!]$. The set of all possible clock formulas, F_{SVP}, is defined with the grammar of `predicate` (DI 5.3) augmented with the following as boolean values:

[7] Every satisfying lattice will have equal states for their least elements (start) and upper bounds (end).

1. Ports: $\hat{p} \equiv \mathfrak{M}_{now}[\![p \neq \perp]\!]$ for port p.
2. Variables: $\hat{v} \equiv \mathfrak{M}_{now}[\![v \neq \perp]\!]$ for variable v.
3. States: $\hat{s} \equiv \mathfrak{M}_{now}[\![State(s)]\!]$ where $State(s)$ means the state machine is currently in state s.
4. Never: $c = \hat{\mathbf{0}} \equiv (\forall t = now : \neg c)$ where c is a clock formula c∈F_{SVP}.
5. Always $c = \hat{\mathbf{1}}_{SVP} \equiv (\forall t = now : c)$ where c is a clock formula c∈F_{SVP}.
6. Difference: f ˆ g ≡ (f̂ **and not** ĝ) for any f,g∈F_{SVP}.
7. Next: v'=e when used in a guard of automata (DI 2.19) means that variable v will hold the value of expression e upon entering the destination state.[8]

[9]

(2) For example, the formula â **and** b̂ = **0̂** stipulates that ports a and b should never be assigned values simultaneously.
[10]

I 2.18 Timed Formula

(1) The clock formula set F_A of an automaton A is inductively extended to the *timed formula* set $F_A^\sharp$ with atoms pertaining to real-time properties of A. Real time properties $f^\sharp \in F_A^\sharp$ are formed with the atoms $n \in \mathbb{N}_0$ and t_p, (resp. t'_p) to mean the date (number of timing periods from start) of the previous, (resp. next) occurrence of p, with integer sub-expressions $f^\sharp + g^\sharp$, $f^\sharp - g^\sharp$, for all $f^\sharp$, $g^\sharp$ in $F_A^\sharp$, and with relations $f^\sharp = g^\sharp$, $f^\sharp < g^\sharp$ for all $f^\sharp + g^\sharp$ in $F_A^\sharp$. [11]

(2) The duration of the timing period need not be the period of the automaton (if it even has one), but is much shorter to discretize time for the system as a whole fine enough to accurately model communication in the real system.
[12]

(3) For example, the synchrony of two ports a, b is expressed as â = b̂ in F_A. In $F_A^\sharp$, it can be approximated by $d \leq t_a < d'$ and $d \leq t_b < d'$, by considering d to be the dispatch signal or date of the parent component. Literally, it means that the dates t_a and t_b of all occurrences of a and b must always occur between the dispatch date d and the next one. [13]

[8] Not to be confused with the use of ' as a temporal operator for periodic threads meaning next period.
[9] JP
[10] JP
[11] JP
[12] JP
[13] JP

I 2.19 Automata

(1) The behavior of a thread or device component defined with a BA2015 annex subclause is equivalent to an automation,[14] defined as a tuple:

$$A = (S_A, s_0, V_A, P_A, F_A, T_A, C_A)$$

where

S_A the set of initial, complete, execute, and final states of A

s_0 the initial state of A

V_A the set of local variables of A; $\mathbb{D}^{V_A}$ is the set of all possible values of local variables

P_A the set of ports of A, $P_A = I_A \cup O_A$, the input and output ports[15] $\mathbb{D}^{I_A}$ and $\mathbb{D}^{O_A}$ are sets of all possible values of inputs and outputs

F_{SVP} is the set of all possible clock formulas over vocabulary $W_A \equiv S_A \cup V_A \cup P_A \cup V'_A$

T_A the set of transitions $T_A \subset S_A \times \mathbb{D}^{V_A} \times \mathbb{D}^{I_A} \times F_{SVP} \rightarrow S_A \times \mathbb{D}^{V_A} \times \mathbb{D}^{O_A} \times F_{SVP}$
where $(s, V_s, I_A, g, d, V_d, O_A, f) \in T_A$

- source state $s \in S_A$,
- variable valuations $\forall v \in V_A : \mathfrak{M}_{V_s}[\![v]\!] \in \mathbb{D}$,
- input port values $\forall i \in I_A : \mathfrak{M}[\![i]\!] \in \mathbb{D}$, and
- source clock (guard) formula $g \in F_{SVP}$

map to

- the destination state $d \in S_A$,
- updated variable values $\forall v' \in V_A : \mathfrak{M}_{V_d}[\![v']\!] \in \mathbb{D}$,
- output port values $\forall o \in O_A : \mathfrak{M}[\![o]\!] \in \mathbb{D}$, and
- destination clock (finish) formula $f \in F_{SVP}$.

C_A the timing constraint $C_A \in F_{SVP}$ must equal $\hat{\mathbf{0}}$ defines timing and synchronization behavior.

[16]

(2) Let the set of all source behavior states of A (DI 2.13.2) and inputs be $Q_A \equiv S_A \times \mathbb{D}^{V_A} \times \mathbb{D}^{I_A}$. Equivalently, let the set of all destination behavior states and outputs of A be $Q'_A \equiv S'_A \times \mathbb{D}^{V'_A} \times \mathbb{D}^{O_A}$. Then $T_A \in Q_A \times F_{SVP} \rightarrow Q'_A \times F_{SVP}$. As shorthand, transitions may be represented as a quadruple $(s, g, d, f) \in T_A$ for *s*ource, *g*uard, *d*estination, and *f*inish, or a triple (s, g, d) where f is assumed to be true. [17]

[14] denoted by a capital letter, here 'A'
[15] in out ports are members of both I_A and O_A
[16] JP
[17] JP

(3) A transition that performs action w when changing from source state s with source clock formula (guard) g to destination state d with destination clock formula (finish) f is written as $T(s, g, d, f)[w]$. This notation will be used to define semantics for actions–particularly when defining complex actions in terms of simpler actions such as loops, action set, and action sequences. [18]

(4) In defining semantics with automata, a single automata may be translated into a *transition system* containing more than one transition, replacing the original transition. For $T \in T_A$:

$$T \Rightarrow T_1 \cup T_2 \equiv (T_A - T) \cup T_1 \cup T_2$$

(T is removed from the set of transitions T_A, to which T_1 and T_2 are added.)[19]

I 2.20 Synchronous Product

(1) The *synchronous product* of automata $A = (S_A, s_0, V_A, P_A, F_A, T_A, C_A)$ and $B = (S_B, t_0, V_B, P_B, F_B, T_B, C_B)$ is defined $A|B = (S_{AB}, (s_0, t_0), V_{AB}, P_{AB}, F_{AB}, T_{AB}, C_{AB})$ as follows:

$S_{AB} = S_A \times S_B$

$V_{AB} = V_A \cup V_B$

$P_{AB} = P_A \cup P_B$

$F_{AB} = F_A \vee F_B$[20]

$T_{AB} = \{((s_1, s_2), g_1 \wedge g_2, (d_1, d_2), f_1 \wedge f_2) \mid (s_1, g_1, d_1, f_1) \in T_A \wedge (s_2, g_2, d_2, f_2) \in T_B$[21]

$C_{AB} = C_A \vee C_B$

[22]

(2) Product is commutative, associative, has neutral element $(\{s\}, s, \emptyset, \emptyset, \emptyset, \emptyset, \hat{0})$ and, for deterministic automata, idempotent. [23]

(3) The synchronous composition (immediate connection) of two automata A and B communicating through a port p is represented by the product $A|FIFOp|B$ where

$$FIFOp = \{(s_1, v' = p_{in}, s_2, true), (s_2, true, s_1, p_{out} = v)\}$$

represents the point-to-point one-place first-in-first-out behavior of port p. A port queue of size n can be specified as a series of n one-place FIFO buffers.[24] [25]

---

[18] JP
[19] JP
[20] check with J.P.
[21] check with J.P.
[22] JP
[23] JP
[24] Wouldn't this force n steps even if the FIFO had only a single element?
[25] JP

I 2.21 Small Step

(1) A *small step* is execution of a single transition of an automaton (DI 2.19), $T = (s, v, i, g, d, v', o, f)$. A small step leaves a source behavior state (s, v), having state label s and (persistent) variable valuation v, with the values of `in` ports i, and guard clock formula g, to enter destination behavior state (d, v'), having state label d and updated variable valuation v', sending values to `out` ports o, satisfying finish clock formula f. [26]

I 2.22 Big Step

(1) A *big step* is a finite series of small steps, such that the source state of the first transition and the destination of the last transition are `complete` states. [27]

(2) Let s be the starting complete state, and e be the ending complete state of a big step B. Then B will be a sequence of small steps $T_1 \ldots T_n$ such that v_0 is the starting values of variables, v_n is the ending values of variables, i is the values received on `in` ports, o is the values sent on `out` ports, g is the dispatch condition, and f is the final formula: [28]

$T_1 = (s, v_0, i, g, s_1, v_1, \bot, true)$
$\ldots$
$T_j = (s_j, v_j, i, true, s_{j+1}, v_{j+1}, \bot, true)$
$\ldots$
$T_n = (s_n, v_n, i, true, e, v_{n+1}, o, f)$
which presumes that input values are frozen at dispatch time, and output values sent at completion. If inputs change during a big step, then replace the i in each transition with i_0, etc. as appropriate. [29]

I 2.23 Trace

(1) A *port trace* is a sequence of (possibly null) values of a port. A synchronous port trace has an entry for each atom $n \in \mathbb{N}_0$ as in DI 2.18 with $\bot$ when the port has no value: $(p : p_0, p_1, \ldots)$. An *execution trace* of a thread is a set of traces of its ports: $\{(p : p_0, p_1, \ldots)(q : q_0, q_1, \ldots)(r : r_0, r_1, \ldots)\}$. An asynchronous trace (marked with $^\sharp$) removes all the null values. [30]

(2) Consider execution traces $B_1 = \{(x : 2, \bot, \bot, \bot)(y : \bot, 2, 1, 0)\}$ and $B_2 = \{(x : 2, \bot, \bot)(y : 2, 1, 0)\}$. The asynchronous trace of B_1 is ${B_1}^\sharp = \{(x : 2)(y : 2, 1, 0)\}$. The asynchronous trace of B_2 is ${B_2}^\sharp = \{(x : 2)(y : 2, 1, 0)\}$. Therefore ${B_1}^\sharp = {B_2}^\sharp$. [31]

[26] JP
[27] JP
[28] JP
[29] JP
[30] JP
[31] JP

Chapter I 3

Lexicon

(1) Numeric literals, whitespace, identifiers and comments follow AS5506B §15 Lexical Elements.[1] String literals are enclosed in ` ' like LaTeX.

I 3.1 Character Set

(1) The only characters allowed outside of comments are the graphic_characters and format_effectors.

```
character ::= graphic_character | format_effector
    | other_control_character

graphic_character ::= identifier_letter | digit | space_character
     | special_character
```

(2) The character repertoire for the text of BLESS annex libraries, subclauses, and properties consists of the collection of characters called the Basic Multilingual Plane (BMP) of the ISO 10646 Universal Multiple-Octet Coded Character Set, plus a set of format_effectors and, in comments only, a set of other_control_functions; the coded representation for these characters is implementation defined (it need not be a representation defined within ISO-10646-1).

(3) The description of the language definition of BLESS uses the graphic symbols defined for Row00: Basic Latin and Row 00: Latin-1 Supplement of the ISO 10646 BMP; these correspond to the graphic symbols of ISO 8859-1 (Latin-1); no graphic symbols are used in this standard for characters outside of Row 00 of the BMP. The actual set of graphic symbols used by an implementation for the visual representation of the text of BLESS is not specified.

(4) The categories of characters are defined as follows:

[1] BA D.7(6)

```
identifier_letter
   upper_case_identifier_letter | lower_case_identifier_letter
upper_case_identifier_letter
   Any character of Row 00 of ISO 10646 BMP whose name begins
  Latin Capital Letter.
lower_case_identifier_letter
  Any character of Row 00 of ISO 10646 BMP whose name begins
  Latin Small Letter.
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
space_character
  The character of ISO 10646 BMP named Space.
special_character
  Any character of the ISO 10646 BMP that is not reserved for a control
  function, and is not the space_character, an identifier_letter,
  or a digit.
format_effector
  The control functions of ISO 6429 called character tabulation (HT),
  line tabulation (VT),  carriage return (CR), line feed (LF), and
  form feed (FF).
other_control_character
  Any control character, other than a format_effector, that is allowed
  in a comment; the set of other_control_functions allowed in comments
  is implementation defined.
```

(5) Table I 3.1 defines names of certain special_characters.

Table I 3.1: Special Character Names

Symbol	Name	Symbol	Name
"	quotation mark	#	number sign
=	equals sign	_	underline
+	plus sign	,	comma
-	minus	.	dot
:	colon	;	semicolon
(	left parenthesis	)	right parenthesis
[	left square bracket	]	right square bracket
{	left curly bracket	}	right curly bracket
&	ampersand	^	caret

I 3.2 Lexical Elements, Separators, and Delimiters

(1) The text of BLESS annex libraries, subclauses, and properties consist of a sequence of separate lexical elements. Each lexical element is formed from a sequence of characters, and is either a delimiter, an identifier, a reserved

word, a numeric_literal, a character_literal, a string_literal, or a comment. The meaning of BLESS annex libraries, subclauses, and properties depends only on the particular sequences of lexical elements that form its compilations, excluding comments.

(2) The text of BLESS annex libraries, subclauses, and properties are divided into lines. In general, the representation for an end of line is implementation defined. However, a sequence of one or more format_effectors other than character tabulation (HT) signifies at least one end of line.

(3) In some cases an explicit *separator* is required to separate adjacent lexical elements. A separator is any of a space character, a format_effector, or the end of a line, as follows:

- A space character is a separator except within a comment, or a string_literal.
- Character tabulation (HT) is a separator except within a comment.
- The end of a line is always a separator.

(4) A *delimiter* is either one of the following special characters

```
( ) [ ] { } , . : ; = * + -
```

or one of the following *compound delimiters* each composed of two or three adjacent special characters

```
:= <> != :: => -> .. -[ ]-> )~>
```

(5) The following names are used when referring to compound delimiters:

Delimiter	Name
:=	assign
<> !=	unequal
::	qualified name separator
=>	association
->	implication
-[	left step bracket
]->	right step bracket
)~>	right conditional bracket

I 3.3 Identifiers

(1) Identifiers are used as names. Identifiers are case sensitive.[2]

```
identifier ::= identifier_letter {[_] letter_or_digit}*

letter_or_digit ::= identifier_letter | digit
```

- An identifier shall not be a reserved word in either BLESS or AADL.
- Identifiers do not contain spaces, or other whitespace characters.

[2] Identifiers in AADL are case insensitive.

I 3.4 Numeric Literals

(1) There are four kinds of *numeric literal*: integer, real, complex, and rational. A *real literal* is a numeric literal that includes a point, and possibly an exponent; an *integer literal* is a numeric literal without a point; a *complex literal* is a pair of real literals separated by a colon; a *rational literal* is a pair of integer literals separated by a bar.

(2) Peculiarly, negative numbers cannot be represented as numeric literals. Instead unary minus preceding a numeric literal represents negative literals instead.

```
numeric_literal ::=
  integer_literal | real_literal | rational_literal | complex_literal
```

(3) Integer values are equivalent to `Base_Types::Integer` values as defined in the AADL Data Modeling Annex B.[3]

```
integer_literal ::= decimal_integer_literal | based_integer_literal
```

```
real_literal ::= decimal_real_literal
```

I 3.4.1 Decimal Literals

(1) A decimal literal is a numeric_literal in the conventional decimal notation (that is, the base is ten).

```
decimal_integer_literal ::= numeral
```

```
decimal_real_literal ::= numeral . numeral [ exponent ]
```

```
numeral ::= digit {[_] digit}*
```

```
exponent ::= (E|e) [+] numeral | (E|e) - numeral
```

(2) An underline character in a numeral does not affect its meaning. The letter E of an exponent can be written either in lower case or in upper case, with the same meaning.

(3) An exponent indicates the power of ten by which the value of the decimal literal without the exponent is to be multiplied to obtain the value of the decimal literal with the exponent.

I 3.4.2 Based Literals

(1) A based literal is a numeric_literal expressed in a form that specifies the base explicitly.

```
based_integer_literal ::= base # based_numeral # [ positive_exponent ]
```

```
base ::= digit [ digit ]
```

```
based_numeral ::= extended_digit [_] extended_digit
```

```
extended_digit ::= digit | A | B | C | D | E | F | a | b | c | d | e | f
```

[3] BA D.7(7)

(2) The base (the numeric value of the decimal numeral preceding the first #) shall be at least two and at most sixteen. The extended_digits A through F represent the digits ten through fifteen respectively. The value of each extended_digit of a based_literal shall be less than the base.

(3) The conventional meaning of based notation is assumed. An exponent indicates the power of the base by which the value of the based literal without the exponent is to be multiplied to obtain the value of the based literal with the exponent. The base and the exponent, if any, are in decimal notation. The extended_digits A through F can be written either in lower case or in upper case, with the same meaning.

I 3.4.3 Rational Literals

A *rational literal* is the ratio of two integers.

```
rational_literal ::=
  [ [-] dividend_integer_literal |  [-] divisor_integer_literal  ]
```

I 3.4.4 Complex Literals

A *complex literal* is a pair of real numbers for the real part and imaginary part.

```
complex_literal ::=
  [  [-] real_literal :  [-] imaginary_part_real_literal  ]
```

I 3.5 String Literals

(1) A string_literal is formed by a sequence of graphic characters (possibly none) enclosed between two string brackets: ‘ and ’.[4]

```
string_literal ::= "{string_element}*"

string_element ::= "" | non_string_bracket_graphic_character
```

(2) The sequence of characters of a string literal is formed from the sequence of string elements between the string bracket characters, in the given order, with a string element that is "" becoming " in the sequence of characters, and any other string element being reproduced in the sequence.

(3) A null string literal is a string literal with no string elements between the string bracket characters.

I 3.6 Comments

(1) A comment starts with two adjacent hyphens and extends up to the end of the line. A comment may appear on any line of a program.

[4] BLESS string literals are different from AADL string literals which use ” as string bracket characters.

```
comment ::= --{non_end_of_line_character}*
```

(2) The presence or absence of comments has no influence on whether a program is legal or illegal. Furthermore, comments do not influence the meaning of a program; their sole purpose is the enlightenment of the human reader.

Chapter I 4

Type

I 4.1 Ideal Types

(1) The AADL core language forces subprogram parameters to be some kind of data. The core grammar allows either data types, or data implementations.

(2) The AADL Data Modeling Annex[1] defines data component classifiers that express the type and representation of values exchanged by active AADL components. This allows interoperability between languages, operating systems, hardware architectures etc. Definitions of BLESS types include Data Modeling Annex equivalents.

(3) SAE International Document AS5506B defines property types in section §11.1.1. BLESS types are equivalent to AADL property types, removing "aadl" from its reserved word to get the BLESS equivalent. Often, values defined as AADL properties need to be used in behaviors and specifications. The equivalence between BLESS and AADL property types make type checking of AADL properties used in BLESS programs straightforward.

Table I 4.1: AADL and BLESS Type Equivalences

AADL Type	BLESS Type
`aadlreal`	`real`
`aadlinteger`	`integer`
`aadlboolean`	`boolean`
`aadlstring`	`string`

BLESS also has ideal types for `natural`, non-negative integers, `rational`, a ratio of integers, `time`, real number restricted to a type of time, and `complex`, a pair of reals.

[1] SAE International Document AS5506/2, January 2011

I 4.2 Types are Sets

(1) A *type* is a set of values. The universe of all values, V, contains all simple values like integers and strings, and all compound values like arrays, records, and variants. A type is a set of elements of V. Moreover when ordered by set inclusion, V forms a lattice of types. The top of this lattice is the set of all values or V itself. The bottom of the lattice is the empty set. The types used by any programming language is only a small subset of V. This chapter defines a type expression language, and mapping from type expressions to sets of values.

(2) Since types are sets, subtypes are subsets. Moreover the semantic assertion"T_1 is a subtype of T_2" corresponds to the mathematical condition $T_1 \subseteq T_2$ in V. Subtyping in the basis for type checking.

I 4.3 BLESS Type Grammar

(1) BLESS uses simple grammar to express simple, constructed types. All persistent values for variables will be statically mapped to memory addresses. No heap is needed. Stack frames will have fixed known size. Recursion prohibition limits stack depth. Much of the safety of BLESS-controlled systems comes from locking-down the type system.

Type expressions may be:

name reference to an AADL data component type having `BLESS::Typed` property.

number **`natural`** , **`integer`** , **`rational`** , **`real`** , **`complex`** , with optionally a range, a unit, or both.

enumeration set of identifier labels

array set of elements indexed by natural number(s)

record set of labelled elements

variant one element from a set of elements determined by an identifier discriminator

boolean either **`true`** or **`false`**

string sequence of characters, §I 3.5

Grammar

```
type ::= data_component_name | number_type | enumeration_type
  | array_type | record_type | variant_type | boolean | string[2]
```

(2) BLESS has no unit type. Therefore unit types in BLESS must be declared as AADL property types.

(3) An AADL package is provided, `BLESS_Types` that extend those in `Base_Types` package defined in the Data Model Annex document. In particular, `BLESS_Types` have a `BLESS_Properties::Supported_Operators` list of operator symbols for types that support arithmetic. Similarly, a `BLESS_Properties::Supported_Relations` list of relation symbols defines what relations can be applied to the type. More information about `BLESS_Types` and `BLESS_Properties` can be found in Chapter §I 12 BLESS Package and Properties.

[2] BLESSDiffers from BA: BA has no types

I 4.4 Data Components as Types

(1) A type may refer to a data component. Data components in other packages may be referenced by a *sequence of*[3] package identifiers separated by double colon. Implementation names are formed by suffixing an identifier to the name of the data component implemented separated by a period.[4]

Grammar

```
data_component_name ::=
  { package_identifier :: }* data_component_identifier
   [ . implementation_identifier ]
```

Legality Rule

(L1) A type name must refer to a visible data component.

Semantics

(S1) The meaning of a type name is the `BLESS::Typed` property of the data component to which it refers.

Example

```
data ResponseFactor  --to motion
  properties
    BLESS::Typed=>"integer 1..16";
end ResponseFactor;
```

I 4.5 Enumeration Type

(1) An *enumeration* type is a sequence of identifiers. Enumeration types are expressed as the reserved word **enumeration** followed by a sequence of identifiers enclosed in parentheses.

Grammar

```
enumeration_type ::= enumeration
    ( defining_enumeration_literal_identifier
    { , defining_enumeration_literal_identifier }* )
```

Property Type

(2) The AADL property type equivalent to **enumeration** (a b c) is **enumeration** (a b c).

Data Model

(3) The Data Model equivalent to **enumeration** (a b c) is

```
data EnumType
  Data_Model::Data_Representation => Enum;
  Data_Model::Enumerators => ("a", "b", "c");
end EnumType;
```

[3] **Reconciliation:** multiple identifier package names

[4] In AADL grammar, an italicized prefix of a component name is merely descriptive.

and then using `EnumType` in its place, prefaced by its package name if declared in a different package.

(4) In general, where *s* is a sequence of identifiers separated by spaces, *s′* is that same sequence of identifiers enclosed in double quotes separated by commas, *N* is an data component identifier, and *P* is a package prefix so that `P::N` is a legal type name, **`enumeration`** (*s*) ≡ *P* :: *N* such that in package *P* there is,

```
data N
  Data_Model::Data_Representation => Enum;
  Data_Model::Enumerators => (s');
end N;
```

Example

```
data Alarm_Type
  properties
    BLESS::Typed=>"enumeration (Pump_Overheated,Defective_Battery,Low_Battery,
      POST_Failure,RAM_Failure,ROM_failure,CPU_Failure,Thread_Monitor_Failure,
      Air_In_Line,Upstream_Occlusion,Downstream_Occlusion,Empty_Reservoir,
      Basal_Overinfusion,Bolus_Overinfusion,Square_Bolus_Overinfusion,No_Alarm)";
    Data_Model::Data_Representation => Enum;
    Data_Model::Enumerators => ("Pump_Overheated","Defective_Battery","Low_Battery",
        "POST_Failure","RAM_Failure","ROM_failure","CPU_Failure","Thread_Monitor_Failure",
        "Air_In_Line","Upstream_Occlusion","Downstream_Occlusion","Empty_Reservoir",
      "Basal_Overinfusion","Bolus_Overinfusion","Square_Bolus_Overinfusion","No_Alarm");
end Alarm_Type;
```

I 4.6 Number Type

(1) A *number type* is the name of a data component that behaves like an indivisible number, possibly restricted to a subrange, and may have units. The **`time`** type is equivalent to **`real`** , but restricted to time units.

Grammar

```
number_type ::=
  ( natural | integer | rational | real | complex | time )
  [ constant_number_range ] [ units aadl_unit_literal_identifier ]

constant_number_range ::=
  [ [-] numeric_constant .. [-] numeric_constant  ]

numeric_constant ::= numeric_literal | numeric_property
```

(2) Number types may be restricted to a range.

Legality Rules

(L1) A number type name (its component classifier reference) must have a corresponding data component.

(L2) The upper and lower bounds of a range must have the same type as that named.

(L3) A **`time`** type may only have units defined by `AADL_Project::Time_Units` : ps, ns, us, sec, min, hr.

Naming Rule

(N1) A unit identifier must correspond to an AADL property unit type.

Semantics

(S1) Number types are sets (§I 2.1):

natural ≡ $\mathbb{N}_0$ denotes the set of all natural numbers, including 0;

integer ≡ $\mathbb{Z}$ denotes the set of all integers;

rational ≡ $\mathbb{Q}$ denotes the set of rational numbers;

real ≡ $\mathbb{R}$ denotes the set of real numbers;

complex ≡ $\mathbb{C}$ denotes the set of complex numbers;

time ≡ $\mathbb{R}$ equivalent to real, having time units.

AADL Property

(3) AADL property types for integers and real numbers have the same grammar as BLESS, except that `aadlinteger` replaces `integer`, `aadlreal` replaces `real`, and constant number ranges may have superfluous unary plus.

(4) AADL property types define a `range_type` which does not define the end points of the range. There is no equivalent to this in BLESS; number types restricted to range must define range bounds.

Data Model

BLESS types are pure types, with unbounded magnitude. These are the closest (finite) Data Model representations.

natural `Base_Types::Natural`

integer `Base_Types::Integer`

rational `Base_Types::Float`

real `Base_Types::Float`

time `Timing_Properties::Time` (predeclared AADL property type)

complex

```
data Complex
  properties
  Data_Model::Data_Representation => Struct;
  Data_Model::Base_Type => (classifier(Base_Types::Float), classifier(Base_Types::Float));
  Data_Model::Element_Names => ("re", "im");  --real and imaginary parts
end Complex;
```

I 4.7 Array Type

(1) An *array type* is a collection indexed by natural numbers. The natural numbers in an array type expression denote the size of the array in successive dimensions. The sizes may be expressed as natural literals, or identifiers of natural number values.[5]

[5] Enumeration types for array indices were removed in v0.13 June 2010. Negative array indices are thus disallowed.

Grammar

```
array_type ::= array [ array_range_list ] of type

array_range_list ::= natural_range { , natural_range }*

natural_range ::= natural_number [ .. natural_number ]

natural_number ::=
 natural_integer_literal | natural_constant_identifier | natural_property
```

Legality Rule

(L1) For all ranges of natural numbers a and b, used to define ranges $a..b$, a must be at most b, $a \leq b$.

Data Model

(2) The Data Model for arrays uses the property `Data_Model::Slice` to define ranges for each array dimension rather than the property `Data_Model::Dimension` which only defines the array size. An single integer literal array dimension is interpreted as a range from zero. The Data Model equivalent to
`array [5, 0..15, May..October] of MyPackage::MyElementType`
is

```
data My_Three_Dimensional_Array
  properties
  Data_Model::Data_Representation => Array;
  Data_Model::Base_Type => (classifier (MyPackage::MyElementType));
  Data_Model::Slice => (0..4,   --5 becomes 0..4
    0..15,  --same range of natural literals
    May..October);  --May and October must identify natural values
end My_Three_Dimensional_Array
```

(3) In general, where n is a sequence of positive integer literals, integer ranges (i.e. 1..10), n' is that same sequence separated by commas having single integer literals replaced by integer ranges starting at zero, and E and T are data component identifiers, and P and R are package prefixes so that `P::T` and `R::E` are legal type names, `array [`n`] of` $R :: E \equiv P :: T$ such that in package P there is,

```
data T
  Data_Model::Data_Representation => Array;
  Data_Model::Base_Type => (classifier (R::E));
  Data_Model::Slice => (n');
end T;
```

(4) The Data Model also allows `Data_Model::Dimension` to be used which may only be a list of integer literals. The equivalent array type uses the same list without commas

Example

```
data Fault_Log  --holds records of faults
  properties
    BLESS::Typed => "array [PCA_Properties::Fault_Log_Size] of PCA_Types::Fault_Record";
    Data_Model::Data_Representation => Array;
    Data_Model::Base_Type => (classifier(Fault_Record));
    Data_Model::Dimension => (PCA_Properties::Fault_Log_Size);
end Fault_Log;
```

I 4.8 Record Type

(1) A *record type* is a collection of types indexed by identifier labels.

Grammar

```
record_type ::= record ( { record_field }+ )

record_field ::= defining_field_identifier : type ;
```

Data Model

The Data Model equivalent to `record (l1:T1; l2:T2; )` is

```
data My_Record
  properties
  Data_Model::Data_Representation => Struct;
  Data_Model::Base_Type => (classifier (T1), classifier (T2));
  Data_Model::Element_Names => ("l1", "l2");
end My_Record;
```

(2) In general, where S is a sequence of pairs of labels and type names, where each label is separated from its type name by a colon and followed by a semicolon,[6] B is a sequence of the second elements of those pairs (type names) of S enclosed in parentheses prefaced by `classifier` separated by commas,[7] and L is a sequence of the first elements of those pairs (labels) of S enclosed in double-quotes and separated by commas,[8] and P is package prefix so that `P::T` is a legal type name, `record ( S )` $\equiv$ $P :: T$ such that in package P there is,

```
data T
  Data_Model::Data_Representation => Struct;
  Data_Model::Base_Type => (B);
  Data_Model::Element_Names => (L);
end T;
```

(3) The Data Model Annex shows an alternate way to represent records (structs) using subcomponents of data component implementations to represent record elements. These are not supported by BLESS. Use the Data Model properties instead.

Example

```
data Fault_Record  --record of fault for log
  properties
   BLESS::Typed => "record (alarm:Alarm_Type; warning:Warning_Type;
        occurrence_time:BLESS_Types::Time;)";
    Data_Model::Data_Representation => Struct;
    Data_Model::Element_Names => ("alarm","warning","occurrence_time");
    Data_Model::Base_Type => ( classifier(Alarm_Type),classifier(Warning_Type),
        classifier(BLESS_Types::Time));
end Fault_Record;
```

[6] i.e. l1:T1; l2:T2; l3:T3

[7] i.e. classifier (T1), classifier (T2), classifier(T3)

[8] i.e. "l1", "l2", "l3"

I 4.9 Variant Type

A *variant type* holds a value of varying type specified by the value of a discriminant. A discriminant holds the value of one of the labels of the record fields, which then determines the type of the variant.

Grammar

```
variant_type ::= variant [ discriminant_identifier ] ( { record_field }+ )
```

Legality Rules

(L1) A value of variant type may only have its discriminant set at creation; discriminants may never be the subject of assignment.

(L2) A value of variant type has the type indicated by its discriminant; accessing that value as any other type is an error.

Data Model

The Data Model equivalent to `variant [d](c1:T1; c2:T2;)` is

```
data My_Variant
  properties
  Data_Model::Data_Representation => Union;
  Data_Model::Base_Type => (classifier (T1), classifier (T2) );
  Data_Model::Element_Names => ("c1", "c2" );
end My_Variant
```

(1) In general, where S is a sequence of pairs of labels and type names, where each label is separated from its type name by a colon and followed by a semicolon,B is a sequence of the second elements of those pairs (type names) of S enclosed in parentheses prefaced by `classifier` separated by commas,and L is a sequence of the first elements of those pairs (labels) of S enclosed in double-quotes and separated by commas,d is a discriminant identifier, and P is package prefix so that `P::T` is a legal type name, `variant [d]( S )` $\equiv$ $P :: T$ such that in package P there is,

```
data T
  Data_Model::Data_Representation => Union;
  Data_Model::Base_Type => (B);
  Data_Model::Element_Names => (L);
end T;
```

(2) The Data Model Annex shows an alternate way to represent variants (unions) using subcomponents of data component implementations to represent record elements. These are not supported by BLESS. Use the Data Model properties instead.

Example

```
data Event_Record  --record of event for log
  properties
    BLESS::Typed => "variant (start_patient_bolus:Start_Patient_Bolus_Event;
      stop_patient_bolus:Stop_Patient_Bolus_Event;)";
    Data_Model::Data_Representation => Union;
    Data_Model::Base_Type => (classifier (Start_Patient_Bolus_Event),
      classifier (Stop_Patient_Bolus_Event));
    Data_Model::Element_Names => ("start_patient_bolus", "stop_patient_bolus" );
end Event_Record;
```

I 4.10 Type Inclusion Rules

A type is included in another type $t \subseteq s$ when every value of one type is also a value of the other. $t \subseteq s \equiv \forall v \in t | v \in s$

In the following type rules,

- type expressions are denoted by s, t, and u,
- $s \rightarrow t$ is a function with domain s and range t;[9]
- type names by a and b, and element labels by L;
- V is the set of all values;
- d is a discriminant label;
- C is a set of inclusion constraints for types;
- $C.a \subseteq b$ is the set C extended with the constraint that type a is included in b;
- $C \models t \subseteq s$ is an assertion that from C we can infer $t \subseteq s$.

TOP ([TOP]). $C \models t \subseteq V$ *(every type is included in the set of all values)*
VAR ([VAR]). $C.a \subseteq t \models a \subseteq t$ *(what it means to extend a type constraint)*
BAS ([BAS]). $C \models a \subseteq a$ *(every type includes itself)*
TRANS ([TRANS]). $\dfrac{C \models s \subseteq t \ \wedge \ C \models t \subseteq u}{C \models s \subseteq u}$ *(type inclusion is transitive)*
FUN ([FUN]). $\dfrac{C \models s \subseteq s_1 \wedge C \models t \subseteq t_1}{C \models (s \rightarrow t) \subseteq (s_1 \rightarrow t_1)}$ *(a function type includes another when its domain includes the other's domain and its range includes the other's range)*
CAR ([CAR]). $\dfrac{C \models s \subseteq t \wedge n \leq m}{C \models \ \texttt{array}\,[n]\ \texttt{of}\ s \subseteq \ \texttt{array}\,[m]\ \texttt{of}\ t}$

(an array type includes another when its element type includes the other's element type, and the other has at most as many elements)
CARM ([CARM]). $\dfrac{C \models s \subseteq t}{C \models \ \texttt{array}\,[n_1, n_2, \ldots, n_k]\ \texttt{of}\ s \subseteq \ \texttt{array}\,[n_1, n_2, \ldots, n_k]\ \texttt{of}\ t}$

(a multi-dimensional array includes another when its element type includes the other's element type, and has exactly the same dimensions)
SLICE ([SLICE]). $\dfrac{C \models s \subseteq t \wedge d \leq a \wedge b \leq e}{C \models \ \texttt{array}\,[a..b]\ \texttt{of}\ s \subseteq \ \texttt{array}\,[d..e]\ \texttt{of}\ t}$

(an array slice includes another when its element type includes the other's element type, and its range includes the other's range)
SLICEM ([SLICEM]). $\dfrac{C \models s \subseteq t \wedge \forall i \in \{1, ..., k\} | d_i \leq a_i \wedge b_i \leq e_i}{C \models \ \texttt{array}\,[a_1..b_1, \ldots, a_k..b_k]\ \texttt{of}\ s \subseteq \ \texttt{array}\,[d_1..e_1, \ldots, d_k..e_k]\ \texttt{of}\ t}$

[9] see §I 10.8 Function Invocation for the form of AADL subprograms to be used as a function by BLESS. For functions with k parameters, s is a tuple of types $(s_1, \ldots, s_k)$.

(a multi-dimensional slice includes another when its element type includes the other's element type, and for each dimension its range includes the other's range)

RECD ([RECD]). $$\frac{C \models s_1 \subseteq t_1 \wedge \cdots \wedge C \models s_n \subseteq t_n}{C \models \ \textbf{record}\ (L_1 : s_1;\ \ldots\ L_n : s_n; \ldots L_m : s_m;) \subseteq \ \textbf{record}\ (L_1 : t_1;\ \ldots\ L_n : t_n;)}$$

(a record type includes another when the other has elements the same labels, and perhaps additional others, and for each label the corresponding element type includes the other's element type for that label)

VART ([VART]). $$\frac{C \models s_1 \subseteq t_1 \wedge \cdots \wedge C \models s_n \subseteq t_n}{C \models \ \textbf{variant}\ (L_1 : s_1;\ \ldots\ L_n : s_n;) \subseteq \ \textbf{variant}\ (L_1 : t_1;\ \ldots\ L_n : t_n;)}$$

(a variant type includes another when the other has elements the same labels, and for each label the corresponding element type includes the other's element type for that label)

I 4.11 Type Rules for Expressions

(1) Type rules for expressions determine types of expressions, especially complex names.

(2) Relation symbols, `= !=`, are treated as functions of pairs of the same element type to **boolean**, $(s, s) \to$ **boolean**, and are defined for every type s.

Relation symbols, `< <= >= >`, are treated as functions of pairs of the same element type to **boolean**, $(s, s) \to$ **boolean**, and are pre-defined for types **natural integer rational real complex**.

(3) Numeric operator symbols, `+ *`, are treated as functions of sequences of the same element type to that element type, $(s, \ldots, s) \to s$, and are pre-defined for types **natural integer rational real complex**.

Numeric operator symbols, `- / mod rem **`, are treated as functions of pairs of the same element type to that element type, $(s, s) \to s$, and are pre-defined for types **natural integer rational real complex**.

Unary `-` is arithmetic negation, $s \to s$, and is pre-defined for types **integer rational real complex**.

(4) Logical operator symbols, `and or xor`, are treated as functions of sequences of **boolean** to **boolean**, (**boolean**, ..., **boolean**) → **boolean**.

Logical operator symbols, `cand cor`, are treated as functions of pairs of **boolean** to **boolean**, (**boolean**, **boolean**) → **boolean**.

Unary **not** is complement, **boolean** → **boolean**.

(5) In the following type rules,

A is a set of type assumptions for variables;

C is a set of inclusion constraints for types;

V is the set of all values;

e is an expression;

s, t are types;

$s \rightarrow t$ is a function with domain s and range t;[10]

x is a variable;

L is a field label;

d is a discriminant label;

$A.x : t$ is the set A extended with the assumption that variable x has type t;

$C, A \models e : t$ means that from the set of constraints C and the set of type assumptions A, we can infer that expression e has type t;

$f : s \rightarrow t$ means f is a function with domain type s and range type t:[11]

```
subprogram f features x:in parameter s; y:out parameter t; end f;
```

ETOP ([ETOP]). $C, A \models e : V$ *(the type of every expression is included in the set of all values)*

EVAR ([EVAR]). $C, A.e : t \models e : t$ *(define extending a type assumption)*

ETRANS ([ETRANS]). $\dfrac{C, A \models e : t \ \wedge \ C \models t \subseteq u}{C, A \models e : u}$ *(type inclusion is transitive for expressions too)*

APPL ([APPL]). $\dfrac{C, A \models f : s \rightarrow t \wedge C, A \models x : s}{C, A \models f(x) : t}$ *(a function of type $s \rightarrow t$, applied to a parameter with type s, has type t)*

ECAR ([ECAR]). $\dfrac{C \models x : \textbf{array}\,[n]\ \textbf{of}\ s \ \wedge \ 0 \leq m < n}{C \models \ x[m] : s}$ *(indexing a variable of array type has the array's element type)*

ECARM ([ECARM]). $\dfrac{C \models x : \textbf{array}\,[n_1, n_2, \ldots, n_k]\ \textbf{of}\ s \quad 0 \leq m_1 < n_1 \wedge \cdots \wedge 0 \leq m_k < n_k}{C \models \ x[m_1, m_2, \ldots, m_k] : s}$ *(indexing a variable of multi-dimensional array type has the array's element type)*

SEL ([SEL]). $\dfrac{C, A \models x : \textbf{record}\,(L_1 : t_1; \ldots L_n : t_n;)}{C, A \models x.L_i : t_i \quad i \in 1..n}$ *(selecting a label of a variable having record type, has the type of the labeled element)*

VSEL ([VSEL]). $\dfrac{C, A \models x : \textbf{variant}\,[d](L_1 : t_1; \ldots L_n : t_n;)}{C, A \models x.L_i : t_i \text{ iff } x.d = L_i \quad i \in 1..n}$ *(selecting a label of a variable having variant type, has the type of the labeled element, only when the label is same as the discriminant)*

[10] For functions with k parameters, s is a tuple of types $(s_1, \ldots, s_k)$.

[11] see §I 10.8 Function Invocation for the form of AADL subprograms to be used as a function by BLESS.

Chapter I 5

BLESS Assertions

(1) Assertion properties may be attached to AADL component features, behavior states, interlaced through actions, or express invariants, and have three forms: predicates, functions, and enumerations.

(2) *Assertion annex libraries* hold labelled Assertions in AADL packages.

(3) *Assertion-predicates* declare truth.

(4) *Assertion-functions* declare value. Assertion-functions specify meaning for data ports or other things with value, or used with other Assertion-functions or Assertions.

(5) Meaning for enumeration-typed ports and variables use *Assertion-enumerations* –a kind of Assertion-function with special grammar associating enumeration identifiers with predicates.

I 5.1 Assertion Annex Library

(1) AADL packages may have annex libraries, not attached to any particular component.[1] An annex library is distinguished by the reserved word **annex** , followed by the identifier of the annex, and user-defined text between **{**** and **}** , terminated with a semicolon.

(2) An assertion annex library contains at least one assertion.

Grammar

```
assertion_annex_library ::= annex Assertion {** { assertion }+ **} ;
```

Example

AADL source code for an assertion annex library used in the definition of behavior of a pulse oximeter:

[1] AS5506B §4.8 Annex Subclauses and Annex Libraries

```
annex Assertion
{** --annex library holding BLESS Assertions
  <<SPO2_LOWER_LIMIT_ALARM: :SensorConnected and not MotionArtifact and
    (SpO2 < SpO2LowerLimit)>>
  <<HEART_RATE_LOWER_LIMIT_ALARM: :SensorConnected and not MotionArtifact and
    (HeartRate < HeartRateLowerLimit)>>
  <<HEART_RATE_UPPER_LIMIT_ALARM: :SensorConnected and not MotionArtifact and
    (HeartRate > HeartRateUpperLimit)>>
  <<SPO2_AVERAGE: :=
     --the sum of good SpO2 measurements
     (sum i:integer in -SpO2MovingAvgWindowSamples..-1 of
        (SensorConnected^(i) and not MotionArtifact^(i)??SpO2^(i):0))
     /  --divided by the number of good SpO2 measurements
     (numberof i:integer in -SpO2MovingAvgWindowSamples..-1
       that (SensorConnected^(i) and not MotionArtifact^(i)))>>
  <<SUPPL_O2_ALARM: :SupplOxyAlarmEnabled^0 and
    (SPO2_AVERAGE())^0 < (SpO2LowerLimit^0+SpO2LevelAdj^0)>>
  <<RAPID_DECLINE_ALARM: :AdultRapidDeclineAlarmEnabled  and
     (exists j:integer in 1 .. NUM_WINDOW_SAMPLES()
      that (SpO2 <= (SpO2^(-j) - MaxSpO2Decline)))>>
  <<MOTION_ARTIFACT_ALARM: :all j:integer
       in 0 ..PulseOx_Properties::Motion_Artifact_Sample_Limit
       are (MotionArtifact^(-j) or not SensorConnected^(-j))>>
  <<SPO2_TREND: : all s:integer in 1 ..num_samples
        are SpO2Trend[s]=(MotionArtifact^(-s) or
          not SensorConnected^(-s)??0:SpO2^(-s))>>
  <<HR_TREND: : all s:integer in 1 ..num_samples are HeartRateTrend[s]=
     (MotionArtifact^(-s) or not SensorConnected^(-s)??0:HeartRate^(-s))>>
  <<AXIOM_CR: :(num_samples-2)<(num_samples-1)>>
**};
```

I 5.2 Assertion

(1) In Behavior Language for Embedded Systems with Software (BLESS), an *assertion* is a temporal logic formula enclosed between << and >>.

Grammar

```
assertion ::=
  << ( assertion_predicate
     | assertion_function
     | assertion_enumeration
     | assertion_enumeration_invocation ) >>
```

I 5.2.1 Formal Assertion Parameter

(1) Assertions may have formal parameters.

Grammar

```
formal_assertion_parameter ::= parameter_identifier [ ~ type_name ]

formal_assertion_parameter_list ::=
  formal_assertion_parameter { , formal_assertion_parameter }*
```

Types for assertion parameters may be data component names, or the reserved word for one of the built-in BLESS types. Types and type checking is defined in .

Grammar

```
type_name ::=
  { package_identifier :: }* data_component_identifier
    [ . implementation_identifier ]
  | natural | integer | rational | real
  | complex | time | string
```

I 5.2.2 Assertion-Predicate

(1) Most Assertions will be predicates and may have a label by which other Assertions can refer to it. An *assertion-predicate* may have formal parameters. If so an assertion-predicate's meaning is textual substitution of actual parameter for formal parameters throughout the body of the assertion.[2]

Grammar

```
assertion_predicate ::=
  [ label_identifier : [ formal_assertion_parameter_list ] : ] predicate
```

(2) If an assertion has no parameters, occurrences of its invocation may be replaced by the text of its predicate. If an assertion has parameters, its label and actual parameters, may be replaced by its predicate with formal parameters replaced by actual parameters.

(3) Any entity may have its `BLESS::Assertion` property associated with the label of an assertion in an assertion annex library.

(4) Semantics for use of assertion-predicates, substitution of actual parameters for formal parameters, is defined in I 5.3.5, Predicate Invocation.

Example

AADL source code for Assertions used in the definition of behavior of a cardiac pacemaker:

```
<<LRL:x:   --Lower Rate Limit
    -- there has been a V-pace or a non-refractory V-sense
    exists t:BLESS_Types::Time
      -- within the previous LRL interval
      in (x-max_cci)..x   --MaxCCI is the maximum cardiac cycle interval
      -- in which a heartbeat was sensed, or caused by pacing
      that (vs or vp)@t  >>
<<LAST_A_WAS_AS:x: exists t:BLESS_Types::Time in x-max_cci..x that
  (as@t and  --A-sense at time t
    not (exists t2:BLESS_Types::Time in t,,x that  --no as or ap since
    (as@t2 or ap@t2))) >>
<<ATR_DURATION:d dur_met:  --wait to be sure a-tachy continues
  ATR_DETECT(d) and  --detection met at time d
  (dur > (numberof t:BLESS_Types::Time in d..dur_met that (vs@t or sp@t)))
  and  (all t2:BLESS_Types::Time in d..dur_met are not ATR_END(t2)) >>
```

[2] If an Assumption has a label, but no parameters, leave a space between to colons so the lexical analyzer emits two colon tokens, not one double-colon token.

I 5.2.3 Assertion-Function

(1) An *assertion-function* abstracts a value, usually numeric. Labeled assertion-functions may be used in assertion-expressions.

Grammar

```
assertion_function ::=
  [ label_identifier : [ formal_assertion_parameter_list ] ] :=
    ( assertion_expression | conditional_assertion_function )
```

(2) Semantics for use of assertion-functions, substitution of actual parameters for formal parameters, is defined in I 5.4.6, Assertion Function Invocation.

Example

An assertion-function defining a moving average, neglecting bad measurements:

```
<<SPO2_AVERAGE: :=
   --the sum of good SpO2 measurements
    (sum i:integer in -SpO2MovingAvgWindowSamples..-1 of
       (SensorConnected^(i) and not MotionArtifact^(i)??SpO2^(i):0))
    /  --divided by the number of good SpO2 measurements
    (numberof i:integer in -SpO2MovingAvgWindowSamples..-1
      that (SensorConnected^(i) and not MotionArtifact^(i)))>>
```

An assertion-function that determines the maximum cardiac cycle interval during atrial tachycardia response fall back:

```
<<FallBack_MaxCCI:dur_met x:= (x-dur_met)*((lrl-url)/fb_time)>>
```

I 5.2.4 Assertion-Enumeration

(1) An *assertion-enumeration* associates an assertion with elements (identifiers) of enumeration types. Assertion-enumerations are usually used as a data port property having enumeration type to define what is true about the system for different elements.

(2) An assertion-enumeration has one parameter for the enumeration value sent or received by an event data port

Grammar

```
assertion_enumeration ::=
  asserion_enumeration_label_identifier : parameter_identifier +=>
    enumeration_pair { , enumeration_pair }*

enumeration_pair ::= enumeration_literal_identifier -> predicate
```

(3) Semantics for use of assertion-enumerations, selection of enumeration pair matching given enumeration value, is defined in I 5.4.7, Assertion Enumeration Invocation.

Example

```
<<ALARM_TYPE: x +=> --has enumeration value of first element
          --when predicate in 2nd element is true
  Pump_Overheated->PUMP_OVERHEATED,
  Defective_Battery->DEFECTIVE_BATTERY,
  Low_Battery->LOW_BATTERY,
  POST_Failure->POST_FAIL,
  RAM_Failure->RAM_FAIL,
  ROM_failure->ROM_FAIL,
  CPU_Failure->CPU_FAIL,
  Thread_Monitor_Failure->THREAD_MONITOR_FAIL,
  Air_In_Line->AIR_IN_LINE,
  Upstream_Occlusion->UPSTREAM_OCCLUSION,
  Downstream_Occlusion->DOWNSTREAM_OCCLUSION,
  Empty_Reservoir->EMPTY_RESERVOIR,
  Basal_Overinfusion->BASAL_OVERINFUSION,
  Bolus_Overinfusion->BOLUS_OVERINFUSION,
  Square_Bolus_Overinfusion->SQUARE_OVERINFUSION,
  No_Alarm->NO_ALARM >>
```

I 5.3 Predicate

(1) A *predicate* is a boolean valued function, when evaluated returns *true* or *false*. An assertion claims its predicate is *true*. The meaning of the logical operators within a predicate have customary meanings. Universal quantification is defined in I 5.3.8, and existential quantification is defined in D I 5.3.9.

Grammar

```
predicate ::=
   universal_quantification |
   existential_quantification |
   subpredicate
      [ { and subpredicate }+
      | { or subpredicate }+
      | { xor subpredicate }+
      | implies subpredicate
      | iff subpredicate
      | -> subpredicate ]
```

Semantics

(S1) Where i is an interval, and A,B are predicate atoms:

$\mathfrak{M}_i[\![A$ **and** $B]\!] \equiv \mathfrak{M}_i[\![A]\!] \wedge \mathfrak{M}_i[\![B]\!]$ *(the meaning of* ***and*** *is conjunction)*
$\mathfrak{M}_i[\![A$ **or** $B]\!] \equiv \mathfrak{M}_i[\![A]\!] \vee \mathfrak{M}_i[\![B]\!]$ *(the meaning of* ***or*** *is disjunction)*
$\mathfrak{M}_i[\![A$ **xor** $B]\!] \equiv \mathfrak{M}_i[\![A]\!] \oplus \mathfrak{M}_i[\![B]\!]$ *(the meaning of* ***xor*** *is exclusive-disjunction)*
$\mathfrak{M}_i[\![A$ **implies** $B]\!] \equiv \mathfrak{M}_i[\![A]\!] \rightarrow \mathfrak{M}_i[\![B]\!]$ *(the meaning of* ***implies*** *is implication)*
$\mathfrak{M}_i[\![A$ **iff** $B]\!] \equiv \mathfrak{M}_i[\![A]\!] \leftrightarrow \mathfrak{M}_i[\![B]\!]$ *(the meaning of* ***iff*** *is if-and-only-if)*
$\mathfrak{M}_i[\![A$ **->** $B]\!] \equiv \mathfrak{M}_i[\![A]\!] \rightarrow \mathfrak{M}_i[\![B]\!]$ *(the meaning of* ***->*** *is implication)*

Example

```
<<(goodSamp[ub mod PulseOx_Properties::Max_Window_Samples] iff
```

```
   (SensorConnected^0 and not MotionArtifact^0)) and GS()>>
```

I 5.3.1 Subpredicate

(1) The meaning of `true`, `false`, and `not` within a predicate have customary meanings. Both parenthesized predicate and name may be followed by a time expression. Being able to express when a predicate will be true makes this a temporal logic able to express useful properties of embedded systems. Predicate invocation is defined in D I 5.3.5.

(2) The reserved word `def` defines a "logic variable" that represents an unknown, or changing value.

Grammar

```
subpredicate ::=
   [ not ]
   ( true | false | stop
   | predicate_relation
   | timed_predicate
   | event_expression
   | def logic_variable_identifier )
```

Semantics

(S2) Where i is an interval, and A is the rest of a subpredicate:

$\mathfrak{M}_i[\![$ **not** $A]\!] \equiv \neg\mathfrak{M}_i[\![A]\!]$ *(the meaning of* **not** *is negation)*
$\mathfrak{M}[\![$ **def** $D]\!] \equiv \exists D$ *(the meaning of* **def** *is definition)*
$\mathfrak{M}[\![$ **stop** $]\!] \equiv$ **stop?**
(the meaning of **stop** *is arrival of event at pre-declared* stop port *implicit for all AADL components)*

I 5.3.2 Timed Predicate

(1) In a *timed predicate*, the time when the predicate holds may be specified. The `'` means the predicate will be true one clock cycle (or thread period) hence; the `@` means the predicate is true when the subexpression, in seconds, is the current time; and the `^` means the predicate is true an integer number of clock ticks from `now`. Grammatically, time expression (I 5.3.3) and period-shift (D I 5.3.4) are time-free (e.g. no `'` `@` or `^` within). Grammar and meaning of a name is defined in I 10.3 Name.

Grammar

```
timed_predicate ::=
   ( name | parenthesized_predicate | predicate_invocation )
   [ ' | @ time_expression | ^ integer_expression ]
```

Legality Rules

(L1) When using `@`, the subexpression must have a time type such as, `Timing_Properties::Time`.

(L2) When using `^`, the value must have integer type.

Semantics

(S3) Where *P* is a name or a parenthesized predicate, *t* is a time, *d* is the duration of a thread's period, and *k* is a period-shift:

$\mathfrak{M}[\![$ `P@t` $]\!] \equiv \mathfrak{M}_t[\![$ `P` $]\!]$ *(the meaning of* `P@t` *is the meaning of* `P` *at time t)*
$\mathfrak{M}_t[\![$ `P^k` $]\!] \equiv \mathfrak{M}_{t+dk}[\![$ `P` $]\!]$
(the meaning of `P^k` *at time t, is the meaning of* `P` *, k period durations hence, or earlier if* $k < 0$*)*
$\mathfrak{M}_t[\![$ `P'` $]\!] \equiv \mathfrak{M}_t[\![$ `P^1` $]\!] \equiv \mathfrak{M}_{t+d}[\![$`P`$]\!]$ *(the meaning of* `P'` *at time t, is the meaning of* `P` *a period duration hence)*

Example

```
<<VS:x: --ventricular sense
  sv@x   --sensing ventricle enabled
  and v@x  --v-signal
  and not tnv@x  --not noisy
  and VRP_EXPIRED(x) >>  --not ventricular refractory period

  <<HR_TREND: : all s:integer in 1..num_samples
        are HeartRateTrend[s]=(MotionArtifact^(-s)
          or not SensorConnected^(-s)??0:HeartRate^(-s))>>
```

I 5.3.3 Time-Expression

(1) Both timed predicate (I 5.3.2 Timed Predicate) and timed expression (I 5.4.1 Timed Expression) require a *time-expression* when using `@` to define when a predicate holds. A time-expression must have type **`time`** , and must not use `@`.

Grammar

```
time_expression ::=
  time_subexpression
  | time_subexpression - time_subexpression
  | time_subexpression / time_subexpression
  | time_subexpression { + time_subexpression
  | time_subexpression { * time_subexpression }+

time_subexpression ::= [ - ]
  ( time_assertion_value
  | ( time_expression )
  | assertion_function_invocation )
```

Legality Rule

(L3) Every `time_expression` must have `time` type.

Semantics

(S4) Where *e* and *f* are time values (real),

$\mathfrak{M}_i[\![$`e+f`$]\!] \equiv \mathfrak{M}_i[\![e]\!] + \mathfrak{M}_i[\![f]\!]$ *(the meaning of + is addition)*
$\mathfrak{M}_i[\![$`e*f`$]\!] \equiv \mathfrak{M}_i[\![e]\!] \times \mathfrak{M}_i[\![f]\!]$ *(the meaning of * is multiplication)*
$\mathfrak{M}_i[\![$`e-f`$]\!] \equiv \mathfrak{M}_i[\![e]\!] - \mathfrak{M}_i[\![f]\!]$ *(the meaning of - is subtraction)*

$\mathfrak{M}_i[\![\texttt{e/f}]\!] \equiv \mathfrak{M}_i[\![e]\!] \div \mathfrak{M}_i[\![f]\!]$ *(the meaning of / is division)*
$\mathfrak{M}_i[\![\texttt{(e)}]\!] \equiv \mathfrak{M}_i[\![e]\!]$ *(the meaning of parentheses is its contents)*
$\mathfrak{M}_i[\![\texttt{-e}]\!] \equiv 0.0 - \mathfrak{M}_i[\![e]\!]$ *(the meaning of unary minus is complement)*

Example

```
<<PACE_ON_MaxCCI:x:     --no intrinsic activity, pace at LRL
  (vp or vs)@(x-max_cci)
      and --and not since
      not (exists t:BLESS_Types::Time
        in x-max_cci,,x
        --with a non-refractory ventricular sense or pace
        that (vs or vp)@t) >>
```

I 5.3.4 Period-Shift

(1) Both timed predicate (I 5.3.2) and timed expression (I 5.4.1) require a *period-shift* when using ^ to shift its time frame by number of thread periods (a.k.a. clock cycles).

```
integer_expression ::=
  [ - ]
  ( integer_assertion_value
  | ( integer_expression - integer_expression )
  | ( integer_expression / integer_expression )
  | ( integer_expression { + integer_expression }+ )
  | ( integer_expression { * integer_expression }+ ) )
```

Legality Rule

(L4) Every `period_shift` must have `integer` type.

Semantics

(S5) Where e and f are integers,

$\mathfrak{M}_i[\![\texttt{(e+f)}]\!] \equiv \mathfrak{M}_i[\![e]\!] + \mathfrak{M}_i[\![f]\!]$ *(the meaning of + is addition)*
$\mathfrak{M}_i[\![\texttt{(e*f)}]\!] \equiv \mathfrak{M}_i[\![e]\!] \times \mathfrak{M}_i[\![f]\!]$ *(the meaning of * is multiplication)*
$\mathfrak{M}_i[\![\texttt{(e-f)}]\!] \equiv \mathfrak{M}_i[\![e]\!] - \mathfrak{M}_i[\![f]\!]$ *(the meaning of - is subtraction)*
$\mathfrak{M}_i[\![\texttt{(e/f)}]\!] \equiv \mathfrak{M}_i[\![e]\!] \,/\, \mathfrak{M}_i[\![f]\!]$ *(the meaning of / is division, neglecting remainder)*
$\mathfrak{M}_i[\![\texttt{-e}]\!] \equiv 0 - \mathfrak{M}_i[\![e]\!]$ *(the meaning of unary minus is complement)*

Example

Examples of period shift from a pulse oximeter smart alarm:

```
<<GOOD: :goodCount=(numberof k:integer in lb..ub-1
  that (SensorConnected^(k-ub) and not MotionArtifact^(k-ub)))>>
<<CTR: :(all k:integer in lb..ub-1
    are spo2_hist[k mod PulseOx_Properties::Max_Window_Samples] = C(k-(ub-1)))
  and (totalSpO2=(sum k:integer in lb..ub-1 of C(k-(ub-1))))
  and (goodCount=(numberof k:integer in lb..ub-1
    that (SensorConnected^(k-(ub-1)) and not MotionArtifact^(k-(ub-1)))))
  and (all k:integer in lb..ub-1
    are goodSamp[k mod PulseOx_Properties::Max_Window_Samples] iff
      (SensorConnected^(k-(ub-1)) and not MotionArtifact^(k-(ub-1))))>>;
```

I 5.3.5 Predicate Invocation

(1) Predicate invocation allows labeled Assertions to be used by other Assertions.

(2) Predicates of the form `<<B:f:P>>` may be invoked as `B(a)` , where `B` is the label, `f` are formal parameters, `P` is a predicate, and `a` are actual parameters. Predicate invocations with single parameter may omit the formal parameter identifier.

Grammar

```
predicate_invocation ::=
  assertion_identifier ( [ assertion_expression |
    actual_assertion_parameter_list  ] )
actual_assertion_parameter_list ::=
  actual_assertion_parameter
    { , actual_assertion_parameter }*
actual_assertion_parameter ::=
  formal_parameter_identifier  :
  actual_parameter_assertion_expression
```

Semantics

(S6) Where B is an assertion label, $f_1 f_2 \ldots f_n$ are formal parameters, and P is a predicate that uses $f_1 f_2 \ldots f_n$, and

$\ll B : f_1 f_2 \ldots f_n : P \gg$ *(there is assertion B with predicate P & formal parameters f)*

then the meaning of predicate invocation is

$\mathfrak{M}_i[\![\texttt{B(f1:a1, f2:a2, ... fn:an)}]\!] \equiv \mathfrak{M}_i[\![B \mid \frac{f_1}{a_1} \mid \frac{f_2}{a_2} \cdots \mid \frac{f_n}{a_n}]\!]$

(the meaning of a predicate invocation is the meaning of the predicate of the assertion with the same label having actual parameters substituted for formal parameters)

Naming Rule

(N1) The identifier of a predicate invocation must be the label of a visible or imported assertion.

Example

Examples of predicate invocation from a cardiac pacemaker:

```
<<VP(now) and URL(now)>>
<<ATR_DURATION(d:detect_time, dur_met:now)>>
```

I 5.3.6 Predicate Relations

(1) *Predicate relations* have conventional meanings. The `in` operators tests membership of a range.

```
predicate_relation ::=
    assertion_subexpression relation_symbol
      assertion_subexpression
  | assertion_subexpression in assertion_range
  | shared_integer_name +=  assertion_subexpression
```

```
relation_symbol ::= = | < | > | <= | >= | != | <>
```

(2) The *range* is defined with ordinary subexpressions (I 10.5). Ranges may be open or closed on either or both ends.

```
assertion_range ::=
  assertion_subexpression range_symbol assertion_subexpression

range_symbol ::= .. | ,. | ., | ,,
```

Semantics

(S7) Where c, d, l, and u are predicate expressions,

$\mathfrak{M}_i[\![c\texttt{=}d]\!] \equiv \mathfrak{M}_i[\![c]\!] = \mathfrak{M}_i[\![d]\!]$ *(the meaning of = is equality)*
$\mathfrak{M}_i[\![c\texttt{<>}d]\!] \equiv \mathfrak{M}_i[\![c\texttt{ != }d]\!] \equiv \mathfrak{M}_i[\![c]\!] \neq \mathfrak{M}_i[\![d]\!]$ *(the meaning of <> and != is inequality)*[3]
$\mathfrak{M}_i[\![c\texttt{<}d]\!] \equiv \mathfrak{M}_i[\![c]\!] < \mathfrak{M}_i[\![d]\!]$ *(the meaning of < is less than)*
$\mathfrak{M}_i[\![c\texttt{>}d]\!] \equiv \mathfrak{M}_i[\![c]\!] > \mathfrak{M}_i[\![d]\!]$ *(the meaning of > is greater than)*
$\mathfrak{M}_i[\![c\texttt{<=}d]\!] \equiv \mathfrak{M}_i[\![c]\!] \leq \mathfrak{M}_i[\![d]\!]$ *(the meaning of <= is at most)*
$\mathfrak{M}_i[\![c\texttt{>=}d]\!] \equiv \mathfrak{M}_i[\![c]\!] \geq \mathfrak{M}_i[\![d]\!]$ *(the meaning of >= is at least)*
$\mathfrak{M}_i[\![c \texttt{ in l..u}]\!] \equiv \mathfrak{M}_i[\![c]\!] \geq \mathfrak{M}_i[\![l]\!] \wedge \mathfrak{M}_i[\![c]\!] \leq \mathfrak{M}_i[\![u]\!]$ *(the meaning of .. is closed interval)*
$\mathfrak{M}_i[\![c \texttt{ in l,.u}]\!] \equiv \mathfrak{M}_i[\![c]\!] > \mathfrak{M}_i[\![l]\!] \wedge \mathfrak{M}_i[\![c]\!] \leq \mathfrak{M}_i[\![u]\!]$ *(the meaning of ,. is open-left interval)*
$\mathfrak{M}_i[\![c \texttt{ in l.,u}]\!] \equiv \mathfrak{M}_i[\![c]\!] \geq \mathfrak{M}_i[\![l]\!] \wedge \mathfrak{M}_i[\![c]\!] < \mathfrak{M}_i[\![u]\!]$ *(the meaning of ., is open-right interval)*
$\mathfrak{M}_i[\![c \texttt{ in l,,u}]\!] \equiv \mathfrak{M}_i[\![c]\!] > \mathfrak{M}_i[\![l]\!] \wedge \mathfrak{M}_i[\![c]\!] > \mathfrak{M}_i[\![u]\!]$ *(the meaning of ,, is open interval)*

(S8) Where v is an identifier of a shared integer variable, and e is an integer-valued expression,

$\mathfrak{M}_i[\![v \texttt{ += e}]\!] \equiv \mathfrak{M}_{end(i)}[\![v]\!] = \mathfrak{M}_{start(i)}[\![v]\!] + \mathfrak{M}_{start(i)}[\![e]\!]$ *(the meaning of += is add to total* [4] *)*

I 5.3.7 Parenthesized Predicate

(1) Parentheses disambiguate precedence.

```
parenthesized_predicate ::= ( predicate )
```

Semantics

(S9) Where *P* is a predicate,

$\mathfrak{M}_i[\![(P)]\!] \equiv \mathfrak{M}_i[\![P]\!]$ *(the meaning of parenthesis is its contents)*

---

[3] **Reconciliation:** inequality

[4] The definition of a single **+=** is straight forward: at the end of the interval, the target will be the target value at the beginning of the interval, plus an expression also valued at the beginning of the interval. Defining concurrent **+=** to the same target, in the same interval, is just like solitary **+=**, using the sum of all concurrent expressions. Concurrent **+=** predicate defines concurrent fetch-add action. Fetch-add is used to access shared data structures without locks, allowing unlimited speed-up. See U.S Pat. No. 5,867,649 DANCE-Multitude Concurrent Computation

I 5.3.8 Universal Quantification

(1) Universal quantification claims its predicate is true for all the members of a particular set. Logic variables must have types. Bounding the domain of quantification to a range, or when some predicate is true, defines the set of values that variables may take.[5] Quantified variables of type time are particularly useful for declaratively expression cyber-physical systems (CPS). A particular combination of events either did or did not occur in a particular interval of time, or what is true about system state during a particular interval of time.

```
universal_quantification ::=
   all logic_variables logic_variable_domain
   are predicate

logic_variables ::=
  logic_variable_identifier { , logic_variable_identifier }*
  : type

logic_variable_domain ::= in
  ( assertion_expression range_symbol assertion_expression
  | predicate )
```

Semantics

(S10) Where v is a logic variable, T is an assertion-type, R is a range, and $P(v)$ is a predicate that uses v,

$\mathfrak{M}_i[\![$`all v:T in R are P(v)`$]\!] \equiv \forall\, v \in \mathfrak{M}_i[\![R]\!] \subseteq \mathfrak{M}_i[\![T]\!] \mid \mathfrak{M}_i[\![P(v)]\!]$
(for all v in R, a subset of T, P(v) is true)

Example

```
<<MOTION_ARTIFACT_ALARM: :all j:integer
      in 0..PulseOx_Properties::Motion_Artifact_Sample_Limit
      are (MotionArtifact^(-j) or not SensorConnected^(-j))>>
```

I 5.3.9 Existential Quantification

(1) Existential quantification claims its predicate is true for at least one member of a particular set.

Grammar

```
existential_quantification ::=
   exists logic_variables logic_variable_domain
   that predicate
```

Semantics

(S11) Where v is a logic variable, T is as assertion-type, R is a range, and $P(v)$ is a predicate that uses v,

$\mathfrak{M}_i[\![$`exists v:T in R that P(v)`$]\!] \equiv \exists\, v \in \mathfrak{M}_i[\![R]\!] \subseteq \mathfrak{M}_i[\![T]\!] \mid \mathfrak{M}_i[\![P(v)]\!]$
(there exists v in R, a subset of T, such that P(v) is true)

Example

[5] Bounding quantification is highly recommended.

```
<<RAPID_DECLINE_ALARM: :AdultRapidDeclineAlarmEnabled  and
    (exists j:integer in 1..NUM_WINDOW_SAMPLES()
      that (SpO2 <= (SpO2^(-j) - MaxSpO2Decline)))>>
```

I 5.3.10 Event

(1) An *event* occurs when either a port or variable has a (non-null) value, or the state machine is in a particular state (see I 2.17 Clock).

Grammar

```
event ::= < port_variable_or_state_identifier >
event_expression ::= [not] event
  | event_subexpression (and event_subexpression)+
  | event_subexpression (or event_subexpression)+
  | event - event
event_subexpression ::=
  [ always | never ] ( event_expression ) | event
```

Semantics

(S12) Where p is a port identifier <p> $\equiv \hat{p} \equiv \mathfrak{M}_{now}[\![p \neq \perp]\!]$.
Where v is a variable identifier <v> $\equiv \hat{v} \equiv \mathfrak{M}_{now}[\![v \neq \perp]\!]$.
Where s is a state identifier <s> $\equiv \hat{s} \equiv \mathfrak{M}_{now}[\![State(s)]\!]$ where $State(s)$ means the state machine is currently in state s.

(S13) Where <x> and <y> are events, <x> - <y> $\equiv \hat{x} \hat{-} \hat{y}$.

(S14) Where ee is an event expression, **never**(ee) ≡ee= $\hat{0}$, and **always**(ee) ≡ee= $1\hat{}_{SVP}$.

(S15) Logical operators not, and, or are complement, conjunction, and disjunction, respectively. Parentheses group.

I 5.4 Assertion-Expression

(1) Other useful quantifiers add, multiply, or count the elements of sets. There is no operator precedence so parentheses must be used to avoid ambiguity. Numeric operators have their usual meanings.

(2) Assertion-expressions differ from expression usually found in programming languages which are intended to be evaluated during execution. Rather, assertion expressions define values derived from over values, usually numeric. Such predicate expressions usually appear within predicates that contain relations between values. Predicate expressions may also used within assertion-functions (I 5.2.3) to define Assertions that return values.

(3) Numeric quantifiers sum, product, and number-of have an optional logic variable domain, but include one whenever possible. Bounding quantification prevents oddities that can occur with infinite domains. In mathematics, sums of an infinite number of ever smaller terms are quite common. But for reasoning about program behavior, stick to bounded quantifications.

Grammar

```
assertion_expression ::=
   sum logic_variables [ logic_variable_domain ]
      of assertion_expression
   | product logic_variables [ logic_variable_domain ]
      of assertion_expression
   | numberof logic_variables [ logic_variable_domain ]
      that subpredicate
   | assertion_subexpression
      [ { + assertion_subexpression }+
      | { * assertion_subexpression }+
      | - assertion_subexpression
      | / assertion_subexpression
      | ** assertion_subexpression
      | mod assertion_subexpression
      | rem assertion_subexpression ]
```

Semantics

(S1) Where v is a logic variable, T is a type, R is a range, $P(v)$ is a predicate that uses v, $E(v)$ is a predicate expression that uses v, and e, f are predicate subexpressions,

$\mathfrak{M}_i[\![$`sum v:T in R of E(v)`$]\!] \equiv \sum_{v \in R} \mathfrak{M}_i[\![E(v)]\!]$
(sum the value E(v) for each v in the range R)
$\mathfrak{M}_i[\![$`product v:T in R of E(v)`$]\!] \equiv \prod_{v \in R} \mathfrak{M}_i[\![E(v)]\!]$
(multiply the value E(v) for each v in the range R)
$\mathfrak{M}_i[\![$`numberof v:T in R that P(v)`$]\!] \equiv ||\{v \in \mathfrak{M}_i[\![R]\!] \mid \mathfrak{M}_i[\![P(v)]\!]\}||$
(cardinality of the set of v in R for which P(v) is true)
$\mathfrak{M}_i[\![$`e+f`$]\!] \equiv \mathfrak{M}_i[\![e]\!] + \mathfrak{M}_i[\![f]\!]$ *(the meaning of + is addition)*
$\mathfrak{M}_i[\![$`e*f`$]\!] \equiv \mathfrak{M}_i[\![e]\!] \times \mathfrak{M}_i[\![f]\!]$ *(the meaning of * is multiplication)*
$\mathfrak{M}_i[\![$`e-f`$]\!] \equiv \mathfrak{M}_i[\![e]\!] - \mathfrak{M}_i[\![f]\!]$ *(the meaning of - is subtraction)*
$\mathfrak{M}_i[\![$`e/f`$]\!] \equiv \mathfrak{M}_i[\![e]\!] \div \mathfrak{M}_i[\![f]\!]$ *(the meaning of / is division)*
$\mathfrak{M}_i[\![$`e**f`$]\!] \equiv \mathfrak{M}_i[\![e]\!]^{\mathfrak{M}_i[\![f]\!]}$ *(the meaning of ** is exponentiation)*
$\mathfrak{M}_i[\![$`e mod f`$]\!] \equiv \mathfrak{M}_i[\![e]\!]$ mod $\mathfrak{M}_i[\![f]\!]$ *(the meaning of* **`mod`** *is modulus)*
$\mathfrak{M}_i[\![$`e rem f`$]\!] \equiv \mathfrak{M}_i[\![e]\!]$ rem $\mathfrak{M}_i[\![f]\!]$ *(the meaning of* **`rem`** *is remainder)*

Legality Rule

(L1) The ranges for sum, product, and numberof predicate expressions must be discrete and finite.

(4) Predicate subexpressions allow optional negation of a timed expression. Negation has the usual meaning.

Grammar

```
assertion_subexpression ::=
  [ - | abs ] timed_expression
  | assertion_type_conversion
```

```
assertion_type_conversion ::=
  ( natural | integer | rational | real | complex | time )
  parenthesized_assertion_expression
```

Semantics

(S2) Where S is a predicate expression,

$\mathfrak{M}_i[\![\texttt{-S}]\!] \equiv 0 - \mathfrak{M}_i[\![S]\!]$ *(the meaning of - is negation)*
$\mathfrak{M}_i[\![\texttt{abs S}]\!] \equiv \mathfrak{M}_i[\![\texttt{(if S>=0 then S else -S)}]\!]$ *(the meaning of* ***abs*** *is absolute value)*[6]

Example

```
<<SPO2_AVERAGE: :=
   --the sum of good SpO2 measurements
    (sum i:integer in -SpO2MovingAvgWindowSamples..-1 of
       (SensorConnected^(i) and not MotionArtifact^(i)??SpO2^(i):0))
    /  --divided by the number of good SpO2 measurements
    (numberof i:integer in -SpO2MovingAvgWindowSamples..-1
      that (SensorConnected^(i) and not MotionArtifact^(i)))>>
```

I 5.4.1 Timed Expression

(1) In a *timed expression*, the time when the expression is evaluated may be specified. The ' means the value of the expression one clock cycle (or thread period) hence; the @ means the value of the expression when the subexpression (to the right of the @), in seconds, is the current time; and the ^ means the value of the expression an integer number of clock ticks from now. Grammatically, time-expression and period-shift are time-free (no ' @ or ^ within).

Grammar

```
timed_expression ::=
    ( assertion_value
      | parenthesized_assertion_expression
      | predicate_incocation )
    [ '
    | ^ integer_expression
    | @ time_expression ]
```

Legality Rules

(L2) When using @, the subexpression must have a time type such as, Timing_Properties::Time.

(L3) When using ^, the value must have integer type.

Semantics

(S3) Where E is a value, a parenthesized predicate expression, or a conditional predicate expression, t is a time, d is the duration of a thread's period, and k is an integer:

[6] **Reconciliation:** absolute value

$\mathfrak{M}[\![\texttt{E@t}]\!] \equiv \mathfrak{M}_t[\![\texttt{E}]\!]$ *(the meaning of* `E@t` *is the meaning of* `E` *at time t)*
$\mathfrak{M}_t[\![\texttt{E\^{}k}]\!] \equiv \mathfrak{M}_{t+dk}[\![\texttt{E}]\!]$ *(the meaning of* `E^k` *at time t, is the meaning of* `E`*, k period durations hence, or earlier if* $k < 0$*)*
$\mathfrak{M}_t[\![\texttt{E'}]\!] \equiv \mathfrak{M}_t[\![\texttt{E\^{}1}]\!] \equiv \mathfrak{M}_{t+d}[\![\texttt{E}]\!]$ *(the meaning of* `E'` *at time t, is the meaning of* `E` *a period duration hence)*

Example

```
<<heart_rate[i]=(MotionArtifact^(1-i) or not SensorConnected^(1-i)
    ??0:HeartRate^(1-i))>>
```

I 5.4.2 Parenthesized Assertion Expression

(1) Parentheses around assertion expressions determine operator precedence. Both conditional assertion expressions and record term have inherent parentheses.

Grammar

```
parenthesized_assertion_expression ::=
    ( assertion_expression )
    | conditional_assertion_expression
    | record_term
```

I 5.4.3 Assertion-Value

(1) An *assertion-value* is atomic, so cannot be further subdivided into simpler expressions. The value of *tops* is the time of previous suspension of the thread which contains it; tops is used commonly in expressions of time-outs. The value of assertion function invocation is given in I 5.3.5. Property values according to AS5506B §11 Properties. Port values according to AS5506B §8.3 Ports.

Grammar

```
assertion_value ::=
  now | tops | timeout
  | value_constant
  | variable_name
  | assertion_function_invocation
  | port_value
```

I 5.4.4 Conditional Assertion Expression

(1) A *conditional assertion expression* determines the value of a predicate expression by evaluating a boolean expression or relation, then choosing between alternative expressions, having the first value if true or the second value if false.

Grammar

```
conditional_assertion_expression ::=
  ( predicate ?? assertion_expression : assertion_expression )
```

Semantics

(S4) Where `t` and `f` are expressions and `B` is a boolean-valued expression or relation:

$$\mathfrak{M}_i[\![\texttt{(B??t:f)}]\!] \equiv \begin{array}{c} \mathfrak{M}_i[\![\texttt{B}]\!] \rightarrow \mathfrak{M}_i[\![\texttt{t}]\!] \\ \neg\mathfrak{M}_i[\![\texttt{B}]\!] \rightarrow \mathfrak{M}_i[\![\texttt{f}]\!] \end{array}$$

(choose first value if true; second value if false)

Example

```
<<(all i:integer in 1 ..num_samples
  are spo2[i]'=(if MotionArtifact^(1-i) or not SensorConnected^(1-i)
         then 0 else SpO2^(1-i)))
      and (num_samples'=PulseOx_Properties::Num_Trending_Samples)>>
```

I 5.4.5 Conditional Assertion Function

(1) A *conditional assertion function* is much like a conditional assertion expression (I 5.4.4), but allows an arbitrary number of choices, each of which is controlled by a predicate. A conditional assertion function is only permitted as a assertion-function value (I 5.2.3).

(2) Conditional assertion-function was added to specify the flow rate of a patient-controlled analgesia (PCA) pump. Rather than a smooth function, the flow rate must be different depending on system state (see example). `PUMP_RATE` is the BLESS::Assertion property of a port of the thread deciding infusion rate. Each of the parenthesized predicates embodies complex conditions that must be true for each of the possible infusion rates. When a value is output from the port, a proof obligation is generated to ensure that the corresponding property holds.

Grammar

```
conditional_assertion_function ::=
    ( condition_value_pair { , condition_value_pair }* )

condition_value_pair ::=
     parenthesized_predicate ->  assertion_expression
```

Semantics

(S5) Where `C1`, `C2`, and `C2` are predicates and `E1`, `E2`, and `E3` are assertion-expressions:

$$\mathfrak{M}_i[\![\texttt{(C1)->E1, (C2)->E2, (C3)->E3}]\!] \equiv \begin{array}{c} \mathfrak{M}_i[\![\texttt{C1}]\!] \rightarrow \mathfrak{M}_i[\![\texttt{E1}]\!] \\ \mathfrak{M}_i[\![\texttt{C2}]\!] \rightarrow \mathfrak{M}_i[\![\texttt{E2}]\!] \\ \mathfrak{M}_i[\![\texttt{C3}]\!] \rightarrow \mathfrak{M}_i[\![\texttt{E3}]\!] \\ \neg\mathfrak{M}_i[\![\texttt{C1}]\!] \wedge \neg\mathfrak{M}_i[\![\texttt{C2}]\!] \wedge \neg\mathfrak{M}_i[\![\texttt{C3}]\!] \rightarrow \bot \end{array}$$

(choose the value corresponding to the true condition, or null in no conditions are true)

Example

Conditional assertion-functions should be used sparingly. The pump-rate example below induced conditional assertion-function's creation to define infusion rate in different conditions.

```
<<PUMP_RATE: :=
  (HALT()) -> 0,                                    --no flow
  (KVO_RATE()) -> PCA_Properties::KVO_Rate,         --KVO rate
  (PB_RATE()) -> PCA_Properties::Patient_Button_Rate, --maximum infusion
  (CCB_RATE()) -> Square_Bolus_Rate,                --square bolus rate
  (PRIME_RATE()) -> PCA_Properties::Prime_Rate,     --pump priming
  (BASAL_RATE()) -> Basal_Rate                      --basal rate, from data port
>>
```

I 5.4.6 Assertion-Function Invocation

Assertion-functions which are declared in the form `<<C:f:=E>>` and may be invoked like functions as a predicate value `C(a)` , where

- `C` is the label,
- `f` are formal parameters,
- `E` is an assertion-expression, and
- `a` are actual parameters.

Grammar

```
assertion_function_invocation ::=
  assertion_function_identifier
    ( [ assertion_expression |
      actual_assertion_parameter { , actual_assertion_parameter }* ] )

actual_assertion_parameter ::=
  formal_identifier : actual_assertion_expression
```

Semantics

(S6) Where C is an assertion-function label, $f_1 f_2 \ldots f_n$ are formal parameters, and E is a predicate expression that uses $f_1 f_2 \ldots f_n$, and

$\ll C : f_1 f_2 \ldots f_n := E \gg$
(there is assertion-function C with predicate expression E and formal parameters f)

(S7) The meaning of assertion-function invocation is

$\mathfrak{M}_i[\![\texttt{C(a1 a2 ... an)}]\!] \equiv \mathfrak{M}_i[\![E \mid \frac{f_1}{a_1} \mid \frac{f_2}{a_2} \cdots \mid \frac{f_n}{a_n}]\!]$
(the meaning of an assertion function invocation is the meaning of the expression of the assertion-function with the same label having actual parameters substituted for formal parameters)

Example

```
<<SUPPL_O2_ALARM: :SupplOxyAlarmEnabled^0 and
  (SPO2_AVERAGE())^0 < (SpO2LowerLimit^0+SpO2LevelAdj^0)>>
```

I 5.4.7 Assertion-Enumeration Invocation

Assertion-enumerations which are declared in the form `<<C:x+=>R>>` and may be invoked like functions as a predicate value `C(a)` , where

- `C` is the label of the assertion-enumeration,
- `a` is an enumeration-element identifier, and
- `R` is a set of enumeration pairs (label `->` predicate).

```
assertion_enumeration_invocation ::=
  +=> assertion_enumeration_label_identifier
    ( actual_assertion_parameter )
```

Semantics

(S8) Where

C is an assertion-enumeration label,

L is a set of enumeration labels $\{l_1,\ l_2,\ \ldots,\ l_n\}$,

a is the formal parameter, an enumeration label $a \in L$,

P is a set of predicates $\{p_1,\ p_2,\ \ldots,\ p_n\}$, and

R is a set of enumeration pairs, $\{l_1 \rightarrow p_1,\ l_2 \rightarrow p_2,\ \ldots,\ l_n \rightarrow p_n\}$ defining the onto relation[7] between enumeration labels and their meaning, $R(j) = q \equiv j \rightarrow q \in R$

and

`<<C:x+=>R>>` *(there is assertion-enumeration C with enumeration pairs R and ignored parameter x)*

(S9) The meaning of assertion-enumeration invocation is

$\mathfrak{M}_i[\![\texttt{C(a)}]\!] \equiv \mathfrak{M}_i[\![R(a)]\!]$

(the meaning of an assertion-enumeration invocation is the predicate paired with given label a)

Example

(1) Enumeration types should be used sparingly. Assertion-enumerations were created to express the meaning of event-data with enumeration type. Ports having enumeration types may only have enumeration literals for out parameters. The following example expressed the meaning of 'On' and 'Off' in section A.5.1.3 of the isolette example in FAA's Requirement Engineering Management Handbook:

```
--A.5.1.3 Manage Heat Source Function
<<HEAT_CONTROL:x+=>
  On  ->  REQMHS2() or --below desired range
          (REQMHS4() and (heat_control^-1=On)),
  Off -> REQMHS1() or --initialization
          REQMHS3() or --above desired range
          REQMHS5() or --failed
          (REQMHS4() and (heat_control^-1=Off)) >>
```

Used to define the meaning of the value of port `heat_control` :

[7] Every label has exactly one predicate defining its meaning.

```
heat_control : out data port Iso_Variables::on_off
  {BLESS::Assertion => "<<+=>HEAT_CONTROL(x)>>";};
```

When an enumeration value is sent out port in state-machine action:

```
  mhsBelow:  --REQ-MHS-2  temp below desired range
  check_temp -[current_temperature? <= lower_desired_temperature?]-> run
    { <<REQMHS2() and not REQMHS1()>>
    heat_control!(On)  --temp below desired range
    ; <<heat_control=On>>
    heat_previous_period' := On
      <<heat_previous_period' = heat_control>>
    };  --end of mhsBelow
```

During transformation from proof outline to complete proof, port output of 'On' and its precondition

```
<<REQMHS2() and not REQMHS1()>>
    heat_control!(On)  --temp below desired range
```

becomes a verification condition, that what's claimed for 'On' holds

```
<<REQMHS2() and not REQMHS1()>>
->
<<REQMHS2() or (REQMHS4() and (heat_control^-1=On))>>
```

(2) If it's just two labels (off/on) use a simple predicate instead. Save the hassle of putting meaning to enumeration labels for when it's unavoidable:

```
--regulator mode Figure A-4. Regulate Temperature Mode Transition Diagram
<<REGULATOR_MODE:x+=>
  Init   -> INI(),
  NORMAL -> REGULATOR_OK() and RUN(),
  FAILED -> not REGULATOR_OK() and RUN() >>
```

Chapter I 6

State Machine

(1) Behavior specifications can be attached to any AADL component types and component implementations using an annex subclause[1] with label `Behavior_Specification`.[2]

```
annex Behavior_Specification {**  . . .  **};
```

(2) When defined within component type specifications, it represents behavior common to all the associated implementations. If a component type or implementation is extended, behavior annex subclauses defined in the ancestor are applied to the descendent except if the later defines its own behavior annex subclause.[3] However, AS5506B §5.4 Threads, defines standard behavior for thread scheduling and interaction. Any component with a behavior specification must conform to the standard for threads, regardless of its component classifier. Therefore, references to 'thread' should be considered applicable to any component with a behavior specification annex subclause.

(3) A behavior annex subclause may be interpreted as a refinement of a call sequence section in a thread or subprogram component implementation. If both a call sequence section and a behavior annex subclause with subprogram call actions are defined for the same component implementation, then all the subprogram calls specified in the former must be reflected in the latter, although the call order may differ.[4][5]

(4) Mode-specific behavior by appending an annex subclause with an **in modes** clause.[6] Alternatively, a mode can be reflected by a complete state of the same name in the Behavior_Specification and mode transition behavior can be modeled as a transition out of such a complete state whose condition identifies the event port named in the mode transition, if specified in the core AADL model.[7]

[1] BA D.3(1)
[2] Implementations may also accept annex labels `BAv2` or `BLESS` equivalently.
[3] BA D.3(1)
[4] BA D.3(22)
[5] **Reconciliation:** call sequence
[6] AS5506B §12 Modes and Mode Transitions
[7] BA D.3(3)

I 6.1 Component Behavior

(1) Component behavior is defined by a *state transition system*. A state transition system has a set of states, a set of local variables some of which may have initial values, and a set of transitions. The transitions of a state transition system specify behavior as a change of the current state from a source state to a destination state.

(2) Component behaviors may have an *assert clause* listing labelled assertions to be used by other assertions.

(3) Component behaviors may have an *invariant clause* that must be true of every state.

Grammar[8] [9]

```
behavior_annex ::=
    [ assert { assertion }+ ]
    [ invariant assertion ]
    [ variables ]
    states { behavior_state }+
    [ transitions ]
```

Legality Rule

(L1) Component behaviors must have at least two states (**`initial`** and **`final`**) and at least one transition.[10]

Naming Rule

(N1) The variable, state, and transition identifiers must be unique within an annex subclause, and may not also be data subcomponents, component features, or mode identifiers–except for complete state identifiers which may be mode identifiers.[11]

Consistency Rules

(C1) If a component type or implementation is extended, behavior specification defined in the ancestor are applied to the descendent except if the later defines its own behavior specification.[12]

(C2) A behavior specification of a subcomponent overrides the behavior specification of its containing component if they conflict.[13]

Semantics

(S1) A `Behavior_Sepcification` defines an automaton (Appendix I 2.19), A, as behavior for the component (usually a thread) which contains it in an annex subclause, $A = (S_A, s_0, V_A, P_A, T_A, C_A)$. Its states, S_A are defined in the `states` section having unique initial state s_0. Its persistent variables, V_A are defined in the `variables` section. Its ports, P_A are defined by its containing component. Its transitions, T_A are defined in the `transitions` section. Its constraints, C_A, are defined in ..., denoted by multi-sorted logical formula F_A.[14][15]

[8] BLESSDiffers from BA: `assert` and `invariant` sections
[9] BLESSDiffers from BA: mandatory `states` keyword
[10] BA D.3(L1)
[11] BA D.3(N1)
[12] BA D.3(C1)
[13] BA D.3(C2)
[14] JP
[15] Constraints such as C_A do not seem to have grammar for them.

I 6.2 Behavior States

(1) The `states` section declares all the states of the automaton. Some states may be qualified as `initial` state, `final` state, or `complete` state, or combinations thereof. A state without qualification will be referred to as execution state. A behavior automaton starts from an initial state and terminates in a final state.[16] A behavior state may have an assertion that holds when that state is current.

(2) The core AADL standard defines runtime execution states for threads.[17] These states include an *initial* state (thread halted), a *complete* state (awaiting dispatch,) and a *final* state (stopped thread).

Grammar

```
behavior_state ::=
  behavior_state_identifier : [initial] [complete] [final] state
    [ assertion ] ;
```
[18] [19]

(3) The behavior specification of components other than subprograms consists of an initial state, one or more final states, one or more complete states, and zero or more execution states. A transition out of the initial state is triggered by the initialize action defined in the core AADL standard. Execution states may be used to represent intermediate initialization steps. Upon completion of initialization a complete state is reached. In a behavior specification, the initial state can be a complete state (i.e. an initial complete state). Such a state is an implicit superposition of two states, an initial state and a complete state, connected by an implicit transition. This implicit transition, from the implicit initial state towards the implicit complete state, is triggered by the initialize action defined in the core standard. No condition can be associated to this implicit transition. No other action than the initialization action defined in the core standard (i.e. call to the `Initialize_Entrypoint` as defined by a property in the core language) can be associated to the implicit transition. Note that entering (resp. exiting) an initial complete state stands for entering (resp. exiting) the implicit complete state. This means that no transition can reach the implicit initial state.[20]

(4) In the case of subprograms, the automaton consists of one `initial` state representing the starting point of a call, zero or more intermediate execution states, and one `final` state. A `final` state represents the completion of a call. The complete state is not used in behavior specifications of subprograms.[21]

(5) When a component has modes it may also have a separate behavior annex subclause for each mode. In this case, a mode transition results in a transition from the `complete` state of the current mode behavior automaton to the `initial` state of the behavior automaton of the new mode.[22]

(6) At least one state must be labeled `final`. There may be no transitions from a `final` state (unless it's also a `complete` state). The `final` state may be entered via a normal transition, abort transition, stop transition, or invocation of the component's *Finalize_Entrypoint*.[23] A state that is qualified as `final`, and is not at the same time `initial` or `complete`, cannot accept outgoing transitions. If the purpose of the behavior annex is to provide a specification of the intended behavior of a component, then the use of several `final` states is

[16] BA D.3(8)
[17] AS5506B §5.4.1 Thread States and Actions
[18] BLESSDiffers from BA: single state identifier allowed
[19] BLESSDiffers from BA: states may have assertions
[20] BA D.3(24)
[21] BA D.3(9)
[22] BA D.3(13)
[23] AS5506B §5.4.1 Thread States and Actions

allowed. Otherwise, if the purpose is to provide a deterministic representation of the implementation of the internal behavior of the component, then only one `final` state must be defined.[24]

(7) Entering a `complete` state suspends the component until its next dispatch. Reaching a `complete` state can be interpreted as calling the `Await_Dispatch` run-time service. Thus a component is suspended if it performs a transition to a complete state, after having executed the action associated to the transition. The next dispatch will restart the thread from that state.[25]

(8) Execution states are transitory allowing computations upon dispatch to be subdivided into steps. From every execute state there must be at least one transition leaving that state with an enabled transition condition. Upon dispatch, a finite number of execute states may occur before entering a complete or final state.

(9) Upon completion of initialization a `complete` state is reached starting from the `initial` state, and perhaps a finite number of execution states.

(10) In a behavior specification, the `initial` state can be a `complete` state (i.e. an `initial complete` state). Such a state is an implicit superposition of two states - an `initial` state and a `complete` state - connected by an implicit transition. This implicit transition from the implicit initial state towards the implicit complete state - is triggered by the initialize action defined in the core standard. No condition can be associated to this implicit transition. No other action than the initialization action defined in the core standard (i.e. call to the `Initialize_Entrypoint` as defined by a property in the core language) can be associated to the implicit transition. Note that entering (resp. exiting) an initial complete state stands for entering (resp. exiting) the implicit complete state.[26] This means that no transition can reach the implicit initial state.

(11) An `initial` state can be a `complete` state and a `final` state as well (i.e. an final complete state). Such a state is an implicit superposition of three states an `initial` state, a `complete` state, and a `final` state - connected by two implicit transitions. The first transition, from the implicit initial state towards the implicit complete state, can only be triggered by the initialization action as defined in the core standard. The second transition, from the implicit complete state and towards the implicit final state, can only be triggered by the reception of a stop event. Note that exiting (respectively entering) an initial final complete state stands for exiting (resp. entering) the implicit complete state. No other action than the initialization action (call to the initialize_entrypoint as defined by a property in the core language) can be associated to the first implicit transition. No other action than the finalization action (represented by the finalize_entrypoint property from the core language) can be associated to the second implicit transition. No execution condition can be associated to those two implicit transitions.[27]

Legality Rules

(L1) A Behavior_Specification component behavior annex specification must define one `initial` state. A Behavior_Specification component behavior annex specification must define at least one `final` state.[28]

(L2) Transitions from an execute source, have execute conditions, which are boolean expressions evaluated by the component.

(L3) Transitions from a `complete` source, have dispatch conditions, evaluated by the AADL runtime services.[29]

[24] BA D.3(12)
[25] BA D.3(12)
[26] BA D.4(7)
[27] BA D.4(8)
[28] BA D.3(L1)
[29] BA D.3(L6), BA D.3(L7)

(L4) Transitions from states that are `final` only (not also `complete` or `initial`) are not allowed.[30]

(L5) A behavior annex specification for a thread, device, and other components awaiting dispatch or awaiting a mode transition, must define at least one complete state and one initial state. This may be the same state.[31]

(L6) A behavior annex specification for threads and other components with initialization and finalization entrypoints may explicitly model the initialization and finalization by including one initial state and one or more final states.[32]

(L7) A behavior annex specification for a subprogram must not define any complete states.[33]

Consistency Rules

(C1) A `Behavior_Specification` for a thread must be consistent with the core AADL semantics.[34]

(C2) If a component type or implementation is *extended*, behavior annex subclause defined in the ancestor are applied to the descendent except if the later defines its own behavior annex subclause.[35]

(C3) A behavior annex subclause of a *subcomponent* overrides the behavior annex subclause of its containing component if they conflict.[36]

(C4) The behavior annex state transition system must not remain blocked in an *execution* state. This means that the logical disjunction of all the execute conditions associated with the transitions out of an execution state must be true.[37]

(C5) If the behavior annex defines transitions from a `complete` state that represents a *mode* in the containing component, then the transition condition associated with these transitions must be consistent with the corresponding mode transition triggers.[38][39]

(C6) In behavior transitions, *mode conditions* can be used to describe mode transitions in any component classifier, except those belonging to the category of threads and subprograms. In components of these categories, execute conditions and/or dispatch conditions should be used to describe behavior transitions.[40]

Semantics

(S1) Entering a `complete` suspends execution until next dispatch, and sends all pending outputs.

(S2) Where S_t is the behavior state of the component at time t, i is a satisfying interval, s is a behavior state, d is a dispatch condition, and A is an assertion:

$\mathfrak{M}_i[\![$ `s initial state;` $]\!] \equiv S_{start(i)} = s$
(the initial state is the state at the start of the interval)
$\mathfrak{M}_i[\![$ `s final state;` $]\!] \equiv S_{end(i)} = s$

[30] BA D.3(L8)
[31] BA D.3(L3)
[32] BA D.3(L4)
[33] BA D.3(L2)
[34] AS5506B §5.4 Threads
[35] BA D.3(C1)
[36] BA D.3(C2)
[37] BA D.3(C3)
[38] AS5506B §12 Modes and Mode Transitions
[39] BA D.3(C4)
[40] BA D.3(C5)

(the final state is the state at the end of the interval)
$\mathfrak{M}_i[\![$ `s complete state;` $]\!] \equiv \forall t \in i \mid (S_t = s) \rightarrow \neg\mathfrak{M}_t[\![d]\!] \wedge suspended(t)$
(for all time, component is suspended and the dispatch condition is false when in a complete state)
$\mathfrak{M}_i[\![$ `s <<A>> state;` $]\!] \equiv \forall t \in i \mid (S_t = s) \rightarrow \mathfrak{M}_t[\![A]\!]$
(for all time, when in a state, its assertion is true)

Example

```
states
  start : initial state;
  fill : complete state
    <<SpO2_INV() and (num_samples<#PulseOx_Properties::Num_Trending_Samples)>>;
  check : state;
  run : complete state;
  halt : final state;  --normal termination
  fail : final state;  --error termination
```

I 6.3 Variables

(1) A `variables` clause declares identifiers that represent either local *behavior variables* in the scope of the current annex subclause, or a reference to an external data component. Variables can be used to keep track of intermediate results within the scope of the annex subclause. They may hold the values of out parameters on subprogram calls to be made available as parameter values to other calls, as output through enclosing out parameters and ports, or as value to be written to a data component in the AADL specification. They can also be used to hold input from incoming port queues or values read from data components in the AADL specification.[41] Values of variables *are* persistent across the various invocations of the same behavior annex subclause.[42] [43]

Grammar

```
variables ::= variables { behavior_variable }+

behavior_variable ::=
  local_variable_declarator { , local_variable_declarator }* :
    [ modifier ] type [ := value_constant ] [ assertion ] ;   [44] [45] [46]

declarator ::= identifier { array_size }*

array_size ::= [ natural_value_constant ]

modifier ::= nonvolatile | constant | shared | spread | final
```

(2) Variables that retain state when the system is powered off are *nonvolatile*. Variables oxymoronically-declared to be *constant* may not be assigned except during initialization. Targets of combinable operations must be declared *shared*. Arrays whose concurrent access is controlled using combinable operations are declared *spread*, as in spread across memory banks to minimize bank conflict on concurrent accesses.[47] Variables that may only be

[41] BA D.3(6)
[42] BLESSDiffers from BA: variable persistence
[43] BLESSDiffers from BA: variables have no property associations
[44] BLESSDiffers from BA: type more general
[45] BLESSDiffers from BA: has variable assertion
[46] BLESSDiffers from BA: no variable properties
[47] If you're not seeking speed-up of computation via concurrent execution, you won't need `shared` or `spread`.

assigned once are labeled *final*.

(3) Behavior variable declarations can indicate that a requires data access is `shared`. Only shared variables may be targets of combinable operations.

Legality Rules

(L1) Variables may have initialization expressions.

(L2) Referenced external data components must be `requires data access` features of the component.[48]

(L3) Variables labeled `final` may only be assigned once.

(L4) Variables labeled `constant` may not be assigned values except by their declaration.

Semantics

(S1) Where v is a behavior variable identifier, T is a type, e is an expression, and d is a data component identifier:

$\mathfrak{M}[\![$ **variables** `v:T;` $]\!] \equiv \exists v \in T$ *(there exists a variable v of type T)*

$\mathfrak{M}[\![$ **variables** `v:T:=e;` $]\!] \equiv \exists v \in T \wedge \mathfrak{M}_{start(i)}[\![v]\!] = \mathfrak{M}[\![e]\!]$
(there exists a variable v of type T with a value of e at the beginning of the interval)

Example

```
variables
  nts : constant integer:=#PulseOx_Properties::Num_Trending_Samples;
  spo2 :  array [1 ..nts] of PulseOx_Types::SpO2:=0;  --holds SpO2 history
  spo2_nxt : array [1 ..nts] of PulseOx_Types::SpO2:=0;
  num_samples :  integer:=0;  --counts samples while filling
```

I 6.4 Transitions

(1) In a `Behavior_Specification`, *transitions* define dynamic behavior. When the component's current state is one of the source states of a particular transition, and the condition for transition evaluates to true, the current state will become the destination state after an action (if supplied) is performed. A transition's Assertion, if supplied, is invariant during the transition.

(2) A transition may be identified by a *label*. The label contains a transition identifier and an optional priority number. Transition priorities control the evaluation order of transition guards.[49] The evaluation order of two transitions with the same priority is non-deterministic. Transitions with no specified priority have the lowest priority.[50]

[48] AS5506B §8.6 Data Component Access
[49] **Reconciliation:** transition priority
[50] BA D.3(19)

(3) Actions can be performed by a transition before entry of the destination state. If a transition is enabled, the actions are performed and then the state specified as the destination of the transition becomes the new current state.[51] [52]

Grammar

```
transitions ::= transitions { behavior_transition }+

behavior_transition ::=
    [ behavior_transition_label : ]
    source_state_identifier { , source_state_identifier }*
    -[ [ transition_condition ] ]-> destination_state_identifier
   [ { [ behavior_actions ] } ] [ assertion ] ;                    [53]

behavior_transition_label ::=
    transition_identifier [ [ priority_natural_literal ] ]

transition_condition ::=
  dispatch_condition | execute_condition
  | mode_condition | internal_condition [54]
```

(4) When the source state of a transition is a state where the component is waiting for dispatch, and if its dispatch protocol is not periodic, then the condition is a *dispatch condition* that specifies the triggering events in terms of event port, event data port, calls received on provides subprogram access features, or time out. Otherwise, when the source state is an execute state of the component, the condition is an *execute condition* on state variables and received input values.

(5) The core AADL standard defines dispatch conditions for threads in terms of a disjunction of trigger conditions as result of arrival of events or event data on incoming ports of subprogram access features. A subset of ports involved in the triggering of a dispatch may be specified through the `Dispatch_Trigger` property. The behavior specification can refine this dispatch condition into a Boolean condition that is associated with a transition out of a complete state.[55]

(6) A dispatch trigger may result in a transition out of a `complete` state and to one of the states defined in the `Behavior_Specification` (either an execution state or a complete state). A dispatch trigger can be the arrival of input on ports, a subprogram call initiated by another thread, or a timed event (periodic dispatch or timeout). Reaching a `complete` state can be interpreted as calling the `Await_Dispatch` run-time service. Thus a component is suspended if it performs a transition to a complete state, after having executed the action associated to the transition. The next dispatch will restart the thread from that state.[56]

(7) When the `Dispatch_Protocol` property is timed or hybrid, the value of the time out dispatch condition is given by the `Period` property of the component.[57]

(8) An empty transition condition is equivalent to a condition that is always true.[58]

[51] **Reconciliation:** behavior action block
[52] BA D.3(20)
[53] BLESSDiffers from BA: transitions may have assertions
[54] BLESSDiffers from BA: mode instead of external condition
[55] BA D.3(26)
[56] BA D.3(27)
[57] BA D.3(28)
[58] BA D.3(N2)

Legality Rule

(L1) A behavior specification for threads and other components must have one initial state and one or more final states.[59]

(L2) Behavior transitions having Assertions must have labels.

(L3) Transitions from states that are final only are not allowed.[60]

(L4) A behavior specification for a thread, device, and other components that can be suspended awaiting dispatch or awaiting a mode transition, must define at least one complete state and one initial state. This may be the same state.[61]

Consistency Rules

(C1) The state transition system must not remain blocked in an execution state. This means that the logical disjunction of all the execute conditions associated with the transitions out of an execution state must be true.[62]

(C2) If the behavior specification defines transitions from a complete state that represents a mode in the containing component, then the `transition_condition` associated with these transitions must be consistent with the corresponding `mode_transition_triggers`.[63][64]

Semantics

(S1) A `behavior_transition` defines multiple transitions $(s, V_s, I_A, g, d, V_d, O_A, f) \in T_A$ of automaton A, because there are many possible input values, variable valuations, and output values for a transition from the source state to the destination state (Annex I 2.19). The transition only occurs when the automaton occupies the source state `transition_condition` is true, which may be an execute condition if the source state is an execution state, or a dispatch condition if the source state is a complete state.[65]

(S2) The `behavior_transition_label`, if present, defines a label, m, which represents the clock (AnnexI 2.17) for the transition, $\hat{\mathrm{m}}$, when the transition occurs. For transition `m: s-[g]-d;`, its clock is $\hat{\mathrm{m}} \Leftrightarrow (s$ **and** $g)$.[66]

Example

```
transitions
  sptt0: start-[ ]->fill{};
  sptt1: fill-[on dispatch]->check { . . . };
  sptt2a: check-[num_samples<#PulseOx_Properties::Num_Trending_Samples]->fill
    {(spo2',num_samples':=spo2,num_samples)};
  sptt2b: check-[num_samples=#PulseOx_Properties::Num_Trending_Samples]->run
    {(spo2',num_samples':=spo2,num_samples)};
  sptt2c: check-[num_samples>#PulseOx_Properties::Num_Trending_Samples]->fail{};
```

[59] BA D.3(L3)
[60] BA D.3(L8)
[61] BA D.3(L8)
[62] BA D.3(C3)
[63] BA D.3(C3)
[64] AS5506B 12 Modes and Mode Transitions
[65] JP
[66] JP

I 6.5 Execute Condition

(1) Any transition leaving an execute state must have an *execute condition*.[67] Execute conditions are boolean expressions that may only contain references visible within the component, such as ports and local variables.

```
execute_condition ::= boolean_expression_or_relation | timeout | otherwise
```

Legality Rule

(L1) Any transition with an execute state as its source must have an execute condition, or nothing which is the same as **true** .

Semantics

(S1) Where s and d are behavior states in S with Assertions A_s and A_d,

```
states . . . s:state <<As>>; d:state <<Ad>>; . . .
```

$S_{start(i)}$ is the behavior state at time $start(i)$, $S_{end(i)}$ is the behavior state at time $end(i)$, b is a behavior condition, w is an asserted action, C is an Assertion, and i is a satisfying interval:

$$\mathfrak{M}_i[\![\ \textbf{transitions}\ \texttt{s-[b]->d}\ ]\!] \equiv \begin{array}{l} S_{start(i)} = s, \\ S_{end(i)} = d, \\ \mathfrak{M}_{start(i)}[\![A_s \wedge b]\!] \rightarrow \mathfrak{M}_{end(i)}[\![A_d]\!] \end{array}$$

(a transition from s to d on condition b over a subinterval i, must start in s with A_s, end in d with A_d, and the conjunction of the condition b and A_s at the beginning, must imply A_d at the end)

$$\mathfrak{M}_i[\![\ \textbf{transitions}\ \texttt{s-[b]->d \{w\}}\ ]\!] \equiv \begin{array}{l} S_{start(i)} = s, \\ S_{end(i)} = d, \\ \mathfrak{M}_{start(i)}[\![A_s \wedge b]\!] \rightarrow \mathrm{wp}(w, \mathfrak{M}_{end(i)}[\![A_d]\!]) \end{array}$$

(a transition from s to d on condition b with action w over a subinterval i, must start in s with A_s, end in d with A_d, and the conjunction of the condition b and A_s at the beginning, must imply the weakest precondition of w and A_d at the end)

$$\mathfrak{M}_i[\![\ \textbf{transitions}\ \texttt{s-[b]->d \{w\} <<C>>}\ ]\!] \equiv \begin{array}{l} S_{start(i)} = s, \\ S_{end(i)} = d, \\ \mathfrak{M}_{start(i)}[\![A_s \wedge b]\!] \rightarrow \mathrm{wp}(w, \mathfrak{M}_{end(i)}[\![A_d]\!]), \\ \mathfrak{M}_i[\![C]\!] \end{array}$$

(a transition from s to d on condition b with action w and Assertion C over a subinterval i, must start in s with A_s, end in d with A_d, and the conjunction of the condition b and A_s at the beginning, must imply the weakest precondition of w and A_d at the end; the Assertion C must be true throughout i)[68]

(S2) Semantics of in data and in event data ports are defined in §?? and §?? respectively.

---

[67] BA D.3(18)

[68] The Assertion in behavior transitions between the action and the terminating semicolon was changed from a post-condition to an invariant that holds during the transition. Previously, during execution of an action, a component was in *no* state. In no state, none of the state Assertions necessarily holds. This made it impossible to write a component invariant that was *always* true. By defining transitions' Assertions to hold during execution of its transition, the intrinsic component invariant becomes the disjunction of all state and transition Assertions.

I 6.6 Internal Conditions

(1) Internal events may be used to represent interactions among annexes. In the scope of a behavior annex subclause, an internal feature may be used to describe under which circumstances an event is sent from either an internal event port or an internal event data port .[69]

Grammar

```
internal_condition ::=  on internal
  internal_port_name { or internal_port_name }*
```

I 6.7 Modal Conditions

(1) The function `in mode` tests whether the current local mode is among the identifiers listed. The mode identifiers must be among those of the behavior annex subclauses in modes clause, if any, and the modes of its thread component.

(2) When the state machine is used to define mode transitions, complete state identifiers match mode identifiers for the component. Leaving a mode-state requires a transition with a *mode condition* which may be triggered by an event (data) arriving or leaving an event (data) port of the component or one of its subcomponents.[70]

Grammar

```
mode_condition ::=  on trigger_logical_expression[71]

trigger_logical_expression ::= event_trigger { logical_operator event_trigger }*

event_trigger ::=
  in_event_subcomponent_port_reference
  | in_event_data_subcomponent_port_reference
  | ( trigger_logical_expression )

subcomponent_port_reference ::=
  subcomponent_identifier { . subcomponent_identifier }*
    . port_identifier                                          [72]

logical_operator ::= and | or | xor | and then | or else[73] [74]
```

Naming Rule

(N1) If any complete state identifier is a mode identifier, then all complete state identifiers in that annex subclause must also be mode identifiers.

Consistency Rules

[69] BA D.5(17)
[70] BLESSDiffers from BA: mode trigger
[71] BLESSDiffers from BA: mode instead of external condition
[72] BLESSDiffers from BA: restricted to subcomponent port
[73] **Reconciliation:** `cand` → `and then`
[74] **Reconciliation:** `cor` → `or else`

(C1) Modal behavior must conform to AS5506B §12, Modes and Mode Transitions.[75]

(C2) If transitions from a complete state that represents a mode in the containing component, then the behavior_condition associated with these transitions must be consistent with the corresponding `mode_transition_triggers` of a `mode_transition`.[76][77][78]

I 6.8 Synchronization

(1) An automaton is said *well synchronized* iff all its transitions from one complete state to another can be performed using one big step (Annex I 2.22). If all series of transitions from one complete state to another in an automaton do not use an output port twice or more, then the automaton is well synchronized. [79]

(2) For a well-synchronized automaton, one can reduce the execution states introduced in the representation of action sequences by substituting variable names by their definitions in the formula that use them. For instance, $\{(s1, g1, s, v = u), (s, g2, s2, f2,)\}$ can be reduced as $\{(s1, g1 \wedge (g2[v/u]), s, v = u \wedge (f2[v/u]))\}$. One can also reduce action sequences and action sets by composing formulas representing independent actions. For instance, $\{(s1, g1, s, p1 = v1), (s, g2, s2, p2 = v2)\}$ can be reduced as $\{(s1, g1 \wedge g2, s2, p1 = v1 \wedge p2 = v2)\}$ iff $p1 \neq p2$, and so on. A well-synchronized automaton can be represented without execution states. [80]

[75] BA D.3(C4)
[76] BA D.4(C4)
[77] AS5506B §12 Modes and Mode Transitions
[78] **Reconciliation:** mode
[79] JP
[80] JP

Chapter I 7

Thread Dispatch

I 7.1 Dispatch Condition

(1) Any transition leaving a `complete` state must have a *dispatch condition* that begins `on dispatch`. When a component has `Periodic` dispatch protocol, no *dispatch expression* is needed. For components with other dispatch protocols, a dispatch expression determines when a component is dispatched.

(2) A dispatch condition must be met to transition from a `complete` state.[1] A dispatch condition determines whether a transition is taken, and an action is performed when the condition evaluates to true. A dispatch condition is a Boolean-valued expression (disjunction of conjunctions of dispatch triggers) that specifies the logical combination of triggering events for the next dispatch. A *dispatch trigger* can be the arrival of an event or event data on an event port or an event data port, the receipt of a call on a provided subprogram access, or a timed event–either periodic dispatch or timeout). The ports used in the dispatch condition must be consistent with the ports listed in the core AADL model as dispatch triggers.

(3) A dispatch trigger can be the arrival of events or event data on ports, calls on provides subprogram access features, the stop event, and occurrence of dispatch related and completion related time outs.[2]

(4) Dispatch conditions must be evaluated by the run-time system, not the component, and must be insensitive to component state. Dispatch conditions must not depend upon which complete state is being resumed from, nor from persistent values of variables. Dispatch conditions must not consume events; dispatch conditions must decide solely on event's existence, not their data, nor queue depth. If no dispatch logical expression is supplied, dispatch occurs upon the default dispatch condition defined for the component's `Dispatch_Protocol`[3] property.

(5) A dispatch condition may be absent (just `on dispatch`) indicating default dispatch at the end of the thread's period. Periodic dispatches are always considered to be implicit unconditional dispatch triggers on complete

[1] BA D.4(2)
[2] BA D.4(3)
[3] AS5506B §A.2 Predeclared Thread Properties

states and handled by dispatch conditions without dispatch trigger condition.[4]

(6) Dispatch conditions are evaluated to determine whether a dispatch occurs. If there are multiple outgoing transitions, the dispatch condition (if present) is evaluated to determine which transition is taken. If multiple transitions are eligible, then the priority value (I 6.4) determines an evaluation ordering, otherwise one of the eligible transitions is taken non-deterministically. The higher the priority value is, the higher the priority of the transition is.[5]

(7) When the `Dispatch_Protocol` property is `Timed` or `Hybrid`, the value of the time out dispatch condition is given by the `Period` property of the component.[6]

Grammar

```
dispatch_condition ::=
  on dispatch [ dispatch_expression ] [ frozen frozen_ports ]
dispatch_expression ::=
  dispatch_conjunction  { or dispatch_conjunction }*
  | stop
  | dispatch_relative_timeout_catch
  | completion_relative_timeout_catch
  | provides_subprogram_access_identifier
dispatch_conjunction ::=    dispatch_trigger  { and dispatch_trigger }*
dispatch_trigger ::=
  in_event_port_name | in_event_data_port_name | port_event_timeout_catch[7]
```

(8) A *dispatch trigger* is an event which causes the dispatch condition to be evaluated. The value of a dispatch condition is a boolean expression of dispatch triggers. Event arrival at either event ports or event data ports causes a dispatch trigger referenced by the port's identifier. The timeout dispatch trigger is covered in section I 7.2 Timeout Dispatch Trigger.

(9) All *stop events* are dispatch triggers, caused by arrival of an event on the implicit *stop port*, to model initiation of finalization and transition from a complete state to the a state, possibly via one or more execution states. If the core property finalize entrypoint is already specified[8] then it can be used as an implicit finalization action, otherwise it can be specified as action on transitions from complete states.[9]

(10) The core AADL standard defines which ports are implicitly frozen at dispatch time, i.e., port that actually triggers a dispatch, or ports that do not trigger a dispatch. In the behavior annex subclause it is possible to explicitly specify as part of the dispatch condition a list of additional ports that must also be frozen although they do not take part to the dispatch condition. Otherwise, the port freeze action, `>>` can be used as a transition action.[10]

Grammar

```
frozen_ports ::= in_port_name { , in_port_name }*
```

[4] BA D.4(4)
[5] BA D.3(27)
[6] BA D.3(28)
[7] BLESSDiffers from BA: `timeout` as dispatch trigger
[8] AS5506B §5.4.1 Thread States and Actions
[9] BA D.4(6)
[10] BA D.4(1)

Naming Rules

(N1) The incoming port identifier in the frozen port list must refer to incoming ports in the component type to which the behavior annex subclause is associated.[11]

(N2) The incoming port identifiers and subprogram access feature identifiers that represent dispatch trigger events must refer to the respective feature in the component type to which the behavior annex subclause is associated.[12]

Legality Rules

(L1) The specification of frozen ports in the dispatch condition must be consistent with that of the core AADL model.[13] [14]

(L2) Table I 7.1 sums up the compatibility rules between the dispatch_protocol property values defined in the core standard and the dispatch_trigger_condition used in a behavior annex. This table is only relevant when the property and the annex are applied to a component of the thread category.[15]

Table I 7.1: Dispatch Protocol-Trigger Compatibility

`dispatch_trigger`	Periodic	Sporadic	Aperiodic	Hybrid	Timed
∅ (none)	X			X	X
`dispatch_expression`		X	X	X	X
Provides Subprogram Access		X	X	X	X
`stop`	X	X	X	X	X
`timeout` (only)					X

Consistency Rules

(C1) The specification of frozen ports in the dispatch condition must be consistent with that of the core AADL model.[16] [17]

Legality Rule

(L3) A behavior annex specification for a subprogram must not contain a dispatch condition in any of its transitions.[18]

Semantics

(S1) Where i is an interval, p is an input event port identifier, e is an event, S is a state (the start node of a satisfying lattice), and A, B, C, and D are dispatch triggers:

$\mathfrak{M}_S[\![A \texttt{ and } B]\!] \equiv \mathfrak{M}_S[\![A]\!] \wedge \mathfrak{M}_S[\![B]\!]$

(dispatch condition may be conjunction of dispatch triggers)

$\mathfrak{M}_S[\![(A \texttt{ and } B) \texttt{ or } (C \texttt{ and } D)]\!] \equiv (\mathfrak{M}_S[\![A]\!] \wedge \mathfrak{M}_S[\![B]\!]) \vee (\mathfrak{M}_S[\![C]\!] \wedge \mathfrak{M}_S[\![D]\!])$

[11] BA D.4(N1)
[12] BA D.4(N2)
[13] BA D.3(L9)
[14] AS5506B §5.4.8 Runtime Support For Threads
[15] BA D.4(L1)
[16] BA D.3(L9)
[17] AS5506B §5.4.8 Runtime Support For Threads (although this section says nothing about frozen ports)
[18] BA D.3(L5)

(dispatch condition may be disjunction of conjunctions of dispatch triggers)
$\mathfrak{M}_S[\![\ \mathtt{p}\ ]\!] \equiv \exists e \in p$
(the meaning of in event port identifier `p` *is when an event exists at port p, it is a dispatch trigger)*

(S2) Execution of a transition occurs when the component is suspended in the transition's source state, its dispatch expression is true, and *no dispatch expression of transition leaving that state has been true since the time-of-previous-suspension* (**`tops`**). For a single transition R, leaving a complete state S, having dispatch expression D,
`R: S -[on dispatch D ]->. . .`,
then R will be dispatched at time t:

$\mathfrak{M}_t[\![dispatch(R)]\!] \equiv \mathfrak{M}_t[\![S]\!] \wedge \mathfrak{M}_t[\![D]\!] \wedge \nexists\, t2 \in \{tops, , t\} \mid \mathfrak{M}_{t2}[\![D]\!]$
(transition R will be dispatched at time t when the component is in state S at time t, dispatch expression D is true at time t, and there was no time t2 since the time-of-previous-suspension tops in which the component was dispatched)

(S3) When multiple transitions lease the same complete state, none of their dispatch conditions must be true since the time-of-previous suspension. For transitions R having dispatch expression D, $R2$ having dispatch expression $D2$, and $R3$ having dispatch expression $D3$, all having complete state S as source,
`R: S -[on dispatch D ]->. . .`,
`R2:S -[on dispatch D2 ]->. . .`,
`R3:S -[on dispatch D3 ]->. . .`,
then R will be dispatched at time t:

$\mathfrak{M}_t[\![dispatch(R)]\!] \equiv \mathfrak{M}_t[\![S]\!] \wedge \mathfrak{M}_t[\![D]\!] \wedge \nexists\, t2 \in \{tops, , t\} \mid (\mathfrak{M}_{t2}[\![D \vee D2 \vee D3]\!])$
(transition R will be dispatched at time t when the component is in state S at time t, dispatch expression D is true at time t, and there was no time t2 since the time-of-previous-suspension tops in which the component was dispatched for any transition leaving state S)

I 7.2 Timeout Dispatch

(1) *Timeout* is a dispatch trigger that is raised after the specified amount of time since the last dispatch or the last completion is expired. In the `Timed` dispatch protocol, the `Timeout` property specifies the timeout value.[19]

Grammar

`dispatch_relative_timeout_catch ::= timeout`

`completion_relative_timeout_catch ::= timeout behavior_time`[20]

(2) Timeouts may include a list of event port identifiers, in or out, data or not. An event, in or out, on a port in the list resets and starts the timeout, regardless of component state. The component need not be in the source state of

[19] BA D.4(5)
[20] BLESSDiffers from BA: port list on port event timeout

the transition having a timeout dispatch trigger to reset/start the timeout. A timeout dispatch trigger may include a port list. In this case, the behavior is as follows:[21][22]

- an event was received or sent by a listed port begins, or resets, the timeout interval
- if no event was received or sent by a listed port during the timeout interval, a dispatch trigger occurs

Grammar

```
port_event_timeout_catch ::=
  timeout ( port_identifier { [ or ] port_identifier }* ) behavior_time[23]
```

(3) A timeout dispatch trigger sans port list and behavior time is dispatch relative using the `Period` property for its duration.[24]

(4) Disjunction(`or`) of port names is optional.

Naming Rule

(N1) A port identifier refers to either an `in` port or an `in event` port.

Legality Rule

(L1) The `dispatch_relative_timeout_catch` condition must only be used for `Timed` threads, and must be declared in only one outgoing transition of a complete state.[25]

Semantics

(S1) Where p_1, p_2, and p_3 are event port identifiers, d is a duration that must either be a literal, or the name of an AADL property of type `Timing_Properties::Time`, and u is an `AADL_Properties:: Time_Units` unit:

$$\mathfrak{M}_t[\![\ \texttt{timeout (p1 p2 p3) d u}\ ]\!] \equiv \begin{array}{l} \mathfrak{M}_{now-d}[\![p_1 \vee p_2 \vee p_3]\!] \wedge \\ \nexists s \in \{now-d, , now\} \mid \mathfrak{M}_s[\![p_1 \vee p_2 \vee p_3]\!] \end{array}$$

(the meaning of `timeout` is an event arrived, or was issued, at one of the listed ports (p_1 p_2 or p_3), d time previously, and no events arrived, or were issued, at any of the listed ports since then)

I 7.3 `abort` and `stop` events

(1) AS5506B defines semantics for `stop` and **`abort`** events.[26] A **`stop`** dispatch trigger occurs when a component is requested to enter its *component halted* state through a *stop* request after completing the execution of a dispatch or while not part of the active mode. In this case, the component may execute a Finalize_Entrypoint before entering the *component halted* state.[27]

[21] BA D.4(5)
[22] BLESSDiffers from BA: timeout
[23] BLESSDiffers from BA: `or` optional in port lists
[24] BA D.4(L2)
[25] BA D.4(L2)
[26] AS5506B §5.4 Threads, esp. Figure 5 Thread States and Actions.
[27] BA D.4(6)

(2) An `abort` dispatch trigger occurs through an *abort* request to cause the component to immediately enter the *component halted* state. For both `stop` and `abort` a `final` state will be entered, never to leave again. The difference is that `stop` executes a `Finalize_Entrypoint` to clean up before halting; that behavior is the action of the `stop` transition.[28]

(3) In a behavior specification, a *final* state can be a complete state (i.e. a *final complete* state). Such a state is an implicit superposition of two states a *complete* state and a *final* state - connected by an implicit transition. This implicit transition from the implicit *complete* state towards the implicit *final* state can only be triggered by the reception of a `stop` event. No other action than the finalization action (represented by the `Finalize_Entrypoint` property from the core language) can be associated to this implicit transition. No execution condition can be associated to this implicit transition. Note that entering (respectively exiting) a final *complete* state stands for entering (resp. exiting) the implicit *complete* state.[29]

(4) In a behavior specification, an `initial` state can be a `complete` state and a `final` state as well (i.e. an `initial final complete state`). Such a state is an implicit superposition of three states an `initial` state, a `complete` state, and a `final` state - connected by two implicit transitions. The first transition, from the implicit `initial` state towards the implicit `complete` state, can only be triggered by the initialization action as defined in the core standard. The second transition, from the implicit `complete` state and towards the implicit `final` state, can only be triggered by the reception of a `stop` event. Note that exiting (respectively entering) an `initial final complete` state stands for exiting (resp. entering) the implicit `complete` state. No other action than the initialization action (call to the `Initialize_Entrypoint` as defined by a property in the core language) can be associated to the first implicit transition. No other action than the finalization action (represented by the `Finalize_Entrypoint` property from the core language) can be associated to the second implicit transition. No execution condition can be associated to those two implicit transitions.[30]

Naming Rules

(N1) The incoming port identifier in the frozen port list must refer to incoming ports in the component type to which the behavior annex subclause is associated.[31]

(N2) The incoming port identifiers and subprogram access feature identifiers that represent dispatch trigger events must refer to the respective feature in the component type to which the behavior annex subclause is associated.[32]

Legality Rules

(L1) `stop` transitions must have `final` states as destinations.

Semantics

(S1) $\mathfrak{M}_S[\![$ **`stop`** $]\!] \equiv \exists e \in$ `stop` and there must be a sequence of zero or more, delay-free execute conditions before reaching a **`final`** state.
(the meaning of ***`stop`*** *is when an event exists at special port* `stop`*, it is a dispatch trigger)*
$\mathfrak{M}_S[\![$ **`abort`** $]\!] \equiv$ immediate component halt
(the meaning of ***`abort`*** *is halt immediately)*[33]

[28] Both `stop` and `abort` will occur automatically, so only users that need to define some special behavior action at their occurrence will use them.
[29] BA D.4(7)
[30] BA D.4(8)
[31] BA D.4(N1)
[32] BA D.4(N2)
[33] AS5506B §5.4.1 Thread States and Actions (20) and Figure 5

Example

(5) This example specifies that the component should be dispatched if either an event arrives at port a, or events have arrived for both ports c and d.

```
annex Behavior_Specification {**
  states S1,S2:state; S3,S4:final state;
  . . .
  transitions
  S1-[on dispatch a or (c and d)]->S2;
  S1-[stop]->S3 {finalize action} ;
  S2-[stop]->S3 {different finalize action} ;
  S1-[abort]->S4 ; --no action, S4 is final for abort
   . . .
**}
```

I 7.4 Thread Providing Subprogram Dispatch

(1) Provides subprogram access features that are declared in a `thread` component type can act as a dispatch triggers. The values of incoming parameters, if any, can then be used by naming the parameter within the scope of the behavior annex.[34]

(2) The core AADL standard supports modeling of remote procedure calls through provides subprogram access features on threads. The arrival of a call acts as a dispatch trigger to the thread. Calls are queued if the thread has not completed a previous dispatch. By default the call is a synchronous call with the calling thread being blocked, which corresponds to a *synchronous* `Subprogram_Call_Type` property.[35] To specify non-blocking calls, a *semi-synchronous* `Subprogram_Call_Type` property must be applied to the subprogram.[36]

[34] BA D.5(20)
[35] AS5506A 5.2
[36] BA D.5(21)

Chapter I 8

Action

(1) Actions associated with transitions are action blocks that are built from basic actions and a minimal set of control structures allowing action sequences, action sets, conditionals and finite loops. Action sequences are executed in order, while actions in actions sets can be executed in any order. Finite loops allow iterations over finite integer ranges.[1]

I 8.1 Behavior Actions

(1) The *behavior actions* may be a single asserted action (I 8.2), sequential composition of actions (I 8.5), or concurrent composition of actions (I 8.6).

```
behavior_actions ::=
  asserted_action | sequential_composition | concurrent_composition
```

I 8.2 Asserted Action

(1) An *asserted action* is an action that may have assertions as pre- and post-conditions.[2] No terminating semicolon occurs after the post-condition. Semicolon is used for sequential composition.

```
asserted_action ::=
  [ precondition_assertion ] action [ postcondition_assertion ]
```

Semantics

(S1) Where P and Q are predicates, and S is an action:

[1] BA D.6(1)
[2] BLESSDiffers from BA: assertions around actions

$\mathfrak{M}_{\underline{i}}[\![\ll P \gg \ S \ \ll Q \gg]\!] \equiv \mathfrak{M}_{start(\underline{i})}[\![P]\!] \wedge \mathfrak{M}_{end(\underline{i})}[\![Q]\!] \wedge \mathfrak{M}_{\underline{i}}[\![S]\!]$
(the meaning of subprogram behavior is that P is true in the stating state of i*, Q is true in the ending state of* i*, and* i *satisfies S)*

Inference Rule

(S2) An asserted action $\ll P \gg \ S \ \ll Q \gg$ is true, if `P` implies the *weakest precondition* (wp) of `S` and `Q`.

WEAKEST PRECONDITION: **[WP]** $\dfrac{P \rightarrow \text{wp}(S, Q)}{\ll P \gg \ S \ \ll Q \gg}$

(S3) Equivalently, $\ll P \gg \ S \ \ll Q \gg$ has the behavior of an automata transition $T(s, true, d, true)[S]$ from state s in which assertion $\ll P \gg$ holds, to state d in which assertion $\ll Q \gg$ holds while performing action S. [3]

Example

```
<<INVW() and (PCA_Properties::Drug_Library_Size=k)>>
No_Drug_Found!    --indicate drug code not found
 <<DL()>>
```

I 8.3 Action

(1) An *action* may be a basic action (I 8.4), an alternative formula (I 8.7), a loop (I 8.10), a for-all (I 8.9), a locking action (I 8.12) or a block (I 8.8).

Grammar

```
action ::=
   basic_action
   | behavior_action_block
   | alternative
   | for_loop
   | forall_action
   | while_loop
   | do_until_loop
   | locking_action
```

I 8.4 Basic Actions

(1) Basic actions can be assignment actions, communication actions or time consuming actions[4], or no action at all (`skip`). Threads can perform actions forbidden for subprograms such as sending and receiving events and data on ports, or assigning values of variables for the following period.

[3] JP
[4] BA D.6(2)

(2) Communication actions can be freezing the content of incoming ports, initiating a send on an event, data, or event data port, initiating a subprogram call or catching a previously raised execution Timeout exception. Some communication actions include implicit assignments, such as the assignment of actual parameters on subprogram calls (see I 9.1).[5]

Grammar

```
basic_action ::=
    skip
    | assignment
    | simultaneous_assignment
    | communication_action
    | timed_action
    | when_throw
    | combinable_operation
    | issue_exception
    | computation_action
```

I 8.4.1 Skip

(1) A `skip` action does nothing at all.[6]

Semantics

(S1) The weakest precondition of `skip` is the same as its postcondition.

SKIP **[S]**:
$\mathfrak{M}_t[\![\text{wp}(\texttt{skip}, Q)]\!] \equiv \mathfrak{M}_t[\![Q]\!]$ *(skip changes nothing)*

I 8.4.2 Assignment

(1) An assignment evaluates an expression and binds a variable to that value. When the variable name is followed by a `'`, the value is bound to the variable one period hence.

(2) Assignments consist of a value expression and a target reference for the value assignment separated by the assignment symbol `:=`. When an assignment action is performed, the result of the evaluation of the right hand side expression is stored into the entity specified by the left hand side target reference. Target references of assignments are local variables, data components acting as persistent state variables, and outgoing features such as ports and parameters.[7]

(3) When assignment actions are used in concurrent composition, then the assigned values are not accessible to

[5] BA D.6(4)
[6] BLESSDiffers from BA: `skip`
[7] BA D.6(3)

expressions of other assignment actions in the same concurrent composition by naming the assignment target.[8]

(4) The keyword `any` should be used to represent non-deterministic behaviors. The purpose of `any` is to represent easily that the assigned value could take any of the possible value determined by the data type. This could be used for formal verification or for simulation purpose using a randomly generated value. The `any` keyword is incompatible with the use of code generation techniques.[9]

Grammar

```
assignment ::=
  variable_name [ ' ] := ( expression | record_term | any )
```

(5) When assigning a variable of record type, the value can be expressed as *record term*.[10]

```
record_term ::= ( { record_value }+ )
record_value ::= field_identifier => value ;
```

Consistency Rule

(C1) The type of the assigned value must be consistent with the type of the assignment target. The corresponding literal values are acceptable values for those types.[11]

Legality Rules

(L1) Only periodic components may delay assignment using `'`.

(L2) In an assignment action, the type of the value expression must match the type of the target.[12]

Semantics

(S2) The effect of assigning the value of an expression to a variable is defined using weakest precondition predicate transformers. Where Q is an Assertion, n is a variable name, e is an expression, t is the time of assignment, d is the duration of the period of a periodic component, and $Q|_e^n$ means to replace every occurrence of expression e in Q with variable name n:

THREAD ASSIGNMENT **[TA]**:
$\mathfrak{M}_t[\![\text{wp}(\texttt{n:=e}, Q)]\!] \equiv \mathfrak{M}_t[\![Q|_e^n]\!]$ *(wp by substitution of variable with expression)*
$\mathfrak{M}_t[\![\text{wp}(\texttt{n':=e}, Q)]\!] \equiv \mathfrak{M}_{t+d}[\![Q|_e^n]\!]$ *(time-shifted wp by substitution)*

I 8.4.3 Simultaneous Assignment

(1) Simultaneous assignment[13] for components is the same as that for subprograms, but allows assignment of next values of variables.

[8] BA D.6(15)
[9] BA D.6(21)
[10] BLESSDiffers from BA: record assignment
[11] BA D.6(16)
[12] BA D.6(L1)
[13] BLESSDiffers from BA: simultaneous assignment

Grammar

```
simultaneous_assignment ::=
  ( variable_name [ ' ] { , variable_name [ ' ] }+
  :=
  ( expression | record_term | any )
    { , ( expression | record_term | any ) }+ )
```

Semantics

(S3) Where Q is an Assertion, n_1, n_2, n_3, ..., are variable names, and e_1, e_2, e_3, ..., are expressions, and $Q|^{n_1,n_2,n_3,\ldots}_{e_1,e_2,e_3,\ldots}$ means to replace every occurrence of variable name n_x listed with the expression e_x in the corresponding position in Q:

THREAD SIMULTANEOUS ASSIGNMENT **[TSA]**:

$\mathfrak{M}_t[\![\text{wp}(\texttt{(n1,n2,n3,...:= e1,e2,e3,...)},Q)]\!] \equiv \mathfrak{M}_t[\![Q|^{n_1,n_2,n_3,\ldots}_{e_1,e_2,e_3,\ldots}]\!]$

(wp by substituting all listed variables with corresponding expression as in §??)

$\mathfrak{M}_t[\![\text{wp}(\texttt{(n1',n2',n3',...:= e1,e2,e3,...)},Q)]\!] \equiv \mathfrak{M}_{t+d}[\![Q|^{n_1,n_2,n_3,\ldots}_{e_1,e_2,e_3,\ldots}]\!]$

(time-shifted substitution of variables by expressions in postcondition)

I 8.4.4 Computation Action

(1) A *computation action* models the duration of execution for scheduling, and timing analysis.[14] Presumably implementation will replace communication actions with behavior actions, and derive information for scheduling and timing from simulations or analyses of compiled code.[15]

(2) **computation(**`min .. max`**)** expresses the use of the CPU for a duration between `min` and `max`. The time is specified in terms of time units as defined by the `Time_Units` property type in the core standard. One value can be specified when `min` and `max` are the same.[16]

Grammar

```
computation_action ::=
  computation ( behavior_time [ .. behavior_time ] )
  [ in binding ( processor_unique_component_classifier_reference
    { , processor_unique_component_classifier_reference }+ ) ]

behavior_time ::= integer_expression unit_identifier
```

Legality Rule

(L3) The unit identifier must be a time unit.

(L4) The time values must be integers.

(L5) The value of the `max` time must be greater than or equal to the value of the `min` time.[17]

[14] **Reconciliation:** computation action
[15] BA D.6(5)
[16] BA D.6(18)
[17] BA D.6(L8)

Semantics

(S4) When a single behavior-time is used, that defines the difference between suspension and dispatch times.

(S5) When two behavior-times are used, that defines the allowed range in the difference between suspension and dispatch times.

I 8.4.5 Issue Exception

An *issue exception* action forces transition to an identified state, and send the message string to the implicit `Exception out event data port`.[18]

Grammar

```
issue_exception ::=
  exception ( [ exception_state_identifier , ] message_string_literal )
```

I 8.5 Sequential Composition

(1) Sequential composition of actions performs them one after another, in order of appearance.[19]

Grammar

```
sequential_composition ::= asserted_action { ; asserted_action }+
```

Semantics

(S1) Where `S1` and `S2` are formulas, and i, j, and m are intervals:

$\mathfrak{M}_{\underline{i}}[\![\ \texttt{S1;S2}\ ]\!] \equiv \exists \underline{j} \subset \underline{i}, \underline{m} \subset \underline{i} \mid \mathfrak{M}_{\underline{j}}[\![\ \texttt{S1}\ ]\!] \wedge \mathfrak{M}_{\underline{m}}[\![\ \texttt{S2}\ ]\!] \wedge \mathit{start}(\underline{m}) = \mathit{end}(\underline{j})$
(there exist subintervals $\underline{j}$ *and* $\underline{m}$ *of* $\underline{i}$ *such that* $\underline{j}$ *satisfies* `S1`, $\underline{m}$ *satisfies* `S2`, *and the least element of* $\underline{m}$ *is the upper bound of* $\underline{j}$*)*

Sequential composition is depicted as sequential lattice combination, $\underline{i}_1 \curvearrowright \underline{i}_2$, in Figure I 2.3.

(S2) Equivalently, `S1 ; S2` has the behavior of an automata transition $T(s,g,d,f)[S1;S2]$ translated to the transition system $T \Rightarrow T_1 \cup T_2$ where $T_1 = T(s,g,e,x)[S1]$ and $T_2 = T(e,x,d,f)[S2]$ by introducing a new execution state e and clock formula x. Sequential composition of more than two actions uses this translation inductively. [20]

Inference Rules

SEQUENTIAL COMPOSITION: **[SC]**
$$\frac{\ll P \gg\ S_1\ \ll R_1 \wedge R_2 \gg \quad \ll R_1 \wedge R_2 \gg\ S_2\ \ll Q \gg}{\ll P \gg\ S_1\ \ll R_1 \gg\ ;\ \ll R_2 \gg\ S_2\ \ll Q \gg}$$

[18] BLESSDiffers from BA: issue exception
[19] BA D.6(11)
[20] JP

SEQUENTIAL COMPOSITION OF K ASSERTED ACTIONS:

$$\textbf{[SCk]}\ \frac{\begin{array}{c}\ll P_1 \gg\ S_1\ \ll Q_1\ \wedge\ P_2 \gg \\ \ll Q_1\ \wedge\ P_2 \gg\ S_2\ \ll Q_2 \wedge P_3 \gg \\ \cdots \\ \ll Q_{k-1}\ \wedge\ P_k \gg\ S_k\ \ll Q_k \gg\end{array}}{\ll P_1 \gg\ S_1\ \ll Q_1 \gg\ ;\ \ll P_2 \gg\ S_2\ \ll Q_2 \gg\ ;\ \cdots\ ;\ \ll P_k \gg\ S_k\ \ll Q_k \gg}$$

Examples

```
la:=StartButton
  <<(la=StartButton) and (Rx_APPROVED())@now  and PB_DURATION()>>
;
Infusion_Flow_Rate!(Basal_Rate)    --infuse at basal rate
  <<(la=StartButton) and (Infusion_Flow_Rate@now=Basal_Rate@now)
    and PB_DURATION()>>
```

```
      <<VS(now) and LAST_AS(now) and LAST_AP(now)>>
    vs!
      <<vs@now and LAST_AS(now) and LAST_AP(now) and AXIOM_CCI()
      and AXIOM_LRLi_gt_URLi_LIMIT(now)>>
    ;
    cci!(now-last_vp_or_vs)
      <<vs@now and LAST_AS(now) and LAST_AP(now)
         and AXIOM_LRLi_gt_URLi_LIMIT(now)>>
    ;
    last_vp_or_vs := now
      <<(last_vp_or_vs=now) and vs@now and LAST_AS(now) and LAST_AP(now)
        and AXIOM_LRLi_gt_URLi_LIMIT(now)>>
```

```
<<E() and ACTUAL_POSITION>0 and ACTUAL_IN_RANGE()>>
  Delta := -1                          --set the delta
  <<E() and Delta= -1 and (ACTUAL_POSITION-1)>=0 and AXIOM_GT(ACTUAL_POSITION)
      and ACTUAL_POSITION<=PCS::MaxPosition>>
  ;  --close valve one step
   ActuatorCommand(pc:Delta)
  <<ACTUAL_POSITION'=(ACTUAL_POSITION+Delta) and Delta= -1
      and (ACTUAL_POSITION-1)>=0 and E()
      and (ACTUAL_POSITION-1)<=PCS::MaxPosition>>
  ;  --set own estimate of position
  EstimatedActualPosition' := (EstimatedActualPosition-1)
  <<EstimatedActualPosition'=ACTUAL_POSITION'
    and ACTUAL_POSITION'>=0 and ACTUAL_POSITION'<=PCS::MaxPosition>>
```

I 8.6 Concurrent Composition

(1) Concurrently-composed actions are order independent; the actions may be performed in any order, or concurrently with the same result.[21]

```
concurrent_composition ::= asserted_action { & asserted_action }+
```

Legality Rules

[21] BA D.6(11)

(L1) The same local variable must not be assigned in different actions of a concurrent composition.[22]

(L2) The same port must not be assigned in different actions of a concurrent composition.[23]

Semantics

(S1) Where S_1 and S_2 are actions; P and Q are assertions:

CONCURRENT COMPOSITION: **[CC]**
$$\frac{\ll P \gg\ S_1\ \ll Q \gg \quad \ll P \gg\ S_2\ \ll Q \gg}{\ll P \gg\ S_1\ \&\ S_2\ \ll Q \gg}$$

(S2) Where $A_1, A_2, \ldots, A_k$ are asserted actions: $A_j = \ll P_j \gg\ S_j\ \ll Q_j \gg$ for $j \in 1..k$; P and Q are assertions:

CONCURRENT COMPOSITION OF K ASSERTED ACTIONS:

$$\textbf{[CCk]}\ \frac{\begin{array}{c} P \to P_1, P \to P_2, \cdots, P \to P_k \\ \ll P_1 \gg\ S_1\ \ll Q_1 \gg \\ \ll P_2 \gg\ S_2\ \ll Q_2 \gg \\ \cdots \\ \ll P_k \gg\ S_k\ \ll Q_k \gg \\ Q_1 \wedge Q_2 \wedge \cdots \wedge Q_k \to Q \end{array}}{\ll P \gg\ \{A_1\ \&\ A_2\ \&\ \ldots\ \&\ A_k\}\ \ll Q \gg}$$

In general, when the optional precondition P_j is omitted from asserted action A_j, then P may be used in its place. When all of the optional postconditions Q_j are omitted, then Q may be used for each. If any postconditions Q_j are included, then **`true`** may be used for omitted postconditions.

(S3) Concurrent composition is depicted as concurrent lattice combination, $\underline{i}_1 \Downarrow \underline{i}_2$, in Figure I 2.3. Where S_1 and S_2 are actions, and i, j, and m are intervals:

$$\mathfrak{M}_{\underline{i}}[\![\texttt{S1 \& S2}]\!] \equiv \exists \underline{j} \subset \underline{i}, \underline{m} \subset \underline{i} \mid \begin{array}{l} \mathfrak{M}_{\underline{j}}[\![S_1]\!] \wedge \mathfrak{M}_{\underline{m}}[\![S_2]\!], \\ start(\underline{i}) = start(\underline{m}) = start(\underline{j}), \\ end(\underline{i}) = end(\underline{m}) = end(\underline{j}) \end{array}$$

(there exist subintervals $\underline{j}$ *and* $\underline{m}$ *of* $\underline{i}$ *such that* $\underline{j}$ *satisfies* `S1`, $\underline{m}$ *satisfies* `S2`, *and* $\underline{i}$, $\underline{j}$, *and* $\underline{m}$ *share least elements and upper bounds)*

Semantics for more than two concurrently-composed actions are defined inductively.

(S4) Equivalently, `S1 &S2` has the behavior of an automata transition $T(s,g,d,f)[S1 \& S2]$ translated to the synchronous composition[24] $(T_1|T_2)[(s,s)/s,(d,d)/d]$ of transition systems where $T_1 = T(s,g,d)[S1]$ and $T_2 = T(s,g,d)[S2]$ substituting the composed states (s,s) and (d,d) by s and d. [25]

[22] BA D.6(L3)
[23] BA D.6(L4)
[24] put reference to synchronous composition here
[25] JP

Example

```
T18_VRP_EXPIRED :  --vs after VRP expired
check_vrp -[sv? and not tnv? and (vrp<=(now-last_vp_or_vs))]-> va
  {<<VS(now) and LAST_VP_OR_VS(now) and LAST_AS(now) and LAST_AP(now)>>
  vs!
   <<vs@now and LAST_AS(now) and LAST_AP(now)>>
      -- and AXIOM_LRLi_gt_URLi_LIMIT(now)
  &
  cci!(now-last_vp_or_vs)
  &
  last_vp_or_vs := now
   <<(last_vp_or_vs=now) and LAST_VP_OR_VS(now)>>};
```

I 8.7 Alternative

(1) An *alternative* action using guarded actions (or commands) makes the proof semantics symmetric. A boolean expression *guards* each alternative; guards may be evaluated in any order.[26] At least one of the guards must be true. If more than one guard is true, any of their alternatives may be performed.

(2) An alternative action using if-elseif-else makes semantics asymmetric.[27] The order is now significant in that cascading alternatives assume that no previous alternative was taken. Sometimes, that important, sometimes not, which leads to misunderstanding and error.

```
alternative ::=
   if  guarded_action { [] guarded_action }+ fi
   |
   if ( boolean_expression_or_relation ) behavior_actions
   { elsif ( boolean_expression_or_relation )
       behavior_actions }*
   [ else behavior_actions ]
   end if
guarded_action ::=
   ( boolean_expression_or_relation )~> behavior_actions
```

Legality Rules

(L1) At least one of the guards must be true.

(L2) The weakest precondition of alternative is least one guard must be true.

Semantics

(S1) The semantics of if-fi alternative is classic[28] guarded commands.

$\mathfrak{M}_i[\![$ `if (B1)->S1[](B2)->S2[] ⋯ [](Bn)->Sn fi` $]\!]$

[26] BLESSDiffers from BA: if [] fi
[27] **Reconciliation:** add if-elsif-else
[28] Dijkstra-Gries

$$
\begin{array}{rc}
 & \mathfrak{M}_{start(\underline{i})}[\![B_1]\!] \rightarrow \mathfrak{M}_{\underline{i}}[\![S_1]\!], \\
 & \mathfrak{M}_{start(\underline{i})}[\![B_2]\!] \rightarrow \mathfrak{M}_{\underline{i}}[\![S_2]\!], \\
\equiv & \vdots \\
 & \mathfrak{M}_{start(\underline{i})}[\![B_n]\!] \rightarrow \mathfrak{M}_{\underline{i}}[\![S_n]\!], \\
 & \mathfrak{M}_{start(\underline{i})}[\![B_1]\!] \vee \mathfrak{M}_{start(\underline{i})}[\![B_2]\!] \vee \cdots \vee \mathfrak{M}_{start(\underline{i})}[\![B_n]\!]
\end{array}
$$

(whenever a guard is true at the beginning of interval i, its action will be true over all of i, and at least one of the guards is true)

(S2) Equivalently, `if (B1)->S1[](B2)->S2[]fi` has the behavior of an automata transition $T(s,g,d,f)$[`if (B1)->S1[](B2)->S2[]fi`] translated to the union of transition systems $T \Rightarrow T_1 \cup T_2$ where $T_1 = T(s, g \wedge B1, d)[S1]$ and $T_2 = T(s, g \wedge B2, d)[S2]$. An arbitrary number of alternatives is defined similarly making a transition system for each alternative. At least one alternative guard must be true. [29]

(S3) The semantics of if-elsif-else alternative is defined in terms of and equivalent if-fi alternative.

$$
\begin{array}{rc}
\mathfrak{M}_{\underline{i}}[\![\texttt{if (B1)S1 elsif (B2)S2 ... elsif (Bn)Sn else Sm end if}]\!] & \\
 & \mathfrak{M}_{start(\underline{i})}[\![B_1]\!] \rightarrow \mathfrak{M}_{\underline{i}}[\![S_1]\!], \\
 & \mathfrak{M}_{start(\underline{i})}[\![B_2]\!] \wedge \neg\mathfrak{M}_{start(\underline{i})}[\![B_1]\!] \rightarrow \mathfrak{M}_{\underline{i}}[\![S_2]\!], \\
\equiv & \vdots \\
 & \mathfrak{M}_{start(\underline{i})}[\![B_n]\!] \wedge \neg\mathfrak{M}_{start(\underline{i})}[\![B_1]\!] \ldots \wedge \neg\mathfrak{M}_{start(\underline{i})}[\![B_{n-1}]\!] \rightarrow \mathfrak{M}_{\underline{i}}[\![S_n]\!], \\
 & \neg\mathfrak{M}_{start(\underline{i})}[\![B_1]\!] \ldots \wedge \neg\mathfrak{M}_{start(\underline{i})}[\![B_n]\!] \rightarrow \mathfrak{M}_{\underline{i}}[\![S_m]\!]
\end{array}
$$

(S4) Equivalently, `if (B1)S1 elsif (B2)S2 else Sm end if` has the behavior of an automata transition $T(s,g,d,f)$[`if (B1)S1 elsif (B2)S2 else Sm end if`] translated to the union of transition systems $T \Rightarrow T_1 \cup T_2 \cup T_m$ where $T_1 = T(s, g \wedge B1, d)[S1]$, $T_2 = T(s, g \wedge B2 \wedge \neg B1, d)[S2]$, and $T_m = T(s, g \wedge \neg B2 \wedge \neg B1, d)[Sm]$. An arbitrary number of alternatives is defined similarly making a transition system for each alternative. [30]

Inference Rules

(S5) Where `B1` , `B2` , and `Bn` are boolean-valued expressions, and `S1` , `S2` , and `Sn` are actions:[31]

ALTERNATIVE:

$$
\frac{\begin{array}{c} P \rightarrow B_1 \vee B_2 \vee \cdots \vee B_n, \\ P \wedge B_1 \rightarrow P_1,\ P \wedge B_2 \rightarrow P_2,\ \ldots,\ P \wedge B_n \rightarrow P_n, \\ \ll B_1 \wedge P \gg S_1 \ll Q_1 \gg,\ \ll B_2 \wedge P \gg S_2 \ll Q_2 \gg,\ \ldots,\ \ll B_n \wedge P \gg S_n \ll Q_n \gg, \\ \ll Q_1 \gg \rightarrow \ll Q \gg,\ \ll Q_2 \gg \rightarrow \ll Q \gg,\ \ldots,\ \ll Q_n \gg \rightarrow \ll Q \gg, \end{array}}{\texttt{<<P>> if (B1)-> <<P1>>S1<<Q1>> [] ... []}\ (\texttt{Bn})\texttt{-> <<Pn>>Sn<<Qn>> fi <<Q>>}} \text{[IF]}
$$

Examples

```
if
  --good SpO2 reading, reset counter
  (SensorConnected? and not MotionArtifact?)~>
```

[29] JP
[30] JP
[31] The ... represent elided guarded actions.

```
      <<SensorConnected^0 and not MotionArtifact^0>>
    numBadReadings := 0
      <<NUMBAD()>>
  []  --bad Sp02, not enough bad reading to alarm
    (MotionArtifact? or not SensorConnected?)~>
      <<all j:integer in 0 ..numBadReadings are
         MotionArtifact^(-j) or not SensorConnected^(-j)>>
    numBadReadings :=numBadReadings+1
      <<NUMBAD()>>
  fi
```

```
  if (SensorConnected? and not MotionArtifact?)
  then
    numBadReadings := 0
  else
    numBadReadings :=numBadReadings+1
  end if
```

```
  if
    (guard_A)~> action_A
  []
    (guard_B)~> action_B
  []
    (guard_C)~> action_C
  []
    (guard_D)~> action_D
  fi
```

I 8.8 Behavior Action Block

(1) A *behavior action block* (optionally) introduces local variables of bounded type and lifetimes.

Grammar

```
behavior_action_block ::=
    [ quantified_variables ] { behavior_actions }
    [ timeout behavior_time ]  [ catch_clause ]

quantified_variables ::= declare { behavior_variable }+
```

(2) The optional `catch_clause` allows specification of behavior upon occurrence of exceptions as defined in I 8.11, Exception Handling.[32]

(3) Quantified variables are local variables, and exist only during lattice construction. Behavior variables are defined in I 6.3, Behavior Variables.[33]

Legality Rule

(L1) Timeout on behavior actions are not allowed on behavior transitions with timeout conditions.[34]

Semantics

(S1) Where v is a variable identifier,t is a type, e is an expression, and S is a formula:

[32] BLESSDiffers from BA: catch clause
[33] BLESSDiffers from BA: local variables for block
[34] BA D.3(L11)

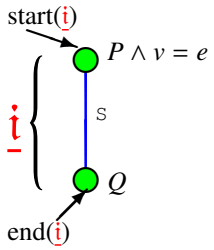


Figure I 8.1: Behavior Action Block Lattice

$\mathfrak{M}_{\underline{i}}[\![$ `<<P>> declare v:t:=e; {S} <<Q>>` $]\!]$
$\equiv$ $\exists$ v∈t | $\mathfrak{M}_{start(\underline{i})}[\![v]\!] = \mathfrak{M}_{start(\underline{i})}[\![e]\!]$
$\wedge\ \mathfrak{M}_{\underline{i}}[\![S]\!]\ \wedge\ \mathfrak{M}_{start(\underline{i})}[\![P]\!] \wedge \mathfrak{M}_{end(\underline{i})}[\![Q]\!]$
(there exists a variable v of type t, where v equals the value of e evaluated at at start(i), P is true at start(i), Q is true at end(i), and i satisfies S)

$\mathfrak{M}_{\underline{i}}[\![$ `{S}` $]\!] \equiv \mathfrak{M}_{\underline{i}}[\![S]\!]$
(the meaning of braces without quantified variables is its contents)

Inference Rule

BLOCK: [B]
$$\frac{\begin{array}{c}\exists v \in t \mid \ll P \wedge v = e \gg \ \rightarrow\ \ll A \gg \\ \ll A \gg\ S\ \ll B \gg \\ \ll B \gg \rightarrow \ll Q \gg\end{array}}{\ll P \gg\ \texttt{declare v:t:=e; \{<<A>>S <<B>>\}}\ \ll Q \gg}$$

(S2) Equivalently, `<<P>> declare v:t:=e; {<<A>> S <<B>>} <<Q>>` has the behavior of an automata transition $T(s, v = e, d, true)[S]$ from state s in which assertion $\ll P \gg$ holds, to state d in which assertion $\ll Q \gg$ holds while performing action S. Additionally, assertion $\ll A \gg$ must be derivable from $\ll P \gg$ with the initial value of v, $\ll P \wedge v = e \gg \rightarrow \ll A \gg$, and also $\ll B \gg \rightarrow \ll Q \gg$ in which v may appear in B, but not Q. [35]

Example

Block from cardiac pacemaker rate controller:

```
declare  --transient, local variables
  siri : real := (msr>(lrl-(f*(xl-thresh)))??msr : lrl-(f*(xl-thresh)));
  z : real := ((lrl-msr)*(lrl+msr)) / (2*(rt-lrl));
  y : real := ((lrl-msr)*(lrl+msr)) / (2*(ct-lrl));
  up_siri : real := ((cci-z)<siri ?? siri : cci-z);
  dn_siri : real := ((cci+y)<siri ?? cci+y : siri);
  down : real :=  cci*(1.0+(drs/100.0));  --down rate smoothing
  up : real := cci*(1.0-(urs/100.0));  --up rate smoothing
{
<<((lrl-url)<>0) and (z=Z()) and (y=Y())
    and (siri=SIRi()) and (dn_siri=DN_SIRi()) and (up_siri=UP_SIRi())
    and (down=DOWN()) and (up=UP())>>
```

[35] JP

```
  dav!((cci*((av-min_av)/(lrl-url))) + min_av)
  &
  min_cci!((url>(up_siri > up??up_siri: up)??url:(up_siri >up??up_siri:up)))
  &
  max_cci!((lrl<(dn_siri<down??dn_siri:down)??lrl:(dn_siri<down??dn_siri:down)))
  <<true>>
  }
```

I 8.9 Forall

(1) To specify concurrent execution of many similar actions *forall action* defines local variables restricted to an integer range,[36] that may then be used as variables within its block

Grammar

```
forall_action ::=
  forall variable_identifier { , variable_identifier }*
    in integer_expression .. integer_expression
    behavior_action_block
```

Semantics

(S1) Two, identical semantics for forall action are given: weakest-precondition predicate transformer and inference rule. Weakest-precondition is much preferred, but can only be used when the wp of the body is known. Semantics for multiple quantified variables is the same as replacing "a" with a sequence of variable identifiers.

(S2) The weakest-precondition predicate transformer for forall action, is the conjunction of the weakest precondition predicate transformed bodies with the quantified variable replaced by each value in the range, and that those transformed, substituted bodies are interference free.[37] The body that uses the quantified variable is $S(a)$.

FORALL ACTION **[FA]**: wp(`forall a in R { S(a) }` , Q) ≡ $\begin{array}{l} \forall a \in R \mid \text{wp}(S(a), Q), \\ \forall a \in R \mid \text{interference-free}(S(a)) \end{array}$

$\mathfrak{M}_{\underline{i}}$[[`forall a in R {<<p(a)>>S(a)<<q(a)>>}`]]
$\quad \exists \underline{i}_1 \in \underline{i}, \underline{i}_2 \in \underline{i}, \ldots, \underline{i}_n \in \underline{i}, \mid$
$\quad \underline{i} = \underline{i}_1 \Downarrow \underline{i}_2 \Downarrow \ldots \Downarrow \underline{i}_n$
≡ $\forall a \in R \mid \mathfrak{M}_{start(\underline{i}_a)}[\![p_a]\!]$
$\quad \forall a \in R \mid \mathfrak{M}_{\underline{i}_a}$[[`<<p(a)>>S(a)<<q(a)>>`]]
$\quad \forall a \in R \mid \mathfrak{M}_{end(\underline{i}_a)}[\![q_a]\!]$
(the satisfying interval is the concurrent composition of intervals satisfying the body for each value a in R)

Inference Rule

(S3) Where $B(i)$ is the value of B after the ith iteration, E is a boolean-valued expression, B is an integer-valued function, S is an action, and P, I, and Q are assertions:

[36] BLESSDiffers from BA: only integer range
[37] Interference freedom is none of the concurrent actions assigns values that other actions either use or assigns.

FORALL ACTION: [FA]
$$\frac{\begin{array}{c} P \rightarrow \forall a \in [lb..ub]\ R_a \\ \ll R_a \gg\ S_a\ \ll T_a \gg \\ \forall a \in [lb..ub]\ T_a\ \rightarrow\ Q \\ \forall a \neq b \in [lb..ub] \text{ interference-free}(S_a, S_b) \end{array}}{\ll P \gg \texttt{ forall a:t in lb..ub \{<<R>>S<<T>>\} } \ll Q \gg}$$

(to prove forall action requires: the precondition imply all inner preconditions, the body is correct, all inner postconditions together imply the postcondition, and all actions are inteference-free)

(S4) Equivalently, `forall a:t in lb..ub <<R>>S<<T>>` has the behavior of an automata transition $T(s,g,d)$[`forall a:t in lb..ub S`] translated to the union of transition systems $T \Rightarrow T_1 \cup T_2 \ldots \cup T_m$ where $T_1 = T(s, g \wedge (t = lb), d)[S]$, $T_2 = T(s, g \wedge (t = lb + 1), d)[S]$, and $T_m = T(s, g \wedge (t = ub), d)[S]$. An arbitrary number of alternatives is defined similarly making a transition system for each alternative. [38]

Example

```
  <<SpO2_INV() and (num_samples<PulseOx_Properties::Num_Trending_Samples)>>
forall i:integer in 1 ..num_samples
  {
    <<spo2[i]=(MotionArtifact^(-i) or not SensorConnected^(-i)??0:SpO2^(-i))>>
   spo2_nxt[i+1]:=spo2[i] --shift old samples
     <<spo2_nxt[i+1]=(MotionArtifact^(-i) or not SensorConnected^(-i)
          ??0:SpO2^(-i))>>
  }
  <<SHFT: :all i:integer in 1 ..num_samples
   are spo2_nxt[i+1]=(MotionArtifact^(-i) or not SensorConnected^(-i)
        ??0:SpO2^(-i))>>
```

I 8.10 Loops

(1) Loops allow actions to be repeated in some controlled manner.

I 8.10.1 While Loop

(1) The *while loop* repeats an action while a guard (boolean expression) is true. While loops may have invariant assertion, and a bound function. The *invariant* must be true before and after each iteration. The *bound function* when positive must imply the guard is true; the bound function when zero or less must imply the guard is false; and, each iteration of the loop must decrease the value of the bound function.

Grammar

```
while_loop ::=
   while ( boolean_expression_or_relation )
   [ invariant assertion ]
   [ bound integer_expression ]
   behavior_action_block
```

Semantics

[38] JP

(S1) Where $B(i)$ is the value of B after the ith iteration, E is a boolean-valued expression, B is an integer-valued function, S is an action, and P, I, and Q are assertions:

Loop: **[L]**
$$\frac{\begin{array}{c} P \rightarrow I \\ I \rightarrow \mathrm{wp}(S, I) \\ (I \wedge \neg E) \rightarrow Q \\ B > 0 \rightarrow E \\ B(i) > B(i+1) \end{array}}{\ll P \gg \texttt{ while (E)invariant <<I>> bound B \{S\} } \ll Q \gg}$$

(S2) Equivalently, **while (E)** { S } has the behavior of an automata transition $T(s, g, d, f)$[**while (E)** { S }] translated to the union of transition systems $T \Rightarrow T_1 \cup T_2 \cup T_3 \cup T_4$ where $T_1 = T(s, g \wedge E, c)[S]$, $T_2 = T(c, E, c)[S]$, $T_3 = T(c, \neg E, d, f)[S]$, and $T_4 = T(s, \neg E, d, f)$, by introducing a new execution state c. If defined, invariant **<<I>>** must hold for states s, d, and c. [39]

Example

```
while ((dc <> dl[k].code) and
    ((PCA_Properties::Drug_Library_Size-k)>0))
  invariant <<INVW()>>
  bound (PCA_Properties::Drug_Library_Size-k)
  {
  <<((PCA_Properties::Drug_Library_Size-k)>0) and INVW()>>
  k:=k+1
  <<(0<(PCA_Properties::Drug_Library_Size-(k-1))) and INVW()>>
  }
<<INVW() and not
   ((dc<>dl[k].code) and
     ((PCA_Properties::Drug_Library_Size-k)>0))>>
```

I 8.10.2 For Loop

(1) A *for loop* is a handy specialization of a while loop, that introduces an integer variable, defined over an integer range,[40] implicitly initialized at the lower bound, incremented after each iteration, and loop termination after the variable equals the upper bound.

Grammar

```
for_loop ::=
  for integer_identifier in integer_expression .. integer_expression
    [ invariant assertion ] { asserted_action }
```

Naming Rule

(N1) The integer identifier of a for control construct represents a variable whose scope is local to the for construct. Such a variable must not be otherwise be visible in scope.[41]

Legality Rules

[39] JP
[40] BLESSDiffers from BA: only integer range
[41] BA D.6(N1)

(L1) The lower bound must be at most the upper bound.

(L2) An integer identifier of a for loop is not a valid target for an assignment action.[42]

Semantics

(S3) Where `a` is a fresh integer variable, `lb` and `ub` are integer-valued expressions for the lower-bound and upper-bound respectively, `I` is a predicate invariant before and after each execution of the loop, and `S(a)` are behavior actions that use `a`:

For: **[FOR]**
$$\frac{\begin{array}{c}\texttt{lb} \le \texttt{ub}\,, \\ \ll P \gg \\ \texttt{variables a:Ideal::integer:=lb;} \\ \texttt{\{while (a<=ub)invariant <<I>>} \\ \texttt{bound ub-a \{S(a);a:=a+1\}\}} \\ \ll Q \gg\end{array}}{\ll P \gg \texttt{ for (a in lb..ub)invariant <<I>> \{ S(a)\} } \ll Q \gg}$$

(S4) Equivalently, **`for (a in lb..ub)`**`{ S(a) }` has the behavior of an automata transition $T(s,g,d)$[**`for (a in lb..ub)`**`{ S(a) }`] translated to the union of transition systems $T \Rightarrow T_1 \cup T_2 \ldots \cup T_m$ where $T_1 = T(s,g,c_1)[S(lb)]$, $T_2 = T(c_1, true, c_2)[S(lb+1)]$, and $T_m = T(c_{m-1}, true, d)[S(ub)]$, by introducing a new execution states $c_1 \ldots c_{m-1}$, where $m = (ub - lb) + 1$. If defined, invariant **`<<I>>`** must hold for states s, d, and $c_1 \ldots c_{m-1}$. [43]

Example

```
for (i in lb..ub) invariant <<A()>> { h[i] := g[i] }
```

I 8.10.3 Do-Until Loop

(1) A *do-until* loop is another specialization of a while loop in which the body is executed unconditionally before evaluating the guard.

```
do_until_loop ::=
  do [ invariant assertion ] [ bound integer_expression ]
  behavior_actions
  until ( boolean_expression_or_relation )
```

Semantics

(S5) Where $B(i)$ is the value of B after the ith iteration, E is a boolean-valued expression, B is an integer-valued function, S is behavior actions, and P, I, and Q are assertions:

Do-Until: **[UNTIL]**
$$\frac{\ll P \gg \texttt{ S <<I>>; while (not E)invariant <<I>> bound B \{ S \} } \ll Q \gg}{\ll P \gg \texttt{ do invariant <<I>> bound B S until (E) } \ll Q \gg}$$

[42] BA D.6(L2)
[43] JP

(S6) Equivalently, **do** S **until** (E) has the behavior of an automata transition $T(s,g,d,f)$[**do** S **until** (E)] translated to the union of transition systems $T \Rightarrow T_1 \cup T_2 \cup T_3$ where $T_1 = T(s,g,c)[S]$, $T_2 = T(c, g \wedge \neg E, c)[S]$, and $T_3 = T(c,E,d,f)$, by introducing a new execution state c. If defined, invariant **<<I>>** must hold for states s, d, and c. [44]

I 8.11 Exception Handling

(1) Safety-critical systems need to define system behavior in every exceptional circumstance. Therefore a way to specify how those exceptions shall be detected, reported, and resolved. BLESSconcerns mostly the reporting part plus some detection. Most importantly, BLESSbehavior does not resolve exceptions; it just emits an event out a port with an error code. The Error Model Annex[45] (EMV2) was made to be used to define system response to faults like BLESSexceptions.

Grammatically, to catch an exception involves adding an optional catch clause to block. Testing for anomalous conditions and raising exceptions adds another basic action.

Grammar

catch_clause ::= **catch** { (exception_label : basic_action) }+

exception_label ::= { *exception*_identifier }+ | **all**

(2) Using **all** as the exception label will catch every exception with the preceding block. Multiple exceptions may cause the same action, catch(x1 x2 x3:a) , or different actions, catch(x1:a1) (x2 x3:a2) .

(3) When exceptions are caught by threads, transition to a special state may be forced with the issue_exception action (I 8.4.5). This is not allowed within subprograms, because they don't have states.

(4) Exceptions may be thrown automatically (i.e. divide by zero) or deliberately with a when-throw action.

when_throw ::= **when** (*boolean*_expression) **throw** *exception*_identifier

Semantics

(S1) Where v is a variable identifier,t is a type, e is an expression, S is a formula, x is an exception identifier, k is an integer-valued expression, and r!(k) is a basic action that sends an event out error data port r with error code k:

$\mathfrak{M}_i$[[**declare** v:t:=e { S } catch(x:r!(k))]]
$\equiv \mathfrak{M}_i$[[**declare** v:t:=e { S }]] $\vee$ ($x \in i \wedge \mathfrak{M}_i$[[r!(k)]])
(either the lattice is constructed normally, or exception x occurred and value k sent out error port r)

(S2) Semantics for multiple exception labels and actions extends that above.[46]

[44] JP
[45] SAE International Standard AS5506B Annex E
[46] SOMEBODY OUGHT TO WRITE A STANDARD LIST OF BUILT-IN EXCEPTIONS LIKE DIVIDE BY ZERO OR ARITHMETIC OVERFLOW.

Example

The following example performs behavior actions when in state `s` and condition `c` is true before transitioning to state `d`. The behavior actions are to do some `work` followed by `morework` concurrently-composed with a when-throw action that raises exception `x`, that when caught sends an event out of port `er`.

```
s -[c]-> d {work;{morework&when(badthing)throw x}catch(x:er!)};
```

I 8.12 Locking Actions

Locking actions are part of the BLESSgrammar to retain backward compatibility with BA programs that use them.[47]

The four locking actions:

***!<** enter critical section

***!>** leave critical section

!< lock data component

!> unlock data component

Grammar

```
locking_action ::= *!< | *!> |
  required_data_access_name !< |  required_data_access_name !>
```

Locking actions were originally omitted from BLESSbecause they void the assumption that the time between dispatch and suspension is 'negligible'. Although the definition of 'negligible' has been deliberately left fuzzy, but stopping execution when some other thread had locked a shared data component, or not exited their mutual critical section. is certainly not negligible.

Anyway, locking actions are a rather blunt mechanism to enforce interference freedom. There are much more adroit means in safety-critical embedded system to share information that don't require locking actions.

Legality Rule

(L1) Accesses to shared data components must be used in a way that no complete state can be reached if a resource has been locked (using for instance `Get_Resource` , or **!<**) and not released (using for instance `Release_Resource` , or **!>**).[48]

Semantics

(S1) Data accesses are similarly subject to a communication protocol between the calling behavior (the client) and the parent component owning the data (the server).

- `required_data_access_name !<`
- `required_data_access_name !>`

[47] **Reconciliation:** locking actions
[48] BA D.6(L7)

- `*!<` and `*!>`

A shared data access lock `dataname!<` is hence encoded by $T(g, s, d)[dataname! <] = (s, g, sds, c), (c, sdf, true, d)$. The output port sds encodes the request to dataname and the input port spf is dispatched when access to dataname is granted. `Get_Resource` and `Release_Resource` actions are treated similarly.

I 8.13 Combinable Operations

Combinable operations are both indivisible and possibly simultaneous. They allow concurrent access to shared data structures. Crucially, combinable operations upon the same target, have the same effect whether executed individually or simultaneously. All combinable operations have three parameters: a target variable of appropriate type, declared to be `shared`; a value to be used in the operation; and an identifier of a local variable to hold the result. Combinable operations on `shared` variables provide concurrent, interference-free access to `spread` data structures, particularly arrays. Used properly, a set of combinable operations has the same effect executed in any order, or simultaneously.

Grammar

```
combinable_operation ::=
  fetchadd
  ( target_variable_name , arithmetic_expression [, result_identifier] )
  |
  ( fetchor | fetchand | fetchxor )
  ( target_variable_name , boolean_expression [, result_identifier] )
  |
  swap
  ( target_variable_name , reference_variable_name , result_identifier )
```

I 8.13.1 Fetch-Add

(1) A single fetch-add operation has the effect of placing the target variable's value into the result variable while indivisibly incrementing the value of the target variable by the value of the expression. Where s is a shared integer name, e is an integer-valued expression, and r is an identifier of an integer variable

$$\mathfrak{M}_{i}[\![\ \texttt{fetchadd(s,e,r)}\ ]\!] \equiv \begin{array}{l} \mathfrak{M}_{end(i)}[\![s]\!] = \mathfrak{M}_{start(i)}[\![s]\!] + \mathfrak{M}_{start(i)}[\![e]\!] \\ \mathfrak{M}_{end(i)}[\![r]\!] = \mathfrak{M}_{start(i)}[\![s]\!] \end{array}$$

(the meaning of fetch-add over an interval i, is the meaning of r at the end of i equals s at the start of i, an s at the end of i equals the sum of s and e at the start of i)

(2) When two fetch-add operations target the same shared integer, the result is non-deterministic, however it must be equivalent to *some* series of fetch-adds. Where s is a shared integer name, e_1 and e_2 are integer-valued expressions, r_1 and r_2 are identifiers of integer variables, and F is the text **`fetchadd(s,e1,r1)& fetchadd(s,e2,r2)`** :

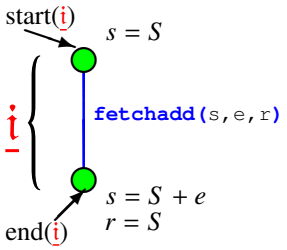


Figure I 8.2: Single Fetch-Add

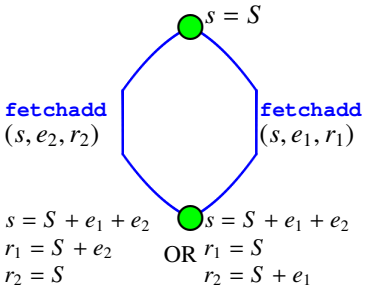


Figure I 8.3: Two Fetch-Adds

$$\mathfrak{M}_{\underline{i}}[\![F]\!] \equiv \begin{array}{l} \mathfrak{M}_{\text{end}(i)}[\![s]\!] = \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] + \mathfrak{M}_{\text{start}(\underline{i})}[\![e_1]\!] + \mathfrak{M}_{\text{start}(\underline{i})}[\![e_2]\!] \\ \mathfrak{M}_{\text{end}(i)}[\![r_1]\!] = \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] \\ \mathfrak{M}_{\text{end}(i)}[\![r_2]\!] = \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] + \mathfrak{M}_{\text{start}(\underline{i})}[\![e_1]\!] \\ \text{OR} \\ \mathfrak{M}_{\text{end}(i)}[\![s]\!] = \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] + \mathfrak{M}_{\text{start}(\underline{i})}[\![e_1]\!] + \mathfrak{M}_{\text{start}(\underline{i})}[\![e_2]\!] \\ \mathfrak{M}_{\text{end}(i)}[\![r_1]\!] = \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] + \mathfrak{M}_{\text{start}(\underline{i})}[\![e_2]\!] \\ \mathfrak{M}_{\text{end}(i)}[\![r_2]\!] = \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] \end{array}$$

(the meaning of concurrent fetch-adds over an interval $\underline{i}$*, is the meaning of* s *at the end of* $\underline{i}$ *equals the sum of* s*,* e_1*, and* e_2 *at the start of* $\underline{i}$*, and either* r_1 *at the end of* $\underline{i}$ *equals* s *at the start of* $\underline{i}$*, and* r_2 *at the end of* $\underline{i}$ *equals the sum of* s *and* e_1 *at the start of* $\underline{i}$*, or* r_2 *at the end of* $\underline{i}$ *equals* s *at the start of* $\underline{i}$*, and* r_1 *at the end of* $\underline{i}$ *equals the sum of* s *and* e_2 *at the start of* $\underline{i}$*)*

(3) If fetch-adds are executed in index order, 1 to n, then the target s will be incremented by the sum of the expressions, each result r_j is the sum of the target and all expressions e_1 to $e_j - 1$, and M is the text **`fetchadd(s,e1,r1); ... ; fetchadd(s,en,rn)`**:[49]

$$\mathfrak{M}_{\underline{i}}[\![M]\!] \equiv \begin{array}{l} \mathfrak{M}_{\text{end}(i)}[\![s]\!] = \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] + \sum_{k=1}^{n} \mathfrak{M}_{\text{start}(\underline{i})}[\![e_k]\!] \\ \mathfrak{M}_{\text{end}(i)}[\![r_1]\!] = \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] \\ \forall j \in 2..n \mid \mathfrak{M}_{\text{end}(i)}[\![r_j]\!] = \begin{array}{l} \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] \\ + \sum_{k=1}^{j-1} \mathfrak{M}_{\text{start}(\underline{i})}[\![e_k]\!] \end{array} \end{array}$$

(4) In the general case of n concurrent fetch-adds, requires use of non-deterministic permutations from §I 2.5. For

[49] semicolon separates elements of action sequences

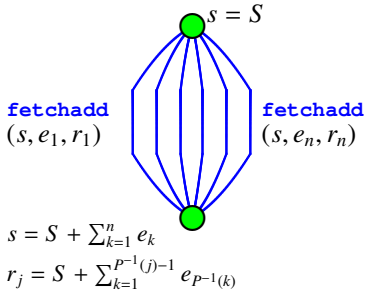


Figure I 8.4: Many Concurrent Fetch-Adds

that, a sequence P is defined to be a permutation of the numbers 1 to n, to indicate any ordering of n fetch-adds, with $P(j)$ being the jth element in P, and $P^{-1}(j)$ being the index of the element of P that holds j. Let C be the text **fetchadd(**s,e1,r1**)&** ... **& fetchadd(**s,en,rn**)** :[50]

$$\mathfrak{M}_{\underline{i}}[\![C]\!] \equiv \begin{array}{l} \mathfrak{M}_{\text{end}(i)}[\![s]\!] = \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] + \sum_{k=1}^{n} \mathfrak{M}_{\text{start}(\underline{i})}[\![e_k]\!] \\ \exists P \leftrightarrows (1,\ldots,n) \mid \forall j \in 1..n \mid \\ \quad \mathfrak{M}_{\text{end}(i)}[\![r_j]\!] = \begin{array}{l} \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] + \\ \sum_{k=1}^{P^{-1}(j)-1} \mathfrak{M}_{\text{start}(\underline{i})}[\![e_{P^{-1}(k)}]\!] \end{array} \end{array}$$

(the target is incremented by sum of the fetch-add parameters; and the results if the fetch-adds occurred in a arbitrary order)

(5) Sometimes, the return value of fetch-add is not needed and omitted. Let C_2 be the text **fetchadd(**s,e1**)&** ... **& fetchadd(**s,en**)**

$$\mathfrak{M}_{\underline{i}}[\![C_2]\!] \equiv \mathfrak{M}_{\text{end}(i)}[\![s]\!] = \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] + \sum_{k=1}^{n} \mathfrak{M}_{\text{start}(\underline{i})}[\![e_k]\!]$$

(the target is incremented by sum of the fetch-add parameters)

(6) However, usually the parameter value is constant 1 or -1. Let I be the text " **fetchadd(**s,1,r1**)&** ... **& fetchadd(**s,1,rn**)** "

$$\mathfrak{M}_{\underline{i}}[\![I]\!] \equiv \begin{array}{l} \mathfrak{M}_{\text{end}(i)}[\![s]\!] = \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] + n \\ \exists P \leftrightarrows (1,\ldots,n) \mid \forall j \in 1..n \mid \\ \quad \mathfrak{M}_{\text{end}(i)}[\![r_j]\!] = \mathfrak{M}_{\text{start}(\underline{i})}[\![s]\!] + P^{-1}(j) - 1 \end{array}$$

(the target is incremented by the number of fetch-add-one operations $s + n$; and the results are some non-deterministic permutation of $(s,\ldots,s+n-1)$)

That each of the results are both in range and different, will be used to for concurrently-accessible data structures to assure interference-freedom. This is the classic "Deli" algorithm wherein patrons take a ticket with a number to await their turn to be served. Fetch-add-one allows an unlimited number of tickets to be issued simultaneously.

(7) Complementing fetch-add-one, is the decrementing parameter -1. Let D be the text **fetchadd(**s,-1,r1**)&** ... **& fetchadd(**s,-1,rn**)** :

[50] ampersand separates elements of action sets

$$\mathfrak{M}_{\text{i}}[\![D]\!] \equiv \begin{array}{l} \mathfrak{M}_{\text{end}(i)}[\![s]\!] = \mathfrak{M}_{\text{start}(\text{i})}[\![s]\!] - n \\ \exists P \leftrightarrows (1, \ldots, n) \mid \forall j \in 1..n \mid \mathfrak{M}_{\text{end}(i)}[\![r_j]\!] = \mathfrak{M}_{\text{start}(\text{i})}[\![s]\!] - P^{-1}(j) + 1 \end{array}$$

I 8.13.2 Fetch-And Fetch-Or Fetch-Xor

Logical operations can be combined too. However, until a need is found, they will be unimplemented.

I 8.13.3 Swap

Dynamic data structures can be concurrently manipulated with swap acting on pointers, or references. However, reference types were deliberately omitted from the type system for BLESS. Therefore, swap is in the grammar, but its use is uncertain, and currently unimplemented.

Chapter I 9

Component Interaction

(1) Threads can interact through shared data component implementations, connected ports and subprogram calls. The AADL execution model defines the way queued event/data of a port are transferred to the thread in order to be processed and when a component is dispatched.[1]

(2) Messages can be received by the component through declared features of the current component type. They can be in or in out data ports; in or in out event ports; in or in out event data ports and in or in out parameters of subprogram access. Event and event data ports are associated with queues.[2]

I 9.1 Communication Action

(1) Communication actions provide interaction with other components. A *communication action* sends or receives values from ports.[3] Actions of `in` ports and `out` ports are covered in following sections.

(2) Actions on ports consist of the input freeze action (`p>>`), the initiate send action with or without value assignment (`p!(v)` or `p!`), and parameterless subprogram calls (`sub()`) or subprogram calls with parameters (`sub(f1:a1, f2:a2, f3:a3)`). Another form of component interaction is through reading and writing of shared data components, which is expressed by the assignment action.[4]

Grammar

```
communication_action ::=
  subprogram_invocation
  | output_port_name ! [ ( expression ) ]
  | input_port_name ? ( target )
  | frozen_input_port_name >>
```

[1] BA D.5(1)
[2] BA D.5(2)
[3] Subprogram invocation is a basic action I 8.4.
[4] BA D.6(10)

```
port_name ::=
  { subcomponent_identifier . }* port_identifier [ [ natural_literal ] ]

target ::=
  local_variable_name
  | output_port_name
  | data_component_reference

data_component_reference ::=
  data_subcomponent_name { . data_subcomponent_name }*
  | data_access_feature_name { . data_field }*
  | data_access_feature_prototype_name { . data_field }*

data_field ::=
  data_subcomponent_name
  | data_access_feature_name
  | data_access_feature_prototype_name
```

Semantics

(S1) Accessing data components outside of a thread break encapsulation of state, is therefore error-prone, and thus stridently discouraged.

I 9.2 Freeze Port

(1) The core language defines that input on ports is determined by default frozen at dispatch time, or at a time specified by the `Input_Time` property[5] and initiated by a `Receive_Input` service call[6] in the source text. From that point in time the input of the port during this execution is not affected by arrival of new data, events, or event data until the next time input is frozen.[7] [8]

(2) Freezing of input port content during execution requires consistency between the `Input_Time` property in the core model and the freeze input action, `p>>`. Similarly, initiating transmission of port output must be consistent between the
`Output_Time` property in the core model and the port output, `p!`.[9]

(3) Ports causing a dispatch event are implicitly frozen at the time specified by the `Input_Time` property if the property specifies a deterministic value. It is also possible to explicitly freeze additional ports if it is consistent with their `Input_Time` property. As long as it remains consistent with the `Input_Time` property of a port, an explicit call to the `Receive_Input` service can be performed thanks to the frozen statement of the dispatch condition. With the same consistency constraints with respect to the `Input_Time` as a transition action.[10]

Consistency Rules

[5] AS5506B §9.2.4 Port Communication Timing
[6] AS5506B §8.3.5 Runtime Support For Ports
[7] BA D.5(3)
[8] **Reconciliation:** >> freeze port
[9] BA D.5(6)
[10] BA D.5(7)

(C1) The specification of frozen ports in the dispatch condition must be consistent with that of the core AADL model.[11]

(C2) Freezing of input port content during execution requires consistency between the `Input_Time` property in the core model and the freeze input action (`p>>`) in the Behavior_Specification.[12] [13]

Semantics

(S1) An input freeze action `p>>` is represented by turning s into a `complete` state with $T(g,s,d)$[`p>>`] = $(s, g?p, true, d)$.[14][15]

I 9.3 In Event Ports

(1) Communication actions do not refer to *in event port*(s). Instead, an event arriving at an event port is a dispatch trigger used in dispatch transition conditions leaving complete states. See I 7.1 Dispatch Condition for thread response to events arriving at an `in event port`.

I 9.4 In Data Ports

(1) An *in data port* holds the most recent value sent to it. The port value may be explicitly assigned to a local variable with a communication action, `p?(v)` , or used in an expression or execute transition conditions (§I 10.1 Value).

(2) The core language defines that data from data ports is made available to the application source code through a port variable with the name of the port. If no new value is available since the previous freeze, the previous value remains available and the variable is marked as not fresh. Freshness can be tested in the application source code via service calls[16].[17]

Table I 9.1: In Data Port AADL Runtime Service Call

Meaning	Grammar	Corresponding service call
read value	`p?`	`Get_Value` and then `Next_Value`
read into variable	`p?(var)`	`Get_Value`

Semantics

(S1) For each `in data port`, r, in the context of interval (state lattice) i:[18]

[11] AS5506B §5.4.8
[12] BA D.5(6)
[13] BA D.5(C1)
[14] JP
[15] This doesn't seem right. No new dispatch, just the value doesn't change until completion. JP, please clarify.
[16] AS5506B §8.3.5 Runtime Support For Ports
[17] BA D.5(4)
[18] The interval i starts at Dispatch time and ends at Completion.

$\mathfrak{M}_t[\![\text{r?(v)}]\!] \equiv \mathfrak{M}_t[\![\text{v}]\!] = \mathfrak{M}_t[\![\text{r}]\!]$
(the variable gets the value of the port at the instant of its evaluation, which because frozen is the value of the port at dispatch)

(S2) To get data from an in data port, a transition read the value of the port and assigns it to the target variable. $T(s, g, d)[r?(v)] = (s, g, c, v = r)$.

Inference Rule

(S3) For an **in data** or **in event data** port p, having assertion P, and variable v

PORT INPUT: **[PI]** $\dfrac{\ll P \wedge v = P \gg \;\rightarrow\; \ll Q \gg}{\ll P \gg \texttt{p?(v)} \ll Q \gg}$

I 9.5 In Event Data Ports

(1) The complexity of `in event data port` behavior comes from its buffer. Unlike plain `in data port` which reports the most recent value received, `in event data port` buffers all the values received while waiting for dispatch, if any. Thus, the need for `updated`, `count`, and `fresh` to monitor the input buffer.

Values of `in event data port` may be used in one of two ways:

p use current value in expression, don't dequeue

p? use current value in expression, dequeue

p?(v) assign current value to variable, dequeue

(2) The core language defines that input on ports is determined by default at dispatch time, or at a time specified by the `Input_Time` property[19] and initiated by a `Receive_Input` service call[20] in the source text.

(3) The core language defines that data from data ports are made available to the component in a port variable.[21] Freshness of this data can be tested as a condition, `p'fresh`.[22]

(4) Event and event data ports have queues and the queues are processed as follows according to the core standard.[23] [24]

- A `Dequeue_Protocol` of `OneItem` makes one item available in a port variable and removes it from the port queue. If the queue is empty the port variable content is considered not fresh.
- A `Dequeue_Protocol` of `AllItems` removes all items from the port queue and places them into a local port queue (local to the state transition system). The component input events of the transition condition and the input actions of the transition action consume data elements from this local port queue. Any data not

[19] AS5506B §8.3.2 Port Input and Output Timing
[20] AS5506B §8.3.5 Runtime Support for Ports
[21] AS5506B §8.3.3 Port Queue Processing
[22] BA D.5(4)
[23] AS5506B §8.3.3 Port Queue Processing
[24] BA D.5(5)

consumed as part of a transition will be lost, when the local queue content is overwritten with new input at the next execution.

- The `Dequeue_Protocol` of `MultipleItems` determines the content of the port queue and makes it available through the local port queue. In this case elements are removed from the queue as they are consumed. Any elements not consumed remain in the queue and become available at the next execution.

Table I 9.2: In Event Data Port AADL Runtime Service Calls

Meaning	Grammar	Corresponding service call
freeze	`p>>`	`Receive_Input`
test of updated	`p'updated`	`Updated`
get unread count	`p'count`	`Get_Count`
test of freshness	`p'fresh`	`Get_Count` > 0
read	`p`	`Get_Value`
read and dequeue	`p?`	`Get_Value` and then `Next_Value`
read into variable and dequeue	`p?(var)`	`Get_Value` and then `Next_Value`

(5) Within a behavior annex subclause, the following constructs are available to get the status and the contents of an input port `p`:[25]

- `p` can be used as a value and returns the most-recent (unless frozen) data stored in the port variable if `p` is a non-empty data port or an event data port with the `OneItem Dequeue_Protocol` using the `Get_Value` runtime service. The value cannot overwritten if the port direction is `in out` by writing to it. It does not dequeue an event and `p'count` is not decremented. Applied to an empty data port, `p` returns `null`.[26]

- `p'count` is equivalent to a call to the `Get_Count` runtime service: `p'count` returns the number of elements available through the port variable. In the case of a data port its value is one, or zero if no new value was received. In the case of an event port or event data port it is the number of frozen elements. If it is strictly positive, `pcount` is decremented when an element is dequeued.[27]

- `p'updated` is equivalent to a call to the Updated runtime service. `pupdated` returns true if some new values were received in port `p` since the last freeze of the port. Note that this operator is used to represent a call to the `Updated` service defined in the core AADL standard. This operation is performed without freezing the port `p`.[28]

- `p'fresh` returns true if the port variable has been refreshed at the previous dispatch. `p'fresh` is equivalent to the expression `Get_Count > 0`. In the case of a data port, this means that it has received a new value by the previous dispatch or freeze. In the case of an event data port this means that one or more elements from the port queue were frozen and are available for processing by the `Behavior_Specification` through `p`. If the port queue is empty at freeze time or the `p?` operation is applied to a port variable with no remaining elements, the value is not considered fresh.[29]

[25] BA D.5(9)
[26] **Reconciliation:** non-dequeued port
[27] BA D.5(9)
[28] BA D.5(9)
[29] BA D.5(9)

- `p?` is equivalent to a call to the `Get_Value` and `Next_Value` runtime services: `p?` dequeues an event or event data from a non empty event port queue. If it is strictly positive, the value of `p'count` is decremented. In the case of an event data port the new first element is available in the port variable.
- When used in a behavior action, `p?(v)` dequeues an event from a non-empty event port queue, returning its value, and `p'count` is decremented using the `Get_Value` and `Next_Value` runtime services. Each use of `p?(v)` dequeues another event, returns its value, and decrements `p'count`. Applied to an empty event data port queue, `p?(v)` causes exception[30] to be thrown.[31]
- `p>>` is equivalent to a call to the `Receive_Input` runtime service, freezing the value so that subsequent references to `p` in the same dispatch receive the same value.[32]

Semantics

(S1) For each `in event data port`, p,

p $\mathfrak{M}_i[\![p]\!] \equiv$ Get_Value
(peak at oldest value w/o removal, use AADL runtime service Get_Value[33])

p? $\mathfrak{M}_i[\![p?]\!] \equiv$ Get_Value then Next_Value
(retrieve oldest value, decrement count use AADL runtime service Next_Value[34])

p?(v) $\mathfrak{M}_i[\![p?(v)]\!] \equiv$ Get_Value then Next_Value
(retrieve oldest value, decrement count use AADL runtime service Next_Value[35])

p'count $\mathfrak{M}_i[\![p'count]\!] \equiv$ Get_Count
(count of values queued, use AADL runtime service Get_Count[36])

p'fresh $\mathfrak{M}_i[\![p'fresh]\!] \equiv$ Get_Count > 0
(new port value, use AADL runtime service Updated[37])

p'updated $\mathfrak{M}_i[\![p'updated]\!] \equiv$ Updated.FreshFlag
(new port value, use AADL runtime service Updated[38])

I 9.6 Concurrency Control

(1) Within a `Behavior_Specification` subclause the value of incoming parameters of the containing subprogram type is returned by the corresponding formal parameter identifier. The value of the parameter has been frozen at the time of the call. Multiple references to a formal parameter return the same value. In the case of

[30] with label `Read_Empty_Event_Data_Port`
[31] BLESSDiffers from BA: empty dequeue exception
[32] By default, port values are frozen at dispatch.
[33] AS5506B §8.3.5 Runtime Support for Ports (50) Get_Value
[34] AS5506B §8.3.5 Runtime Support for Ports (52) Next_Value
[35] AS5506B §8.3.5 Runtime Support for Ports (52) Next_Value
[36] AS5506B §8.3.5 Runtime Support for Ports (51) Get_Count
[37] AS5506B §8.3.5 Runtime Support for Ports (53) Updated
[38] AS5506B §8.3.5 Runtime Support for Ports (53) Updated

subprogram calls, outgoing parameters return their result by assigning them to the local variable named as the corresponding formal parameter identifier. The local variable can then be referenced to return its value.[39]

(2) Access to shared data subcomponents is controlled according to the `Concurrency_Control_Protocol` property specified associated with this data subcomponent.[40] If concurrency control is enabled, critical sections boundaries are defined by one of the following ways:

- By explicit definition of the time range over which a set of referenced shared data subcomponent are accessed. This is done using the `*!<` for starting time (resp. `*!>`) for ending time) locking actions (see I 8.12 Locking Actions) within a behavior action block. If the critical section contains references to several shared data subcomponents, then resource locking will be done in the same order as the occurrence of the references to the shared data subcomponents and resource unlocking will be done in the reverse order. These operators may be used to refine the value of the `Access_Time` property[41] if it has been specified.
- By calls to appropriate provides subprogram access of the corresponding data component that have been explicitly defined to implement the concurrency control protocol.
- By explicit calls to the `Get_Resource` (resp. `Release_Resource`) runtime service[42] that can be achieved using the `!<` (resp. `!>`) operators applied to the shared data subcomponent identifier.

[43]

(3) A transition action can write data values to a shared data component by naming the data component directly or the data access identifiers declared in the component type on the left-hand side of an assignment, i.e., `d := v`.[44]

I 9.7 Out Ports

(1) Messages can be sent within a `Behavior_Specification` subclause through declared features of the current component type. They can be: `out` or `in out` data ports; `out` or `in out` event ports; `out` or `in out` event data ports.[45]

(2) The sending of messages is consistent with the timing semantics of the core language. The core language specifies the output time through an `Output_Time`[46] property and the sending of the output is initiated by a `Send_Output` service call in the source text. For data ports the output is implicitly initiated at completion time (or deadline in the case of delayed data port connections).[47]

(3) Within a `Behavior_Specification` subclause, the following constructs are available to set the value of an out port p:[48].

[39] BA D.5(11)
[40] AS5506B §5.1 Data
[41] AS5506B §8.6 Data Component Access
[42] AS5506B §5.1.1 Runtime Support For Shared Data Access
[43] BA D.5(12)
[44] BA D.5(16)
[45] BA D.5(13)
[46] AS5506B §8.3.2 Port Input and Output Timing
[47] BA D.5(14)
[48] BA D.5(15)

Table I 9.3: Out Communication Actions

Meaning	Grammar	Corresponding service call
put	`p := v`	`Put_Value`
send	`p!`	`Send_Output`
put and send	`p!(v)`	`Put_Value` and then `Send_Output`

- `p!` calls `Put_Value` on an event or event data port. The event is sent to the destination with assigned data, if any, according to the `Output_Time` property.
- `p := d` calls `Send_Output` data or event data port. Data is transferred to the destination port according to the `Output_Time` property.
- `p!(d)` writes data `d` to the event data port `p` and calls the `Put_Value` and then `Output_Time` services. The event is sent to the destination according to the `Output_Time` property.

Consistency Rule

(C1) If the sending time of an output port is specified with the `Output_Time` property[49], then no send output action must be specified in the corresponding behavior actions of the `Behavior_Specification` subclause, or the two statements must be equivalent.[50]

Semantics

(S1) The precondition of port output must imply the assertion property of the port. The conjunction of the precondition of port output and event occurrence must imply the postcondition of port output.

EVENT PORT OUTPUT:

$$\textbf{[EPOut]}\ \frac{A \wedge p@now \rightarrow B \quad A \rightarrow \mathfrak{M}[\![p]\!]}{\ll A \gg\ p!\ \ll B \gg}$$

(S2) For data and event data output

(S3) For each `out event port`, p,

$\mathfrak{M}_i[\![p!]\!] \equiv$ Put_Value()
(send event from port, use AADL runtime service Put_Value with no DataValue parameter[51]; issued at `Output_Time`)

(S4) Equivalently, to send an out port event, a transition causes the clock of the port to be true.
$T(s,g,d)[p!] = (s,g,d,\hat{p})$. [52]

(S5) For each `out event data port`, p,

[49] AS5506B 8.3.2
[50] BA D.5(C2)
[51] AS5506B §8.3.5 Runtime Support for Ports (45) Put_Value
[52] JP

$\mathfrak{M}_i[\![p!(e)]\!] \equiv \mathfrak{M}_i[\![p:=e]\!] \equiv$ Put_Value(e)
(send event data from port, use AADL runtime service Put_Value; issued at `Output_Time`*)*

(S6) Equivalently, to send data on an out event data port, a transition causes the value of the port to be the data sent, and then reset.
$T(s,g,d)[p!(e)] = \{(s,g,c,p=e)(c,g,d,\neg\hat{p})\}$ by introducing a new execution state c. [53]

(S7) To send data on an out data port, a transition causes the value of the port to be the data sent, but not reset.
$T(s,g,d)[p!(e)] = T(s,g,d)[p := e] = (s,g,c,p=e)$. [54]

I 9.8 Subprogram Invocation

(1) Subprograms may be invoked (called) as an action.

Grammar

```
subprogram_invocation ::=
  subprogram_name ( [parameter_list] ) [55]

subprogram_name ::= subprogram_prototype_name
  | required_subprogram_access_name
  | subprogram_subcomponent_name
  | subprogram_unique_component_classifier_reference
  | required_data_access_name . provided_subprogram_access_name
  | local_variable_name . provided_subprogram_access_name

parameter_list ::= parameter { , parameter }*

parameter ::= [ formal_parameter_identifier : ] actual_parameter

actual_parameter ::= target | expression
```

(2) Requires subprogram access features that are declared in the component type can be called inside an action block of a transition. Call parameters can be previously received subprogram parameters, ports value or assigned temporary variables.[56]

(3) Subprogram invocations can be used in sequential composition or concurrent composition. In sequential composition they represent synchronous subprogram calls, while in concurrent composition they represent semi-synchronous calls, i.e., multiple calls are initiated to be performed simultaneously. The concurrent composition is considered to have completed when all subprogram calls within that set have completed. Note that the results from out parameters of one simultaneous call cannot be used as input to another call or other action in the same concurrent composition.[57]

[53] JP
[54] JP
[55] BLESSDiffers from BA: no ! for subprogram invocation
[56] BA D.5(18)
[57] BA D.6(14)

(4) Within Behavior_Specification subclauses, the following constructs are available to call required subprogram `s`:[58]

- `s()` calls the parameter-less subprogram referred to by the requires subprogram access feature `s`.
- `s(f1:a1,...,fn:an)` calls the subprogram referred to by the requires subprogram access feature `s` with the corresponding formal-actual parameter list.[59]

(5) By default the subprogram invocation is synchronous with the calling thread being blocked, which corresponds to a `Synchronous Subprogram_Call_Type` property.[60] To specify non-blocking calls, a `SemiSynchronous Subprogram_Call_Type` property must be applied to the subprogram.[61]

Legality Rule

(L1) In a subprogram invocation, the parameter list must match the signature of the subprogram being invoked.[62]

Semantics

(S1) Subprogram invocation is lattice construction after actual for formal substitution.

Where p is a subprogram name and $f_1 : a_1, \ldots, f_k : a_k$ are formal-actual parameter pairs:

$\mathfrak{M}_{\underline{i}}[\![p(f_1 : a_1, \ldots, f_k : a_k)]\!] \equiv \mathfrak{M}_{\underline{i}}[\![p|^{f_1,\ldots,f_k}_{a_1,\ldots,a_k}]\!]$

(S2) The weakest-precondition semantics of subprogram invocation are defined by substituting actual parameters for formal parameters in the subprogram's pre- and post-conditions.

Where p is a subprogram name and $f_1 : a_1, \ldots, f_k : a_k$ are formal-actual parameter pairs, $pre|^{f_1,\ldots,f_k}_{a_1,\ldots,a_k}$ and $post|^{f_1,\ldots,f_k}_{a_1,\ldots,a_k}$ are the precondition and postcondition of p after actual parameters are substituted for formal parameters:

Subprogram Invocation **[SI]**: $\text{wp}(p(f_1 : a_1, \ldots, f_k : a_k), Q) \equiv Q|^{pre}_{post}$

These two semantics for subprogram invocation are equivalent because the precondition applies to the start state of the lattice, and the postcondition applies to the end state of the lattice.

(S3) Subprogram invocations are specified using the communication protocols `HSER`, `LSER` or `ASER` defined in (TBD).[63] A subprogram invocation is hence translated by the composition of the client (the caller) and server (the callee) with the behavior of the calling protocol.

[58] BA D.5(19)
[59] BLESSDiffers from BA: formal-actual subprogram parameters
[60] AS5506B §5.2 Subprograms and Subprogram Calls
[61] BA D.5(21)
[62] BA D.6(L5)
[63] Wherever D.8 Synchronization Protocols are to be defined

Chapter I 10

Behavior Expression

I 10.1 Value

(1) For threads, `value` has all the options as subprograms[1] plus port values, a test of the current mode, and reference to property constants for this component.

(2) Values are evaluated from incoming ports and parameters, local variables, referenced data subcomponents, as well as port count, port fresh, and port dequeue.[2]

Grammar

```
value ::=
  now | tops | timeout null | in mode ( { mode_identifier }+ )
  | value_constant | variable_name | function_call | port_value
```

now the present instant with type time

tops time-of-previous-suspension

timeout AADL runtime service for hybrid dispatch protocol threads: (now-tops) ≤ Timing_Properties::Period

in mode is true when AADL mode is among those listed; false otherwise

value_constant defined in §I 10.2, Value Constant

***variable*_name** defined in §I 10.3, Name

function_call defined in §I 10.8, Function Invocation

port_value defined in §I 10.9, Port Value

[1] see §I 11.3
[2] BA D.7(3)

I 10.2 Value Constant

(1) Value constants are Boolean, numeric or string literals, property constants or property values.[3]

Grammar

```
value_constant ::=
  true | false | numeric_literal | string_literal
  | property_constant | property_reference
```

(2) Numeric literals are defined in §I 3.4 Numeric Literals. String literals are defined in §I 3.5 String Literals.

Semantics

$\mathfrak{M}_{\underline{i}}[\![$ **true** $]\!] \equiv \top$ *(the meaning of* ***true*** *is customary)*
$\mathfrak{M}_{\underline{i}}[\![$ **false** $]\!] \equiv \bot$ *(the meaning of* ***false*** *is customary)*

I 10.2.1 Property Constant

(1) Property constants are values that are defined in AADL property sets.[4]

Grammar

```
property_constant ::=
  property_set_identifier :: property_constant_identifier
```

Semantics

(S1) The meaning of property constants are defined by the AADL standard, AS5506C §11.1.3 Property Constants.

I 10.2.2 Property Reference

(1) Property values may be defined in property sets, or attached to a component or feature.[5]

Grammar

```
property_reference ::=
  ( # [ property_set_identifier :: ]
  | component_element_reference #
  | unique_component_classifier_reference #
  | self )
  property_name
```

(2) The property may be relative to the component containing the behavior annex subclause: a subcomponent, a bound prototype, a feature, or the component itself.

Grammar

[3] BA D.7(4)
[4] AS5506C §11.1.3 Property Constants
[5] BLESSDiffers from BA: no local variable properties

```
component_element_reference ::=
  subcomponent_identifier | bound_prototype_identifier
  | feature_identifier | self
```

(3) Because AADL property values may be arrays or records, a property name may include array indices or record field identifiers.

(4) When the property is a range, the upper bound or lower bound of the property value can be referenced using `upper_bound` and `lower_bound` keywords.[6]

(5) When a property is a record, the field of a property value can be referenced using a dot separator between the property identifier and the field identifier.[7]

(6) When a property is an array, elements of the property value can be referenced using an integer value between brackets.[8]

Grammar

```
property_name ::= property_identifier { property_field }*

property_field ::=
  [ integer_value ] | . field_identifier | . upper_bound | . lower_bound
```

(7) Property values may be from any component specified by its package name, type identifier, and optionally implementation identifier.

```
unique_component_classifier_reference ::=
  { package_identifier :: }* component_type_identifier
  [ . component_implementation_identifier ]
```

I 10.3 Name

(1) A *name* is a sequence of identifiers, with optional array indices, separated by periods. Section §I 4, Types, defines the relationship between names and elements of values having constructed types: arrays, records, and variants. A slice, or portion of an array, may be named by an integer-valued range as its array index.

Grammar

```
name ::=
  root_identifier { [ index_expression_or_range ] }*
   { . field_identifier  { [ index_expression_or_range ] }*  }*
```

(2) An array index must be an integer-valued expression (§I 10.4), or a *slice* defined as an integer-valued range: lower bound `..` upper bound.

```
index_expression_or_range ::= integer_expression [ ..  integer_expression ]
```

Legality Rules

[6] BA D.7(9)
[7] BA D.7(10)
[8] BA D.7(11)

(L1) Array indices must be non-negative.

(L2) An array index or slice must be in the array's range. Names with array indexes outside of the array's range have undefined value and have undefined type.

(L3) A slice's lower bound must be at most its upper bound.

Semantics

(S1) Where x is a variable name,[9] y is a value, s is a state, and the pair $(x, y) \in s$:

$\mathfrak{M}_s[\![x]\!] \equiv y$ *(the meaning of a variable name in a state is its value)*

Where a is an array name, i is an integer value or values for a multidimensional array, y is a value, s is a state, and the pair $(a[i], y) \in s$:

$\mathfrak{M}_s[\![a[i]]\!] \equiv y$ *(the meaning of an array in a state is the value associated with its index)*

Where r is an record name, l is a label, y is a value, s is a state, and the pair $(r.l, y) \in s$:

$\mathfrak{M}_s[\![r.l]\!] \equiv y$ *(the meaning of a record in a state is its value of its selected label)*

Where v is a variant name with discriminator d, l is a label, y is a value, s is a state, and the pairs $(v.d, l), (v.l, y) \in s$:

$\mathfrak{M}_s[\![v.l]\!] \equiv y$ *(the meaning of a variant is the value of the element having the label of the discriminator)*

I 10.4 Expression

(1) An *expression* defines a value derived from other values by numeric or boolean operations. The type of subexpressions must be compatible with the expression's operator. The conditional boolean operators, `and then` and `or else`, demand evaluation of its left-side subexpression before its right-side subexpression, which is then evaluated only if it makes a difference to the result.[10] The other numeric and boolean operators have customary meanings.[11]

(2) Expressions have been defined to perform calculations with the complexity of programming languages such as Ada.[12] This expression language is derived from ISO/IEC 8652:1995(E), Ada95 Reference Manual §4.4.[13]

Grammar

[9] A name may be a simple identifier, or a compound name using indexes and/or labels. Here that name must correspond to a variable. In the following the name must correspond to an array, record or variant.
[10] BA R.7(12)
[11] BA D.7(1)
[12] BA D.7(2)
[13] BA D.7(5)

```
expression ::= subexpression
    [ { + numeric_subexpression }+
    | { * numeric_subexpression }+
    |   - numeric_subexpression
    |   / numeric_subexpression
    |   mod natural_subexpression
    |   rem integer_subexpression
    |   ** numeric_subexpression
    | { and boolean_subexpression }+
    | { or boolean_subexpression }+
    | { xor boolean_subexpression }+
    |   and then boolean_subexpression
    |   or else boolean_subexpression  ]
```

Legality Rules

(L1) Operators have no precedence; parentheses must disambiguate operator order.[14]

(L2) Operands of logical operators must be boolean.[15]

(L3) Operands of numeric operators must be numeric.[16]

Semantics

(S1) Where `e` and `f` are numeric-valued expressions, and A and B are boolean-valued expressions:

$\mathfrak{M}_i[\![\ \texttt{e+f}\ ]\!] \equiv \mathfrak{M}_i[\![e]\!] + \mathfrak{M}_i[\![f]\!]$ *(the meaning of + is addition)*
$\mathfrak{M}_i[\![\ \texttt{e*f}\ ]\!] \equiv \mathfrak{M}_i[\![e]\!] \times \mathfrak{M}_i[\![f]\!]$ *(the meaning of * is multiplication)*
$\mathfrak{M}_i[\![\ \texttt{e-f}\ ]\!] \equiv \mathfrak{M}_i[\![e]\!] - \mathfrak{M}_i[\![f]\!]$ *(the meaning of - is subtraction)*
$\mathfrak{M}_i[\![\ \texttt{e/f}\ ]\!] \equiv \mathfrak{M}_i[\![e]\!] \div \mathfrak{M}_i[\![f]\!]$ *(the meaning of / is division)*
$\mathfrak{M}_i[\![\ \texttt{e**f}\ ]\!] \equiv \mathfrak{M}_i[\![e]\!]^{\mathfrak{M}_i[\![f]\!]}$ *(the meaning of ** is exponentiation)*
$\mathfrak{M}_i[\![\ \texttt{e}\ \textbf{mod}\ \texttt{f}\ ]\!] \equiv \mathfrak{M}_i[\![e]\!] \bmod \mathfrak{M}_i[\![f]\!]$ *(the meaning of mod is modulus)*
$\mathfrak{M}_i[\![\ \texttt{e}\ \textbf{rem}\ \texttt{f}\ ]\!] \equiv \mathfrak{M}_i[\![e]\!] - (\mathfrak{M}_i[\![f]\!] \times (\mathfrak{M}_i[\![e]\!] \div \mathfrak{M}_i[\![f]\!]))$ *(the meaning of rem is remainder)*[17]
$\mathfrak{M}_i[\![A\ \textbf{and}\ B]\!] \equiv \mathfrak{M}_i[\![A]\!] \wedge \mathfrak{M}_i[\![B]\!]$ *(the meaning of* **and** *is conjunction)*
$\mathfrak{M}_i[\![A\ \textbf{or}\ B]\!] \equiv \mathfrak{M}_i[\![A]\!] \vee \mathfrak{M}_i[\![B]\!]$ *(the meaning of* **or** *is disjunction)*
$\mathfrak{M}_i[\![A\ \textbf{xor}\ B]\!] \equiv \mathfrak{M}_i[\![A]\!] \oplus \mathfrak{M}_i[\![B]\!]$ *(the meaning of* **xor** *is exclusive-disjunction)*
$\mathfrak{M}_i[\![A\ \textbf{and then}\ B]\!] \equiv \begin{array}{l} \mathfrak{M}_i[\![A]\!] \rightarrow \mathfrak{M}_i[\![B]\!] \\ \neg\mathfrak{M}_i[\![A]\!] \rightarrow \bot \end{array}$ *(second term not evaluated if first term is false)*[18]
$\mathfrak{M}_i[\![A\ \textbf{or else}\ B]\!] \equiv \begin{array}{l} \mathfrak{M}_i[\![A]\!] \rightarrow \top \\ \neg\mathfrak{M}_i[\![A]\!] \rightarrow \mathfrak{M}_i[\![B]\!] \end{array}$ *(second term not evaluated if first term is true)*[19]

[14] BLESSDiffers from BA: operator precedence
[15] BA D.7(L3)
[16] BA D.7(L5)
[17] **Reconciliation:** rem
[18] **Reconciliation:** and then
[19] **Reconciliation:** or else

I 10.5 Subexpression

(1) A *subexpression* allows negation, complement and grouping with parentheses, or is a conditional expression (§I 10.6).

Grammar

```
subexpression ::=
  [ - | not | abs ]
  ( value | ( expression_or_relation )
    | conditional_expression | case_expression )

expression_or_relation ::= subexpression [ relation_symbol subexpression ]
```

Semantics

(S1) Where `e` is a numeric-valued expression, `A` is a boolean-valued expression, and c, d are expressions:

$\mathfrak{M}_i[\![\texttt{-e}]\!] \equiv 0 - \mathfrak{M}_i[\![\texttt{e}]\!]$ *(the meaning of - is negation)*
$\mathfrak{M}_i[\![\texttt{abs e}]\!] \equiv \mathfrak{M}_i[\![\texttt{(if e>=0 then e else -e)}]\!]$ *(the meaning of* ***abs*** *is absolute value)*[20]
$\mathfrak{M}_i[\![\texttt{not A}]\!] \equiv \neg\mathfrak{M}_i[\![\texttt{A}]\!]$ *(the meaning of* ***not*** *is complement)*
$\mathfrak{M}_i[\![\texttt{(A)}]\!] \equiv \mathfrak{M}_i[\![\texttt{A}]\!]$ *(the meaning of parenthesis is its contents)*
$\mathfrak{M}_i[\![c\texttt{=}d]\!] \equiv \mathfrak{M}_i[\![c]\!] = \mathfrak{M}_i[\![d]\!]$ *(the meaning of = is equality)*
$\mathfrak{M}_i[\![c\texttt{<>}d]\!] \equiv \mathfrak{M}_i[\![c\texttt{!=}d]\!] \equiv \mathfrak{M}_i[\![c]\!] \neq \mathfrak{M}_i[\![d]\!]$ *(the meaning of != is inequality)*[21]
$\mathfrak{M}_i[\![c\texttt{<}d]\!] \equiv \mathfrak{M}_i[\![c]\!] < \mathfrak{M}_i[\![d]\!]$ *(the meaning of < is less than)*
$\mathfrak{M}_i[\![c\texttt{>}d]\!] \equiv \mathfrak{M}_i[\![c]\!] > \mathfrak{M}_i[\![d]\!]$ *(the meaning of > is greater than)*
$\mathfrak{M}_i[\![c\texttt{<=}d]\!] \equiv \mathfrak{M}_i[\![c]\!] \leq \mathfrak{M}_i[\![d]\!]$ *(the meaning of <= is at most)*
$\mathfrak{M}_i[\![c\texttt{>=}d]\!] \equiv \mathfrak{M}_i[\![c]\!] \geq \mathfrak{M}_i[\![d]\!]$ *(the meaning of >= is at least)*

I 10.6 Conditional Expression

(1) A *conditional expression* determines the value of an expression by evaluating a boolean expression or relation, then choosing between alternative expressions, returning the first if true or the second if false.

Grammar

```
conditional_expression ::=
  ( boolean_expression_or_relation ?? expression : expression )
| ( if boolean_expression_or_relation then expression else expression )
```

Semantics

(S1) Where `t` and `f` are expressions and `B` is a boolean-valued expression or relation:

[20] **Reconciliation:** absolute value
[21] **Reconciliation:** inequality

$\mathfrak{M}_i[\![$ `(if B then t else f)` $]\!] \equiv \mathfrak{M}_i[\![$ `(B??t:f)` $]\!] \equiv$
$\mathfrak{M}_i[\![\texttt{B}]\!] \rightarrow \mathfrak{M}_i[\![\texttt{t}]\!]$
$\neg\mathfrak{M}_i[\![\texttt{B}]\!] \rightarrow \mathfrak{M}_i[\![\texttt{f}]\!]$

(choose first expression if true; second expression if false)

Consistency Rule

(C1) All expressions of a conditional expression must have the same type.

Examples

```
(if SensorConnected? and not MotionArtifact? then SpO2? else 0)
(lrl<(dn_siri<down??dn_siri:down)??lrl:(dn_siri<down??dn_siri:down))
```

I 10.7 Case Expression

(1) A *case expression* extends the conditional expression to more than two choices. If one guard (boolean expression or relation) of a case choice is true, then the value of the case expression is the expression of that choice. If more than one case choice guard is true, the value is chosen non-deterministically from their expressions. If no case choice guard is true, the value is null.

Grammar

`case_expression ::= ( case_choice { , case_choice }+ )`

`case_choice ::= ( boolean_expression_or_relation ) -> expression`

Semantics

(S1) Where `B1 ... Bj` are boolean-valued expressions or relations, and `e1 ... ej` are expressions of the same type:

$\mathfrak{M}_i[\![$ `( (B1)->e1, ..., (Bj)->ej )` $]\!] \equiv$
$\mathfrak{M}_i[\![\texttt{B1}]\!] \rightarrow \mathfrak{M}_i[\![\texttt{e1}]\!]$
$\ldots$
$\mathfrak{M}_i[\![\texttt{Bj}]\!] \rightarrow \mathfrak{M}_i[\![\texttt{ej}]\!]$
$\neg\mathfrak{M}_i[\![\texttt{B1}]\!] \wedge \ldots \wedge \neg\mathfrak{M}_i[\![\texttt{Bj}]\!] \rightarrow \bot$

(choose expression with true guard, or null if all guards are false)

Consistency Rule

(C1) All expressions of case choices in a case expression must have the same type.

I 10.8 Function Invocation

(1) A *function call* is the invocation of a subprogram having a special form.[22] AADL subprogram components that may be invoked as functions must have one **out** parameter, preceded by any number of **in** parameters.

```
subprogram f
features
  p1 : in parameter t1;
    .
    .
    .
  pk : in parameter tk;
  result : out parameter resultType;
annex Action {** . . . **};
end mul;
```

(2) The identifiers of the formal parameters in a function call, correspond to the **in** parameters of the AADL subprogram component. Subprograms invoked as functions must use formal-actual pairs as arguments. These substitutions of actual for formal parameters are applied to the subprogram's pre- and post-conditions when verification conditions for function invocation are generated.

(3) Subprograms in other packages may be invoked as functions by prefacing their identifier with package identifiers separated by double colons.[23]

Grammar

```
function_call ::=
 { package_identifier :: }* function_identifier ( [ function_parameters ] )

function_parameters ::= formal_expression_pair { , formal_expression_pair }*

formal_expression_pair ::= formal_identifier => actual_expression
```

Semantics

(S1) Where C is the name of a function having formal parameters $f_1, \ldots, f_k$, and $e_1, \ldots, e_k$ are expressions:

$\mathfrak{M}_i[\![C(f_1 \Rightarrow e_1, \ldots, f_k \Rightarrow e_k)]\!] \equiv \mathfrak{M}[\![C]\!](f_1 \Rightarrow \mathfrak{M}_i[\![e_1]\!], \ldots, f_k \Rightarrow \mathfrak{M}_i[\![e_k]\!])$
(the meaning of a function call, is the meaning of its name, applied to the meanings of its parameters)

Legality Rule

(L1) Subprograms invoked as functions must have all but the last parameter an `in` parameters, and the last parameter must be an `out` parameter.

(L2) Subprograms invoked as functions must be side-effect free; their only result is the value returned.

Examples

This example shows the use of variable labels as temporary variables to pass data between successive actions.

[22] Ordinarily, function calls cannot be used within expressions, because AADL doesnt have a pure function, distinct from its subprogram classifier. Because an AADL subprogram is not limited to determining its return value solely from passed values, evaluation of AADL subprograms may have side effects. Functions are AADL subprograms that are purely functional. Then function calls can be used in expressions within BA2015.

[23] **Reconciliation:** removed $ from function invocation

```
data number
end number;

subprogram mul
features
  --this is in the special form for a subprogram to be a function,
  --single out parameter preceded by any number of in parameters
  --may invoked within expressions as mul(e1,e2),
  --or actions as mul(v1,v2,v3)
  x : in parameter number;
  y : in parameter number;
  z : out parameter number;
annex Action {**
  post z=x*y  --postcondition relating result to values of inputs
  { z := x*y <<z=x*y>> }
**};
end mul;

subprogram cube
features  --cube is also a function
  x : in parameter number;
  y : out parameter number;
      --features other than parameters follow the out parameter
  mul : requires subprogram access mul;
annex Action {**
  post  y=x*x*x
  variables --existential quantification introduces local variables
    tmp : number;
  { mul(x,x,tmp) --invoke mul as subprogram
    ; <<tmp=x*x>>   --sequential composition
    y := mul(tmp,x) <<y=x*x*x>> } --invoke mul as function
**};
end cube;
```

I 10.9 Port Value

(1) The core language defines that data from data ports is made available to the application source code (and Behavior_Specification) through a port variable with the name of the port. If no new value is available since the previous freeze, the previous value remains available and the variable is marked as not fresh. Freshness can be tested in the application source code via service calls[24] and in the Behavior_Specifcation via functions.[25]

Grammar

`port_value ::= in_port_name ( ? | 'count | 'fresh | 'updated )`[26]

(2) The meaning of port values is defined in **??**, In Event Data Ports.

[24] AS5506C §8.3.5 Runtime Support for Ports

[25] BLESSDiffers from BA: port names must have suffix: ? or '

[26] BLESSDiffers from BA: port identifiers must have ? or '

Chapter I 11

Subprogram

(1) Subprogram behavior is defined using the *Action annex sublanguage*. Only subprogram components have `Action` annexes.

Grammar

```
subprogram_annex_subclause ::=
  annex Action {** subprogram_behavior **} ;
```

I 11.1 Subprogram Behavior

(1) An `Action` annex consists of a behavior action block that may be preceded by Assertions visible in the scope of the subprogram, a precondition, and a postcondition. A precondition that must be true of the subprogram parameters is preceded by pre. A postcondition that will be true after execution of the subprogram is preceded by post.

Grammar

```
subprogram_behavior ::=
   [ assert { assertion }+ ]
   [ pre assertion ]
   [ post assertion ]
   [ invariant assertion ]
   behavior_action_block
```
[1]

(2) In most programming languages, a subprogram is comprised from imperative commands that assign values of expressions to variables or control the flow of execution with branches and loops. For BAv2 subprograms, the temporal logic formula comprising the main body of the subprogram is satisfied by lattices of states (§I 2.9). Execution of a BAv2

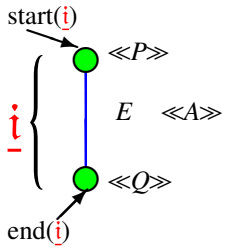


Figure I 11.1: Subprogram Satisfying Lattice

[1] BLESSDiffers from BA: subprograms have no transitions

subprogram constructs a satisfying state lattice. Typically many different lattices satisfy the same temporal logic formula, all of which will have identical bindings of values to variables in their start and end states.

(3) Figure I 11.1 depicts a lattice that satisfies a subprogram having precondition P, post condition Q, by constructing an interval i, satisfying E, its existential lattice quantification. Although depicted as a single arc from the interval's start node to its end node, satisfying lattices will have many intermediate nodes and arcs.

(4) Subprograms may also assert invariants that must be true in every state. Figure I 11.1 depicts an invariant, A that must hold in every state in i. Both E and A are logic formulas, of different logics. The difference is that e is an interval temporal logic satisfied by the combined structure of states (nodes) and transitions (arcs), while A is a first-order predicate applied to each of the states individually.

(5) Within the behavior annex subclause the value of a data component is returned by naming the data subcomponent, the requires data access, or the provides data access feature. Multiple references to this name represent multiple reads that may return different values, if the data component is shared and a write has been performed concurrently between the two reads. Concurrent writes may be prevented by a value of the `Concurrency_Control_Protocol` property that ensures mutual exclusion over an execution sequence with multiple reads.[2]

(6) A transition action can assign a return value to an outgoing parameter of the containing subprogram type by naming the parameter on the left-hand side of the assignment, i.e., `par :=v`. A transition action can assign a value to an incoming parameter of a subprogram call by specifying the value v in place of the formal parameter.[3]

Legality Rule

(L1) Assertions of subprograms must not have temporal operators `@`, `^`, or `'`.[4]

Semantics

(S1) Where A, P, and Q are predicates, and E is existential lattice quantification:

$\mathfrak{M}_{i}[\![\textbf{assert}\ \ll A \gg\ \textbf{pre}\ \ll P \gg\ \textbf{post}\ \ll Q \gg\ E]\!]$
$\equiv \mathfrak{M}_{start(i)}[\![P]\!] \wedge \mathfrak{M}_{end(i)}[\![Q]\!] \wedge \mathfrak{M}_{i}[\![E]\!] \wedge \mathfrak{M}_{i}[\![A]\!]$
(the meaning of subprogram behavior is: P is true in the stating state of i, Q is true in the ending state of i, A is true throughout i, and i satisfies E)

(S2) Equivalently, **assert** `<<A>>` **pre** `<<P>>` **post** `<<Q>>` `E` has the behavior of an automata transition $T(s, true, d, true)[E]$ from `initial` state s in which assertion $\ll P \gg$ holds to `final` state d in which assertion $\ll Q \gg$ holds while performing action E.[5]

Example

```
subprogram minimum3
features
  a: in parameter BAv2_Types::Real;
  b: in parameter BAv2_Types::Real;
  c: in parameter BAv2_Types::Real;
  result: out parameter BAv2_Types::Real;
annex Action
```

[2] BA D.5(10)
[3] BA D.5(17)
[4] Without temporal operators, assertions are first-order predicates.
[5] JP

```
{**
assert <<MIN3:a b c:=(a<MIN(a:b,b:c) ?? a : MIN(a:b,b:c))>>
pre <<a>0 and b>0 and c>0>>
post <<result=MIN3(a:a,b:b,c:c)>>
  {
    <<true>>
  result := (c<(a < b ?? a : b) ?? c : (a < b ?? a : b))
  <<result=MIN3(a:a,b:b,c:c)>>
  }
**};
end minimum3;
```

I 11.2 Subprogram Basic Actions

(1) Within a Action annex, the only basic actions are skip, assignment, simultaneous assignment, and exception throwing.[6] For threads `basic_action` includes other actions not performed by subprograms (§I 8.4).

Grammar

```
basic_action ::=
  skip | assignment  | simultaneous_assignment | when_throw
  | subprogram_invocation
```

I 11.3 Value for Subprograms

(1) An *value* is indivisible and may be a variable name, a function call, a reserved word, a property constant or a literal.[7] Literals have the same representation as the core language.[8]

Grammar

```
value ::=
  variable_name | value_constant | function_call
  | incoming_subprogram_parameter_identifier | null
```

[6] BLESSDiffers from BA: subprogram basic actions
[7] BLESSDiffers from BA: subprogram values
[8] AS5506B §15.4 Numeric Literals

Chapter I 12

BLESS Package and Properties

Two property sets and a package of data types are predeclared.

(1) Property set `BLESS` defines:

Assertion what is true about an event or data sent by, or arriving at a port

Typed data type as defined in Chapter I 4

Invariant what is always true about a component

Precondition what must be true before a subprogram is called

Postcondition what will be true when a subprogram returns

```
property set BLESS is
  Assertion : aadlstring applies to ( all );
  Typed : aadlstring applies to ( all );
  Invariant : aadlstring applies to ( all );
  Precondition : aadlstring applies to ( subprogram );
  Postcondition : aadlstring applies to ( subprogram );
end BLESS;
```

(2) Property set `BLESS_Properties` defines:

Supported Operators what operators apply to elements of a type

Supported Relations what relations apply to elements of a type

Radix radix position for fixed-point types

```
property set \package_Properties is
  with AADL_Project;
  Supported_Operators : list of aadlstring applies to ( data );
    --used to define arithmetic operator symbols supported by a type
  Supported_Relations : list of aadlstring applies to ( data );
    --used to define relation symbols supported by a type
  Radix : AADL_Project::Size_Units applies to ( data );
    --location of the radix point for fixed-point representation
    --counting from most significant bit
```

```
end BLESS_Properties;
```

(3) These data components in the package, `BLESS_Types` represent ideal values: integers without upper or lower bounds, real numbers of infinite precision, strings with unbound length. Actual types, with ranges and bounds, must substitute for ideal types, either explicitly or automatically.

(4) Chapter I 4 uses the standard Data Modeling annex (`Data_Model` and `Base_Types`) to define correspondence with types built in to BLESS.

```
package BLESS public
with Base_Types, BLESS_Properties, Data_Model, BLESS;
  data Integer extends Base_Types::Integer
  properties  --operators and relation symbols defined for Integer
    BLESS::Typed => "integer";
    BLESS_Properties::Supported_Operators =>
        ("+", "*", "-", "/", "mod", "rem", "**");
    BLESS_Properties::Supported_Relations => ("=", "!=", "<", "<=", ">=", ">");
  --how should conversion routines be declared?
end Integer;

data Natural extends Base_Types::Natural
  properties  --operators and relation symbols defined for Natural
    BLESS::Typed => "natural";
    BLESS_Properties::Supported_Operators =>
        ("+", "*", "-", "/", "mod", "rem", "**");
    BLESS_Properties::Supported_Relations => ("=", "!=", "<", "<=", ">=", ">");
end Natural;

data Real extends Base_Types::Float
  properties  --operators and relation symbols defined for Float
    BLESS::Typed => "real";
    BLESS_Properties::Supported_Operators => ("+", "*", "-", "/", "**");
    BLESS_Properties::Supported_Relations => ("=", "!=", "<", "<=", ">=", ">");
end Real;

data String extends Base_Types::String
  properties  --operators and relation symbols defined for String
    BLESS::Typed => "string";
    BLESS_Properties::Supported_Operators => ("+", "-"); --just concatenation
    BLESS_Properties::Supported_Relations => ("=", "!=", "<", "<=", ">=", ">");
end String;

data Fixed_Point
  properties  --operators and relation symbols defined for fixed-point arithmetic
    BLESS::Typed => "rational";
    BLESS_Properties::Supported_Operators => ("+", "*", "-", "/", "**");
    BLESS_Properties::Supported_Relations => ("=", "!=", "<", "<=", ">=", ">");
    Data_Model::Data_Representation => Integer;
end Fixed_Point;

data Time extends Real
end Time;

data flag extends Base_Types::Boolean --boolean flag
  properties
    BLESS::Typed=>"boolean";
end flag;

end BLESS_Types;
```

Part II

Verification of BLESS Behaviors

This Part explains verification of BLESS programs with formal proofs.

During the AADL standard committee's[1] deliberations on a state-machine language which has come to be known as BA, similarities of the action language to the Declarative Axiomatic Notation for Concurrent Execution (DANCE) were noticed.

DANCE allowed annotations to be interspersed with text for actions like comments. These annotations were first-order predicates (boolean-valued functions) applied to values of program variables at the location of the annotation during execution. When applied, the predicates were (supposed to be) true, thus state some true fact about the state of the program at that location. Thus the predicates were called *assertions* indicating that they were always supposed to be true. When a DANCE program was sufficiently annotated to be a *proof outline* a proof engine could transform the proof outline into a sequence of theorems, each of which was an axiom, given, or derived from earlier theorems by a sound inference rule. The last theorem would state that when the program began with its precondition true, it would terminate with its postcondition being true.

If BA could be augmented with such annotations, the same proof engine could transform proof outlines into proofs. Thus BLESS began as BA augmented with non-executed assertions.

Annotating programs with assertions to form proof outlines was first proposed long ago, but has been little used in practice. Treating programs, their specifications, and their executions as mathematical objects, then formally showing that every execution upholds its specification has been too hard. Every engineering discipline–other than software engineering–models its subject mathematically to make matter and energy obey human will. Yet programs can be perfect in ways that physical objects can never be. Perhaps software engineers reluctance to treat programs as mathematical objects can be overcome with tool support–especially when the consequences of failure can be human injury or death.

Although other researchers legitimately characterize verification performed by their formal verification as 'proof', here we use 'proof' to mean sequence of theorems, each of which is given or axiomatic, or derived from earlier theorems by sound inference rules, in which the last theorem states that every possible execution meets its specification.

II 0.1 What is *Proof*?

In *How to Write a* 21^{st} *Century Proof* [28] Leslie Lamport advocates formal, inductive proofs–a sequence of theorems, each of which is either an assumption, axiom, or derived from prior theorems by stated, sound, inference rules. Lamport uses TLA+ in his example, and aids understanding that the sequence of theorems is really a tree with indentation and theorem numbering.

> In principle, the proof of a theorem should show that the theorem can be formally deduced from axioms by the application of proof rules. In practice, we never carry a proof down to that level of detail. However, a mathematician should always be able to keep answering the question *why?* about a proof, all the way down to the level of axioms. A completely formal proof is the Platonic ideal.

BLESS proofs that all executions of programs meet their specifications strives for this Platonic ideal.

[1] SAE International AS-2C

Most formal methods claim to provide "proof" for this or that. This "proof" being convincing evidence. Model checker's failure to find a counterexample is proof that the searched-for property holds. Static analyzers can prove that all variables are initialized, and that un-checked buffer overflow can't occur. Theorem provers like Coq exhibit sequences of tactics starting with '`Proof.`' and ending with '`QED.`' as "proof" (although examining a sequence of tactics provides no clue as to what is "proved").

Fine. Call whatever evidence produced by a formal method "proof". However, here we are trying to explain how a program–annotated with assertions to be a proof outline–can be transformed using human-selected tactics into a completely-formal, inductive *proof* that every program execution will meet its specification.

II 0.2 Background

Our methodology traces back to the use of pre/postconditions in Floyd/Hoare logic [15] [22] which characterizes the behavior of a program statement S using triples of the form

$$\{P\}\ S\ \{Q\}$$

where both P (the precondition) and Q (the postcondition) are predicates (boolean-valued formulas) over program variable values. Because the state-machine language to be annotated with assertions already used braces for delimiting program blocks, double angle brackets `<<>>` were used around assertions instead:

```
<<P>> S <<Q>>
```

David Gries, *The Science of Programming* [18] provides a gentle introduction to proof outlines, and reasoning about program triples.

II 0.3 Method

The method for transforming proof outlines into proofs has four phases:

1. Extract verification conditions (proof obligations)
2. Reduce proof obligations with complex actions into simpler ones until atomic
3. Reduce atomic proof obligations into implications
4. Reduce implications into axioms and assumptions (givens)

A person guides the proof engine by selecting tactics. When all verification conditions have been reduced to axioms, the proof engines creates the sequence of theorems which formally proves conformance of behavior to specification.

One advantage of this method is that all the proof obligations (later theorems) are expressed in the same language as the program itself. Another advantage is the easy tracing of unsolved (or unsolvable) proof obligations back to the program source from which they originated. This makes correcting the program much easier than problems with proving having translated the program into native language of Why3, Coq, Isabelle/HOL, PVS, or ACL2. Finally, having a list of unsolved proof obligations makes progress towards proof manifest.

Proof of a component using proofs of its subcomponents is considered in Chapter II 4 Verifying Composition.

Before a program can be formally verified, what it's supposed to do must be formally specified.

Chapter II 1

BLESS Specification

Safety-critical software often needs to meet timing-related requirements. Before formal verification, such timing needs formal specification. BLESS assertions can be used as a behavior interface specification language (BISL) that allows safety-critical timing specification–declaratively–by extending first-order predicates with simple temporal operators. Component behavior is specified by associating with each port an assertion, plus an invariant assertion of the component as a whole. Assertions of *out* ports declare what is guaranteed about the system when events and/or data are issued by the component. Complementarily, assertions of *in* ports declare what is assumed about the system when events and/or data are received by the component.

II 1.1 Introduction

Behavior interface specification languages (BISLs) provide formal annotations to express the intended behavior of programs. Such annotations may be used within programs to help show, formally, that programs' behaviors meet their formal specifications. This paper concerns an important subset of programs, namely, those that control safety-critical, cyber-physical systems. Errors in safety-critical software may cause injury, or even death. Therefore, embedded programs warrant special scrutiny, and a specialized BISL for their specification and verification.

The behavior of cyber-physical systems commonly involves timing. Therefore, embedded programs controlling cyber-physical systems need temporal specifications. The temporal logic we use as BISL is extension of first-order predicates with simple temporal operators that determine when predicates hold.

II 1.2 Behavior Interface Specification Languages

In [19] BISLs were surveyed with a focus toward automatic program verification. They define a software *behavioral specification* as a precise description of the intended behavior of some computing system or its components.

One domain identified for BISLs is safety-critical programs systems including avionics, medical devices, automotive control sys- tems, nuclear power plants, as well as systems associated with critical infrastructure. Verification and validation is a key component of the development and certification of such systems. In this context, formal specifications provide a canonical declaration of a systems intended functionality against which an implementation can be verified. Formal specifications can also guide test case construction, but that is not the concern of this paper. The benefits of specifications are amplified when they are written in a formal specification language–a mathematically precise notation for recording intended properties of software.

Formalizing the syntax of the specification language enables specifications to be processed by software development tools and checked for well-formedness. Formalizing the semantics helps make specifications unambiguous, less dependent on cultural norms, and thus less likely to be misunderstood. Most importantly, formalizing the semantics enables tools to provide automated reasoning about specifications and their relationship to associated code.

Summary of BISLs from [19]:[1]

> Formal specifications can be leveraged by tools throughout the entire software life-cycle. At design time, using automated deduction and SAT-based techniques, specifications can be checked automatically for consistency and queried to deter- mine if desired system behaviors are implied by the specifications [23]. As coding begins, static analysis tools can check implementations against specifications, e.g., a method body can be checked to determine if it correctly implements its contract, that it satisfies the preconditions of any methods that it calls, and that all assertions in the method body hold [3, 14]. Specifications can also serve as a starting place for transformational development in which specifications are systematically refined into code [1, 20, 26, 35–37, 39]. Formal specifications need not specify full correctness to be useful; light-weight annotations including those that restrict ranges of numeric variables and that specify the non-nullness of reference variables can be easily incorporated during development to guide tools that seek to find bugs used deductive techniques [3, 14] or abstract interpretation [5]. During testing, executable representations of assertions can be checked, test cases can be generated automatically from formal specifications [6–8], or implementations can be exercised directly from specifications as in model-based testing techniques [24]. Formal specifications can be compiled to code-based oracles that determine when a particular test passes or fails [21]. Even after systems have been deployed, code generated from specifications can provide run-time monitoring of a systems execution and aid in the implementation of fault-recovery mechanism [4, 9].

Most relevant to this work:

> Finally, in critical systems, tools based on combinations of automatic and interactive theorem-proving can be used for verification–proving that an implementation is free of bugs and that it satisfies its formal specification in every possible execution [2, 13].

BLESS assertions were developed to declaratively specify cardiac pacemaker function [42]. Both properties of particular instants of time (sensed, intrinsic heartbeat, or paced, caused heartbeat) and intervals of time. Other temporal logics were tried, principally interval temporal logic (ITL) [38], duration calculus (DC) [43], linear time logic (LTL) [40], and temporal logic of actions [27], but could not be used to specify pacemaker function by the author (which may be the result of limitations of the author rather than limitations of the logics).

[1] Note to reviewers: No better concise summary of BISLs could be found. Long quotation deemed better than plagiarism, or re-wording.

Classical, first-order logic [17] extended with simple temporal operators that defined when predicates were evaluated sufficed to specify pacemaker function.

II 1.2.1 BISLs Expressing Timing

Quotations about BISLs from [19]:

> Verification and validation is a key component of the development and certification of such systems. . . . [S]pecifications provide a canonical declaration of a systems intended functionality against which an implementation can be verified.
>
> *[V]erification*–proving that an implementation is free of bugs and that it satisfies its formal specification in every possible execution.
>
> *Functional behavior properties* describe the (data) values associated with system operations or state changes.
>
> *Temporal properties* describe properties of a systems sequences of states, along with the relationship between system events and state transitions. Timing constraints are especially important for the modeling and analysis of real- time systems.
>
> *Resource properties* describe constraints on how much of some resource, such as time or space, may be used by an operation or may be used between a pair of events.
>
> Verification technology is also closely tied to semantics. A *verification logic* is a formal reasoning system that allows proofs that establish that code satisfies a specification.
>
> Pre/postconditions are one example of structured use of assertions. A precondition is an assertion that must hold whenever the procedure is called (after parameter passing). A postcondition is an assertion that must hold immediately after the procedure completes its execution. The use of pre/-postconditions in program specification can be traced back to Floyd/Hoare which characterizes the behavior of program statement C using triples of the form $\{P\}\ C\ \{Q\}$, where both P (the precondition) and Q (the postcondition) are boolean formula over variable values.
>
> [R]easoning about the correctness of an event-driven system seems to be beyond the state-of-the-art techniques.

These brief quotations are the principles addressed by the BISL defined here. Although BISLs can be used for much more than specification, that timing behavior of safety-critical software can be *specified*. In [29] we argue that such specifications can be *validated*–that natural language requirements can clearly be understood to be expressed by formal specification. In [32] we argue that thread behaviors, annotated with assertions in this BISL, can form a proof outline that can be automatically transformed (with some human guidance) into a complete proof that every execution will uphold its specification. In [31] we argue that composition of formally specified components can be similarly proved correct.

Although many legitimately characterize verification performed by their formal verification as ‘proof’, here we use ‘proof’ to mean sequence of theorems, each of which is given or axiomatic, or derived from earlier theorems by sound inference rules, in which the last theorem states that every possible execution meets its specification.

In total:

- Behavior of safety-critical systems having crucial timing can be declaratively, formally specified.
- Such formal specifications can be validated by domain experts.

- Proofs can be constructed that individual component meet their specifications.
- Proofs can be constructed that compositions of verified components meet their specifications.

II 1.3 BLESS Assertions

The Architecture Analysis and Design Language (AADL) [41] is a SAE International standard language intended for design of safety-critical cyber-physical systems. AADL embodies much system engineering experience with avionics, but is applicable to many other cyber-physical domains such as land vehicles, medical devices, and train control systems. AADL supports development of system *architecture*–the logical and physical structure of the system–recursively decomposing complex functionality into smaller, simpler components, both graphically and textually. Crucially, the interfaces and connections between components are carefully designed so that when the components are integrated, they work together correctly.

Unlike other architecture languages, AADL has well-defined semantics, albeit using natural language. AADL is extensible by user-defined properties and annex languages. Annex subclauses are attached to individual components while annex libraries are contained in packages not specific to any component(s). The Behavioral Language for Embedded Systems with Software (BLESS) in an AADL annex sublanguage. BLESS annex subclauses define state-transition machines representing the implementation of behavior as programs. BLESS assertions are used to declaratively specify behavior–attached as AADL properties to components and their ports. BLESS assertions are also used within BLESS annex subclauses as annotations forming a proof outline. Such use is considered in [32]. Here we use BLESS assertions to formally specify behavior by attaching them to AADL architectural elements as AADL properties. For convenience, BLESS assertions that are used in many places may be declared in an AADL annex library.

Grammar and formal semantics for BLESS assertions given in Chapter I 5 BLESS Assertions.

Specifications are most useful when they are declarative, stating *what* should happen, not *how*. The task of verification shows the 'how' properly implements the 'what'. Verifying an operational implementation meets declarative specification provides an orthogonal test akin to auditing double-entry bookkeeping or type checking. Refinement of one operational implementation into another merely verifies that errors have been faithfully preserved.

BLESS assertions are enclosed in double-angle brackets **<<>>** . For convenience, labeled assertions may be declared, and then used in other assertions, equivalent to textual replacement, first replacing formal parameters with actual parameters, if any.

Given that `max_cci` is the name of an **`in data port`** having time type, `vs` and `vp` are names of **`out event port`** s, an assertion labeled LRL is defined as

```
<<LRL:x: exists t:time in (x-max_cci)..x
  that (vs or vp)@t>>
```

Then the text `LRL(now)` could be replaced with

```
exists t:time in (now-max_cci)..now
  that (vs or vp)@t
```

When occurrence of an intrinsic, non-refractory ventricular sensed contraction is indicated by an event on port `vs` , an induced ventricular paced contraction is indicated by an event on port `vp` , and the longest period without either intrinsic or induced ventricular contractions is `max_cci}` for maximum cardiac cycle interval), then `LRL` **`(now)`** expresses the fundamental effectiveness property of cardiac pacing–the most recent heartbeat ocurred no further in the past than the maximum cardiac cycle interval.

II 1.4 Specification of AADL Components

Specification of AADL threads consists of assertion properties of its ports, and a thread invariant, much like a loop invariant. AADL subprograms are passively invoked having pre/postcondition specifications.

Assertion properties of **`in`** ports are assumed. Assertion properties of **`out`** ports are guaranteed. Thread invariants must always hold at dispatch and suspension, much like loop invariants before and after each iteration of the loop.

For **`event`** ports and **`data`** ports with Boolean type, assertion-predicates are used. For other types of **`data`** ports the value is specified with an assertion-function of the form **`<<`**`:=expression`**`>>`** or an assertion predicate of the form **`<<`**`portname = expression`**`>>`** , but are not used in the following example.

II 1.4.1 Continuous-Time Example: DDD pacing

For an example of specification using **`@`** , we use a thread to control a cardiac pacemaker with DDD-mode pacing as defined in [42]. Designations of pacing mode are common throughout the pacemaker industry. Each 'D' stands for 'dual': sensing of cardiac activity in both the right-atrium and right-ventricle, pacing in both of those heart chambers, and both inhibition and tracking behavior. Pacing is delivery of a short (half-millisecond) pulse of single-digit voltage, as chosen by the cardiologist.

Inhibition means when the heart is beating fast enough, the pacemaker inhibits pacing, doing nothing but monitoring. A cardiologist prescribes a Lower-Rate Limit (LRL) for a patient which defines "fast enough" for the pacemaker.

Tracking means that pacing in the ventricle follows (tracks) sensed or paced contractions in the atrium. If no ventricular contraction is sensed within a prescribed atrioventricular (AV) delay, a ventricular pace will be issued, keeping ventricular contractions synchronized with atrial contractions. However, because pacing sick hearts too fast is harmful, tracking is limited to a prescribed Upper-Rate Limit (URL).

LRL defines the fundamental *effectiveness* property, pacing when the intrinsic heart rate is too slow. URL defines the fundamental *safety* property, not pacing too fast.

Figure II 1.1, shows a simple AADL architecture for a pacemaker. Leads are implanted into a patient's right atrium and ventricle connecting through a header to an analog 'front-end' which filters and amplifies millivolt-level signals conducted by leads from the inside of the heart. The front-end will also apply paces on those same leads when commanded.

In AADL, threads must be contained in a *process* component which represents a protected address space. In this example, the 'ddd' process contains a single thread with the same ports shown in Figure II 1.2. The `as` and

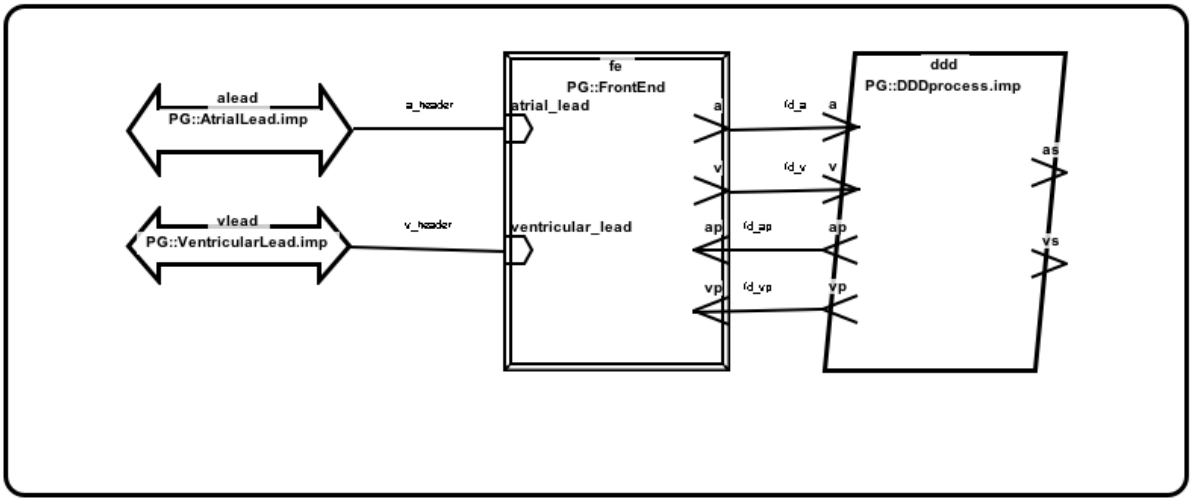


Figure II 1.1: AADL Architecture for DDD

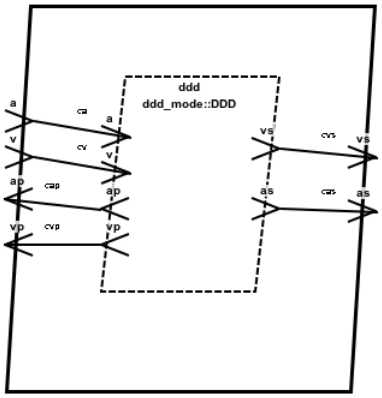


Figure II 1.2: Process Containing Thread

vs ports would connect to components reporting telemetry and recording history in more complete pacemaker models, but here are otherwise unused.

Despite careful filtering, signals detected by the front-end do not always represent atrial or ventricular contractions. To help distinguish which signals represent contractions, the thread ignores signals during 'refractory' periods. A signal deemed non-refractory from the atrium (resp. ventricle) would generate an event on the as (resp. vs) port.

The durations of the refractory periods are also prescribed by a cardiologist, making specification of when DDD-mode should pace each chamber challenging. Because once prescribed, parameter values are constant, they are modeled as AADL property constants rather than data ports.

II 1.4.2 Specification of Port Behavior

This is the AADL for the DDD-mode thread shown in figure II 1.1:

```
thread DDD
  features
  a: in event port; --atrial signal
  v: in event port; --ventricular signal
  ap: out event port  --pace atrium
    {BLESS::Assertion=>"<<AP(now)>>";};
  vp: out event port  --pace ventricle
    {BLESS::Assertion=>
      "<<VP(now) and URL(now)>>";};
  as: out event port --atrial sense
    {BLESS::Assertion=>"<<AS(now)>>";};
  vs: out event port  --ventricular sense
    {BLESS::Assertion=>"<<VS(now)>>";};
 . . .
  --maximum cardiac cycle interval
  max_cci: in data port ms
    {BLESS::Assertion=>"<<:=MaxCCI()>>";};
 . . .
  properties
    Dispatch_Protocol => Aperiodic;
    BLESS::Invariant=>"<<LRL(now)>>";
end DDD;
```

Ports a and v receive events front the front-end when it senses signals from the atrium and ventricle respectively. Ports ap and vp send events to the front-end to cause paces in the atrium and ventricle. Ports as and vs send events when atrial or ventricular signals are deemed 'non-refractory' representing actual heart contractions. Each of the output ports has a **BLESS::Assertion** property that must hold at the instant and event is issued by the port.

It is often convenient to define labelled assertions, with optional parameters, that may be re-used by other assertions. Here, the assertion properties of ports reference labelled assertions with **now** as the parameter indicating that the referenced assertion holds at the instant the event is sent by the port.

II 1.4.3 Specification of Invariant Behavior

The Lower-Rate Limit effectiveness property is expressed by the **BLESS::Invariant** property, **<<LRL(now)>>** .

```
<<LRL:x: exists t:time
   in (x-#PP::Lower_Rate_Limit_Interval)..x
   that (vs or vp)@t>>
```

Even though a cardiologist may express LRL as bpm (beats/minute), pacemakers always use the equivalent interval in milliseconds. Therefore a typical LRL=60 bpm is equivalent to a `PP::Lower_Rate_Limit_Interval` of 1000 ms. So the thread invariant, substituting the actual parameter **`now`** for the formal parameter `x` states that there will be either a non-refractory ventricular sense or a non-refractory ventricular pace within the previous Lower-Rate Limit interval. Given LRL=60 bpm, that means there will always be a ventricular contraction, either sensed or paced, in the previous second.

In [] we showed validation of natural language requirements in the PACEMAKER System Specification [] were faithfully expressed as formal BLESS assertions. Below we present just the assertions for brevity.

For an atrial pace caused by an event issued by the `ap` port, **`<<`**`AP`**`(now)>>`** must hold:

```
<<AP:x:
  (vp or vs)@(x-(#PP::Lower_Rate_Limit_Interval
      -#PP::Fixed_AV_Delay))
  and not (exists tv:time
     in (x-(#PP::Lower_Rate_Limit_Interval
         -#PP::Fixed_AV_Delay)),,x
     that (vs or vp or as or ap)@tv)>>
```

For a ventricular pace caused by an event issued by the `vp` , both **`<<`**`VP`**`(now)>>`** and **`<<`**`URL`**`(now)>>`** must hold. Cause ventricular pace either if intrinsic rate is to slow, or tracking an atrial sense.

```
<<VP:x: PACE_ON_LRL(x) or PACE_ON_SAV_DELAY(x)>>
<<PACE_ON_LRL:x: (vp or vs)@(x
      -#PP::Lower_Rate_Limit_Interval)
    and not (exists t:time
      in (x-#PP::Lower_Rate_Limit_Interval),,x
      that (vs or vp)@t)>>
<<PACE_ON_SAV_DELAY:x:
  as@(x-#PP::Sensed_AV_Delay)
  and not (exists tu:time
    in x-#PP::Upper_Rate_Limit_Interval,,x
    that (vs or vp)@tu )>>
```

However, must not ventricular pace too soon after either ventricular pace or sense.

```
<<URL:x: not (exists tu:time
  in (x-#PP::Upper_Rate_Limit_Interval),,x
  that (vs or vp)@tu)>>
```

Discerning which signals or non-refractory senses is crucial for determining when to pace. Atrial senses must not be during the atrial refractory period (following atrial senses and paces), not the post-ventricular atrial refractory period (following ventricular senses or paces).

```
<<AS:x: a@x and not PVARP(x) and not ARP(x)>>
<<PVARP:x: exists tv:time
  in x-#PP::PVARP,,x that (vs or vp)@tv>>
<<ARP:x: exists tar:time
  in x-#PP::Atrial_Refractory_Period,,x
  that (as or ap)@tar>>
```

The complex timing of DDD-mode pacing can be declaratively specified with BLESS assertions.

II 1.4.4 Discrete-Time Example

A heart-rate trend thread is used to illustrate discrete-time specification using BLESS assertions. The output of the thread would be displayed as a bar-chart to graphically depict a patient's heart rate over time as detected by a pulse oximeter. However, measurements are only valid if the pulse oximeter's sensor is connected to the patient's appendage (finger, toe, or earlobe), and are not disturbed by patient motion.

The heart-rate trend thread is one of six threads providing extra functions from a stream of data from a pulse oximeter as shown in Figure II 1.3 (top most). The `NumSamples` port provides the number of samples in the trend buffer when it is not full, and is not otherwise used.

This example shows a pattern found to be useful in other behaviors: the position of data in the buffer indicates its age.

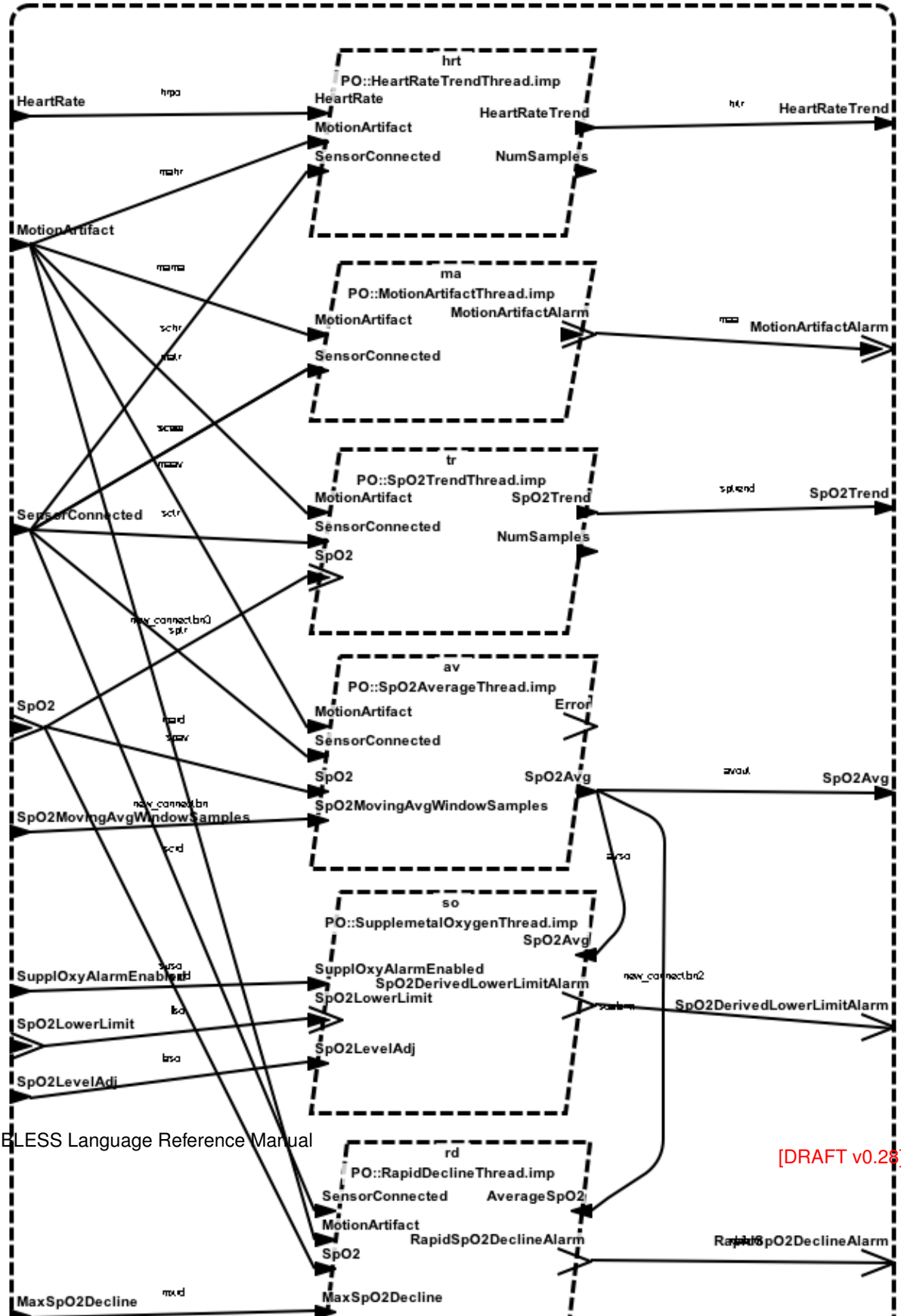
The inputs to `HeartRateTrendThread` thread are heart-rate (`HeartRate`) and whether the sensor is connected
(`SensorConnected`), or is affected by patient motion
(`MotionArtifact`). The outputs are an array of past heart rates, subject to connection and motion artifact and the number of samples for which heart-rate is available, `NumSamples`.

```
thread HeartRateTrendThread
 features
  HeartRate : in data port
    PulseOx_Types::Heart_Rate;
  SensorConnected : in data port
    Base_Types::Boolean;
  MotionArtifact : in data port
    Base_Types::Boolean;
  HeartRateTrend : out data port
    PulseOx_Types::HeartRateSamples
    {BLESS::Assertion=>"<<all s:integer
      in 1 ..num_samples are HeartRateTrend[s]=
   (MotionArtifact^(-s) or not SensorConnected^(-s)
     ?? 0 : HeartRate^(-s))>>";};
  NumSamples : out data port
    Base_Types::Integer;
 properties
   Dispatch_Protocol => Periodic;
   Period => PulseOx_Properties::
     Time_Between_Trending_Samples;
end HeartRateTrendThread;
```

The `BLESS::Assertion` expresses how position in the array output indicates the age of the element. Universal quantification over the array index allow specification for the whole array in a single BLESS assertion. The caret (^) indicates a time shift, in thread periods. In the trend buffer, all of the shifts are negative: back in time. For instance, `HeartRateTrend[1]` holds the value of `HeartRate` received during the previous period, if the sensor was then connected having no motion artifact, or zero otherwise. Each consecutive element of `HeartRateTrend` holds values further back in time, up to the number of samples collected, if less than a whole array has been sampled.

Another thread (fourth from top in Fig II 1.3) computes a moving average of blood oxygenation, SpO_2, which is also affected by motion artifact and sensor disconnect, which must not be included in the moving average. This thread also uses the timing buffer pattern, keeping a running total of SpO_2 measurements, and a count of those that have the sensor connected, without motion artifact.

```
<<SPO2_AVERAGE: :=
  --the sum of good SpO2 measurements
```


hrt
PO::HeartRateTrendThread.imp
HeartRate
MotionArtifact
SensorConnected
HeartRateTrend
NumSamples
HeartRate
HeartRateTrend
MotionArtifact
ma
PO::MotionArtifactThread.imp
MotionArtifact
SensorConnected
MotionArtifactAlarm
MotionArtifactAlarm
tr
PO::SpO2TrendThread.imp
MotionArtifact
SensorConnected
SpO2
SpO2Trend
NumSamples
SpO2Trend
SensorConnected
av
PO::SpO2AverageThread.imp
MotionArtifact
SensorConnected
SpO2
SpO2MovingAvgWindowSamples
Error
SpO2Avg
SpO2Avg
SpO2
SpO2MovingAvgWindowSamples
so
PO::SupplemetalOxygenThread.imp
SupplOxyAlarmEnabled
SpO2LowerLimit
SpO2LevelAdj
SpO2Avg
SpO2DerivedLowerLimitAlarm
SpO2DerivedLowerLimitAlarm
SupplOxyAlarmEnabled
SpO2LowerLimit
SpO2LevelAdj
rd
PO::RapidDeclineThread.imp
SensorConnected
MotionArtifact
SpO2
MaxSpO2Decline
AverageSpO2
RapidSpO2DeclineAlarm
RapidSpO2DeclineAlarm
MaxSpO2Decline

```
(sum i:integer in
    -SpO2MovingAvgWindowSamples..-1 of
 (SensorConnected^(i) and not MotionArtifact^(i)
       ?? SpO2^(i) : 0))
/
  --divided by the number of good measurements
numberof i:integer in
    -SpO2MovingAvgWindowSamples..-1
     that (SensorConnected^(i) and not
       MotionArtifact^(i)))>>
```

This BLESS assertion shows usefulness of `sum` (summation) and `numberof` (counting) quantifiers to specify periodic behavior.

II 1.5 Comparison with other BISLs

BISLs were comprehensively surveyed in [19]. Rather than show BLESS assertions differ from every other BISL, we compare with the principles quoted earlier.

First of all, other BISLs address functions that BLESS assertions ignore completely: heap allocation, stack overflow, database transactions, confidentiality-authentication-anonymity-nonrepudiation, and others. For safety-critical systems typically, dynamic memory allocation is disallowed so there is no heap, recursion is prohibited to maximum stack size can be statically determined, and databases are not provided.

Security concerns about safety-critical systems are growing, particularly for interoperable medical devices, so must be addressed some other way. BLESS assertions may be able help assure security, but that's not their central purpose.

"Verification and validation is a key component of the development and certification of such systems. . . . [S]pecifications provide a canonical declaration of a systems intended functionality against which an implementation can be verified." This is the purpose of BLESS assertions.

"*. . . [V]erification*–proving that an implementation is free of bugs and that it satisfies its formal specification in every possible execution." This is the purpose of BLESS as a whole.

BLESS assertions express *functional behavior properties* and *temporal properties*, but not *resource properties*: "constraints on how much of some resource, such as time or space, may be used by an operation". For safety-critical systems behavior and temporal properties determine safety and performance of intended function. Resource properties have not been critical in any of the examples we have studied. When resource properties are critical, one of the specification languages listed in [19] could be used in place of, or in addition to, BLESS: timed automata, TPTL, metric temporal logic, HighSpec, CS-OZ-DC, UPPAAL, Esterel, Lustre, and the duration calculus.

"Verification technology is also closely tied to semantics. A *verification logic* is a formal reasoning system that allows proofs that establish that code satisfies a specification." The verification logic that uses BLESS assertions to transform program proof outlines into complete proofs using BLESS assertions is discussed in [32].

"Pre/postconditions are one example of structured use of assertions. A precondition is an assertion that must hold whenever the procedure is called (after parameter passing). A postcondition is an assertion that must hold

immediately after the procedure completes its execution." Verification of AADL subprograms with BLESS behaviors use pre/postconditions. BLESS assertions used as pre/postconditions do not include timing, reverting to first-order predicates.

The semantic foundations of BLESS "can be traced back to Floyd-/Hoare which characterizes the behavior of program statement C using triples of the form $\{P\}C\{Q\}$" but with angle brackets, **<<>>** , replacing braces. BLESS strives for 'total' correctness which includes termination, requiring both loop invariants and bound functions to ensure termination.

BLESS assertions do not deal with exceptional behavior. BLESS programs may use constructs that detect and handle exceptions, but when they occur, conditions required for correctness proofs are violated, rendering proofs invalid.

BLESS assertions allow 'ghost' (a.k.a. logical) variables which do not occur in BLESS actions. Similarly, labelled assertions perform what [19] calls, logic functions (a.k.a. logic predicates).

BLESS supports both public and private invariants. In the example here `BLESS::Invariant=>"<<LRL(now)>>";` declares its public invariant. In [32] we show a private invariant that includes persistent variables not visible outside the thread. A verification obligation that the private invariant implies the public invariant must be discharged.

Finally, "reasoning about the correctness of an event-driven system" is no longer beyond the state-of-the-art.

Chapter II 2

BLESS Verification Conditions

For subprograms like procedures and functions, preconditions and postconditions are sufficient to specify behavior, giving rise to a single verification condition. If the precondition (applied to values of program variables) holds when invoked, the subprogram will terminate with the postcondition (applied to values of program variables) will hold.

For an Architecture Analysis and Design Language (AADL) thread component behavior, verification conditions (VC) are needed for every state and transition in the state-transition machine defining its behavior. Each of these verification conditions will be a triple–some having no statement(s): `<<P>> -> <<Q>>` For these we try to show that if P (is true), then Q (will be true), or P implies Q.

A state-transition machine has some local variables, a set of states, and transitions between states. Transitions have source states, transition condition, destination state, and perhaps an action. A proof outline of a state machine annotates state declarations with assertions about what is true about the system when in a particular state, and intersperses assertions within actions. A transition with source state a, destination state b, transition condition c, and action w would be written `a-[c]->b{w};`. If state a has assertion A, and state b has assertion B, then the verification condition will be `<<A and c>> w <<B>>`.

Unfortunately, it's not quite so simple. There are four kinds of states, and two kinds of transition conditions. Each state machine must have an *initial* state, that may not be the destination of any transition, one or more *final* states that may not be the source of any transitions. Entering a *complete* state suspends execution until the thread is next dispatched. Transitions leaving a complete state must be *dispatch* conditions evaluated by the thread dispatcher. Between dispatch and suspension a finite number of *execution* states may be encountered. Transitions leaving execution states have execute conditions which are ordinary binary-valued expressions which may refer to values of local variables.

II 2.1 Execution States

Executions states must be transitory. Thread execution may not stall or wait in an execution state. Therefore there must always be an enabled, outgoing transition. Each execution state has a verification condition that what is true about the system in that state, must imply the disjunction of transitions leaving the execution state.

Suppose execution state e, having assertion E, has three outgoing transitions with transition conditions t_1, t_2, t_3. The verification condition for e would then be:

$E \rightarrow t_1 \vee t_2 \vee t_3$

If e was declared on line 9 as

```
e : state <<E>>;
```

Its verification condition generated by the proof engine would be:

```
P [9] <<E>>
S [9] ->
Q [9] <<t1 or t2 or t3>>
```

II 2.2 Complete States

Entering a **complete** state indicates that the current computation has been completed, so execution is suspended. Transitions leaving a complete state must have dispatch conditions, which are evaluated by the dispatcher (scheduler). Because the state machine may remain in complete states for some time, the assertion of what is true about the system when in that state must imply the component (thread) invariant.

Suppose complete state c, having assertion C, is part of a state machine with invariant I. The verification condition for e would then be:

$C \rightarrow I$

If c was declared on line 25 (with invariant I was defined on line 13) as

```
invariant <<I>>;
```

```
c : complete state <<C>>;
```

Its verification condition generated by the proof engine would be:

```
P [13] <<I>>
S [25] ->
Q [25] <<C>>
```

II 2.3 Transitions with Execute Conditions

Transitions from execution states have execute conditions which are boolean expressions which may reference local variables or values of **in** ports. If the transition has no action, then the conjunction of the source state's assertion with the execute condition must imply the destination state's assertion.

Suppose a transition, L, from source state s, having assertion S, to destination state d, having assertion D, has execute condition t with no action.

```
L: s -[ t ]-> d {};
```

The verification condition for L is:

$S \wedge t \rightarrow D$

If s was declared on line 17, d was declared on line 19, and the transition L was on line 30, the verification condition generated by the proof tool would be:

```
P [17] <<S and t>>
S [30] ->
Q [19] <<D>>
```

Empty execute conditions are always true.

Execute conditions of transitions leaving an execution state need not be disjoint. If more than on execute condition is true, the transition taken will be chosen non-deterministically. When all transitions are proved, any choice will result in correct action.

Transition may perform actions after leaving the source state and before entering the destination state. The conjunction of the source state's assertion with the execute condition becomes the action's precondition, and the destination state's assertion becomes its postcondition. Consider the same transition as before, but with action w.

```
L: s -[ t ]-> d {w};
```

The verification condition generated by the proof tool would be:

```
P [17] <<S and t>>
S [30]  w
Q [19] <<D>>
```

Because the action may be composite having loops and sequential/concurrent composition, the action can be arbitrarily long. However, it must terminate in a negligible moment. Therefore locking actions (I 8.12) and **computation** (I 8.4.4) may not be used if you want to prove correctness.

II 2.4 Transitions with Dispatch Conditions

Transitions leaving **complete** states must have dispatch conditions. For periodic threads, the dispatch condition is merely **on dispatch** which will occur automatically every period. Then the assertions of source

and destination become the pre- and post-condition (resp.) of the transition's action. Almost always, a single transition leaves each complete state for periodic threads.

For aperiodic, sporadic, and timed threads, dispatch conditions may be complex: disjunction of conjunction of dispatch triggers which may include timeouts. Usually, dispatch triggers are the arrival of events at **event port** s or event-data at **event data port** s. Timeouts can be reset by events at listed ports, triggering when no events occurred since its duration.

Verification conditions for transitions with dispatch conditions are much like transition with execute conditions– except what gets conjuncted with the source state's assertion:

- the dispatch condition is true, and
- no dispatch condition leaving the source state has been true since the time-of-previous-suspension (tops).

The time-of-previous-suspension was when the source state was most recently entered.

As a simple first example, consider a complete state c having assertion C with two outgoing transitions L_1 and L_2 with no actions going to destination states d_1 and d_2 having assertions D_1 and D_2 and dispatch conditions of event arrival at event ports p_1 and p_2.

```
L1: c -[on dispatch p1]-> d1 {};
L2: c -[on dispatch p2]-> d2 {};
```

The verification condition for L_1 would be conjunction of source assertion C, that an event arrived at port p_1 (now), and no events arrived at either p_1 or p_2 in the open interval between the time-of-previous-suspension and now, implies destination state assertion D_1:

$(C \wedge p_1@now \wedge \neg\exists u \in [tops, , now] \mid (p_1 \vee p_2)@u) \rightarrow D_1$

The verification condition generated by the proof tool would be:

```
P [27] <<C and p1@now and not (exists u:time in tops,,now that (p1 or p2)@u)>>
S [28]  ->
Q [19] <<D1>>
```

A timeout dispatch trigger occurs when an event arrives or leaves at a listed port the timeout duration previously, and not since. Consider a complete state c having assertion C with a single outgoing transition, L_3, with a timeout dispatch condition on port p with duration X milliseconds to destination state d having assertion D:

```
L3: c -[on dispatch timeout (p) X ms]-> d {};
```

The verification condition for L_3 would be conjunction of source assertion C, that an event arrived at port p, X milliseconds previously, and not since, and there was no timeout since the time-of-previous-suspension, implies destination assertion D:

$C \wedge p@(now - X) \wedge \neg(\exists t \in [now - X, , now] \mid p@t) \wedge$
$\neg(\exists u \in [tops, , now] \mid p@u \wedge (\neg\exists t \in [u - X, , u] \mid p@t)) \rightarrow D$

The verification condition generated by the proof tool would be:

```
P [37] <<C and p@(now-X)
  and not (exists t:time
       in now-X,,now
       that p@t )
```

```
  and
  not (exists u:time
    in tops,,now
    that ((p@(u-X)
           and
           not (exists t:time
             in u-X,,u
             that p@t ))) )
S [38]  ->
Q [14] <<D>>
```

Transitions having dispatch conditions with actions generate verification conditions with pre- and post-conditions. If transition L_3 above had action w, then the `S` would have action `w` instead of an arrow.

For the general case, suppose there are j transitions leaving complete state c, having assertion C. Let the dispatch expressions be $e_1 \ldots e_j$, the destinations states $d_1 \ldots d_j$ having state assertions $D_1 \ldots D_j$, and actions $W_1 \ldots w_j$:

$L_1 : c -$ **[on dispatch** e_1 **]->** $d_1\{w_1\}$;
$L_2 : c -$ **[on dispatch** e_2 **]->** $d_2\{w_2\}$;
...
$L_j : c -$ **[on dispatch** e_j **]->** $d_j\{w_j\}$;

From §I 7.1, each dispatch expression $e_1 \ldots e_j$ is a disjunction of conjunctions of dispatch triggers. Form assertion-expressions $E_1 \ldots E_j$ from $e_1 \ldots e_j$ by replacing each event (data) port identifier p_k with p_k@ now, and each port-event-timeout-catch **timeout** (p_1 **or** ... **or** p_m) X u with

$(p_1 \vee \ldots \vee p_m)@(\text{now} - X) \wedge \neg\exists t \in [\text{now} - X, , \text{now}] \mid (p_1 \vee \ldots \vee p_m)@t$

For transition L_1 (other are similar) make verification condition:

$\ll C \wedge E_1 \wedge \neg\exists u \in [\text{tops}, , \text{now}] \mid E_1 \vee \ldots \vee E_j \gg w_1 \ll D_1 \gg$

II 2.5 Transitions from Initial State

Each state machine should have a single transition from its initial state having empty (default true) transition condition. This prevents cases where there is no enabled transition leaving the initial state.

Chapter II 3

Transforming Verification Conditions into Inductive Proofs

After the proof engine generates verification conditions, a human selects tactics that successively simplifies proof obligations (POs) until they are axioms. (Verification conditions are initial proof obligations.)

Generally,

- POs having composite actions are reduced to (multiple) POs having atomic actions,
- POs having atomic actions are reduced to implications ($P \rightarrow Q$), and
- POs that are implications are transformed into normal form and recognized as axioms.

II 3.1 Reducing Composite Actions

POs with composite actions are reduced to simpler actions recursively into atomic actions and implications.

II 3.1.1 Reducing Sequential Composition

Action sequences, separated by semicolons, must be performed in order. If we can prove

`<<P>>` `A1` `<<R>>` and `<<R>>` `A2` `<<Q>>`

then we can prove `<<P>>` `A1` `;` `A2` `<<Q>>` .

Someday, a smart proof engine could "guess" `<<R>>` , but for now, the proof engine requires a smart human being to supply `<<R>>` :

`<<P>>` `A1` `;` `<<R>>` `A2` `<<Q>>`

(`<<R>>` can be placed on either side of the semicolon)

Any number of actions can be sequentially composed, and thus reduced.

`<<P>> A1 ; <<R1>> A2 ; <<R2>> A3 ; <<Q>>` would be reduced to

`<<P>> A1 <<R1>>`, `<<R1>> A2 <<R2>>`, and `<<R2>> A3 <<Q>>`

Inference rules for sequential composition are defined in §I 8.5.

II 3.1.2 Reducing Concurrent Composition

Concurrent composition can be easier to prove than sequential composition.

`<<P>> S1 & S2 <<Q>>` reduces to

`<<P>> S1 <<Q>>` and `<<P>> S2 <<Q>>`

For multiple concurrent composition, put them in a block (`{ }`):

`<<P>> {<<P1>> S1 <<Q1>> & . . . & <<Pk>> Sk <<Qk>>} <<Q>>`

reduces to

`<<P>> -> <<P1>>, <<P>> -> <<P2>>,..., <<P>> -> <<Pk>>`

`<<P1>> S1 <<Q1>>,..., <<Pk>> Sk <<Qk>>`

`<<Q1 and Q2 and . . . and Qk>> -> <<Q>>`

Inference rules for concurrent composition are defined in §I 8.6.

II 3.1.3 Reducing Alternative

Alternative (`if []fi`) forms a set of guarded actions (commands). Whichever guard is true (and at least one must be true) its action is performed.

```
<<P>> if
(B1)~> <<P1>>S1<<Q1>>
[] (B2)~> <<P2>>S2<<Q2>>
[] . . .
[] (Bn)~> <<Pn>>Sn<<Qn>>
fi <<Q>>
```

reduces to

`<<P>> -> <<B1 or B2 or ... or Bn>>,`

`<<P and B1>> -> <<P1>>,..., <<P and Bn>> -> <<Pn>>,`

`<<P1>> S1 <<Q1>>,..., <<Pn>> Sn <<Qn>>,`

`<<Q1>> -> <<Q>>,..., <<Qn>> -> <<Q>>`

If the precondition of an action is missing, `<<P and Bk>>` will be used instead. Similarly if the postcondition is missing `<<Q>>` will be used.

Inference rules for alternative are defined in §I 8.7.

II 3.1.4 Reducing Forall

Inference rules for forall are defined in §I 8.9.

II 3.1.5 Reducing While Loops

Inference rules for while loop are defined in §I 8.10.1.

II 3.1.6 Reducing For Loops

Inference rules for 'for' loop are defined in §I 8.10.2.

II 3.1.7 Reducing Do-Until Loops

Inference rules for do-until' loop are defined in §I 8.10.3.

II 3.1.8 Reducing Blocks

A behavior action block (§I 8.8) introducing local variables is existential lattice quantifiaction. Otherwise it's just convenient grouping.

Inference rules for block are defined in §I 8.8.

II 3.2 Reducing Atomic Actions

After composite actions have been reduced to atomic actions, then atomic action can be reduced to implications.

II 3.2.1 Reducing Assignment

Assignment reduces to implication of the precondition by a weakest-precondition (wp) predicate transformer applied to the postcondition.

The formula for assignment wp is defined in §I 8.4.2.

II 3.2.2 Reducing Simultaneous Assignment

Simultaneous assignment also reduces using wp extended to apply to all the assignments together. Simultaneous assignment is *not* concurrent assignment, in that the assignments may be performed/reduced individually in an arbitrary order. If variables to be assigned appear on the r.h.s. of any equation, different orders may produce different results.

Operationally, all of the 'old' values of variables would be frozen; computed 'new' values would be cached; and once they all have been computed, cached values replace their previous values.

Commonly, simultaneous assignment is used at the end of periodic threads' actions to assign the 'next' values of all persistent variables.

The formula for simultaneous assignment wp is defined in §I 8.4.3.

II 3.2.3 Reducing Port Output

Reducing port output produces two implications:

- the precondition implies the port's assertion
- the conjunction of the precondition and equality of the port to its new value implies the postcondition.

Events and event values are actually sent at completion, but are assumed to happen immediately. For event ports, `p` , the added term is `p@t` ; for event data ports, `p(e)` the added term is `(p=e)` .

Inference rules for reducing port output can be found in §I 9.7.

II 3.2.4 Reducing Port Input

Port input assigns the value of an **`in data`** port to a variable. The reduced proof obligation has the conjunction of the precondition and equality of the variable with the port's assertion imply the post condition.

Where `G` is the assertion of port `g`

```
<<P>> g?(v) <<Q>>
```

reduces to

```
<<P and (g=v)>> -> <<Q>>
```

However, AADL allows **`in event data`** to have a buffer to which more than one data item may be stored. Currently the same inference rule is used to reduce both **`in data`** and **`in event data`** port output. Therefore, there *must* be exactly one data item in an **`in event data`** port buffer.

Inference rule for reducing port input can be found in §**??**.

II 3.2.5 Reducing Combinable Operations

Combinable operations allowed interference-free highly-concurrent access to data structures in BLESS's predecessor, DANCE. That's where viewing actions as interval temporal logic formulas satisfied by lattices of states comes from. In the original BA standard both forall and actions sets (concurrent composition) were included in the grammar. Also semicolon was used in action sequences for sequential composition.

II 3.2.6 Reducing Subprogram Invocation

II 3.3 Irreducible Actions

There are several actions that had been left out of BLESS, but put back during the reconciliation with BA.

All of the locking actions (§I 8.12), and the **`computation`** action (§I 9.1)were left out because they spoil the assumption that the time between dispatch and suspension was negligible.

Similarly, exception handling (§I 8.4.5 and §I 8.11) have no proof rules. When an exception occurs, something is broken, and the assumptions of the proof are no longer valid. Generally, `exception` will be used within an alternative, such that the non-exception condition is a different guard. When someone wants to prove a program that throws exceptions, use of `exception` will be restricted to be the sole action of a guarded command. When reducing alternative, guarded commands would be ignored. The corresponding `catch` could then assume that the guard of the exception is true.

II 3.4 Pounding Implications Into Axioms

Much of the effort in guiding the proof engine to a complete proof of a BLESS program or system comes from pounding implications into normal form so they can be recognized as axioms. Chief among them are 'normalize', which changes ordering, and removes parentheses, and 'laws' which applies common laws of logic. This takes a bit of practice, particularly knowing what all the tactics do. Hopefully, pounding implications into axioms can be completely automated.

For now, implications can be exported to Coq. However, Coq also requires users to guide its proof engine through selection of tactics. Nevertheless, Coq has had much more effort and scrutiny applied to it which may improve confidence in correctness proofs' soundness.

Chapter II 4

Verifying Composition

Proving that operational state machines uphold their declarative specifications provides great confidence that a lowest-level AADL component will behave as intended. Proving that whole systems meet their specifications is different–a component having subcomponents proves its behavior meets its specification from the proved-correct specifications of its subcomponents.

Like state machines, behavior of composite components is specified by the assertions of its features (ports) and an invariant assertion. This gives rise to to kinds of verification condition:

- each connection entails an assume-guarantee contract
- the conjunction of subcomponent invariants implies the invariant of the whole

Each level in a hierarchical AADL architecture can be thus proved from the subcomponents of containing components.

II 4.1 Connection Assume-Guarantee Contracts

Assume-guarantee contracts were first used by Chandy and Misra [34] and Jones [25]. More recently, Cimatti and Tonetta used linear temporal logic (LTL) to describe contracts using model checking with NuSMV to determine contract validity [11]. Boolean algebra has also been used to reason about assume-guarantee contracts [16]. Perhaps the closest work to ours is OCRA (Othello Contracts Refinement Analysis) [10] which uses a variant of LTL where formulas represent sets of hybrid traces, mixing discrete- and continuous-time steps, and is therefore amenable to model properties of timed and hybrid systems, and AGREE (Assume Guarantee Reasoning Environment) [12] which also uses LTL. Our assume-guarantee contracts differ in that they apply to architectural connections between components using a temporal logic allowing quantification over continuous time [30].

Correctness of a connection between components in an AADL architecture is determined by an assume-guarantee contract. The `BLESS::Assertion` property of the `out` port must imply the `BLESS::Assertion` property of the `in` port. When a component sends an event from an `out event` port, it guarantees that the

BLESS::Assertion property of that port holds at the instant the event is sent. Conversely, when a component receives an event on an in event it assumes that the BLESS::Assertion property of that port holds at the instant the event is received.

AADL supports both immediate and delayed connections. For immediate connections the assume-guarantee contract that must be verified is simple implication. If the sending port of an immediate connection has BLESS::Assertion property <<S>> and the receiving port has BLESS::Assertion property <<R>>, then the contract to be verified is <<S>> -> <<R>>. For delayed connections the contract would be <<S^(-1)>> -> <<R>> indicating that the sending assertion being true one period previously must imply the receiving assertion when the event arrives.

For data and event data ports, the BLESS::Assertion properties of the ports usually refer to the port identifier as the value of data sent or received. Because the port identifiers can be different, the port identifiers are replaced by the identifier of the connection in the assume-guarantee contract.

II 4.2 Component Invariants

Specification of individual component behavior consists of the BLESS::Assertion properties of its ports and the BLESS::Invariant property of the component, which is also a BLESS assertion.

A component invariant is much like a loop invariant which must be true before and after each iteration of the loop, but not necessarily during execution of the loop's body. Similarly, component invariants must always hold, except during the negligible time between dispatch and suspension. (What constitutes 'negligible' depends on the application, but the intent is that component invariants are always true, except for brief moments when it doesn't matter. Towards this end, execute states must always have an enabled, out going transition; a finite number of execute states may be entered between leaving a complete state upon dispatch and entering a complete state upon suspension; actions performed during transitions must terminate; and, spin-waiting on locks or semaphores is prohibited.)

In [32] we show how the proof outline of thread behavior is transformed into a complete proof. For a component composed of subcomponents, its invariant must be implied by the conjunction of the invariant properties of its subcomponents. Essentially, what is true of the whole is derived from what is true of the parts. If a component with invariant <<I>> has three subcomponents with invariants <<J>>, <<K>>, and <<L>> then we would verify <<J and K and L>> -> <<I>>.

In [33] we use component invariant derived from the invariants of its subcomponents as part of medical device virtual integration (MD-VI) to show that interoperable medical devices composed with a control application (will) have combined safety or functional properties.

II 4.3 Composition Example: DDDR With Everything

For our example, we use an AADL model of a dual-chamber cardiac pacemaker having almost[1] all of the functionality defined in Chapter 5, Bradycardia Therapy, of The PACEMAKER System Specification [42]. A cardiac

[1]Triggered response to sensing for modes AAT and VVT was not implemented because it is a non-therapeutic mode legacy from the earliest years of implanted pacemakers before telemetry used only in clinical settings to show device sensing by triggering an immediate pace

pacemaker has many options, and no patient will use all of them at once, but an electrophysiologist may use any combination to treat a patient's specific needs. The letters DDDR are industry standard expression of pacing mode meaning (in order): dual chamber (both right-atrium and ventricle) pacing, dual chamber sensing, dual response to sensing (both inhibition of pacing, and tracking of atrial senses by ventricular paces), and rate response to increase pacing rate with patient activity.

Obviously, a program that used a special case for each possible combination of options would be monstrously complex, and impossible to formally verify. Instead, function is specified using a collection of BLESS assertions, and implemented by simple threads, each doing its own thing, whose composition delivers the therapy chosen by the physician.

II 4.3.1 Architecture

The AADL architecture for this example is shown in Figure II 4.1. The analog front-end connects to the leads into the right-atrium and right-ventricle through a header (not shown). The leads conduct millivolt-level signals of cardiac activity to the pacemaker, and deliver the single-digit volt paces to cause contraction for patients suffering from some form of slow heart rate (a.k.a. bradycardia). When cardiac signals exceed the programmed threshold, the front end sends events to the software (`a` and `v`), and delivers paces to the leads when it receives events from software (`ap` and `vp`). The front-end also detects excessive ambient noise which prevents detection of cardiac signals (`tna` and `tnv`).

A Hall-effect switch detects the presence of a powerful magnet to cause pacing at a constant rate indicative of remaining battery life. Magnet mode behavior has not been implemented, but would not be difficult.

An accelerometer detects patient motion for rate response, increasing pacing rate according to patient activity.

The history subsystem records data about device therapy (statistics and episode recordings) that is used by the following cardiologist to adjust parameter settings at quarterly check-ups.

The telemetry subsystem allows a clinician to query current parameters, read and reset the history, and relay pacemaker function in real time to a device controller-monitor (DCM).

All of the software that determines when, and if, to pace is contained in the DDD process. In AADL, a process is a protected address space within which software threads must be contained.

II 4.3.2 Software

The contents of the DDD process is shown in Figure II 4.2. Unfortunately, the many connections makes the diagram hard to read, but a simple description should suffice for understanding our message that complex functionality can be achieved, and formally verified, by composition of simple(r) components.[2]

parameters The thread on the left gets parameters from telemetry.

markers The thread on the upper right determines 'markers' which are recorded by history, or sent in real time through telemetry to the device recorder-monitor.

visible as a spike on an electrocardiogram.

[2]The full AADL project including diagrams, specification, proofs, and scripts of tactics can be found at `bless.santoslab.org`.

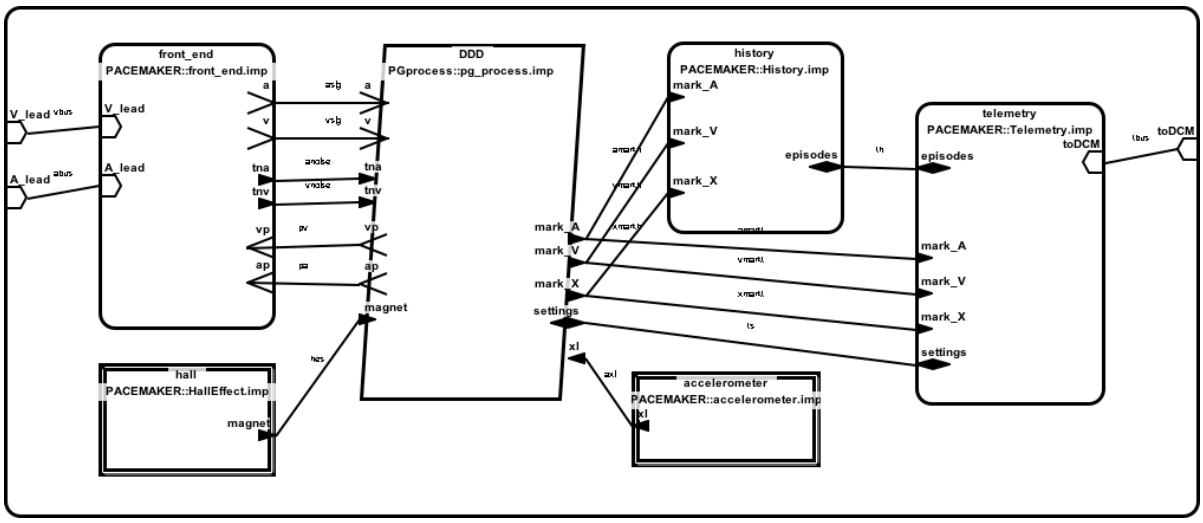


Figure II 4.1: AADL Architecture for DDDR With Everything

paces The central thread determines when, and if, paces are delivered is very similar to the thread for DDD mode pacing used as example in [32]. Instead of parameters that were constant, defined by AADL properties, they are input through ports.

rate The thread on the lower right determines pacing rate with three parameters:

- maximum cardiac cycle interval (equal to the Lower Rate Limit interval for plain DDD)
- minimum cardiac cycle interval (equal to the Upper Rate Limit interval for plain DDD)
- dynamic AV delay (equal to the AV Delay for plain DDD)

ATR atrial tachycardia response–switch to ventricle-only mode when atrium goes wild

II 4.3.3 Atrial Tachycardia Response (ATR)

Atrial Tachycardia Response limits tracking of ventricular paces following atrial senses when the atrium beats abnormally fast (tachycardia). Some patients are susceptible to "atrial storms" in which their atria beat abnormally fast, intermittently. The Upper Rate Limit prevents tracking ventricular paces from being hazardously fast, but it is still unpleasant, and damaging for sick hearts to beat so fast for long. The solution is to determine when the atrium is overly fast, switching to ventricle-only pacing and sensing, until the atrial storm subsides. When mode switching occurs, the pacing rate needs to be gradually lowered from the Upper Rate Limit to the Lower Rate Limit–changing pacing rate abruptly is especially discomforting, and risks triggering life-threatening ventricular tachycardias.

All pacemaker manufacturers have ATR, but use different algorithms, each with defensible claims that theirs is the best for some patients. Requirements for ATR in [42] were deliberately obscured from the proprietary algorithm, but all seek to not mode switch precipitously, and fall back from URL to LRL gradually.

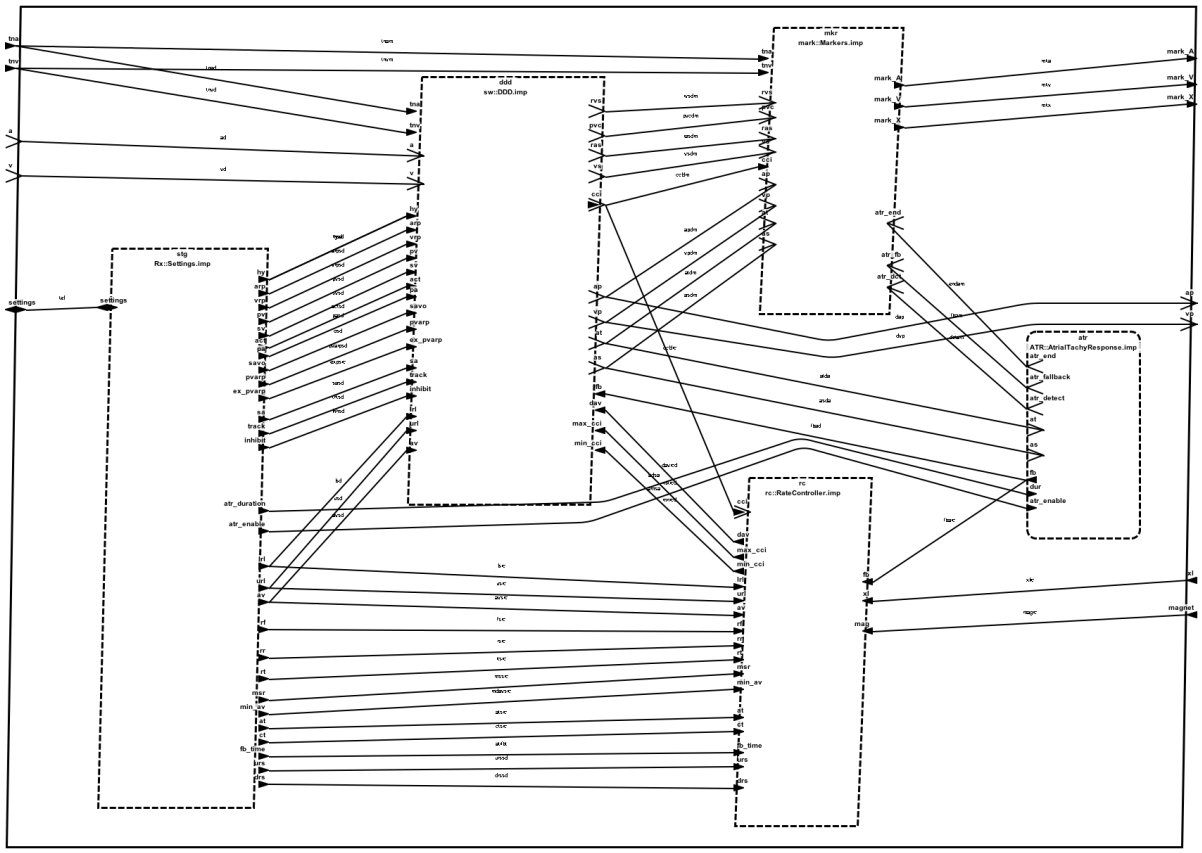


Figure II 4.2: Process Containing Threads

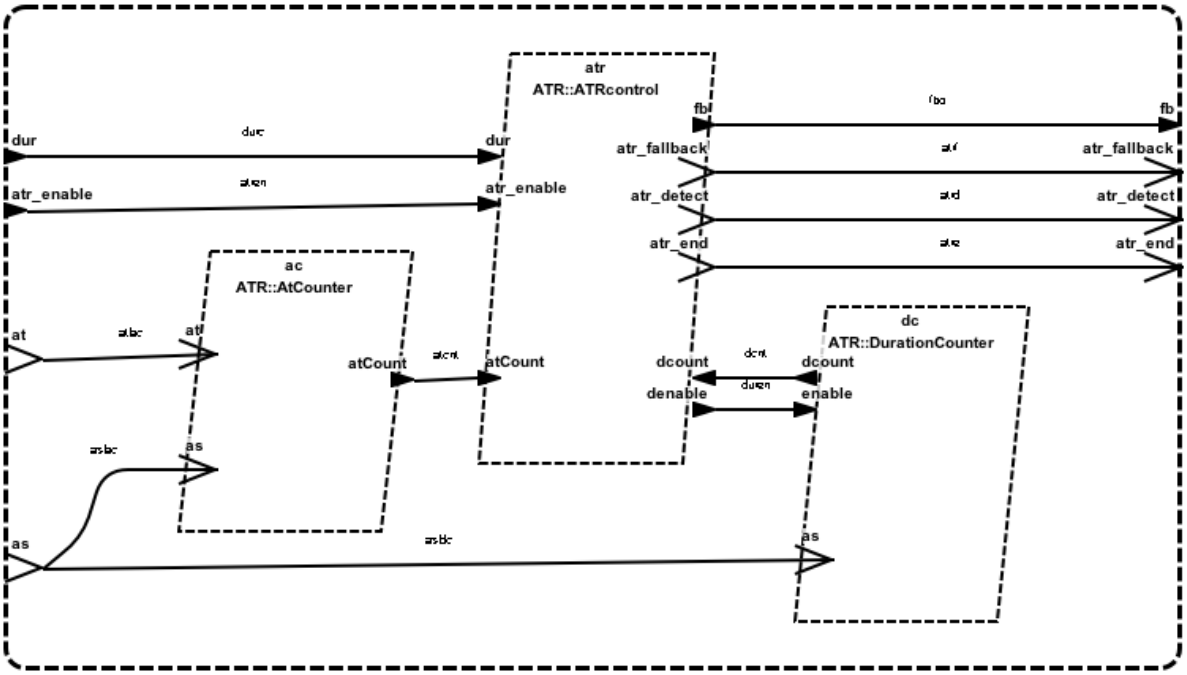


Figure II 4.3: ATR Threads

Originally, ATR was implemented as a single thread, but was too convoluted and difficult to verify. Therefore it was partitioned into three threads. `ATR::AtCounter` keeps track of how many of the most recent atrial senses (as) were faster than the Upper Rate Limit (at). `ATR::DurationCounter` counts cardiac cycles once ATR has been detected. `ATR::ATRcontrol` determines when ATR is detected (at least 5 of 7 atrial senses were fast), which starts the duration counter, when the episode ends (at most 3 of 7 atrial senses were fast), and tells the rate controller when to begin fall-back from URL to LRL.

II 4.4 Proving Assume-Guarantee Contracts

Although the `BLESS::Assertion` properties may be complex, the assume-guarantee contracts are almost always trivially solved. For example, the maximum cardiac cycle interval, `MaxCCI()` determines how long after a ventricular sense or pace to wait before delivering the next ventricular pace. It is the minimum of the hysteresis lower rate limit interval `LRL_Hy(now)`, the sensor rate interval including recovery `DN_SIRi()`, and the rate-smoothing interval when the rate is decreasing `DOWN()`.

```
<<MaxCCI: := MIN3(a:LRL_Hy(now), b:DN_SIRi(), c:DOWN())>>

<<LRL_Hy:x:  --lower Rate Limit with Hysteresis
  exists t:time  --there was a moment
  in x-HyLRL(x)..x  --within the previous Hysteresis Pacing interval
  that (n@t or p@t) >>  --with a pace or non-refractory sense

<<DN_SIRi: := MIN(a:(CCI+Y()), b:SIRi() )>>  --includes recovery
<<Y: := ((lrl-msr)*(lrl+msr)) / (2.0*(ct-lrl))>>  --down rate smoothing for recovery time
<<SIRi: := MAX(a:msr,b:(lrl- time(rf*(xl-at))))>>  --sensor indicated rate interval

<<DOWN: := CCI*(1.0+(drs/100.0))>>  --down rate smoothing
```

The calculation of `MaxCCI()` by the Rate Controller thread is complex, but the assume-guarantee contract is simple.

```
[serial 1071]: PGprocess::pg_process.imp.mxrcd:
   rc::RateController.max_cci -> sw::DDD.max_cci
P [4] <<mxrcd = (MaxCCI())>>
S [2]->
Q [5] <<mxrcd = (MaxCCI())>>
  What for: Composition of Subcomponents via Directional Connection
  PGprocess::pg_process.imp.mxrcd: rc::RateController.max_cci -> sw::DDD.max_cci
```

It comes from the `mxrcd` connection in `pg_process.imp` between the `max_cci` ports of `RateController` and `DDD`.

```
process implementation pg_process.imp
  subcomponents
    ddd: thread sw::DDD.imp;
    rc: thread rc::RateController.imp;
    atr: thread group ATR::AtrialTachyResponse.imp;
    mkr: thread mark::Markers.imp;
    stg: thread Rx::Settings.imp;
  connections
. . .
    mxrcd: port rc.max_cci->ddd.max_cci;
. . .
end pg_process.imp;

thread RateController  --choose minimum and maximum cardiac cycle intervals
  features
. . .
  max_cci: out data port ms  --maximum allowed CCI, like dynamic LRLi
    {BLESS::Assertion=>"<<:=MaxCCI()>>";};
. . .
end RateController;

thread DDD  --main PACEMAKER behavior thread
  features
. . .
  max_cci: in data port ms
    {BLESS::Assertion=>"<<:=MaxCCI()>>";};
. . .
end DDD;
```

Similarly, the containing component invariant is derived directly from its subcomponents' invariants:

```
[serial 1095]: PACEMAKER::PG.imp
P [1] <<(LRL(now))>>
S ->
Q [2] <<LRL(now)>>
  What for: Subcomponent's Invariant implies PG.imp's Invariant
```

Chapter II 5

Pacemaker Thread Example

(1) To consider a more realistic thread behavior, the PACEMAKER System Specification[1] is used to illustrate crucial timing behavior. VVI is a pacing mode that let's a patient's heart beat on its own above a prescribed rate, but take over to emit a short current to cause contraction when the patient's intrinsic rate fell below the prescribed rate.

(2) The first "V" of "VVI" says pace ventricle (right-ventricle unless otherwise indicated), the second "V" says sense ventricle, and the "I" says to inhibit pacing when sensed beats are sufficiently fast. The lower rate limit (LRL) is the heart rate, prescribed by the physician in beats per minute at which the pacemaker will not let the heart beat more slowly. In practice, the lower rate limit is less thought of by its rate in beats-per-minute, but by its duration in milliseconds.

(3) The invariant that keeps the patient lively is:

> "There will always be a pace or a (non-refractory) sense in the previous lower-rate limit interval."

Not that the average of the last 100 cardiac cycles exceeded the LRL interval, or count of beats in a minute, but that the last heart beat, intrinsic and sensed or deliberately paced, happened recently. Long pauses between heartbeats will not occur. An LRL of 60 beats-per-minute (bpm) has an LRL interval of 1000 ms.

(4) However, there is a ventricular refractory period following a sense or pace in which senses are often spurious and should be ignored.

(5) In BLESS, the reserved word **`assert`** introduces labeled assertions that may be referred-to elsewhere in the thread behavior.

(6) The following VVI.aadl defines a thread component with an **`in event port`** called `vs` for ventricular sense, and an **`out event port`** called `vp` for ventricular pace. An **`in event port`**, `stat_pace`, causes pacing to begin at implant. Occurrence of `stat_pace` defines t=0, after which, the prescription for LRL is fulfilled.

[1] sqrl.mcmaster.ca/pacemaker.htm

(7) `stop` is an implicit in event port of AADL components, here used for controlled thread termination causing transition to final state, `off`.

(8) The first assertion, LRL takes a parameter theTime. The invariant, VVI, says LRL will be true `now`, whenever `now` happens to be. `LRL(now)` says there is a time in the previous lower rate limit interval, at which a pace or a non-refractory sense occurred.

(9) The assertion `inVRP` also has a formal parameter, theTime, returning "true" if in the ventricular refractory period prior to `theTime`, there occurred a pace or a non-refractory sense, and "false" otherwise. The recursive nature of inVRP being defined in terms of itself will be of interest later.

(10) There are six states, `start off pace sense check_pace_vrp check_sense_vrp`, and one persistent variable `last_vp_or_vs`.

(11) The `start` state is the initial state when in the box before implant. The `off` state is the final state after being turned off by stop.

(12) In the `pace` state, there has been a pace in the last LRL interval ms. Similarly, in the `sense` state, an non-refractory sense occurred in the last LRL interval ms.

(13) The `check_pace_vrp` and `check_sense_vrp` occur when a sense happens, to determine if it was in VRP.

II 5.1 VVI.aadl Source Text

```
1  --VVI.aadl
2  --simple single-chamber pacemaker, VVI mode
3
4  package vvi_mode
5  public
6  with Timing_Properties; --predeclared property set for time
7  with PP;  --pacing properties, settings that define the behavior of the device
8
9  system VVI
10   features
11   vs: in event port;  --a ventricular contraction has been sensed
12   vp: out event port  --pace ventricle
13     {BLESS::Assertion=>"<<VP_vvi()>>";};
14   nr_vs: out event port --non-refractory ventricular sense
15     {BLESS::Assertion=>"<<VS_vvi()>>";};
16 annex BLESS
17 {**
18 assert
19       --lower rate limit: there was a pace or non-refractory
20       --sense before theTime, within the LRL interval
21   <<LRL_vvi:theTime: exists t:Timing_Properties::Time
22     in theTime-PP::Lower_Rate_Limit_Interval..theTime
23     that (nr_vs@t or vp@t)  >>
24
25       --vetricular refractory period:
26       --there was a pace or non-refractory sense
27       --before now, within VRP
28   <<notVRP: : (vp or nr_vs)@last_vp_or_vs and
29       (now-last_vp_or_vs)>=PP::Ventricular_Refractory_Period>>
30
31       --meaning of VS marker is nr_vs out port event
32       --should get from Assertion annexed to AADL port from OSATE
33   <<VS_vvi: : vs@now and notVRP() >>
34
35       --meaning of VP marker is vp out port event
36       --for VVI, vp! means (vp or nr_vs) occurred LRL interval ago
```

```
37        --and not since
38    <<VP_vvi: :  --last pace or sense LRL interval ago,or stat pace
39      (vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
40      and --there does not exist a time
41        not (exists t:Timing_Properties::Time
42          --since then, note ",,"
43          in now-PP::Lower_Rate_Limit_Interval,,now
44          --with a non-refractory ventricular sense or pace
45          that (nr_vs or vp)@t) >>
46
47        --a ventricular pace occurred in the previous LRL interval
48    <<PACE:theTime:vp@last_vp_or_vs and
49      (exists t:Timing_Properties::Time  --there is a time
50        in theTime-PP::Lower_Rate_Limit_Interval..theTime
51          --in the previous LRL interval ms
52        that vp@t) >>  --with a ventricular pace
53
54        --a ventricular sense occurred in the previous LRL interval
55    <<SENSE:theTime:nr_vs@last_vp_or_vs and
56      (exists t:Timing_Properties::Time  --there is a time
57        in theTime-PP::Lower_Rate_Limit_Interval..theTime
58          --in the previous LRL interval ms
59        that nr_vs@t) >>  --with a non-refractory VS
60
61        --the last VP or non-refractory VS occurred at last_vp_or_vs
62    <<LAST: :(vp or nr_vs)@last_vp_or_vs>>
63
64  invariant
65    <<LRL_vvi(now)>>  --LRL is true, whenever "now" is
66
67  variables
68    last_vp_or_vs : persistent Timing_Properties::Time;
69      --time of last ventricular pace or sense
70
71  states
72    power_on : initial state  --powered-up,
73      <<VP_vvi()>>; --okay to pace immediately
74    pace : complete state
75        --a ventricular pace has occured in the
76        --previous LRL-interval milliseconds
77      <<PACE(now)>>;
78    sense : complete state
79        --a ventricular sense has occured in the
80        --previous LRL-interval milliseconds
81      <<SENSE(now)>>;
82    check_pace_vrp : state
83        --an execute state to check if vs is in vrp
84      <<vs@now and PACE(now)>>;
85    check_sense_vrp : state
86        --need a different check state for sense
87      <<vs@now and SENSE(now)>>;
88    off : final state;  --upon "stop"
89
90  transitions
91    T1_POWER_ON:  --start pacing immediately
92    power_on -[ ]-> pace
93      {<<VP_vvi()>>vp!<<vp@now>>   --cause first pace
94      &last_vp_or_vs:=now<<last_vp_or_vs=now>>};
95
96    T3_PACE_LRL_AFTER_VP: --pace when LRL times out
97    pace -[on dispatch timeout (vp nr_vs)
98          PP::Lower_Rate_Limit_Interval ms]-> pace
99      { <<VP_vvi()>>
100     vp!<<vp@now>>
101     &last_vp_or_vs:=now<<last_vp_or_vs=now>>};
102
103   T4_VS_AFTER_VP: --sense after pace=>check if in VRP
104   pace -[on dispatch vs]-> check_pace_vrp{};
105
106   T5_VS_AFTER_VP_IN_VRP:  -- vs in VRP,  go back to "pace" state
107   check_pace_vrp -[
```

```
108     (now-last_vp_or_vs)<PP::Ventricular_Refractory_Period
109     ]-> pace{};
110
111   T6_VS_AFTER_VP_IS_NR: --vs after VRP,
112     --go to "sense" state, send nr_vs!, reset timeouts
113   check_pace_vrp -[(now-last_vp_or_vs)>=
114       PP::Ventricular_Refractory_Period]-> sense
115     {  <<VS_vvi()>>
116     nr_vs!<<nr_vs@now>>  --send nr_vs! to reset timeouts
117     &last_vp_or_vs:=now<<last_vp_or_vs=now>>};
118
119   T7_PACE_LRL_AFTER_VS: --pace when LRL times out after VS
120   sense -[on dispatch timeout (vp nr_vs)
121       PP::Lower_Rate_Limit_Interval ms]-> pace
122     {<<VP_vvi()>>
123     vp! <<vp@now>>
124     &last_vp_or_vs:=now<<last_vp_or_vs=now>>};
125
126   T8_VS_AFTER_VS: --check if vs in VRP
127   sense -[on dispatch vs]-> check_sense_vrp{};
128
129   T9_VS_AFTER_VS_IN_VRP:  -- vs in VRP,  go back to "sense" state
130   check_sense_vrp -[(now-last_vp_or_vs)<
131       PP::Ventricular_Refractory_Period]-> sense{};
132
133   T10_VS_AFTER_VS_IS_NR:  --vs after VRP is non-refractory
134   check_sense_vrp -[(now-last_vp_or_vs)>=
135       PP::Ventricular_Refractory_Period]-> sense
136     --reset timeouts with nr_vs! port send
137     {  <<VS_vvi()>>
138     nr_vs!<<nr_vs@now>>  --non-refractory ventricular sense
139     &last_vp_or_vs:=now<<last_vp_or_vs=now>>};
140
141   T11_STOP:   --turn off pacing
142   pace,sense -[on dispatch stop]-> off{};
143 **};  --end of annex subclause
144
145 end VVI;
146 end vvi_mode;
147 --end of VVI.aadl
```

II 5.2 Initial VVI Proof Obligations

The following proof obligations were extracted by the BLESS proof tool. Each proof obligation is issued a serial number when created. Theorem numbers are only created when the proof is complete.

II 5.2.1 VVI Complete State Proof Obligations

The complete states, `sense` and `pace`, must uphold the invariant, LRL:

```
[serial 1003]:
P [77] <<PACE(now)>>
S [65]->
Q [65] <<LRL_vvi(now)>>
  What for: <<M(pace)>> -> <<I>> from invariant I when complete state pace has
    Assertion <<M(pace)>> in its definition.


[serial 1004]:
P [81] <<SENSE(now)>>
S [65]->
```

```
Q [65] <<LRL_vvi(now)>>
  What for: <<M(sense)>> -> <<I>> from invariant I when complete state sense has
    Assertion <<M(sense)>> in its definition.
```

II 5.2.2 VVI Execute State Proof Obligations

The execute states, `check_pace_vrp` and `check_sense_vrp`, must uphold Serban's Theorem:

```
[serial 1005]:
P [84] <<vs@now and PACE(now)>>
S [84]->
Q [84] <<((now-last_vp_or_vs) < PP::Ventricular_Refractory_Period)
or ((now-last_vp_or_vs) >= PP::Ventricular_Refractory_Period)>>
  What for: Serban's Theorem:  disjunction of execute conditions leaving
    execution state check_pace_vrp, <<M(check_pace_vrp)>> => <<e1 or e2 or . . . en>>
```

```
[serial 1006]:
P [87] <<vs@now and SENSE(now)>>
S [87]->
Q [87] <<((now-last_vp_or_vs) < PP::Ventricular_Refractory_Period)
or ((now-last_vp_or_vs) >= PP::Ventricular_Refractory_Period)>>
  What for: Serban's Theorem:  disjunction of execute conditions leaving
    execution state check_sense_vrp, <<M(check_sense_vrp)>> => <<e1 or e2 or . . . en>>
```

II 5.2.3 VVI Initial Transition Proof Obligation

For transition `T1_POWER_ON` from the `power_on` initial state, which has no assertion because it is not operating:

```
[serial 1007]:
P [73] <<VP_vvi()>>
S [94]<<VP_vvi()>>
vp!
<<vp@now>>
&
last_vp_or_vs := now
<<last_vp_or_vs = now>>
Q [77] <<PACE(now)>>
  What for:  <<M(power_on)>> A <<M(pace)>> for T1_POWER_ON:power_on-[ ]->pace{A};
```

II 5.2.4 VVI Dispatch Condition Proof Obligations

For transitions `T3_PACE_LRL_AFTER_VP` and `T4_VS_AFTER_VP` from the `pace` complete state, upon event arrival at port `vs`:

```
[serial 1008]:
P [97] <<(PACE(now))
  and
  ((vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
    and
    not (exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval,,now
      that (vp or nr_vs)@t ))
  and
  not (exists u:Timing_Properties::Time
```

```
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
S [101]<<VP_vvi()>>
vp!
<<vp@now>>
&
last_vp_or_vs := now
<<last_vp_or_vs = now>>
Q [77] <<PACE(now)>>
  What for:  <<M(pace) and x>> A <<M(pace)>> for T3_PACE_LRL_AFTER_VP:pace-[x]->pace{A};
```

```
[serial 1009]:
P [104] <<(PACE(now))
  and
  vs@now
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
S [104]->
Q [84] <<vs@now and PACE(now)>>
  What for:  <<M(pace) and x>> => <<M(check_pace_vrp)>> for
    T4_VS_AFTER_VP:pace-[x]->check_pace_vrp{};
```

For transitions `T7_PACE_LRL_AFTER_VS` and `T8_VS_AFTER_VS` from the `sense` complete state, upon event arrival at port `vs`:

```
[serial 1012]:
P [120] <<(SENSE(now))
  and
  ((vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
    and
    not (exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval,,now
      that (vp or nr_vs)@t ))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
```

```
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
S [124]<<VP_vvi()>>
vp!
<<vp@now>>
&
last_vp_or_vs := now
<<last_vp_or_vs = now>>
Q [77] <<PACE(now)>>
  What for:  <<M(sense) and x>> A <<M(pace)>> for T7_PACE_LRL_AFTER_VS:sense-[x]->pace{A};
```

```
[serial 1013]:
P [127] <<(SENSE(now))
  and
  vs@now
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
S [127]->
Q [87] <<vs@now and SENSE(now)>>
  What for:  <<M(sense) and x>> => <<M(check_sense_vrp)>> for
    T8_VS_AFTER_VS:sense-[x]->check_sense_vrp{};
```

II 5.2.5 VVI Execute Condition Proof Obligations

Execution states `check_pace_vrp` and `check_sense_vrp` decide whether a `vs` occurs within the Ventricular Refractory Period (VRP). If in VRP, then it's a refractory ventricular sense; ignore it. Otherwise, send event `nr_vs!` to reset the timeout.

For transitions `T5_VS_AFTER_VP_IN_VRP` and `T6_VS_AFTER_VP_IS_NR` from the `check_pace_vrp` execute state, determining event arrival at port `vs` is in the ventricular refractory period:

```
[serial 1010]:
P [84] <<vs@now and PACE(now) and
          ((now-last_vp_or_vs) < PP::Ventricular_Refractory_Period)>>
S [107]->
Q [77] <<PACE(now)>>
  What for:  <<M(check_pace_vrp) and x>> => <<M(pace)>> for
    T5_VS_AFTER_VP_IN_VRP:check_pace_vrp-[x]->pace{};
```

```
[serial 1011]:
P [84] <<vs@now and PACE(now) and
          ((now-last_vp_or_vs) >= PP::Ventricular_Refractory_Period)>>
S [117]<<VS_vvi()>>
nr_vs!
```

```
<<nr_vs@now>>
&
last_vp_or_vs := now
<<last_vp_or_vs = now>>
Q [81] <<SENSE(now)>>
  What for:  <<M(check_pace_vrp) and x>> A <<M(sense)>> for
    T6_VS_AFTER_VP_IS_NR:check_pace_vrp-[x]->sense{A};
```

For transitions `T9_VS_AFTER_VS_IN_VRP` and `T10_VS_AFTER_VS_IS_NR` from the `check_sense_vrp` execute state, determining event arrival at port `vs` is in the ventricular refractory period:

```
[serial 1014]:
P [87] <<vs@now and SENSE(now) and
          ((now-last_vp_or_vs) < PP::Ventricular_Refractory_Period)>>
S [130]->
Q [81] <<SENSE(now)>>
  What for:  <<M(check_sense_vrp) and x>> => <<M(sense)>> for
     T9_VS_AFTER_VS_IN_VRP:check_sense_vrp-[x]->sense{};
```

```
[serial 1015]:
P [87] <<vs@now and SENSE(now) and
         ((now-last_vp_or_vs) >= PP::Ventricular_Refractory_Period)>>
S [139]<<VS_vvi()>>
nr_vs!
<<nr_vs@now>>
&
last_vp_or_vs := now
<<last_vp_or_vs = now>>
Q [81] <<SENSE(now)>>
  What for:  <<M(check_sense_vrp) and x>> A <<M(sense)>> for
     T10_VS_AFTER_VS_IS_NR:check_sense_vrp-[x]->sense{A};
```

II 5.2.6 VVI Stop Event Proof Obligations

For the transition `T11_STOP` from complete states `pace` and `sense` to final state `off` caused by a **`stop`** event:

```
[serial 1016]:
P [142] <<(PACE(now))
  and
  stop
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
S [142]->
Q [88] <<true>>
  What for:  <<M(pace) and x>> => <<M(off)>> for T11_STOP:pace-[x]->off{};
```

```
[serial 1017]:
P [142] <<(SENSE(now))
```

```
  and
  stop
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
S [142]->
Q [88] <<true>>
  What for:  <<M(sense) and x>> => <<M(off)>> for T11_STOP:sense-[x]->off{};
 [serial 1016]
```

II 5.3 Proof of VVI Obligations

(1) Though rather long, inspecting the generated proof is the means to convince oneself that all of the obligations have indeed been proved. The proof rules invoked are presented in section **??**, but the reader should be able to understand why a theorem is an axiom, or by what inference rule and previous theorem it was derived.

II 5.3.1 VVI Complete State Proofs

(1) The first four theorems prove that the Assertion of complete state `pace` upholds the thread invariant. Theorem (1) invokes an axiom to create a tautological implication. It seems strange when an axiom pops into existence– why did the proof tool start with that? It didn't. It ended with an axiom because the proof tool creates proofs *backwards*, beginning with the conclusion desired. The conclusion is always the last theorem in the proof, here Theorem (119).

Theorem (1) uses the most popular axiom, And-Elimination/Or-Introduction Schema. No matter what predicates P, Q, and R, the implication is always true: $(P \wedge Q) \rightarrow (P \vee R)$.

```
Theorem (1)                                  [serial 1020]
 77 {P} <<(exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that vp@t )
  and
  vp@last_vp_or_vs>>
 65 S  ->
 65 {Q} <<(exists t:Timing_Properties::Time
  in now-PP::Lower_Rate_Limit_Interval..now
  that nr_vs@t )
or (exists t:Timing_Properties::Time
  in now-PP::Lower_Rate_Limit_Interval..now
  that vp@t )>>
by And-Elimination/Or-Introduction Schema (ctao): (P and Q)->(P or R)
and Normalization Axioms:
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
What for: normalization of [serial 1019]
```

Theorem (2) adds some parentheses and changes the order of terms in {P}.

```
Theorem (2)                                  [serial 1019]
 77 {P} <<(vp@last_vp_or_vs
  and
  (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that vp@t ))>>
 65 S  ->
 65 {Q} <<((exists t:Timing_Properties::Time
  in now-PP::Lower_Rate_Limit_Interval..now
  that nr_vs@t )
or (exists t:Timing_Properties::Time
  in now-PP::Lower_Rate_Limit_Interval..now
  that vp@t ))>>
by Normalization
and Normalization Axioms:
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (1)
What for: Combine Existential Quantifications: exists x:t in R
    that (A or B) = (exists x:t in R that A) or (exists x:t in R that B) for [serial 1018]
```

Theorem (3) combines the existential quantifications in {Q}/

```
Theorem (3)                                  [serial 1018]
 77 {P} <<(vp@last_vp_or_vs
  and
  (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that vp@t ))>>
 65 S  ->
 65 {Q} <<(exists t:Timing_Properties::Time
  in now-PP::Lower_Rate_Limit_Interval..now
  that (nr_vs@t or vp@t) )>>
by Combine Existential Quantifications: exists x:t in R
   that (A or B) = (exists x:t in R that A) or (exists x:t in R that B)
and Theorem (2)
What for: substituted Assertions' predicates for  labels  for [serial 1003]
```

Finally Theorem (4) substitutes Assertion labels for their formulas. This solves the first proof obligation.

```
Theorem (4)                                  [serial 1003]
 77 {P} <<PACE(now)>>
 65 S  ->
 65 {Q} <<LRL_vvi(now)>>
by Substitution of Assertion Labels
and Theorem (3)
What for: <<M(pace)>> -> <<I>> from invariant I when complete state pace
   has Assertion <<M(pace)>> in its definition.
```

The theorems (5) through (8) prove that the Assertion of complete state `sense` upholds the thread invariant.

```
Theorem (5)                                  [serial 1024]
 81 {P} <<(exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that nr_vs@t )
  and
  nr_vs@last_vp_or_vs>>
 65 S  ->
 65 {Q} <<(exists t:Timing_Properties::Time
  in now-PP::Lower_Rate_Limit_Interval..now
  that nr_vs@t )
or (exists t:Timing_Properties::Time
  in now-PP::Lower_Rate_Limit_Interval..now
  that vp@t )>>
by And-Elimination/Or-Introduction Schema (ctao): (P and Q)->(P or R)
```

```
and Normalization Axioms:
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
What for: normalization of [serial 1023]
```

```
Theorem (6)                                    [serial 1023]
 81 {P} <<(nr_vs@last_vp_or_vs
  and
  (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that nr_vs@t ))>>
 65 S  ->
 65 {Q} <<((exists t:Timing_Properties::Time
  in now-PP::Lower_Rate_Limit_Interval..now
  that nr_vs@t )
or (exists t:Timing_Properties::Time
  in now-PP::Lower_Rate_Limit_Interval..now
  that vp@t ))>>
by Normalization
and Normalization Axioms:
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (5)
What for: Combine Existential Quantifications: exists x:t in R
   that (A or B) = (exists x:t in R that A) or (exists x:t in R that B) for [serial 1022]
```

```
Theorem (7)                                    [serial 1022]
 81 {P} <<(nr_vs@last_vp_or_vs
  and
  (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that nr_vs@t ))>>
 65 S  ->
 65 {Q} <<(exists t:Timing_Properties::Time
  in now-PP::Lower_Rate_Limit_Interval..now
  that (nr_vs@t or vp@t) )>>
by Combine Existential Quantifications: exists x:t in R
   that (A or B) = (exists x:t in R that A) or (exists x:t in R that B)
and Theorem (6)
What for: substituted Assertions' predicates for  labels  for [serial 1004]
```

```
Theorem (8)                                    [serial 1004]
 81 {P} <<SENSE(now)>>
 65 S  ->
 65 {Q} <<LRL_vvi(now)>>
by Substitution of Assertion Labels
and Theorem (7)
What for: <<M(sense)>> -> <<I>> from invariant I when complete state sense has
     Assertion <<M(sense)>> in its definition.
```

II 5.3.2 VVI Execute State Proofs

The theorems (9) through (11) prove Serban's Therorem for execute state `check_pace_vrp`.

Theorem (9) uses the axiom that whatever predicate P may be, $P \rightarrow$true is a tautology.

```
Theorem (9)                                    [serial 1028]
 84 {P} <<PACE(now) and vs@now>>
 84 S  ->
 84 {Q} <<true>>
by True Conclusion Schema (tc): P->true
What for: Law of Excluded Middle: P or not P is tautology for [serial 1026]
```

```
Theorem (10)                                    [serial 1026]
 84 {P} <<PACE(now) and vs@now>>
 84 S  ->
 84 {Q} <<not (now-last_vp_or_vs) < PP::Ventricular_Refractory_Period
or (now-last_vp_or_vs) < PP::Ventricular_Refractory_Period>>
by Law of Excluded Middle: P or not P is tautology
and Normalization Axioms:
    At Most Is Not Less Than: (a<=b) = not(b<a)
    Reflexivity of Disjunction: (m or k) = (k or m)
    Irreflexivity of At Least: (a>=b) = (b<=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (9)
What for: normalization of [serial 1005]
```

```
Theorem (11)                                    [serial 1005]
 84 {P} <<vs@now and PACE(now)>>
 84 S  ->
 84 {Q} <<((now-last_vp_or_vs) < PP::Ventricular_Refractory_Period)
or ((now-last_vp_or_vs) >= PP::Ventricular_Refractory_Period)>>
by Normalization
and Normalization Axioms:
    At Most Is Not Less Than: (a<=b) = not(b<a)
    Reflexivity of Disjunction: (m or k) = (k or m)
    Irreflexivity of At Least: (a>=b) = (b<=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (10)
What for: Serban's Theorem:  disjunction of execute conditions leaving
  execution state check_pace_vrp, <<M(check_pace_vrp)>> => <<e1 or e2 or . . . en>>
```

The theorems (12) through (14) prove Serban's Therorem for execute state `check_sense_vrp`.

```
Theorem (12)                                    [serial 1031]
 87 {P} <<SENSE(now) and vs@now>>
 87 S  ->
 87 {Q} <<true>>
by True Conclusion Schema (tc): P->true
What for: Law of Excluded Middle: P or not P is tautology for [serial 1029]
```

```
Theorem (13)                                    [serial 1029]
 87 {P} <<SENSE(now) and vs@now>>
 87 S  ->
 87 {Q} <<not (now-last_vp_or_vs) < PP::Ventricular_Refractory_Period
or (now-last_vp_or_vs) < PP::Ventricular_Refractory_Period>>
by Law of Excluded Middle: P or not P is tautology
and Normalization Axioms:
    At Most Is Not Less Than: (a<=b) = not(b<a)
    Reflexivity of Disjunction: (m or k) = (k or m)
    Irreflexivity of At Least: (a>=b) = (b<=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (12)
What for: normalization of [serial 1006]
```

```
Theorem (14)                                    [serial 1006]
 87 {P} <<vs@now and SENSE(now)>>
 87 S  ->
 87 {Q} <<((now-last_vp_or_vs) < PP::Ventricular_Refractory_Period)
or ((now-last_vp_or_vs) >= PP::Ventricular_Refractory_Period)>>
by Normalization
and Normalization Axioms:
    At Most Is Not Less Than: (a<=b) = not(b<a)
    Reflexivity of Disjunction: (m or k) = (k or m)
    Irreflexivity of At Least: (a>=b) = (b<=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
```

```
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (13)
What for: Serban's Theorem:  disjunction of execute conditions leaving execution
state check_sense_vrp, <<M(check_sense_vrp)>> => <<e1 or e2 or . . . en>>
```

II 5.3.3 Transition `T1_POWER_ON`

The theorems (15) through (28) prove transition `T1_POWER_ON`.

```
Theorem (15)                                [serial 1032]
 73 {P} <<VP_vvi()>>
 93 S  ->
 93 {Q} <<VP_vvi()>>
by Identity (id):  P->P is tautology
What for: P => Pj in concurrent composition for [serial 1007]
```

```
Theorem (16)                                [serial 1046]
 94 {P} <<vp@now and vp@last_vp_or_vs and now = last_vp_or_vs>>
 93 S  ->
 77 {Q} <<vp@now and vp@last_vp_or_vs>>
by And Introduction Schema (aiswl):  (X and Y and Z)->(X and Y)
What for: Remove Equivalent Term: P(a) and P(b) and a=b is P(a) and a=b for [serial 1044]
```

```
Theorem (17)                                [serial 1044]
 94 {P} <<vp@now and now = last_vp_or_vs>>
 93 S  ->
 77 {Q} <<vp@now and vp@last_vp_or_vs>>
by Remove Equivalent Term: P(a) and P(b) and a=b is P(a) and a=b
and Normalization Axiom:
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (16)
What for: normalization of [serial 1043]
```

```
Theorem (18)                                [serial 1043]
 94 {P} <<vp@now and now = last_vp_or_vs>>
 93 S  ->
 77 {Q} <<(vp@now) and vp@last_vp_or_vs>>
by Normalization
and Normalization Axiom:
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (17)
What for: replace exists with upper or lower bound for [serial 1041]
```

```
Theorem (19)                                [serial 1041]
 94 {P} <<vp@now and now = last_vp_or_vs>>
 93 S  ->
 77 {Q} <<(exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that vp@t )
  and
  vp@last_vp_or_vs>>
by Introduction of Existential Quantification
and Normalization Axioms:
    Reflexivity of Equality: (a=b) = (b=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (18)
What for: normalization of [serial 1040]
```

```
Theorem (20)                                    [serial 1040]
 94 {P} <<vp@now and (last_vp_or_vs = now)>>
 93 S  ->
 77 {Q} <<(vp@last_vp_or_vs
  and
  (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that vp@t ))>>
by Normalization
and Normalization Axioms:
    Reflexivity of Equality: (a=b) = (b=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (19)
What for: substituted Assertions' predicates for  labels  for [serial 1033]
```

```
Theorem (21)                                    [serial 1033]
 94 {P} <<vp@now and (last_vp_or_vs = now)>>
 93 S  ->
 77 {Q} <<PACE(now)>>
by Substitution of Assertion Labels
and Theorem (20)
What for: Q1 and Q2 and  . . . and Qn => Q in concurrent composition for [serial 1007]
```

```
Theorem (22)                                    [serial 1036]
 93 {P} <<VP_vvi()>>
 93 S  ->
 13 {Q} <<VP_vvi()>>
by Identity (id):  P->P is tautology
What for: applied port output <<pre>> -> <<M(vp)>> [serial 1034]
```

```
Theorem (23)                                    [serial 1037]
 93 {P} <<(VP_vvi()) and vp@now>>
 93 S  ->
 93 {Q} <<vp@now>>
by And Introduction Schema (aisph):  (X and Y)->X
What for: applied port output <<pre and vp@now>> -> <<post>> [serial 1034]
```

```
Theorem (24)                                    [serial 1034]
 93 {P} <<VP_vvi()>>
 93 S  vp!
 93 {Q} <<vp@now>>
by Port Event Output: when <<A and p@now>> -> <<B>> and
   <<A>> -> <<M(p)>> then <<A>> p! <<B>>
and Theorems (22) (23)
What for: <<Pj>> Sj <<Qj>> in concurrent composition for [serial 1007]
```

```
Theorem (25)                                    [serial 1039]
 73 {P} <<VP_vvi()>>
 94 S  ->
 94 {Q} <<true>>
by True Conclusion Schema (tc): P->true
What for: Equality Law (idistr):  a=a <-> true for [serial 1038]
```

```
Theorem (26)                                    [serial 1038]
 73 {P} <<VP_vvi()>>
 94 S  ->
 94 {Q} <<now = now>>
by Equality Law (idistr):  a=a <-> true
and Theorem (25)
What for: applied wp to assignment of [serial 1035]
```

```
Theorem (27)                                    [serial 1035]
 73 {P} <<VP_vvi()>>
 94 S  last_vp_or_vs := now
 94 {Q} <<last_vp_or_vs = now>>
by Assignment Rule:
 <<P>> -> <<wp(x:=e,Q)>> which is <<Q[x/e]>>
 ---------------------------------------
            <<P>> x:=e <<Q>>
and Theorem (26)
What for: <<P>> Sj <<Qj>> in concurrent composition for [serial 1007]
```

```
Theorem (28)                                    [serial 1007]
 73 {P} <<VP_vvi()>>
 94 S  <<VP_vvi()>>
vp!
<<vp@now>>
&
last_vp_or_vs := now
<<last_vp_or_vs = now>>
 77 {Q} <<PACE(now)>>
by Concurrent Composition Rule:
 <<P>> -> <<P1>>,  <<P>> -> <<P2>>, . . . , <<P>> -> <<Pn>>
 <<P1>> S1 <<Q1>>, <<P2>> S2 <<Q2>>, . . . , <<Pn>> Sn <<Qn>>
 <<Q1 and Q2 and  . . . and Qn>> -> <<Q>>
 ---------------------------------------------
       <<P>> S <<Q>> where S is
 <<P1>> S1 <<Q1>> & <<P2>> S2 <<Q2>> & . . . & <<Pn>> Sn <<Qn>>
and Theorems (15) (21) (24) (27)
What for:  <<M(power_on)>> A <<M(pace)>> for T1_POWER_ON:power_on-[ ]->pace{A};
```

II 5.3.4 Transition `T3_PACE_LRL_AFTER_VP`

The theorems (29) through (48) prove transition `T3_PACE_LRL_AFTER_VP`.

```
Theorem (29)                                    [serial 1062]
 97 {P} <<((exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval..now
      that vp@t )
    and
    vp@last_vp_or_vs)
  and
  (nr_vs or vp)@(now-PP::Lower_Rate_Limit_Interval)
  and
  not (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval,,now
    that (nr_vs or vp)@t )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that (nr_vs or vp)@(u-PP::Lower_Rate_Limit_Interval)
          and
          not (exists t:Timing_Properties::Time
            in u-PP::Lower_Rate_Limit_Interval,,u
            that (nr_vs or vp)@t ) )
  and
  not stop>>
 99 S  ->
 99 {Q} <<(nr_vs or vp)@(now-PP::Lower_Rate_Limit_Interval)
  and
  not (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval,,now
```

```
    that (nr_vs or vp)@t )>>
by And Introduction Schema (aiswl):  (X and Y and Z)->(X and Y)
and Normalization Axioms:
    Reflexivity of Disjunction: (m or k) = (k or m)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
What for: normalization of [serial 1060]
```

```
Theorem (30)                                [serial 1060]
 97 {P} <<(vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
  and
  not (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval,,now
    that (vp or nr_vs)@t )
  and
  ((vp@last_vp_or_vs
    and
    (exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval..now
      that vp@t )))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (stop)>>
 99 S  ->
 99 {Q} <<((vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
  and
  not (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval,,now
    that (nr_vs or vp)@t ))>>
by Normalization
and Normalization Axioms:
    Reflexivity of Disjunction: (m or k) = (k or m)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (29)
What for: substituted Assertions' predicates for  labels  for [serial 1056]
```

```
Theorem (31)                                [serial 1056]
 97 {P} <<(vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
  and
  not (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval,,now
    that (vp or nr_vs)@t )
  and
  (PACE(now))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (stop)>>
```

```
 99 S  ->
 99 {Q} <<VP_vvi()>>
by Substitution of Assertion Labels
and Theorem (30)
What for: Introduction of (unused) Existential Quantification for [serial 1054]
```

```
Theorem (32)                                [serial 1054]
 97 {P} <<(vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
  and
  not (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval,,now
    that (vp or nr_vs)@t )
  and
  (PACE(now))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 99 S  ->
 99 {Q} <<VP_vvi()>>
by Introduction of (unused) Existential Quantification
and Theorem (31)
What for: Associativity: (b.c).a = a.b.c for [serial 1047]
```

```
Theorem (33)                                [serial 1047]
 97 {P} <<(PACE(now))
  and
  ((vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
    and
    not (exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval,,now
      that (vp or nr_vs)@t ))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 99 S  ->
 99 {Q} <<VP_vvi()>>
by Associativity: (b.c).a = a.b.c
and Theorem (32)
What for: P => Pj in concurrent composition for [serial 1008]
```

```
Theorem (34)                                [serial 1069]
 101 {P} <<vp@now and vp@last_vp_or_vs and now = last_vp_or_vs>>
```

```
 100 S  ->
 77 {Q} <<vp@now and vp@last_vp_or_vs>>
by And Introduction Schema (aiswl):  (X and Y and Z)->(X and Y)
What for: Remove Equivalent Term: P(a) and P(b) and a=b is P(a) and a=b for [serial 1067]
```

```
Theorem (35)                                [serial 1067]
 101 {P} <<vp@now and now = last_vp_or_vs>>
 100 S  ->
 77 {Q} <<vp@now and vp@last_vp_or_vs>>
by Remove Equivalent Term: P(a) and P(b) and a=b is P(a) and a=b
and Normalization Axiom:
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (34)
What for: normalization of [serial 1065]
```

```
Theorem (36)                                [serial 1065]
 101 {P} <<vp@now and now = last_vp_or_vs>>
 100 S  ->
 77 {Q} <<(vp@now) and vp@last_vp_or_vs>>
by Normalization
and Normalization Axiom:
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (35)
What for: replace exists with upper or lower bound for [serial 1061]
```

```
Theorem (37)                                [serial 1061]
 101 {P} <<vp@now and now = last_vp_or_vs>>
 100 S  ->
 77 {Q} <<(exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that vp@t )
  and
  vp@last_vp_or_vs>>
by Introduction of Existential Quantification
and Normalization Axioms:
    Reflexivity of Equality: (a=b) = (b=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (36)
What for: normalization of [serial 1059]
```

```
Theorem (38)                                [serial 1059]
 101 {P} <<vp@now and (last_vp_or_vs = now)>>
 100 S  ->
 77 {Q} <<(vp@last_vp_or_vs
  and
  (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that vp@t ))>>
by Normalization
and Normalization Axioms:
    Reflexivity of Equality: (a=b) = (b=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (37)
What for: substituted Assertions' predicates for  labels  for [serial 1048]
```

```
Theorem (39)                                [serial 1048]
 101 {P} <<vp@now and (last_vp_or_vs = now)>>
 100 S  ->
 77 {Q} <<PACE(now)>>
by Substitution of Assertion Labels
and Theorem (38)
What for: Q1 and Q2 and  . . . and Qn => Q in concurrent composition for [serial 1008]
```

```
Theorem (40)                                [serial 1051]
 99 {P} <<VP_vvi()>>
 100 S  ->
 13 {Q} <<VP_vvi()>>
by Identity (id):  P->P is tautology
What for: applied port output <<pre>> -> <<M(vp)>> [serial 1049]
```

```
Theorem (41)                                [serial 1052]
 100 {P} <<(VP_vvi()) and vp@now>>
 100 S  ->
 100 {Q} <<vp@now>>
by And Introduction Schema (aisph):  (X and Y)->X
What for: applied port output <<pre and vp@now>> -> <<post>> [serial 1049]
```

```
Theorem (42)                                [serial 1049]
 99 {P} <<VP_vvi()>>
 100 S  vp!
 100 {Q} <<vp@now>>
by Port Event Output: when <<A and p@now>> -> <<B>> and
   <<A>> -> <<M(p)>> then <<A>> p! <<B>>
and Theorems (40) (41)
What for: <<Pj>> Sj <<Qj>> in concurrent composition for [serial 1008]
```

```
Theorem (43)                                [serial 1058]
 97 {P} <<(vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
  and
  not (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval,,now
    that (vp or nr_vs)@t )
  and
  (PACE(now))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (stop)>>
 101 S  ->
 101 {Q} <<true>>
by True Conclusion Schema (tc): P->true
What for: Introduction of (unused) Existential Quantification for [serial 1057]
```

```
Theorem (44)                                [serial 1057]
 97 {P} <<(vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
  and
  not (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval,,now
    that (vp or nr_vs)@t )
  and
  (PACE(now))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
```

```
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 101 S  ->
 101 {Q} <<true>>
by Introduction of (unused) Existential Quantification
and Theorem (43)
What for: Associativity: (b.c).a = a.b.c for [serial 1055]
```

```
Theorem (45)                                [serial 1055]
 97 {P} <<(PACE(now))
  and
  ((vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
    and
    not (exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval,,now
      that (vp or nr_vs)@t ))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 101 S  ->
 101 {Q} <<true>>
by Associativity: (b.c).a = a.b.c
and Theorem (44)
What for: Equality Law (idistr):  a=a <-> true for [serial 1053]
```

```
Theorem (46)                                [serial 1053]
 97 {P} <<(PACE(now))
  and
  ((vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
    and
    not (exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval,,now
      that (vp or nr_vs)@t ))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 101 S  ->
 101 {Q} <<now = now>>
```

```
by Equality Law (idistr):  a=a <-> true
and Theorem (45)
What for: applied wp to assignment of [serial 1050]
```

```
Theorem (47)                                    [serial 1050]
 97 {P} <<(PACE(now))
  and
  ((vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
    and
    not (exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval,,now
      that (vp or nr_vs)@t ))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 101 S  last_vp_or_vs := now
 101 {Q} <<last_vp_or_vs = now>>
by Assignment Rule:
 <<P>> -> <<wp(x:=e,Q)>> which is <<Q[x/e]>>
 ---------------------------------------
              <<P>> x:=e <<Q>>
and Theorem (46)
What for: <<P>> Sj <<Qj>> in concurrent composition for [serial 1008]
```

```
Theorem (48)                                    [serial 1008]
 97 {P} <<(PACE(now))
  and
  ((vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
    and
    not (exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval,,now
      that (vp or nr_vs)@t ))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 101 S  <<VP_vvi()>>
vp!
<<vp@now>>
&
last_vp_or_vs := now
<<last_vp_or_vs = now>>
 77 {Q} <<PACE(now)>>
by Concurrent Composition Rule:
 <<P>> -> <<P1>>,  <<P>> -> <<P2>>, . . . , <<P>> -> <<Pn>>
```

```
 <<P1>> S1 <<Q1>>, <<P2>> S2 <<Q2>>, . . . , <<Pn>> Sn <<Qn>>
 <<Q1 and Q2 and  . . . and Qn>> -> <<Q>>
 ------------------------------------------------
       <<P>> S <<Q>> where S is
 <<P1>> S1 <<Q1>> & <<P2>> S2 <<Q2>> & . . . & <<Pn>> Sn <<Qn>>
and Theorems (33) (39) (42) (47)
What for:  <<M(pace) and x>> A <<M(pace)>> for
   T3_PACE_LRL_AFTER_VP:pace-[x]->pace{A};
```

II 5.3.5 Transition T4_VS_AFTER_VP

The theorems (49) and (50) prove transition T4_VS_AFTER_VP.

```
Theorem (49)                                  [serial 1070]
 104 {P} <<PACE(now)
  and
  vs@now
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that (nr_vs or vp)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (nr_vs or vp)@t ) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 104 S  ->
 84 {Q} <<PACE(now) and vs@now>>
by And Introduction Schema (aiswl):  (X and Y and Z)->(X and Y)
and Normalization Axioms:
    Reflexivity of Disjunction: (m or k) = (k or m)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
What for: normalization of [serial 1009]
```

```
Theorem (50)                                  [serial 1009]
 104 {P} <<(PACE(now))
  and
  vs@now
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 104 S  ->
 84 {Q} <<vs@now and PACE(now)>>
by Normalization
and Normalization Axioms:
```

```
    Reflexivity of Disjunction: (m or k) = (k or m)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (49)
What for:  <<M(pace) and x>> -> <<M(check_pace_vrp)>> for
   T4_VS_AFTER_VP:pace-[x]->check_pace_vrp{};
```

II 5.3.6 Transition T5_VS_AFTER_VP_IN_VRP

The theorem (51) proves transition T5_VS_AFTER_VP_IN_VRP.

```
Theorem (51)                                [serial 1010]
 84 {P} <<vs@now and PACE(now) and
      ((now-last_vp_or_vs) < PP::Ventricular_Refractory_Period)>>
 107 S  ->
 77 {Q} <<PACE(now)>>
by And Introduction Schema (aisph):  (X and Y)->X
What for:  <<M(check_pace_vrp) and x>> -> <<M(pace)>> for
   T5_VS_AFTER_VP_IN_VRP:check_pace_vrp-[x]->pace{};
```

II 5.3.7 Transition T6_VS_AFTER_VP_IS_NR

The theorems (52) through (73) prove transition T6_VS_AFTER_VP_IS_NR.

```
Theorem (52)                                [serial 1097]
 84 {P} <<(exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that vp@t )
  and
  vp@last_vp_or_vs
  and
  vs@now
  and
  PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs)>>
 115 S  ->
 115 {Q} <<(nr_vs@last_vp_or_vs and vs@now
      and PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs))
  or (vp@last_vp_or_vs and vs@now
     and PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs))>>
by Premise Has All Terms of Conjunction within Disjunction (animporan):
     |- ( /\( l1 l2 l3 ) -> \/( l4 /\( l2 ) l5 ) )
 for proof obligations of the form <<a1 and ... and an>> -> <<b1 or ... or bm>>
  find any bj=(c1 and ... and cj) such that forall c in {c1,...,cj} there exists a in {a1,...,an}
What for: Distribution of preconditions, and over or, and
     distribution of postcondtitions, or over and for [serial 1096]
```

```
Theorem (53)                                [serial 1096]
 84 {P} <<(exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that vp@t )
  and
  vp@last_vp_or_vs
  and
  vs@now
  and
  PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs)>>
 115 S  ->
 115 {Q} <<(nr_vs@last_vp_or_vs or vp@last_vp_or_vs)
  and
  vs@now
  and
```

```
  PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs)>>
by Distribution of preconditions, and over or, and
     distribution of postcondtitions, or over and
and Theorem (52)
What for: Combine Timed Atoms: (a or b)@t = (a@t or b@t)
     (a and b)@t = (a@t and b@t) for [serial 1089]
```

```
Theorem (54)                                [serial 1089]
 84 {P} <<(exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that vp@t )
  and
  vp@last_vp_or_vs
  and
  vs@now
  and
  PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs)>>
 115 S  ->
 115 {Q} <<(nr_vs or vp)@last_vp_or_vs
  and
  vs@now
  and
  PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs)>>
by Combine Timed Atoms: (a or b)@t = (a@t or b@t)  (a and b)@t = (a@t and b@t)
and Normalization Axiom:
    Reflexivity of Conjunction: (m and k) = (k and m)
and Theorem (53)
What for: normalization of [serial 1087]
```

```
Theorem (55)                                [serial 1087]
 84 {P} <<(exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that vp@t )
  and
  vp@last_vp_or_vs
  and
  vs@now
  and
  PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs)>>
 115 S  ->
 115 {Q} <<(nr_vs or vp)@last_vp_or_vs
  and
  PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs)
  and
  vs@now>>
by Normalization
and Normalization Axiom:
    Reflexivity of Conjunction: (m and k) = (k and m)
and Theorem (54)
What for: Associativity: (b.c).a = a.b.c for [serial 1084]
```

```
Theorem (56)                                [serial 1084]
 84 {P} <<((exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval..now
      that vp@t )
    and
    vp@last_vp_or_vs)
  and
  vs@now
  and
  PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs)>>
 115 S  ->
 115 {Q} <<((nr_vs or vp)@last_vp_or_vs and
  PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs))
  and
  vs@now>>
by Associativity: (b.c).a = a.b.c
and Normalization Axioms:
```

```
    Reflexivity of Disjunction: (m or k) = (k or m)
    Irreflexivity of At Least: (a>=b) = (b<=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (55)
What for: normalization of [serial 1082]
```

```
Theorem (57)                                [serial 1082]
 84 {P} <<vs@now
  and
  (vp@last_vp_or_vs
    and
    (exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval..now
      that vp@t ))
  and
  ((now-last_vp_or_vs) >= PP::Ventricular_Refractory_Period)>>
 115 S  ->
 115 {Q} <<(vs@now
  and
  ((vp or nr_vs)@last_vp_or_vs and
    (now-last_vp_or_vs) >= PP::Ventricular_Refractory_Period))>>
by Normalization
and Normalization Axioms:
    Reflexivity of Disjunction: (m or k) = (k or m)
    Irreflexivity of At Least: (a>=b) = (b<=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (56)
What for: substituted Assertions' predicates for  labels  for [serial 1080]
```

```
Theorem (58)                                [serial 1080]
 84 {P} <<vs@now
  and
  (vp@last_vp_or_vs
    and
    (exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval..now
      that vp@t ))
  and
  ((now-last_vp_or_vs) >= PP::Ventricular_Refractory_Period)>>
 115 S  ->
 115 {Q} <<(vs@now and notVRP())>>
by Substitution of Assertion Labels
and Theorem (57)
What for: substituted Assertions' predicates for  labels  for [serial 1072]
```

```
Theorem (59)                                [serial 1072]
 84 {P} <<vs@now and PACE(now) and ((now-last_vp_or_vs) >=
      PP::Ventricular_Refractory_Period)>>
 115 S  ->
 115 {Q} <<VS_vvi()>>
by Substitution of Assertion Labels
and Theorem (58)
What for: P => Pj in concurrent composition for [serial 1011]
```

```
Theorem (60)                                [serial 1095]
 117 {P} <<nr_vs@now and nr_vs@last_vp_or_vs and now = last_vp_or_vs>>
 116 S  ->
 81 {Q} <<nr_vs@now and nr_vs@last_vp_or_vs>>
by And Introduction Schema (aiswl):  (X and Y and Z)->(X and Y)
What for: Remove Equivalent Term: P(a) and P(b) and a=b is P(a) and a=b for [serial 1093]
```

```
Theorem (61)                                [serial 1093]
 117 {P} <<nr_vs@now and now = last_vp_or_vs>>
```

```
 116 S  ->
 81 {Q} <<nr_vs@now and nr_vs@last_vp_or_vs>>
by Remove Equivalent Term: P(a) and P(b) and a=b is P(a) and a=b
and Normalization Axiom:
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (60)
What for: normalization of [serial 1091]
```

```
Theorem (62)                                [serial 1091]
 117 {P} <<nr_vs@now and now = last_vp_or_vs>>
 116 S  ->
 81 {Q} <<(nr_vs@now) and nr_vs@last_vp_or_vs>>
by Normalization
and Normalization Axiom:
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (61)
What for: replace exists with upper or lower bound for [serial 1083]
```

```
Theorem (63)                                [serial 1083]
 117 {P} <<nr_vs@now and now = last_vp_or_vs>>
 116 S  ->
 81 {Q} <<(exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that nr_vs@t )
  and
  nr_vs@last_vp_or_vs>>
by Introduction of Existential Quantification
and Normalization Axioms:
    Reflexivity of Equality: (a=b) = (b=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (62)
What for: normalization of [serial 1081]
```

```
Theorem (64)                                [serial 1081]
 117 {P} <<nr_vs@now and (last_vp_or_vs = now)>>
 116 S  ->
 81 {Q} <<(nr_vs@last_vp_or_vs
  and
  (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that nr_vs@t ))>>
by Normalization
and Normalization Axioms:
    Reflexivity of Equality: (a=b) = (b=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (63)
What for: substituted Assertions' predicates for  labels  for [serial 1073]
```

```
Theorem (65)                                [serial 1073]
 117 {P} <<nr_vs@now and (last_vp_or_vs = now)>>
 116 S  ->
 81 {Q} <<SENSE(now)>>
by Substitution of Assertion Labels
and Theorem (64)
What for: Q1 and Q2 and  . . . and Qn => Q in concurrent composition for [serial 1011]
```

```
Theorem (66)                                [serial 1076]
 115 {P} <<VS_vvi()>>
 116 S  ->
 15 {Q} <<VS_vvi()>>
by Identity (id):  P->P is tautology
What for: applied port output <<pre>> -> <<M(nr_vs)>> [serial 1074]
```

```
Theorem (67)                                    [serial 1077]
 116 {P} <<(VS_vvi()) and nr_vs@now>>
 116 S  ->
 116 {Q} <<nr_vs@now>>
by And Introduction Schema (aisph):  (X and Y)->X
What for: applied port output <<pre and nr_vs@now>> -> <<post>> [serial 1074]
```

```
Theorem (68)                                    [serial 1074]
 115 {P} <<VS_vvi()>>
 116 S  nr_vs!
 116 {Q} <<nr_vs@now>>
by Port Event Output: when <<A and p@now>> -> <<B>> and
     <<A>> -> <<M(p)>> then <<A>> p! <<B>>
and Theorems (66) (67)
What for: <<Pj>> Sj <<Qj>> in concurrent composition for [serial 1011]
```

```
Theorem (69)                                    [serial 1079]
 84 {P} <<vs@now and PACE(now) and ((now-last_vp_or_vs) >=
      PP::Ventricular_Refractory_Period)>>
 117 S  ->
 117 {Q} <<true>>
by True Conclusion Schema (tc): P->true
What for: Equality Law (idistr):  a=a <-> true for [serial 1078]
```

```
Theorem (70)                                    [serial 1078]
 84 {P} <<vs@now and PACE(now) and ((now-last_vp_or_vs) >=
      PP::Ventricular_Refractory_Period)>>
 117 S  ->
 117 {Q} <<now = now>>
by Equality Law (idistr):  a=a <-> true
and Theorem (69)
What for: applied wp to assignment of [serial 1075]
```

```
Theorem (71)                                    [serial 1075]
 84 {P} <<vs@now and PACE(now) and ((now-last_vp_or_vs) >=
      PP::Ventricular_Refractory_Period)>>
 117 S  last_vp_or_vs := now
 117 {Q} <<last_vp_or_vs = now>>
by Assignment Rule:
 <<P>> -> <<wp(x:=e,Q)>> which is <<Q[x/e]>>
 ---------------------------------------
            <<P>> x:=e <<Q>>
and Theorem (70)
What for: <<P>> Sj <<Qj>> in concurrent composition for [serial 1011]
```

```
Theorem (72)                                    [serial 1011]
 84 {P} <<vs@now and PACE(now) and ((now-last_vp_or_vs) >=
      PP::Ventricular_Refractory_Period)>>
 117 S  <<VS_vvi()>>
nr_vs!
<<nr_vs@now>>
&
last_vp_or_vs := now
<<last_vp_or_vs = now>>
 81 {Q} <<SENSE(now)>>
by Concurrent Composition Rule:
 <<P>> -> <<P1>>,  <<P>> -> <<P2>>, . . . , <<P>> -> <<Pn>>
 <<P1>> S1 <<Q1>>, <<P2>> S2 <<Q2>>, . . . , <<Pn>> Sn <<Qn>>
 <<Q1 and Q2 and  . . . and Qn>> -> <<Q>>
 ---------------------------------------------
       <<P>> S <<Q>> where S is
 <<P1>> S1 <<Q1>> & <<P2>> S2 <<Q2>> & . . . & <<Pn>> Sn <<Qn>>
and Theorems (59) (65) (68) (71)
What for:  <<M(check_pace_vrp) and x>> A <<M(sense)>> for
     T6_VS_AFTER_VP_IS_NR:check_pace_vrp-[x]->sense{A};
```

```
Theorem (73)                                [serial 1113]
 120 {P} <<((exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval..now
      that nr_vs@t )
    and
    nr_vs@last_vp_or_vs)
  and
  (nr_vs or vp)@(now-PP::Lower_Rate_Limit_Interval)
  and
  not (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval,,now
    that (nr_vs or vp)@t )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that (nr_vs or vp)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (nr_vs or vp)@t ) )
  and
  not stop>>
 122 S  ->
 122 {Q} <<(nr_vs or vp)@(now-PP::Lower_Rate_Limit_Interval)
  and
  not (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval,,now
    that (nr_vs or vp)@t )>>
by And Introduction Schema (aiswl):  (X and Y and Z)->(X and Y)
and Normalization Axioms:
    Reflexivity of Disjunction: (m or k) = (k or m)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
What for: normalization of [serial 1111]
```

II 5.3.8 Transition `T7_PACE_LRL_AFTER_VS`

The theorems (74) through (92) prove transition `T7_PACE_LRL_AFTER_VS`.

```
Theorem (74)                                [serial 1111]
 120 {P} <<(vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
  and
  not (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval,,now
    that (vp or nr_vs)@t )
  and
  ((nr_vs@last_vp_or_vs
    and
    (exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval..now
      that nr_vs@t )))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
```

```
  and
  not (stop)>>
 122 S  ->
 122 {Q} <<((vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
  and
  not (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval,,now
    that (nr_vs or vp)@t ))>>
by Normalization
and Normalization Axioms:
    Reflexivity of Disjunction: (m or k) = (k or m)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (73)
What for: substituted Assertions' predicates for  labels  for [serial 1107]
```

```
Theorem (75)                                [serial 1107]
 120 {P} <<(vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
  and
  not (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval,,now
    that (vp or nr_vs)@t )
  and
  (SENSE(now))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (stop)>>
 122 S  ->
 122 {Q} <<VP_vvi()>>
by Substitution of Assertion Labels
and Theorem (74)
What for: Introduction of (unused) Existential Quantification for [serial 1105]
```

```
Theorem (76)                                [serial 1105]
 120 {P} <<(vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
  and
  not (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval,,now
    that (vp or nr_vs)@t )
  and
  (SENSE(now))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 122 S  ->
 122 {Q} <<VP_vvi()>>
```

```
by Introduction of (unused) Existential Quantification
and Theorem (75)
What for: Associativity: (b.c).a = a.b.c for [serial 1098]
```

```
Theorem (77)                                    [serial 1098]
 120 {P} <<(SENSE(now))
  and
  ((vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
    and
    not (exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval,,now
      that (vp or nr_vs)@t ))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 122 S  ->
 122 {Q} <<VP_vvi()>>
by Associativity: (b.c).a = a.b.c
and Theorem (76)
What for: P => Pj in concurrent composition for [serial 1012]
```

```
Theorem (78)                                    [serial 1120]
 124 {P} <<vp@now and vp@last_vp_or_vs and now = last_vp_or_vs>>
 123 S  ->
 77 {Q} <<vp@now and vp@last_vp_or_vs>>
by And Introduction Schema (aiswl):  (X and Y and Z)->(X and Y)
What for: Remove Equivalent Term: P(a) and P(b) and a=b is P(a) and a=b for [serial 1118]
```

```
Theorem (79)                                    [serial 1118]
 124 {P} <<vp@now and now = last_vp_or_vs>>
 123 S  ->
 77 {Q} <<vp@now and vp@last_vp_or_vs>>
by Remove Equivalent Term: P(a) and P(b) and a=b is P(a) and a=b
and Normalization Axiom:
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (78)
What for: normalization of [serial 1116]
```

```
Theorem (80)                                    [serial 1116]
 124 {P} <<vp@now and now = last_vp_or_vs>>
 123 S  ->
 77 {Q} <<(vp@now) and vp@last_vp_or_vs>>
by Normalization
and Normalization Axiom:
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (79)
What for: replace exists with upper or lower bound for [serial 1112]
```

```
Theorem (81)                                    [serial 1112]
 124 {P} <<vp@now and now = last_vp_or_vs>>
 123 S  ->
 77 {Q} <<(exists t:Timing_Properties::Time
```

```
    in now-PP::Lower_Rate_Limit_Interval..now
    that vp@t )
  and
  vp@last_vp_or_vs>>
by Introduction of Existential Quantification
and Normalization Axioms:
    Reflexivity of Equality: (a=b) = (b=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (80)
What for: normalization of [serial 1110]
```

```
Theorem (82)                                [serial 1110]
 124 {P} <<vp@now and (last_vp_or_vs = now)>>
 123 S  ->
 77 {Q} <<(vp@last_vp_or_vs
  and
  (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that vp@t ))>>
by Normalization
and Normalization Axioms:
    Reflexivity of Equality: (a=b) = (b=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (81)
What for: substituted Assertions' predicates for  labels  for [serial 1099]
```

```
Theorem (83)                                [serial 1099]
 124 {P} <<vp@now and (last_vp_or_vs = now)>>
 123 S  ->
 77 {Q} <<PACE(now)>>
by Substitution of Assertion Labels
and Theorem (82)
What for: Q1 and Q2 and  . . . and Qn => Q in concurrent composition for [serial 1012]
```

```
Theorem (84)                                [serial 1102]
 122 {P} <<VP_vvi()>>
 123 S  ->
 13 {Q} <<VP_vvi()>>
by Identity (id):  P->P is tautology
What for: applied port output <<pre>> -> <<M(vp)>> [serial 1100]
```

```
Theorem (85)                                [serial 1103]
 123 {P} <<(VP_vvi()) and vp@now>>
 123 S  ->
 123 {Q} <<vp@now>>
by And Introduction Schema (aisph):  (X and Y)->X
What for: applied port output <<pre and vp@now>> -> <<post>> [serial 1100]
```

```
Theorem (86)                                [serial 1100]
 122 {P} <<VP_vvi()>>
 123 S  vp!
 123 {Q} <<vp@now>>
by Port Event Output: when <<A and p@now>> -> <<B>> and <<A>> -> <<M(p)>>
     then <<A>> p! <<B>>
and Theorems (84) (85)
What for: <<Pj>> Sj <<Qj>> in concurrent composition for [serial 1012]
```

```
Theorem (87)                                [serial 1109]
 120 {P} <<(vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
  and
  not (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval,,now
```

```
    that (vp or nr_vs)@t )
  and
  (SENSE(now))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (stop)>>
 124 S  ->
 124 {Q} <<true>>
by True Conclusion Schema (tc): P->true
What for: Introduction of (unused) Existential Quantification for [serial 1108]
```

```
Theorem (88)                                [serial 1108]
 120 {P} <<(vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
  and
  not (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval,,now
    that (vp or nr_vs)@t )
  and
  (SENSE(now))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 124 S  ->
 124 {Q} <<true>>
by Introduction of (unused) Existential Quantification
and Theorem (87)
What for: Associativity: (b.c).a = a.b.c for [serial 1106]
```

```
Theorem (89)                                [serial 1106]
 120 {P} <<(SENSE(now))
  and
  ((vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
    and
    not (exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval,,now
      that (vp or nr_vs)@t ))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
```

```
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 124 S  ->
 124 {Q} <<true>>
by Associativity: (b.c).a = a.b.c
and Theorem (88)
What for: Equality Law (idistr):  a=a <-> true for [serial 1104]
```

```
Theorem (90)                                [serial 1104]
 120 {P} <<(SENSE(now))
  and
  ((vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
    and
    not (exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval,,now
      that (vp or nr_vs)@t ))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 124 S  ->
 124 {Q} <<now = now>>
by Equality Law (idistr):  a=a <-> true
and Theorem (89)
What for: applied wp to assignment of [serial 1101]
```

```
Theorem (91)                                [serial 1101]
 120 {P} <<(SENSE(now))
  and
  ((vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
    and
    not (exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval,,now
      that (vp or nr_vs)@t ))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 124 S  last_vp_or_vs := now
 124 {Q} <<last_vp_or_vs = now>>
by Assignment Rule:
 <<P>> -> <<wp(x:=e,Q)>> which is <<Q[x/e]>>
```

```
 ---------------------------------------
            <<P>> x:=e <<Q>>
and Theorem (90)
What for: <<P>> Sj <<Qj>> in concurrent composition for [serial 1012]
```

```
Theorem (92)                                [serial 1012]
 120 {P} <<(SENSE(now))
  and
  ((vp or nr_vs)@(now-PP::Lower_Rate_Limit_Interval)
    and
    not (exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval,,now
      that (vp or nr_vs)@t ))
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 124 S  <<VP_vvi()>>
vp!
<<vp@now>>
&
last_vp_or_vs := now
<<last_vp_or_vs = now>>
 77 {Q} <<PACE(now)>>
by Concurrent Composition Rule:
 <<P>> -> <<P1>>,  <<P>> -> <<P2>>, . . . , <<P>> -> <<Pn>>
 <<P1>> S1 <<Q1>>, <<P2>> S2 <<Q2>>, . . . , <<Pn>> Sn <<Qn>>
 <<Q1 and Q2 and  . . . and Qn>> -> <<Q>>
 ---------------------------------------------
       <<P>> S <<Q>> where S is
 <<P1>> S1 <<Q1>> & <<P2>> S2 <<Q2>> & . . . & <<Pn>> Sn <<Qn>>
and Theorems (77) (83) (86) (91)
What for:  <<M(sense) and x>> A <<M(pace)>> for T7_PACE_LRL_AFTER_VS:sense-[x]->pace{A};
```

II 5.3.9 Transition T8_VS_AFTER_VS

The theorems (93) and (94) prove transition T8_VS_AFTER_VS.

```
Theorem (93)                                [serial 1121]
 127 {P} <<SENSE(now)
  and
  vs@now
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that (nr_vs or vp)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (nr_vs or vp)@t ) )
```

```
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 127 S  ->
 87 {Q} <<SENSE(now) and vs@now>>
by And Introduction Schema (aiswl):  (X and Y and Z)->(X and Y)
and Normalization Axioms:
    Reflexivity of Disjunction: (m or k) = (k or m)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
What for: normalization of [serial 1013]
```

```
Theorem (94)                                [serial 1013]
 127 {P} <<(SENSE(now))
  and
  vs@now
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 127 S  ->
 87 {Q} <<vs@now and SENSE(now)>>
by Normalization
and Normalization Axioms:
    Reflexivity of Disjunction: (m or k) = (k or m)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (93)
What for:  <<M(sense) and x>> -> <<M(check_sense_vrp)>> for
     T8_VS_AFTER_VS:sense-[x]->check_sense_vrp{};
```

II 5.3.10 Transition `T9_VS_AFTER_VS_IN_VRP`

The theorem (95) prove transition `T9_VS_AFTER_VS_IN_VRP`.

```
Theorem (95)                                [serial 1014]
 87 {P} <<vs@now and SENSE(now) and
      ((now-last_vp_or_vs) < PP::Ventricular_Refractory_Period)>>
 130 S  ->
 81 {Q} <<SENSE(now)>>
by And Introduction Schema (aisph):  (X and Y)->X
What for:  <<M(check_sense_vrp) and x>> -> <<M(sense)>> for
     T9_VS_AFTER_VS_IN_VRP:check_sense_vrp-[x]->sense{};
```

II 5.3.11 Transition `T10_VS_AFTER_VS_IS_NR`

The theorems (96) through (116) prove transition `T10_VS_AFTER_VS_IS_NR`.

```
Theorem (96)                                [serial 1148]
 87 {P} <<(exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that nr_vs@t )
  and
  nr_vs@last_vp_or_vs
  and
  vs@now
  and
  PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs)>>
 137 S  ->
 137 {Q} <<(nr_vs@last_vp_or_vs and vs@now
      and PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs))
  or (vp@last_vp_or_vs and vs@now
      and PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs))>>
by Premise Has All Terms of Conjunction within Disjunction (animporan):
   |- ( /\( l1 l2 l3 ) -> \/( l4 /\( l2 ) l5 ) )
 for proof obligations of the form <<a1 and ... and an>> -> <<b1 or ... or bm>>
  find any bj=(c1 and ... and cj) such that
    forall c in {c1,...,cj} there exists a in {a1,...,an}
What for: Distribution of preconditions, and over or, and
     distribution of postcondtitions, or over and for [serial 1147]
```

```
Theorem (97)                                [serial 1147]
 87 {P} <<(exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that nr_vs@t )
  and
  nr_vs@last_vp_or_vs
  and
  vs@now
  and
  PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs)>>
 137 S  ->
 137 {Q} <<(nr_vs@last_vp_or_vs or vp@last_vp_or_vs)
  and vs@now
  and PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs)>>
by Distribution of preconditions, and over or, and
     distribution of postcondtitions, or over and
and Theorem (96)
What for: Combine Timed Atoms: (a or b)@t = (a@t or b@t)
   (a and b)@t = (a@t and b@t) for [serial 1140]
```

```
Theorem (98)                                [serial 1140]
 87 {P} <<(exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that nr_vs@t )
  and
  nr_vs@last_vp_or_vs
  and
  vs@now
  and
  PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs)>>
 137 S  ->
 137 {Q} <<(nr_vs or vp)@last_vp_or_vs
  and vs@now
  and PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs)>>
by Combine Timed Atoms: (a or b)@t = (a@t or b@t)  (a and b)@t = (a@t and b@t)
and Normalization Axiom:
    Reflexivity of Conjunction: (m and k) = (k and m)
and Theorem (97)
What for: normalization of [serial 1138]
```

```
Theorem (99)                                [serial 1138]
 87 {P} <<(exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that nr_vs@t )
```

```
  and
  nr_vs@last_vp_or_vs
  and
  vs@now
  and
  PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs)>>
 137 S  ->
 137 {Q} <<(nr_vs or vp)@last_vp_or_vs
  and PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs)
  and vs@now>>
by Normalization
and Normalization Axiom:
    Reflexivity of Conjunction: (m and k) = (k and m)
and Theorem (98)
What for: Associativity: (b.c).a = a.b.c for [serial 1135]
```

```
Theorem (100)                                [serial 1135]
 87 {P} <<((exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval..now
      that nr_vs@t )
    and
    nr_vs@last_vp_or_vs)
  and
  vs@now
  and
  PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs)>>
 137 S  ->
 137 {Q} <<((nr_vs or vp)@last_vp_or_vs
  and PP::Ventricular_Refractory_Period <= (now-last_vp_or_vs))
  and vs@now>>
by Associativity: (b.c).a = a.b.c
and Normalization Axioms:
    Reflexivity of Disjunction: (m or k) = (k or m)
    Irreflexivity of At Least: (a>=b) = (b<=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (99)
What for: normalization of [serial 1133]
```

```
Theorem (101)                                [serial 1133]
 87 {P} <<vs@now
  and
  (nr_vs@last_vp_or_vs
    and
    (exists t:Timing_Properties::Time
      in now-PP::Lower_Rate_Limit_Interval..now
      that nr_vs@t ))
  and
  ((now-last_vp_or_vs) >= PP::Ventricular_Refractory_Period)>>
 137 S  ->
 137 {Q} <<(vs@now
  and ((vp or nr_vs)@last_vp_or_vs
  and (now-last_vp_or_vs) >= PP::Ventricular_Refractory_Period))>>
by Normalization
and Normalization Axioms:
    Reflexivity of Disjunction: (m or k) = (k or m)
    Irreflexivity of At Least: (a>=b) = (b<=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (100)
What for: substituted Assertions' predicates for  labels  for [serial 1131]
```

```
Theorem (102)                                [serial 1131]
 87 {P} <<vs@now
  and
  (nr_vs@last_vp_or_vs
    and
    (exists t:Timing_Properties::Time
```

```
      in now-PP::Lower_Rate_Limit_Interval..now
      that nr_vs@t ))
  and
  ((now-last_vp_or_vs) >= PP::Ventricular_Refractory_Period)>>
 137 S  ->
 137 {Q} <<(vs@now and notVRP())>>
by Substitution of Assertion Labels
and Theorem (101)
What for: substituted Assertions' predicates for  labels  for [serial 1123]
```

```
Theorem (103)                              [serial 1123]
 87 {P} <<vs@now and SENSE(now) and
      ((now-last_vp_or_vs) >= PP::Ventricular_Refractory_Period)>>
 137 S  ->
 137 {Q} <<VS_vvi()>>
by Substitution of Assertion Labels
and Theorem (102)
What for: P => Pj in concurrent composition for [serial 1015]
```

```
Theorem (104)                              [serial 1146]
 139 {P} <<nr_vs@now and nr_vs@last_vp_or_vs and now = last_vp_or_vs>>
 138 S  ->
 81 {Q} <<nr_vs@now and nr_vs@last_vp_or_vs>>
by And Introduction Schema (aiswl):  (X and Y and Z)->(X and Y)
What for: Remove Equivalent Term: P(a) and P(b) and a=b is P(a) and a=b for [serial 1144]
```

```
Theorem (105)                              [serial 1144]
 139 {P} <<nr_vs@now and now = last_vp_or_vs>>
 138 S  ->
 81 {Q} <<nr_vs@now and nr_vs@last_vp_or_vs>>
by Remove Equivalent Term: P(a) and P(b) and a=b is P(a) and a=b
and Normalization Axiom:
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (104)
What for: normalization of [serial 1142]
```

```
Theorem (106)                              [serial 1142]
 139 {P} <<nr_vs@now and now = last_vp_or_vs>>
 138 S  ->
 81 {Q} <<(nr_vs@now) and nr_vs@last_vp_or_vs>>
by Normalization
and Normalization Axiom:
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (105)
What for: replace exists with upper or lower bound for [serial 1134]
```

```
Theorem (107)                              [serial 1134]
 139 {P} <<nr_vs@now and now = last_vp_or_vs>>
 138 S  ->
 81 {Q} <<(exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that nr_vs@t )
  and
  nr_vs@last_vp_or_vs>>
by Introduction of Existential Quantification
and Normalization Axioms:
    Reflexivity of Equality: (a=b) = (b=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (106)
What for: normalization of [serial 1132]
```

```
Theorem (108)                              [serial 1132]
 139 {P} <<nr_vs@now and (last_vp_or_vs = now)>>
```

```
 138 S  ->
 81 {Q} <<(nr_vs@last_vp_or_vs
  and
  (exists t:Timing_Properties::Time
    in now-PP::Lower_Rate_Limit_Interval..now
    that nr_vs@t ))>>
by Normalization
and Normalization Axioms:
    Reflexivity of Equality: (a=b) = (b=a)
    Reflexivity of Conjunction: (m and k) = (k and m)
    Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (107)
What for: substituted Assertions' predicates for  labels  for [serial 1124]
```

```
Theorem (109)                                [serial 1124]
 139 {P} <<nr_vs@now and (last_vp_or_vs = now)>>
 138 S  ->
 81 {Q} <<SENSE(now)>>
by Substitution of Assertion Labels
and Theorem (108)
What for: Q1 and Q2 and  . . . and Qn => Q in concurrent composition for [serial 1015]
```

```
Theorem (110)                                [serial 1127]
 137 {P} <<VS_vvi()>>
 138 S  ->
 15 {Q} <<VS_vvi()>>
by Identity (id):  P->P is tautology
What for: applied port output <<pre>> -> <<M(nr_vs)>> [serial 1125]
```

```
Theorem (111)                                [serial 1128]
 138 {P} <<(VS_vvi()) and nr_vs@now>>
 138 S  ->
 138 {Q} <<nr_vs@now>>
by And Introduction Schema (aisph):  (X and Y)->X
What for: applied port output <<pre and nr_vs@now>> -> <<post>> [serial 1125]
```

```
Theorem (112)                                [serial 1125]
 137 {P} <<VS_vvi()>>
 138 S  nr_vs!
 138 {Q} <<nr_vs@now>>
by Port Event Output: when <<A and p@now>> -> <<B>> and <<A>> -> <<M(p)>>
     then <<A>> p! <<B>>
and Theorems (110) (111)
What for: <<Pj>> Sj <<Qj>> in concurrent composition for [serial 1015]
```

```
Theorem (113)                                [serial 1130]
 87 {P} <<vs@now and SENSE(now) and
      ((now-last_vp_or_vs) >= PP::Ventricular_Refractory_Period)>>
 139 S  ->
 139 {Q} <<true>>
by True Conclusion Schema (tc): P->true
What for: Equality Law (idistr):  a=a <-> true for [serial 1129]
```

```
Theorem (114)                                [serial 1129]
 87 {P} <<vs@now and SENSE(now) and
      ((now-last_vp_or_vs) >= PP::Ventricular_Refractory_Period)>>
 139 S  ->
 139 {Q} <<now = now>>
by Equality Law (idistr):  a=a <-> true
and Theorem (113)
What for: applied wp to assignment of [serial 1126]
```

```
Theorem (115)                                   [serial 1126]
 87 {P} <<vs@now and SENSE(now) and
      ((now-last_vp_or_vs) >= PP::Ventricular_Refractory_Period)>>
 139 S  last_vp_or_vs := now
 139 {Q} <<last_vp_or_vs = now>>
by Assignment Rule:
 <<P>> -> <<wp(x:=e,Q)>> which is <<Q[x/e]>>
 ---------------------------------------
            <<P>> x:=e <<Q>>
and Theorem (114)
What for: <<P>> Sj <<Qj>> in concurrent composition for [serial 1015]
```

```
Theorem (116)                                   [serial 1015]
 87 {P} <<vs@now and SENSE(now) and
      ((now-last_vp_or_vs) >= PP::Ventricular_Refractory_Period)>>
 139 S  <<VS_vvi()>>
nr_vs!
<<nr_vs@now>>
&
last_vp_or_vs := now
<<last_vp_or_vs = now>>
 81 {Q} <<SENSE(now)>>
by Concurrent Composition Rule:
 <<P>> -> <<P1>>,  <<P>> -> <<P2>>, . . . , <<P>> -> <<Pn>>
 <<P1>> S1 <<Q1>>, <<P2>> S2 <<Q2>>, . . . , <<Pn>> Sn <<Qn>>
 <<Q1 and Q2 and  . . . and Qn>> -> <<Q>>
 ---------------------------------------------
       <<P>> S <<Q>> where S is
 <<P1>> S1 <<Q1>> & <<P2>> S2 <<Q2>> & . . . & <<Pn>> Sn <<Qn>>
and Theorems (103) (109) (112) (115)
What for:  <<M(check_sense_vrp) and x>> A <<M(sense)>> for
     T10_VS_AFTER_VS_IS_NR:check_sense_vrp-[x]->sense{A};
```

II 5.3.12 Transition `T11_STOP`

The theorems (117) and (118) prove transition `T11_STOP`.

```
Theorem (117)                                   [serial 1016]
 142 {P} <<(PACE(now))
  and
  stop
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 142 S  ->
 88 {Q} <<true>>
by True Conclusion Schema (tc): P->true
What for:  <<M(pace) and x>> -> <<M(off)>> for T11_STOP:pace-[x]->off{};
```

```
Theorem (118)                                   [serial 1017]
 142 {P} <<(SENSE(now))
```

```
  and
  stop
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that ((vp or nr_vs)@(u-PP::Lower_Rate_Limit_Interval)
           and
           not (exists t:Timing_Properties::Time
             in u-PP::Lower_Rate_Limit_Interval,,u
             that (vp or nr_vs)@t )) )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that vs@u )
  and
  not (exists u:Timing_Properties::Time
    in tops,,now
    that stop )>>
 142 S  ->
 88 {Q} <<true>>
by True Conclusion Schema (tc): P->true
What for:  <<M(sense) and x>> -> <<M(off)>> for T11_STOP:sense-[x]->off{};
```

II 5.3.13 VVI Final Theorem

Finally, Theorem (119) claims that VVI behavior is correct because all of its proof obligations have been proved from Theorems (4) (8) (11) (14) (28) (48) (50) (51) (72) (92) (94) (95) (116) (117) (118).

```
Theorem (119)                                [serial 1002]
     {P} <<VVI proof obligations>>
 65 S  ->
     {Q} <<VVI proof obligations>>
by Initial Thread Obligations
and Theorems (4) (8) (11) (14) (28) (48) (50) (51) (72) (92) (94) (95) (116) (117) (118)
What for: Initial proof obligations for VVI
```

Q.E.D.

Part III

BLESS Proof Tool Manual

Chapter III 1

Introduction

Part III combines the user manual for the BLESS proof tool, together with its soundness proof.

Chapter III 2, BLESS Menu, describes operation of a selection from the "BLESS" pulldown menu, or equivalent hot key or proof script step. Operations that apply sets of proof rules to the current set of proof obligations, describes how the proof engineer looks for opportunities to apply those rules, and then links to the soundness proofs of corresponding inferences or axioms in the generated correctness proof.

BLESS correctness proofs are created "backwards", starting with proof goals (your initial obligations), then pounding them into axioms with successive application of proof rule sets. When all proof obligations have been solved, each initial obligation forms the root of a theorem tree, and every proof obligation is part of some[1] theorem tree. Theorem numbers are assigned by walking proof trees depth-first. Because all solved theorem trees have axioms for leaves, Theorem 1 will always be an axiom. Theorem 2 is almost always an application of some inference rule to Theorem 1. Theorem 1 popped into existence when the "axioms" set of proof rules were applied to some set of proof obligations, finding one of them to be axiomatic, thus solved. Theorem 2 was created by the last application of some set of proof rules that finally bashed a proof obligation into normal form that could be recognized as an axiom. The reasoning expressed in the resulting proof is the opposite of the reasoning you do while solving proof obligations with the BLESS proof tool. Therefore, removing unnecessary parentheses with the proof tool becomes "add unnecessary parentheses for no good reason" in the proof.

III 1.1 Installation

- Install OSATE from `http://osate.github.io/download-and-install.html`; choose "Stable version"
- Click the 'latest/' version (currently 2.2.1), then "products"

[1] Theorem trees (edges representing dependency of a inference rule truthfulness upon the truthfulness of prior theorems) are unnecessarily disjoint. Proof obligations are not shared; each exists in exactly one theorem tree. Memoizing by checking whether newly-created proof obligations have already been solved will eliminate unnecessary redundancy thus shrinking proofs, at an addition of $O(N^2)$ time for checking.

- Choose Linux, Mac OS, or Windows
- Download, unzip, and put it somewhere you can find it (i.e. Applications folder for Mac OS)
- Launch unzipped "osate" (i.e. double-click for Mac OS)[2]
- Choose a Workspace
- Help → Install New Software; Click "Add" Button
- Enter "BLESS" as name and `https://bless.santoslab.org/update` as location.
- Select all the plugins found
- Click "Next"[3]
- Read the licenses. If acceptable click "Next" again[4]
- Click "Yes" when it asks to restart.
- OSATE relaunches, and you're good to go!

III 1.2 Invocation

Click the praying hands icon to awaken the BLESS plug-in. Note dump and script file locations in BLESS console.

If you want to use hot keys instead of always using the "BLESS" pull-down menu:
Preferences → General → Keys then choose scheme "BLESS hot keys".

III 1.3 BLESS Menu

All proof tool operations are found under the "BLESS" pull-down menu (§III 2). Many of them can also be invoked with a proof script (see §III 1.4).

III 1.4 Proof Scripts

Proof scripts may be invoked from the BLESS drop-down menu.

get new script selects proof script to be used

step script invokes a single tactic from the proof script

run script execute all tactics in the script

[2]Depending on your security preferences, you may need to right-click it the first time to launch.
[3]May get "Install Remediation Page". That's okay.
[4]May get "Security Warning". Click "OK".

When run, scripts are executed until a "stop" command is encountered, or three consecutive tactics do nothing. Effect of commands in proof scripts is summarized in Table III 1.1.

Table III 1.1: Proof Script Commands

Command	Effect	Menu
alldistˆ	completely distribute ˆ and @	§III 2.14.1
and-over-or-post	distribute postconditions and-over-or	
and-over-or-pre	distribute preconditions and-over-or	
apply-conditional	apply conditional function	
asynchronous	use proof rules for asynchronous components	§**??**
atomic	reduce atomic actions	§III 2.8.2
axioms	apply axioms to solve proof obligations	§III 2.8.4
cnf	put into conjunctive normal form	§III 2.14.4
completesub	completely substitute assertions for labels	
console-off	suppress console output	§III 2.7.1
console-on	enable console output	§III 2.7.1
contract-uqr	contract universally-quantified ranges	§III 2.13.5
counting	apply `numberof` laws	§III 2.13.7
DeMorgan	apply DeMorganś law	§III 2.14.3
distribute	distribute or over and	§III 2.14.6
distˆ	distribute ˆ over terms	§III 2.14.2
elim-subtract	change subtraction to adding unary minus	§III 2.9.9
equivalent	add equivalent terms to preconditions	§III 2.15.1
extend-eqr	extend existentially-quantified ranges	§III 2.13.4
guided-sub-equals[A]	choose A for substitution of equals	§III 2.9.5
laws	apply laws of logic	§III 2.8.5
load	load and parse BLESS annex subclauses	
make-all	make all proof obligations	§**??**
make-an	make one proof obligation	§III 2.3
normalize	normalize	§III 2.8.3
now	assume present	
or-over-and-post	distribute postconditions or-over-and	
or-over-and-pre	distribute preconditions or-over-and	
periodic	use proof rules for periodic components	§**??**
push	push unsolved proof obligations back to unsolved list	§III 2.6.1
qtiming	apply quantification timing rules	§III 2.13.2
quant	apply quantification rules	§III 2.13.1
range-exp	replace ranges with boolean expressions	§III 2.9.15
reduce	reduce composite actions	§III 2.8.1
remove-axioms-post	remove axioms from postconditions	§III 2.10.2
remove-axioms-pre	remove axioms from preconditions	§III 2.10.1
replace-port	replace port names with their assertions	§III 2.9.14
replace-qv	replace quantified variables with #1#, etc.	§III 2.13.3

Command	Effect	Menu
replace<=	replace a<=b with not (b<a)	§III 2.10.1
replace<>	replace inequality with not equality	
replace->	replace a→b with not a or b	§III 2.9.13
replaceNEwithLTorGT	replace inequality with less than or greater than	
rev-distribute	distribute and over or	§III 2.14.5
sort-by-component[N]	sort obligations by component name N	
split-post	split postconditions	§III 2.11.1
split-quant	split quantifiers	§III 2.11.2
split@	split timed atoms	§**??**
stop	stop executing script	
stripExist	remove existential quantification	
sub-equals	substitute an equals	§III 2.9.6
sub-all-eq	substitute all equals	§III 2.9.7
sub-equals-and	substitute equals within conjunction	§III 2.9.8
subscript[d/f]	read subscript file, where d is a sub directory of the current script	
substitute-both	substitute assertion labels for predicates	§III 2.9.1
substitute-post	substitute assertion labels for predicates in postconditions	§III 2.9.3
substitute-pre	substitute assertion labels for predicates in preconditions	§III 2.9.2
transitive	add transitive relations to conjunctions having inequalities	§III 2.15.3
zeroquantlb	shift quantification lower-bound to zero	§III 2.13.6
??	apply conditional expression rules	§III 2.15.2
@toˆ	change at to caret	§III 2.12.1
ˆto@	change caret to at	§III 2.12.2
<=to<	make partial orders into total orders	§III 2.15.4
Comment	lines begin with #	

Chapter III 2

BLESS Menu

The "BLESS" pull-down menu is the primary means for human command of the BLESS proof tool.[1] Following sections describe each entry in the pull-down menu.

III 2.1 get new script

Bring up a file selection dialog box to choose a (new) script of proof tactics.

Sometimes after getting a new script repeatedly, Eclipse may behave strangely. Restart Eclipse whenever this happens.

III 2.2 load model

Cause all open projects in the workspace to be analyzed to find AADL components/libraries/features having BLESS (or subBLESS) subclauses or BLESS::Assertion properties. Builds data structures from which proof obligations are derived.

All initial proof obligations (a.k.a verification conditions) are created during load.

III 2.3 make an obligation

Rather than actually making an obligation, get an obligation from the initial obligations to prove. Generally, initial proof obligations are solved one at a time.

[1] To use BLESS hot-key bindings select: Preferences → General → Keys then choose scheme"BLESS hot key".

III 2.4 run script

Execute proof script rules until "stop", end of file, or three consecutive tactics do nothing. Select proof script from file selection dialog box if script not already selected.[2]

Important: After a proof is stopped, most commands from the menu are locked out. Step the script once to re-enable menu commands.

III 2.5 step script

Execute the next proof script rule. Select proof script from file selection dialog box if script not already selected.

III 2.6 Actions Menu

Actions do something, but don't apply proof rules.

III 2.6.1 push obligations back

Push all but one current proof obligation back into the initial proof obligation pool.

Sometimes, complex transition actions decompose into dozens of P→Q proof obligations. Because proof rule sets are applied to all current proof obligations, many proof obligations may be transformed by application of a rule in the set. Especially as a beginning BLESS prover, several pages of current proof obligations can be confusing. Push lets you focus on solving one proof obligation at a time. You can bluntly control the order in which proof obligations are chosen with the "sort by line", or "sort by serial" options.

III 2.6.2 close dump file

During operation, a "dump.txt" file is written with a record of everything that happened and a "script.txt" file is written with those tactics that had effect. Eclipse buffers file writes. This command flushes the buffers and closes the files.

A script.txt file is being written for you every time you run the tool. Reviewing and re-running these generated script files can be helpful. Particularly when you have solved some particularly-difficult proof obligation, and you can't remember what you did, and in what order. The script.txt file is closed when dump.txt is closed. Closing before opening loses no text. Copy interesting script files to new names, *before* overwriting it with the next launch of BLESS.

[2] All BLESS proof tool operations are written to a script.txt file (see Console for full path) that can be used as a proof script. Save a copy somewhere else before using the script.

III 2.6.3 make all obligations

Get all initial obligations to apply the same proof rules to all of them, together. Generally, different tactics are needed to prove different initial proof obligations, so this command is most used to list all the initial proof obligations.

III 2.6.4 display derivation

Show the tree structure of the proof constructed thus far. Particularly when encountering a proof obligation that can't possibly be solved, tracing its geneology can reveal how it came to be, and which lines of source code. Consider that an unsolvable proof obligation is the expected result when attempting to prove an incorrect program. Consider that your unsolvable proof obligation may be trying to tell you *exactly where in your source code your error resides*.

III 2.6.5 sort by component name

Opens a dialog box. Type the fully-qualified name of the AADL component you want to prove first.

This allows you to concentrate on the desired component's proof when there are many components in your model.

III 2.6.6 show script terms

List all of the terms allowed in a script in the Console.

III 2.6.7 Translate submenu

Translate BLESS to other languages, displayed in the Console.

emit English assertions

Attempt to translate BLESS assertions into English language. Doesn't work very well and identifiers must be selected to be words.

emit Signal

Translate BLESS annex subclauses to Signal–a process language used by Polychrony for timing analysis. On-going cooperation with Jean-Pierre Talpin and his team at IRISA will continue to improve until proved-correct BLESS behaviors can be seamlessly analyzed with Polychrony.

emit SAL

Translate BLESS into SAL–a model checker by SRI. Requested by Brendan Hall at Honeywell. SAL wasn't expressive enough for events. Brendan claims that a SAL theory for calendar could provide the missing capability.

emit BA

Translate BLESS annex subclauses to standard BA balloted by the AADL standard committee in 2016[3]. This should work, absolutely. Will be working closely with Etienne Borde at TELECOM ParisTech to ensure the translation conforms to the revised BA grammar.

III 2.6.8 Export submenu

Export unsolved proof obligations to other proof tools.

Export to Coq

Export proof obligations of the form $\ll P \gg \rightarrow \ll Q \gg$ to the Cog language to be proved with the Coq Proof Assistant.

This might be helpful for solving proof obligations that have been reduced to implications, that do not have quantification over time and temporal operators. However, because so much reduction must be done by the BLESS proof engine, exporting implications to Coq will not improve trust in validity.

III 2.7 Options Menu

Options change subsequent proof tool behavior.

III 2.7.1 suppress output

Inhibit writing of output to the BLESS console. Dramatically improves proof script execution time. Re-enables if suppressed. Default: false.

III 2.7.2 display trees

Show a pop-up window with current proof obligations after application of proof rules does something. Default: false.

[3] AS5506/2 Annex B: Behavior Model Annex

III 2.7.3 sort by line

Sort proof obligations by line number of action. This only works well when the workspace has a single package

III 2.7.4 sort by serial

Sort proof obligations by serial number.

III 2.7.5 routinely normalize

Invoke normalize (§III 2.8.3) after every operation. Both crams proof obligations together (one theorem covers and inference rule and some normaizations), and causes unnecessary proof obligations created for unnecessary normalizations. Makes finding proofs the first time easier, but routinely normalize should not be used when proof is a verification artifact. Toggles. Default: false.

III 2.8 Proof Menu

The Proof menu has the most commonly-used proof rules.

III 2.8.1 reduce composite

Look for action compositions (sequential, concurrent, loops, . . .), replace with several, simpler proof obligations. These reductions are summarized in Table III 2.1.

Grammatically, BLESS replaces production "action" in BA with production "asserted_action" (§I 8.2), which permits optional assertions as pre- and post-conditions. During parsing, a synthetic "ACTION" AST node is created to hold the action together with its assertions. You won't see an ACTION node unparsed, just that the S in `<<P>>S <<Q>>` has its own pre- and post-conditions.

III 2.8.2 reduce atomic

Look for atomic actions, replace with pure assertion proof obligations ($P \rightarrow Q$). These reductions are summarized in Table III 2.2.

III 2.8.3 normalize

Put into normal form. This is the most common proof tool command.

Reflexive terms are sorted. Expressions of numeric literals are computed. Cancel opposite terms.

Table III 2.1: Reduce Composite Action

root	reduction	semantics	mm	proof
;	reduce sequential composition	§I 8.5	bl.sck	§1
&	reduce concurrent composition	§I 8.6	bl.cck	§2
if	reduce alternative	§I 8.7	bl.iffi	§3
while	reduce iterative	§I 8.10.1	bl.loop	§4
do	reduce iterative	§I 8.10.3	bl.until	§5
for	reduce iterative	§I 8.10.2	bl.for	§6
{	reduce existential (lattice) quantification	§I 8.8	bl.elq	§7
forall	reduce universal (lattice) composition	§I 8.9	bl.ulq	§8
ACTION	reduce asserted action	§I 8.2	bl.aab	§9
	EXISTENTIAL_QUANTIFICATION		bl.aapre	§10
	_INTRODUCTION		bl.aapost	§11
			bl.aanone	§12

Table III 2.2: Reduce Atomic Action

root	reduction	semantics	mm	proof
skip	skip	§I 11.2	bl.skip	§13
:=	assignment	§I 11.2	bl.a	§14
+=	fetchadd +1	§I 8.13.1	bl.fap1	§15
	fetchadd -1	§I 8.13.1	bl.fam1	§16
	fetchadd e	§I 8.13.1	bl.fae	§17
C(e1,e2,)	subprogram invocation	§IV 6.0.4	bl.si	§18
!	port output event	§I 9.7	bl.poe	§19
	port output value	§I 9.7	bl.pov	§20
?	port input	§I 9.3	bl.pi	§21
		§I 9.4		
		§I 9.5		
<<P>>S <<Q>>	asserted action	§I 8.2	bl.aab	§9
			bl.aapre	§10
			bl.aapost	§11
			bl.aanone	§12

Unlike most commands, normalize may perform multiple operations in a single proof obligation. The normalization reasons are gathered into a set.

In Tables III 2.3, III 2.4, III 2.5, III 2.6, and III 2.7 an equivalence means: find patterns matching l.h.s.; replace with r.h.s. Pounding proof obligations into normal form creates most theorems in correctness proofs.

Abbreviations: nl stands for numeric literal; ps stands for property set identifier; pn stands for property name identifier; j and k stand for integer values.

Table III 2.3: Arithmetic Normalization

reason	**effect**	**mm**	**proof**
add/subtract same	$(a+b)-a \equiv b$ $(a+b)-b \equiv a$ $a-(a+b) \equiv -b$ $b-(a+b) \equiv -a$ $b+(a-b) \equiv a$ $(a-b)+b \equiv a$	??	??
arithmetic division	divide numeric literals	??	??
arithmetic minus	subtract numeric literals	??	??
arithmetic plus	add numeric literals	??	??
arithmetic times	multiply numeric literals	??	??
associativity	$(a-b)-(c-d) \equiv (a+d)-(b+c)$ $(-a)+b \equiv b-a$ $-a+b \equiv b-a$ $y+(-x) \equiv y-x$ $y+-x \equiv y-x$ $a+(c-d)+e1+e2... \equiv (a+c+e1+e2...)-d$ $(c-d)+e1+e2+... \equiv (c+e1+e2+...)-d$ $(a-b)+(c-d) \equiv (a+c)-(b+d)$ $(a-b)-c \equiv a-(b+c)$ $a-(c-d) \equiv (a+c)-d$ $(-c+d)-a \equiv d-(c+a)$ $(c+-d)-a \equiv c-(d+a)$	??	??
change $\geq$ to $\leq$	$a \geq b \equiv b \leq a$	??	??
change $\leq$ to not $<$	$a \leq b \equiv (\neg(b < a))$ (Only occurs when III 2.10.1)	??	??
change $>$ to $<$	$a > b \equiv b < a$	??	??
remove zero addition	$0+x \equiv x$ $x+0 \equiv x$	??	??
remove zero subtraction	$0-x \equiv -x$ $x-0 \equiv x$	??	??
unary minus	$-(x-y) \equiv y-x$	??	??
negate numeric literals		??	??

Table III 2.4: Boolean Normalization

reason	**effect**	**mm**	**proof**
complement	$\neg\top \equiv \bot$ $\neg\bot \equiv \top$ $\neg\neg x \equiv x$ $\neg(\neg x) \equiv x$	??	??
only children	and/or of single term (df-lan1) (df-lor1)		?? ??

Table III 2.5: Parentheses Normalization

reason	effect	mm	proof
remove unnecessary parentheses	$(x) \equiv x$	??	??
remove unnecessary parentheses from conditional expression	`((b??t:f)) ≡ ((b)??t:f) ≡ (b??(t):f) ≡ (b??t:(f)) ≡ (b??t:f)`	?? ??	?? ??
removeUnnecessaryParentheses from range bounds	`(lb)..ub ≡ lb..(ub) ≡ lb..ub (lb).,ub ≡ lb.,(ub) ≡ lb.,ub`	?? ??	?? ??

Table III 2.6: Reflexive Normalization

reason	effect	mm	proof
reflexivity of addition	`z+y+x ≡ x+y+z`	??	??
reflexivity of multiplication	`z*y*x ≡ x*y*z`	??	??
reflexivity of conjunction	`c and b and a ≡ a and b and c`	??	??
reflexivity of disjunction	`c or b or a ≡ a or b or c`	??	??
reflexivity of xor	`c xor b xor a ≡ a xor b xor c`	??	??
reflexivity of equality	`z=x ≡ x=z`	??	??
reflexivity of inequality	`z<>x ≡ x<>z`	??	??

Table III 2.7: Timing Normalization

reason	effect	mm	proof
caret composition	`x^j^k ≡ x^(j+k)  (x^j)^k ≡ x^(j+k)`	??	??

timed constants at	`(true)@t ≡ true  (false)@t ≡ false`
	`(nl)@t ≡ nl  (ps::pn)@t ≡ ps::pn`
	`-(nl)@t ≡ -nl  -(ps::pn)@t ≡ -ps::pn`
		??	??

timed constants caret	`(true)^b ≡ true  (false)^b ≡ false`
	`(nl)^b ≡ nl  (ps::pn)^b ≡ ps::pn`
	`-(nl)^b ≡ -nl  -(ps::pn)^b ≡ -ps::pn`
		??	??

timed constants tick	`(true)' ≡ true  (false)' ≡ false`
	`(nl)' ≡ nl  (ps::pn)' ≡ ps::pn`
	`-(nl)' ≡ -nl  -(ps::pn)' ≡ -ps::pn`
		??	??

next is same	only after distribute caret	??	??

III 2.8.4 axioms

Check if an axiom (tautology). All the leaves of a theorem tree must be axioms to be a proof. Table III 2.8

Table III 2.8: Axiom Recognition

reason	effect	mm	ref
true-conclusion	`<<A>>` → `<<true>>`	bl.tc	??
identity	`<<A>>` → `<<A>>`	id	IV 3.2
or-introduction	`<<A>>` → `<<A or D>>`	bl.orcwl	??
and-elimination	`<<A and B>>` → `<<A>>` `<<A and B and C>>` → `<<A and B>>`	??	?? ??
and-elimination or-introduction	`<<A and B>>` → `<<A or D>>` `<<A and B and C>>` → `<<(A and C)or D>>`	??	?? ??

III 2.8.5 laws

Check laws of logic. Tables III 2.9 III 2.10 III 2.11 III 2.12

Table III 2.9: Order Laws

reason	effect	mm	proof
SIMPLE_EQUALITY	$x = x \equiv$ `true`	??	??
TOTAL_ORDER	$x < x \equiv$ `false`	??	??
PARTIAL_ORDER	$x \leq x \equiv$ `true`	??	??

Table III 2.10: Boolean Laws

reason	effect	mm	proof
CONTRADICTION	$A \wedge B \wedge \neg C \equiv$ `false` (incl. (A) $(\neg C)$ $(\neg(C))$)	??	??
EXCLUDED_MIDDLE	$A \vee B \vee \neg A \equiv$ `true` (incl. (A) $(\neg A)$ $(\neg(A))$)	??	??
FALSEimpliesPisTRUE	`false ->A ≡ true`	??	??
TRUEimpliesPisP	`true ->A ≡ A`	??	??
PimpliesFALSEisNotP	`A ->false ≡ not A`	??	??
PimpliesTRUEisTRUE	`A ->true ≡ true`	??	??
PandPisP	`A and A ≡ A`	??	??
PandTrueisP	`A and true ≡ A`	??	??
PandFalseisFalse	`A and false ≡ false`	??	??
PorPisP	`A or A ≡ A`	??	??
PorTrueisTrue	`A or true ≡ true`	??	??
PorFalseisP	`A or false ≡ A`	??	??

Table III 2.11: Associativity

reason	effect	mm	proof
and	(B and C)and A ≡ B and C and A	??	??
or	(B or C)or A ≡ B or C or A	??	??
xor	(B xor C)xor A ≡ B xor C xor A	??	??
+	(B+C)+A ≡ B+C+A	??	??
*	(B⋆C)⋆A ≡ B⋆C⋆A	??	??

Table III 2.12: Other Laws

reason	effect	mm	proof
MULTIPLY_BOTH_SIDES	$x \star y = x \star z \equiv y = z$ (for $x \neq 0$)	??	??
ADD_BOTH_SIDES	$x + y = x + z \equiv y = z$ $y - x = z - x \equiv y = z$	??	??
ADD_BOTH_SIDES_SUBTRACTION	$(x + y) - (x + z) \equiv y - z$	??	??

III 2.9 Substitute Menu

Substitution requires a formula and a pair of equal things; one of the equal things gets replaced by the other for each occurrence in the formula–usually exactly one.

Replacement of an assertion label with its predicate is a common substitution, with actual parameters substituted for formal parameters (if any). Given assertion `A` with formal parameters `f1`, `f2`, and `f3`, having predicate X:

```
<<A:f1 f2 f3:X>>
```

and some other predicate that uses `A` with actual parameters `a1`, `a2`, and `a3`:

```
<< . . . A(a1,a2,a3). . . >>
```

replace `A(a1,a2,a3)` with `X` after substituting `a1` for `f1`, `a2` for `f2`, and `a3` for `f3`:

$$A(a_1, a_2, a_3) \equiv X|^{a_1 a_2 a_3}_{f_1 f_2 f_3}$$

III 2.9.1 substitute assertion labels

Substitute occurrence of assertion labels by the assertion's predicate, in both pre- and post-conditions.

III 2.9.2 substitute assertions in preconditions

Substitute occurrence of assertion labels by the assertion's predicate, in preconditions.

III 2.9.3 substitute assertions in postconditions

Substitute occurrence of assertion labels by the assertion's predicate, in postconditions.

III 2.9.4 completely substitute

Substitute assertion labels, and normalize, until no assertion labels remain.

This command is very powerful, but should be used sparingly.

III 2.9.5 guided substitution of equals

Opens a dialog box for a hint to guide substitution. Looks through terms of precondition conjunction, finds an equality that matches the hint, then substitutes hint with its equal in postcondition.

Usually, substitute equals, or substitute all equals, performs the substitution needed. When they don't, guided substitution of equals will perform exactly the substitution desired.

III 2.9.6 substitute (an) equals

Look for equations that are terms in precondition conjunctions; choose the equation having the highest "score"; replace occurrences in its postcondition of an equal thing with the other.

For the first several proofs, the highest score equation was invariably the substitution needed to transform a proof obligation into an axiom. The scoring broke-down when the precondition was conjunction including four equations needing substitution. This motivated the following command.

III 2.9.7 substitute all equals

Substitute *all* equations that are terms in precondition conjunctions, in the postcondition.

III 2.9.8 substitute equals within conjunction

Look for conjunctions have equation terms; replace an equal thing with the other, in other terms of the conjunction. This can be handy to manipulate preconditions.

III 2.9.9 substitute adding negation for subtraction

Replace each subtraction with adding negation. Sometimes, arithmetic expressions including subtraction are hard to bash into normal form. This seems especially common in quantification bounds.

`x+(y-1)≠(x+y)-1`

But converting subtraction to adding negation makes

`x+(y+ -1)≠(x+y)+ -1`

then with associativity of addition

`x+y+ -1≤x+y+ -1.`

III 2.9.10 replace `a<>b` with `a<b or b<a`

Replace occurrences of inequality with disjunction of total ordering.

III 2.9.11 replace `a<>b` with `not a=b`

Replace occurrences of inequality with the complement of equality.

III 2.9.12 replace `x<=y` with `not y<x`

Replace occurrences of at most, with not less than. Handy for solving Serban's theorems.

III 2.9.13 replace `A->B` with `not A or B`

Replace implication with not premise or consequence.

Sets Global.replaceImplicationWithNotAorB then invokes Strategy.applyLaws(). Probably should write its own ANTLR tree-filter grammar rather than check (all) laws.

III 2.9.14 replace port names

Replace port names with their **`BLESS::Assertion`** properties.

III 2.9.15 range to expression

Change quantification ranges into boolean expressions. Currently only exists and all.

Replace `x:T in l..u` with `x:T in l<=x and x<=u`.
Replace `x:T in l,,u` with `x:T in l<x and x<u`.
Replace `x:T in l,.u` with `x:T in l<x and x<=u`.
Replace `x:T in l.,u` with `x:T in l<=x and x<u`.

III 2.10 Remove Menu

Sometimes you need some pesky fact like `j-1<j`. Because (almost) every assertion in the proof outline will be used for both pre- and postconditions, little axioms get copied everywhere, even though needed for a single proof obligation.

Instead, define a special axiom whose label begins with "axiom", stick it where you need it, and then remove it when unneeded. In your assertion annex library add **`<<`**`AXIOMxm1ltx:x:x-1<x`**`>>`**, then use it in some conjunction `AXIOMxm1ltx(j)` where you need your pesky fact.

Of course, this should axiom-assertions be used sparingly, and receive special, human, scrutiny to be sure what's claimed is truly an axiom.

III 2.10.1 remove axioms from precondition

Replace any Assertion labels used in a precondition, that begin with "axiom", with **`true`**.

III 2.10.2 remove axioms from postcondition

Replace any assertion labels used in a postcondition, that begin with "axiom", with **true**.

III 2.10.3 remove existential quantification

Change all occurrences of existential quantification to its predicate:

exists v:t **in** lb..ub **that** X becomes X

Use only when no quantified variable (v) occurs free in its predicate (X).

III 2.11 Split Menu

Splitting is a fundamental way of transforming proof obligations. Splitting postconditions makes simpler proof obligations. Splitting quantifications and timed atoms make proof obligations have more, but simpler terms.

Splitting can mess things up if done too soon. Use carefully and sparingly.

III 2.11.1 split postconditions

Find proof obligations having postconditions with **and** root; replace each such proof obligation with a set of proof obligations having the original precondition, and a postcondition that was a term child of the **and**.[4]

Make **<<**A**>>**→**<<**B **and** C**>>**
into **<<**A**>>**→**<<**B**>>** and **<<**A**>>**→**<<**C**>>**.

III 2.11.2 split quantifications

Split universal quantification of conjunction into conjunction of universal quantifications; split existential quantification of disjunction into disjunction of existential quantifications.

all x:T **in** R **are** (A **and** B)≡(**all** x:T **in** R **are** A)**and** (**all** x:T **in** R **are** B)
exists x:T **in** R **that** (A **or** B)≡(**exists** x:T **in** R **that** A)**or** (**exists** x:T **in** R **that** B)

III 2.11.3 split @

Distribute timed atoms over **and**, **or**

(A **and** B)@t ≡ A@t **and** B@t
(A **or** B)@t ≡ A@t **or** B@t

[4] This is the essence of sequent calculus: prove each term individually.

III 2.12 Timing Menu

Convert between discrete and continuous time.

III 2.12.1 @ to ^

Make `x@now` into `x^0`.

Make `x@t` into `x^((t-now)/period)`.

III 2.12.2 ^ to @

Make `x^0` into `x@now`.

Make `x^j` into `x@(now + j*period)`.

III 2.12.3 now

Assume evaluation at present instant.

Make `x@now` into `x`.

Make `x^0` into `x`.

III 2.13 Quantification Menu

Quantification rules are difficult to use.

III 2.13.1 quantification laws

Reduce obvious quantifications (Table III 2.13).

III 2.13.2 quantification timing

Extend or contract ranges of quantified formulas (Table III 2.14 and III 2.15). This makes "windows" of data like that used by the pulse-ox smart alarm averaging thread. Put the new value in the next record, and increment the upper- and lower-bounds of the invariant that says what your data means over time.[5]

For **sum**, we know the total over a range of integer, and wish to add the next element (F(ub+1):

```
total = (sum j:integer in lb..ub of F(j))+F(ub+1)
```

[5] like a Z-transform

Table III 2.13: Quantification Laws

reason	effect	mm	proof
CONSTANT_BODY_ALL_TRUE	all a:t in R are true ≡ true	??	??
CONSTANT_BODY_ALL_FALSE	all a:t in R are false ≡ false	??	??
CONSTANT_BODY_ EXISTS_TRUE	exists a:t in R that true ≡ true	??	??
CONSTANT_BODY_ EXISTS_FALSE	exists a:t in R that false ≡ false	??	??
CONSTANT_BODY_SUM_K	sum a:t in lb..ub of k ≡ k*(ub-lb)	??	??
CONSTANT_BODY_NUMBEROF	numberof a:t in lb..ub of true ≡ ((ub+1)-lb)	??	??
EMPTY_RANGE_ALL	all a:t in false are V ≡ true	??	??
EMPTY_RANGE_EXISTS	exists a:t in false that V ≡ false	??	??
EMPTY_RANGE_NUMBEROF	numberof a:t in false that V ≡ 0	??	??
EMPTY_RANGE_SUM	sum a:t in false of V ≡ 0	??	??
EXISTS_OPEN_LEFT	exists a:t in j,.j that V ≡ false	??	??
EXISTS_OPEN_RIGHT	exists a:t in j.,j that V ≡ false	??	??
EXISTS_OPEN_BOTH	exists a:t in j,,j that V ≡ false	??	??
QUANTIFIED_FETCH_ADD	all a:t in R are v+=e(a) ≡ v=v0 + sum a:t in R of e(a)	??	??
REMOVE_UNUSED_VARIABLES_ALL	all a:t in R are X ≡ X when a not in X	??	??
REMOVE_UNUSED_VARIABLES_EXISTS	exists a:t in R that X ≡ X when a not in X	??	??
SOLITARY_RANGE_ALL	all a:t in j..j are V ≡ V[j/a]	??	??
SOLITARY_RANGE_SUM	sum a:t in j..j of V ≡ V[j/a]	??	??
SOLITARY_RANGE_EXISTS	exists a:t in j..j that V ≡ V[j/a]	??	??
SOLITARY_RANGE_NUMBEROF	numberof a:t in j..j that true ≡ 1	??	??

→

```
total = (sum j:integer in lb..ub+1 of F(j))
```

However, in the proof, which is backwards, will "contract" summation instead. Two pairs of rules support adding or subtracting elements to either end of the range.

For **numberof**, we know the count over a range of integer, and wish to increment the count if our condition (B(ub+1) is true:

```
count = (numberof j:integer in lb..ub that B(j))+(B(ub+1)??1 :0)
```

→

```
 count = (numberof j:integer in lb..ub+1 that B(j))
```

Two pairs of rules support incrementing or decrementing elements to either end of the range.

For **all**, we know some property holds over a range of integer, and wish to state its holds for the next element as well:

```
(all j:integer in lb..ub are P(j))and P(ub+1)
```

→

```
all j:integer in lb..ub+1 are P(j))
```

Why not the other 3? (extend all, extend and contract exists)

There's so much to be said about applying ^ to quantifications over discrete time. When all uses of the quantified variable are timed atoms (`^j` or `^(-j)`), then applying `^k` shifts the range up or down (Table III 2.15).

III 2.13.3 replace quantified variables with #_#

Two formulas can be identical, except for identifiers of quantified variables. By replacing every quantified variable with a unique identifier, exclusive to quantified variables, then formulas identical except for choice of of q.v. identifiers can be recognized as the same.

Replacement identifiers are a natural number (base 10) surrounded by #: `#5#`. Tests of equality keep a scratch pad of quantified variable matching, so that if `#5#` matched with `#13#` at one place in the formula, they must match at all places.

```
all #5#:integer in lb..ub are P(#5#)
```

↔

```
all #13#:integer in lb..ub are P(#13#)
```

III 2.13.4 extend exists range

First inequalities in a precondition are found, and then ranges of existential quantifications in both pre- and postcondition are extended in an inequality applies. Don't use this except when you deliberately need to expand some existential quantification.

The inequality relations `< <= > >=` are only recognized if at top-level conjunction of precondition:

```
<<(m<lb)and L(now)and not (n<=ub)and . . . >>->><<Q>>
```

Complements of inequalities are also loaded.

Table III 2.14: Extend and Contract Quantification Range

reason	effect	mm	proof
EXTEND_SUMMATION	when `P(ub)=term` `(sum j:integer in lb..ub of P(j))-term` ≡ `(sum j:integer in lb..ub-1 of P(j))` when `P(lb)=term` `(sum j:integer in lb..ub of P(j))-term` ≡ `(sum j:integer in lb+1..ub of P(j))`	??	??
CONTRACT_SUMMATION	when `P(ub+1)=term` `(sum j:integer in lb..ub of P(j))+term` ≡ `(sum j:integer in lb..ub+1 of P(j))` when `P(lb-1)=term` `(sum j:integer in lb..ub of P(j))+term` ≡ `(sum j:integer in lb-1..ub of P(j))`	??	??

EXTEND_NUMBEROF	when `P(ub)=term`
	`(numberof j:integer in lb..ub that P(j))`
	`-(term??1:0)`
	≡ `(numberof j:integer in lb..ub-1 that P(j))`
	when `P(lb)=term`
	`(numberof j:integer in lb..ub that P(j))`
	`-(term??1:0)`
	≡ `(numberof j:integer in lb+1..ub that P(j))`
		??	??

CONTRACT_NUMBEROF	when `P(ub+1)=term`
	`(numberof j:integer in lb..ub that P(j))`
	`-(term??1:0)`
	≡ `(numberof j:integer in lb..ub+1 that P(j))`
	when `P(lb-1)=term`
	`(numberof j:integer in lb..ub that P(j))`
	`-(term??1:0)`
	≡ `(numberof j:integer in lb-1..ub that P(j))`
		??	??

CONTRACT_UNIVERAL _QUANTIFICATION	when `P(ub+1)=term` `(all j:integer in lb..ub are P(j))` ≡ `(all j:integer in lb..ub+1 are P(j))` when `P(lb-1)=term` `(all j:integer in lb..ub are P(j))` ≡ `(all j:integer in lb-1..ub are P(j))`	??	??

Table III 2.15: Quantification Shift

reason	effect	mm	proof
SHIFT_QUANT_IN_TIME	when all uses of quantified variable `j` in `P(j)` are `^(-j)` then shifts in discrete time of universal quantification `(all j:integer in lb..ub are P(j))^k` is the same as changing range bounds `≡all j:integer in lb-k..ub-k are P(j)` shifts in discrete time of existential quantification `(exists j:integer in lb..ub that P(j))^k` is the same as changing range bounds `≡exists j:integer in lb-k..ub-k that P(j)`	??	??
MEMBER_OF_ALL	`all j:integer in lb..ub are P(j) → P(lb)` `all j:integer in lb..ub are P(j) → P(ub)`	??	??

Then all existential quantifications of the form:
`exists k:integer in lb..ub that P(k)` have expressions for `lb` and `ub` compared with the inequality relations. If something is true in a range, it's also true in every range that contains it. If `m<lb` or `m<=lb` then replace

```
exists k:integer in lb..ub that P(k)
```
with `exists k:integer in m..ub that P(k).`

Similarly, if `n<ub` or `n<=ub` then replace

```
exists k:integer in lb..ub that P(k)
```
with `exists k:integer in lb..n that P(k).`

III 2.13.5 contract all range

Same as III 2.13.4 except, universal quantification range bounds are compared with the inequalities to make the range smaller. Same caution about indiscriminate use.

If `lb<m` or `lb<=m` then replace

```
all k:integer in lb..ub are P(k)
```
with `all k:integer in m..ub are P(k).`

Similarly, if `ub<n` or `ub<=n` then replace

```
all k:integer in lb..ub are P(k)
```
with `all k:integer in lb..n are P(k).`

III 2.13.6 shift lower bound to 0

Sometimes quantifications may be the same, although have different ranges. Shifting lower bounds to 0 can make them normal form, and then recognized as being the same.

```
all k:integer in lb..ub are P(k) ≠ all k:integer in lb+1..ub+1 are P(k-1).
```

Every quantification is equivalent is equivalent to a quantification of the same length, with lower-bound of 0.

```
    all k:integer in lb..ub are P(k)
≡ all k:integer in 0..ub-lb are P(k+lb).
```

Therefore

```
    all k:integer in lb+1..ub+1 are P(k-1)
≡ all k:integer in 0..(ub+1)-(lb+1)are P((k-1)+(lb+1))
    ≡ all k:integer in 0..ub-lb are P(k+lb)
      ≡ all k:integer in lb..ub are P(k).
```

III 2.13.7 counting rules

Change counting quantifier equal to 1 to single occurrence.

```
p@t ≡ 1 = numberof u in t..t that p@u
```

III 2.14 Distribute Menu

The Distribute Menu has proof rules to distribute.[6]

Two distributions are used in different combinations:

and-over-or: (make into conjunction of disjunctions)

```
(a and b)or (c and d) ≡ (a or c)and (a or d)and (b or c)and (b or d)
```

or-over-and: (make into disjunction of conjunctions)

```
(a or b)and x and z ≡ (a and x and z)or (b and x and z)
```

III 2.14.1 completely distribute time

Repeatedly invoke §III 2.14.2 until no changes occur.

[6] It's really hard to express what distribute does, replacing a higher-level something with application of it to its children, or something like that.

III 2.14.2 distribute time

Move application of temporal operator (`@ ^`) to contents of parentheses.

Replace `(P)^k` with `^k` applied to each term in `P`.

Where `op` is one of
{`and or xor iff + * - / = <> < > <= >=`}:

```
(A op B)^k ≡ (A^k op B^k)
```

Negation and complement:

```
(not A)^k ≡ not (A)^k
   (-x)^k ≡ -(x)^k
```

Quantifiers:

```
    (all x:T in R are P)^k ≡ (all x:T in R are (P )^k )
(exists x:T in R that P)^k ≡ (exists x:T in R that (P )^k )
     (sum x:T in R of P)^k ≡ (sum x:T in R of (P )^k )
 (product x:T in R of P)^k ≡ (product x:T in R of (P )^k )
(numberof x:T in R that P)^k ≡ (numberof x:T in R that (P )^k )
```

Conditional:

```
(b??y:z)^k ≡ (b^k ??y^k:z^k)
```

Now:

```
P^0 ≡ P
```

Double time shift:

```
(P^a)^b ≡ P^(a+b)
```

Array:[7]

```
  A[n]^b ≡ A[n^b]
A[n].r^b ≡ A[n^b].r
```

Tick (next value):

```
P' ≡ P^(1)
```

Replace `(x)@t` with `@t` applied to each term in `x`:

Where `op` is one of
{`and or xor iff + * - / = <> < > <= >=`}:

```
(A op B)@t ≡ (A@t op B@t)
```

Negation and complement:

[7] Must only apply when the array is invariant w.r.t. time.

```
(not A)@t ≡ not (A)@t
(-x)@t ≡ -(x)@t
```

Quantifiers:

```
(all x:T in R are P)@t ≡ (all x:T in R are (P )@t )
(exists x:T in R that P)@t ≡ (exists x:T in R that (P )@t )
(sum x:T in R of P)@t ≡ (sum x:T in R of (P )@t )
(product x:T in R of P)@t ≡ (product x:T in R of (P )@t )
(numberof x:T in R that P)@t ≡ (numberof x:T in R that (P )@t )
```

Conditional:

```
(b??y:z)@t ≡ (b@t ??y@t:z@t)
```

Now:

```
P@now ≡ P
```

III 2.14.3 DeMorgan's Law

Push knots toward leaves.

Table III 2.16: DeMorgan's Laws

reason	effect	mm	proof
DEMORGANS_LAW_ NOT_AND_IS_OR_NOT	not (A and B and C) ≡ (not A or not B or not C)	??	??
DEMORGANS_LAW_ NOT_OR_IS_AND_NOT	not (A or B or C) ≡ (not A and not B and not C)	??	??
DEMORGANS_LAW_ NOT_EXISTS_IS_ALL_NOT	not (exists x:T in R that P) ≡ (all x:T in R are not(P))	??	??
DEMORGANS_LAW_ NOT_ALL_IS_EXISTS_NOT	not (all x:T in R are P) ≡ (exists x:T in R that not(P))	??	??
DOUBLE_NEGATION	not(not A) ≡ not not A ≡ A	??	??

III 2.14.4 conjunctive normal form

Conjunctive normal form tries a sequence of proof rules that attempts to flatten preconditions and postconditions into conjunctions of disjunctions.

1. replace `A->B` with `not A or B` (set flag; apply laws)
2. DeMorgan's Law
3. normalize
4. distribute and-over-or for both pre- and postcondition

III 2.14.5 and-over-or

Distribute and-over-or for precondition; or-over-and for postcondition.

III 2.14.6 or-over-and

Distribute and-over-or for postcondition; or-over-and for precondition. Also, splits existential quantification and timed atoms:[8]

```
exists x:T in R that (A or B) ≡ (exists x:T in R that A)or (exists x:t in R that B)
                                (A or B)@t ≡ (A)@t or (B)@t
```

III 2.14.7 and-over-or precondition

Distribute and-over-or for precondition.

III 2.14.8 and-over-or postcondition

Distribute and-over-or for postcondition.

III 2.14.9 or-over-and precondition

Distribute or-over-and for precondition.

III 2.14.10 or-over-and postcondition

Distribute or-over-and for postcondition.

III 2.15 Special Menu

These commands don't fit in other categories.

III 2.15.1 add equivalent terms

Add equivalent terms to precondition by substitution of equalities.

Look for the first equality in a precondition's conjunction, add for all other terms, check if substituting one equal thing for the other makes and difference. If so, add the new term to the conjunction.

[8] wonder why and-over-or doesn't split `all` and `(A and B)@t`

```
<<x=y and G(x)>> → <<Q>> ≡ <<G(x)and G(y)and x=y>> → <<Q>>
```

The equal term used for substitution is put back on the end of the conjunction, so the operation can be performed successively with several equal terms.

III 2.15.2 conditional expression (b??t:f)

When B then `(B??x:y) ≡ (x)`
When not B then `(B??x:y) ≡ (y)`

III 2.15.3 add transitive relations

For preconditions, apply transitivity to `<=` and `<`.

```
<<x<y and y<z>> → <<Q>> ≡ <<x<y and y<z and x<z>> → <<Q>>
<<x<=y and y<z>> → <<Q>> ≡ <<x<=y and y<z and x<z>> → <<Q>>
<<x<y and y<=z>> → <<Q>> ≡ <<x<y and y<=z and x<z>> → <<Q>>
<<x<=y and y<=z>> → <<Q>> ≡ <<x<=y and y<=z and x<=z>> → <<Q>>
```

III 2.15.4 <= to <

Make partial orders into total orders. Remember we're reasoning backwards. In the resultant proof, less-than implies at-most.

$x < y \implies x <= y$

III 2.15.5 apply conditional function

A conditional assertion function (§I 5.4.5) is like a conditional expression, except it has more than two alternatives.

Suppose we have an equality relation such as

```
e = ((P1)->e1,(P2)->e2,(P3)->e3)
```

If `e` is the same as one of the expressions in the conditional assertion function, then the corresponding predicate must be true. Suppose `e=e2`, then the whole thing can be replaced by `P2`.

```
P2 ≡ e2 = ((P1)->e1,(P2)->e2,(P3)->e3)
```

Be sure the predicates are disjoint (exactly one may be true).

Conditional assertion functions were added because the flow rate of the Open PCA Pump had different conditions for multiple rates (off, KVO, bolus, square bolus, prime, flush).

Part IV

BLESS Soundness Proof

A deduction is *sound* when it derives true facts from true facts. BLESS behaviors (programs) are deliberately constructed such that a sound deduction can conclude a behavior meets its specification, in its entire state space. A *soundness proof* proves that each step in the deduction is sound, and by induction over the length of a proof, that the proof itself is sound.[1]

A logic system is *sound* when only true theorems can be proved from true theorems. If the logic is inconsistent or unsound, then its proofs are suspect, if not worthless.

To prove soundness for BLESS induction over all proof rules used, that each proof rule is sound, therefore proofs which are a sequence of sound rules will also be sound. Each of the axioms or inference rules used by BLESS will have its own soundness proof.

[1] I§**??**

Chapter IV 1

Metamath Proof System

IV 1.1 Metamath Preliminaries

Metamath[1] will be used to prove as theorems in set theory, what are assumed by the BLESS proof tool as axioms and inference rules. Metamath is a

- language to express mathematics, thus "meta-"
- tool suite; especially Metamath Proof Explorer (checker)
- library of over 10,000 proofs, individually named and consecutively numbered, starting from set theory axioms

The "meta-" in Metamath means its subject is math. Metamath does math about math.

Metamath allows you to define your own mathematical system. Once defined you can prove interesting things about your mathematical system.

Metamath language allows expression of (other) mathematical systems, and prove theorems about those mathematical systems, which will in turn be used to prove theorems about the math you want to use. Here Metamath will used to prove soundness about the proof rules used by the BLESS proof engine. For BLESS, Metamath is used to prove that each proof rule is *sound*, either tautology or inference rule, always producing true facts from true facts.

IV 1.2 Metamath Symbols

Metamath symbols (Table IV 1.2) follow customary usage. Symbols introduced especially for BLESS (Table IV 1.3) are explained when their use is defined. Variable symbols (Table IV 1.1) have what are effectively

[1] http://us.metamath.org/index.html

types.

Most of the steps in a Metamath proof assure the syntax is correct. As part of this, variable symbols can be used in ways such that the syntax enforces types. The proofs pretty-printed in LaTeXomit steps that assure syntactical correctness.

Table IV 1.1: Metamath Variables

wff	well-formed formula declaration
$\varphi\ \psi\ \chi\ \theta\ \tau\ \eta\ \zeta$	label representing a wff
set	set variable declaration
$x\ y\ z\ w\ v\ u\ t$	label representing a set variable
class	class variable declaration
$A\ B\ C\ D$	label representing a class variable
wl	well-formed formula list declaration
$l_1\ l_2\ l_3\ l_4\ l_5$	label representing a well-formed formula list
csl	comma separated list declaration
$csl_1\ csl_2$	label representing a comma separated list
act	action declaration
$S\ S_1\ S_2\ S_3\ S_n$	label representing an action
assrt	assertion declaration
$P\ P_1\ P_2\ P_3\ P_n\ Q\ Q_1\ Q_2\ Q_3\ Q_n$	label representing an assertion
aa	asserted action declaration
$AA_1\ AA_2\ AA_3\ AA_n$	label representing an asserted action

Table IV 1.2: Metamath Symbols

$\vdash$	a proof exists for
$\Rightarrow$	given l.h.s, r.h.s can be proved
$\top$	true
$\bot$	false
$\rightarrow$	implies
$\leftrightarrow$	equivalence (if-and-only-if)
( )	precedence
$\vee\ \wedge\ \neg$	and or not
$=\ <\ \leq\ \neq$	equal less-than at-most unequal
$+\ -\ \times\ /$	add subtract multiply divide
$\forall\ \exists$	forall exists
$\in\ \notin$	element-of not-element-of

For example, the proof for bl.that (**??**) is represented in the bless.mm database as

```
$(
bl.that.1 $e |- ph $.
$( theorems are true $)
bl.that $p |- ( ph <-> true )
  $=
  $( wffs for bitr4i  ph= ph  ps= ( ph -> ph )  ch= true  $)
    wph wph wph wi wtrue
  $( wffs for a1bi ph= ph  ps= ph $)
```

Table IV 1.3: BLESS Symbols

$\bigwedge( )$	multi-term conjunction
$\bigvee( )$	multi-term disjunction
$\bar{[}\psi/\chi\bar{]}\varphi$	replace all ψ with χ in φ
$\ll \gg$	assertion delimiters
; &	sequential and concurrent composition
wp	weakest-precondition

```
    wph wph
  bl.that.1
  a1bi  $( |- ( ph <-> ( ph -> ph ) )  $)
  wph df-true $( |- ( true <-> ( ph -> ph ) ) $)
  $( by bitr4i |- ( ph <-> ps )  and  |- ( ch <-> ps )  =>  |- ( ph <-> ch )
    with ph= ph  ps= ( ph -> ph )  ch= true  $)
  bitr4i
  $( |- ( ph <-> true ) $)
  $.
$}
$( end of bl.that $)
```

Symbols `$(` and `$)` delimit comments; `${` and `$}` delimit scope.

IV 1.3 Metamath Proofs

Metamath proofs are inductive, consisting of a sequence of theorems, each of which is an axiom (or definition) or derived from earlier theorem(s) by sound inference rules.

ADD MUCH MORE ABOUT METAMATH HERE, INCLUDING ALL THE CORRECT GRAMMAR/WFF

IV 1.4 Metamath Theorems Used

Metamath Lemma 1. **3anass** *Associative law for triple conjunction. (Contributed by NM, 8-Apr-1994.)*

$$\vdash ((\varphi \wedge \psi \wedge \chi) \leftrightarrow (\varphi \wedge (\psi \wedge \chi))) \tag{IV 1.1}$$

Metamath Lemma 2. **3orass** *Associative law for triple disjunction. (Contributed by NM, 8-Apr-1994.)*

$$\vdash ((\varphi \vee \psi \vee \chi) \leftrightarrow (\varphi \vee (\psi \vee \chi))) \tag{IV 1.2}$$

Metamath Lemma 3. **3bitri** *A chained inference from transitive law for logical equivalence. (Contributed by NM, 5-Aug-1993.)*

$$\vdash (\varphi \leftrightarrow \psi) \quad \& \quad \vdash (\psi \leftrightarrow \chi) \quad \& \quad \vdash (\chi \leftrightarrow \theta) \quad \Rightarrow \quad \vdash (\varphi \leftrightarrow \theta) \tag{IV 1.3}$$

Metamath Lemma 4. **3imtr3i** *A mixed syllogism inference, useful for removing a definition from both sides of an implication. (Contributed by NM, 10-Aug-1994.)*

$$\vdash (\varphi \rightarrow \psi) \quad \& \quad \vdash (\varphi \leftrightarrow \chi) \quad \& \quad \vdash (\psi \leftrightarrow \theta) \quad \Rightarrow \quad \vdash (\chi \rightarrow \theta) \tag{IV 1.4}$$

Metamath Lemma 5. **3imtr4i** *A mixed syllogism inference, useful for applying a definition to both sides of an implication. (Contributed by NM, 5-Aug-1993.)*

$$\vdash (\varphi \rightarrow \psi) \quad \& \quad \vdash (\chi \leftrightarrow \varphi) \quad \& \quad \vdash (\theta \leftrightarrow \psi) \quad \Rightarrow \quad \vdash (\chi \rightarrow \theta) \tag{IV 1.5}$$

Metamath Lemma 6. **a1bi** *Inference rule introducing a theorem as an antecedent. (Contributed by NM, 5-Aug-1993.) (Proof shortened by Wolf Lammen, 11-Nov-2012.)*

$$\vdash \varphi \quad \Rightarrow \quad \vdash (\psi \leftrightarrow (\varphi \rightarrow \psi)) \tag{IV 1.6}$$

Metamath Lemma 7. **a1tru** *Anything implies* $\top$. *(Contributed by FL, 20-Mar-2011.) (Proof shortened by Anthony Hart, 1-Aug-2011.)*

$$\vdash (\varphi \rightarrow \top) \tag{IV 1.7}$$

Metamath Lemma 8. **anbi12i** *Conjoin both sides of two equivalences. (Contributed by NM, 5-Aug-1993.)*

$$\vdash (\varphi \leftrightarrow \psi) \quad \& \quad \vdash (\chi \leftrightarrow \theta) \quad \Rightarrow \quad \vdash ((\varphi \wedge \chi) \leftrightarrow (\psi \wedge \theta)) \tag{IV 1.8}$$

Metamath Lemma 9. **anbi1i** *Introduce a right conjunct to both sides of a logical equivalence. (Contributed by NM, 5-Aug-1993.) (Proof shortened by Wolf Lammen, 16-Nov-2013.)*

$$\vdash (\varphi \leftrightarrow \psi) \quad \Rightarrow \quad \vdash ((\varphi \wedge \chi) \leftrightarrow (\psi \wedge \chi)) \tag{IV 1.9}$$

Metamath Lemma 10. **anbi2i** *Introduce a left conjunct to both sides of a logical equivalence. (Contributed by NM, 5-Aug-1993.) (Proof shortened by Wolf Lammen, 16-Nov-2013.)*

$$\vdash (\varphi \leftrightarrow \psi) \quad \Rightarrow \quad \vdash ((\chi \wedge \varphi) \leftrightarrow (\chi \wedge \psi)) \tag{IV 1.10}$$

Metamath Lemma 11. **ancom** *Commutative law for conjunction. Theorem *4.3 of [WhiteheadRussell] p. 118. (Contributed by NM, 25-Jun-1998.) (Proof shortened by Wolf Lammen, 4-Nov-2012.)*

$$\vdash ((\varphi \wedge \psi) \leftrightarrow (\psi \wedge \varphi)) \tag{IV 1.11}$$

Metamath Lemma 12. **andir** *Distributive law for conjunction. (Contributed by NM, 12-Aug-1994.)*

$$\vdash (((\varphi \vee \psi) \wedge \chi) \leftrightarrow ((\varphi \wedge \chi) \vee (\psi \wedge \chi))) \tag{IV 1.12}$$

Metamath Lemma 13. **anidm** *Idempotent law for conjunction. (Contributed by NM, 5-Aug-1993.) (Proof shortened by Wolf Lammen, 14-Mar-2014.)*

$$\vdash ((\varphi \wedge \varphi) \leftrightarrow \varphi) \tag{IV 1.13}$$

Metamath Lemma 14. **ax-1** *Axiom* {Simp}. Axiom A1 of [Margaris] p. 49. One of the 3 axioms of propositional calculus. The 3 axioms are also given as Definition 2.1 of [Hamilton] p. 28. This axiom is called {*Simp*} *or "the principle of simplification" in* {Principia Mathematica} (Theorem *2.02 of [WhiteheadRussell] p. 100) because "it enables us to pass from the joint assertion of φ and ψ to the assertion of φ simply." (Contributed by NM, 5-Aug-1993.)

$$\vdash (\varphi \rightarrow (\psi \rightarrow \varphi)) \tag{IV 1.14}$$

Metamath Lemma 15. **ax-mp** *Rule of Modus Ponens. The postulated inference rule of propositional calculus. See e.g. Rule 1 of [Hamilton] p. 73. The rule says, "if φ is true, and φ implies ψ, then ψ must also be true." This rule is sometimes called "detachment," since it detaches the minor premise from the major premise. "Modus ponens" is short for "modus ponendo ponens," a Latin phrase that means "the mood that by affirming affirms" [Sanford] p. 39. This rule is similar to the rule of modus tollens* `mto`.

Note: In some web page displays such as the Statement List, the symbols "&" and "=}" informally indicate the relationship between the hypotheses and the assertion (conclusion), abbreviating the English words "and" and "implies." They are not part of the formal language. (Contributed by NM, 5-Aug-1993.)

$$\vdash \varphi \quad \& \quad \vdash (\varphi \rightarrow \psi) \quad \Rightarrow \quad \vdash \psi \tag{IV 1.15}$$

Metamath Lemma 16. **bicomi** *Inference from commutative law for logical equivalence. (Contributed by NM, 5-Aug-1993.)*

$$\vdash (\varphi \leftrightarrow \psi) \quad \Rightarrow \quad \vdash (\psi \leftrightarrow \varphi) \tag{IV 1.16}$$

Metamath Lemma 17. **biid** *Principle of identity for logical equivalence. Theorem *4.2 of [WhiteheadRussell] p. 117. (Contributed by NM, 5-Aug-1993.)*

$$\vdash (\varphi \leftrightarrow \varphi) \tag{IV 1.17}$$

Metamath Lemma 18. **biimpi** *Infer an implication from a logical equivalence. (Contributed by NM, 5-Aug-1993.)*

$$\vdash (\varphi \leftrightarrow \psi) \quad \Rightarrow \quad \vdash (\varphi \rightarrow \psi) \tag{IV 1.18}$$

Metamath Lemma 19. **biimpri** *Infer a converse implication from a logical equivalence. (Contributed by NM, 5-Aug-1993.) (Proof shortened by Wolf Lammen, 16-Sep-2013.)*

$$\vdash (\varphi \leftrightarrow \psi) \quad \Rightarrow \quad \vdash (\psi \rightarrow \varphi) \tag{IV 1.19}$$

Metamath Lemma 20. **bitr2i** *An inference from transitive law for logical equivalence. (Contributed by NM, 5-Aug-1993.)*

$$\vdash (\varphi \leftrightarrow \psi) \quad \& \quad \vdash (\psi \leftrightarrow \chi) \quad \Rightarrow \quad \vdash (\chi \leftrightarrow \varphi) \tag{IV 1.20}$$

Metamath Lemma 21. **bitr3** *Closed nested implication form of* `bitr3i`. *Derived automatically from* `bitr3VD`. *(Contributed by Alan Sare, 31-Dec-2011.) (Proof modification is discouraged.) (New usage is discouraged.)*

$$\vdash ((\varphi \leftrightarrow \psi) \rightarrow ((\varphi \leftrightarrow \chi) \rightarrow (\psi \leftrightarrow \chi))) \tag{IV 1.21}$$

Metamath Lemma 22. **bitr3i** *An inference from transitive law for logical equivalence. (Contributed by NM, 5-Aug-1993.)*

$$\vdash (\psi \leftrightarrow \varphi) \quad \& \quad \vdash (\psi \leftrightarrow \chi) \quad \Rightarrow \quad \vdash (\varphi \leftrightarrow \chi) \tag{IV 1.22}$$

Metamath Lemma 23. **bitr4i** *An inference from transitive law for logical equivalence. (Contributed by NM, 5-Aug-1993.)*

$$\vdash (\varphi \leftrightarrow \psi) \quad \& \quad \vdash (\chi \leftrightarrow \psi) \quad \Rightarrow \quad \vdash (\varphi \leftrightarrow \chi) \tag{IV 1.23}$$

Metamath Lemma 24. **bitri** *An inference from transitive law for logical equivalence. (Contributed by NM, 5-Aug-1993.) (Proof shortened by Wolf Lammen, 13-Oct-2012.)*

$$\vdash (\varphi \leftrightarrow \psi) \quad \& \quad \vdash (\psi \leftrightarrow \chi) \quad \Rightarrow \quad \vdash (\varphi \leftrightarrow \chi) \tag{IV 1.24}$$

Metamath Lemma 25. **con1bii** *A contraposition inference. (Contributed by NM, 5-Aug-1993.) (Proof shortened by Wolf Lammen, 13-Oct-2012.)*

$$\vdash (\neg\varphi \leftrightarrow \psi) \quad \Rightarrow \quad \vdash (\neg\psi \leftrightarrow \varphi) \tag{IV 1.25}$$

Metamath Lemma 26. **con4bii** *A contraposition inference. (Contributed by NM, 21-May-1994.)*

$$\vdash (\neg\varphi \leftrightarrow \neg\psi) \quad \Rightarrow \quad \vdash (\varphi \leftrightarrow \psi) \tag{IV 1.26}$$

Metamath Lemma 27. **df-an** *Define conjunction (logical 'and'). Definition of [Margaris] p. 49. When both the left and right operand are true, the result is true; when either is false, the result is false. For example, it is true that* $(2 = 2 \wedge 3 = 3)$. *After we define the constant true* $\top$ *(*`df-tru`*) and the constant false* $\bot$ *(*`df-fal`*), we will be able to prove these truth table values:* $((\top \wedge \top) \leftrightarrow \top)$ *(*`truantru`*),* $((\top \wedge \bot) \leftrightarrow \bot)$ *(*`truanfal`*),* $((\bot \wedge \top) \leftrightarrow \bot)$ *(*`falantru`*), and* $((\bot \wedge \bot) \leftrightarrow \bot)$ *(*`falanfal`*).*

Contrast with $\vee$ *(*`df-or`*),* $\rightarrow$ *(*`wi`*),* $\overline{\wedge}$ *(*`df-nan`*), and* $\veebar$ *(*`df-xor`*) . (Contributed by NM, 5-Aug-1993.)*

$$\vdash ((\varphi \wedge \psi) \leftrightarrow \neg(\varphi \rightarrow \neg\psi)) \tag{IV 1.27}$$

Metamath Lemma 28. **df-or** *Define disjunction (logical 'or'). Definition of [Margaris] p. 49. When the left operand, right operand, or both are true, the result is true; when both sides are false, the result is false. For example, it is true that* $(2 = 3 \vee 4 = 4)$ *(*`ex-or`*). After we define the constant true* $\top$ *(*`df-tru`*) and the constant false* $\bot$ *(*`df-fal`*), we will be able to prove these truth table values:* $((\top \vee \top) \leftrightarrow \top)$ *(*`truortru`*),* $((\top \vee \bot) \leftrightarrow \top)$ *(*`truorfal`*),* $((\bot \vee \top) \leftrightarrow \top)$ *(*`falortru`*), and* $((\bot \vee \bot) \leftrightarrow \bot)$ *(*`falorfal`*).*

This is our first use of the biconditional connective in a definition; we use the biconditional connective in place of the traditional "{=*def*=}*", which means the same thing, except that we can manipulate the biconditional connective directly in proofs rather than having to rely on an informal definition substitution rule. Note that if we mechanically substitute* $(\neg\varphi \rightarrow \psi)$ *for* $(\varphi \vee \psi)$*, we end up with an instance of previously proved theorem* `biid`*. This is the justification for the definition, along with the fact that it introduces a new symbol* $\vee$*. Contrast with* $\wedge$ *(*`df-an`*),* $\rightarrow$ *(*`wi`*),* $\overline{\wedge}$ *(*`df-nan`*), and* $\veebar$ *(*`df-xor`*) . (Contributed by NM, 5-Aug-1993.)*

$$\vdash ((\varphi \vee \psi) \leftrightarrow (\neg\varphi \rightarrow \psi)) \tag{IV 1.28}$$

Metamath Lemma 29. **df-tru** *Definition of* $\top$*, a tautology.* $\top$ *is a constant true. In this definition* `biid` *is used as an antecedent, however, any true wff, such as an axiom, can be used in its place. (Contributed by Anthony Hart, 13-Oct-2010.)*

$$\vdash (\top \leftrightarrow (\varphi \leftrightarrow \varphi)) \tag{IV 1.29}$$

Metamath Lemma 30. **eqid** *Law of identity (reflexivity of class equality). Theorem 6.4 of [Quine] p. 41.*

This law is thought to have originated with Aristotle ({Metaphysics}, Book VII, Part 17). (Thanks to Stefan Allan for this information.) (Contributed by NM, 5-Aug-1993.)

$$\vdash A = A \tag{IV 1.30}$$

Metamath Lemma 31. **exmid** *Law of excluded middle, also called the principle of* {tertium non datur}. Theorem *2.11 of [WhiteheadRussell] p. 101. It says that something is either true or not true; there are no in-between values of truth. This is an essential distinction of our classical logic and is not a theorem of intuitionistic logic. (Contributed by NM, 5-Aug-1993.)

$$\vdash (\varphi \vee \neg\varphi) \tag{IV 1.31}$$

Metamath Lemma 32. **falim** $\bot$ *implies anything. (Contributed by FL, 20-Mar-2011.) (Proof shortened by Anthony Hart, 1-Aug-2011.)*

$$\vdash (\bot \to \varphi) \tag{IV 1.32}$$

Metamath Lemma 33. **idd** *Principle of identity with antecedent. (Contributed by NM, 26-Nov-1995.)*

$$\vdash (\varphi \to (\psi \to \psi)) \tag{IV 1.33}$$

Metamath Lemma 34. **imim2i** *Inference adding common antecedents in an implication. (Contributed by NM, 5-Aug-1993.)*

$$\vdash (\varphi \to \psi) \quad \Rightarrow \quad \vdash ((\chi \to \varphi) \to (\chi \to \psi)) \tag{IV 1.34}$$

Metamath Lemma 35. **imim2i** *Inference adding common antecedents in an implication. (Contributed by NM, 5-Aug-1993.)*

$$\vdash (\varphi \to \psi) \quad \Rightarrow \quad \vdash ((\chi \to \varphi) \to (\chi \to \psi)) \tag{IV 1.35}$$

Metamath Lemma 36. **impbii** *Infer an equivalence from an implication and its converse. (Contributed by NM, 5-Aug-1993.)*

$$\vdash (\varphi \to \psi) \quad \& \quad \vdash (\psi \to \varphi) \quad \Rightarrow \quad \vdash (\varphi \leftrightarrow \psi) \tag{IV 1.36}$$

Metamath Lemma 37. **leid** *'Less than or equal to' is reflexive. (Contributed by NM, 18-Aug-1999.)*

$$\vdash (A \in \mathbb{R} \to A \leq A) \tag{IV 1.37}$$

Metamath Lemma 38. **notbii** *Negate both sides of a logical equivalence. (Contributed by NM, 5-Aug-1993.) (Proof shortened by Wolf Lammen, 19-May-2013.)*

$$\vdash (\varphi \leftrightarrow \psi) \quad \Rightarrow \quad \vdash (\neg\varphi \leftrightarrow \neg\psi) \tag{IV 1.38}$$

Metamath Lemma 39. **notnot** *Double negation. Theorem *4.13 of [WhiteheadRussell] p. 117. (Contributed by NM, 5-Aug-1993.)*

$$\vdash (\varphi \leftrightarrow \neg\neg\varphi) \tag{IV 1.39}$$

Metamath Lemma 40. **olc** *Introduction of a disjunct. Axiom *1.3 of [WhiteheadRussell] p. 96. (Contributed by NM, 30-Aug-1993.)*

$$\vdash (\varphi \to (\psi \vee \varphi)) \tag{IV 1.40}$$

Metamath Lemma 41. **orbi12i** *Infer the disjunction of two equivalences. (Contributed by NM, 5-Aug-1993.)*

$$\vdash (\varphi \leftrightarrow \psi) \quad \& \quad \vdash (\chi \leftrightarrow \theta) \quad \Rightarrow \quad \vdash ((\varphi \vee \chi) \leftrightarrow (\psi \vee \theta)) \tag{IV 1.41}$$

Metamath Lemma 42. **orbi1i** *Inference adding a right disjunct to both sides of a logical equivalence. (Contributed by NM, 5-Aug-1993.)*

$$\vdash (\varphi \leftrightarrow \psi) \quad \Rightarrow \quad \vdash ((\varphi \vee \chi) \leftrightarrow (\psi \vee \chi)) \tag{IV 1.42}$$

Metamath Lemma 43. **orbi2i** *Inference adding a left disjunct to both sides of a logical equivalence. (Contributed by NM, 5-Aug-1993.) (Proof shortened by Wolf Lammen, 12-Dec-2012.)*

$$\vdash (\varphi \leftrightarrow \psi) \quad \Rightarrow \quad \vdash ((\chi \vee \varphi) \leftrightarrow (\chi \vee \psi)) \tag{IV 1.43}$$

Metamath Lemma 44. **orc** *Introduction of a disjunct. Theorem *2.2 of [WhiteheadRussell] p. 104. (Contributed by NM, 30-Aug-1993.)*

$$\vdash (\varphi \rightarrow (\varphi \vee \psi)) \tag{IV 1.44}$$

Metamath Lemma 45. **orcom** *Commutative law for disjunction. Theorem *4.31 of [WhiteheadRussell] p. 118. (Contributed by NM, 5-Aug-1993.) (Proof shortened by Wolf Lammen, 15-Nov-2012.)*

$$\vdash ((\varphi \vee \psi) \leftrightarrow (\psi \vee \varphi)) \tag{IV 1.45}$$

Metamath Lemma 46. **ordir** *Distributive law for disjunction. (Contributed by NM, 12-Aug-1994.)*

$$\vdash (((\varphi \wedge \psi) \vee \chi) \leftrightarrow ((\varphi \vee \chi) \wedge (\psi \vee \chi))) \tag{IV 1.46}$$

Metamath Lemma 47. **oridm** *Idempotent law for disjunction. Theorem *4.25 of [WhiteheadRussell] p. 117. (Contributed by NM, 5-Aug-1993.) (Proof shortened by Andrew Salmon, 16-Apr-2011.) (Proof shortened by Wolf Lammen, 10-Mar-2013.)*

$$\vdash ((\varphi \vee \varphi) \leftrightarrow \varphi) \tag{IV 1.47}$$

Metamath Lemma 48. **pm2.61** *Theorem *2.61 of [WhiteheadRussell] p. 107. Useful for eliminating an antecedent. (Contributed by NM, 5-Aug-1993.) (Proof shortened by Wolf Lammen, 22-Sep-2013.)*

$$\vdash ((\varphi \rightarrow \psi) \rightarrow ((\neg\varphi \rightarrow \psi) \rightarrow \psi)) \tag{IV 1.48}$$

Metamath Lemma 49. **pm2.86i** *Inference based on* `pm2.86`. *(Contributed by NM, 5-Aug-1993.) (Proof shortened by Wolf Lammen, 3-Apr-2013.)*

$$\vdash ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi)) \quad \Rightarrow \quad \vdash (\varphi \rightarrow (\psi \rightarrow \chi)) \tag{IV 1.49}$$

Metamath Lemma 50. **pm4.44** *Theorem *4.44 of [WhiteheadRussell] p. 119. (Contributed by NM, 3-Jan-2005.)*

$$\vdash (\varphi \leftrightarrow (\varphi \vee (\varphi \wedge \psi))) \tag{IV 1.50}$$

Metamath Lemma 51. **pm4.45** *Theorem *4.45 of [WhiteheadRussell] p. 119. (Contributed by NM, 3-Jan-2005.)*

$$\vdash (\varphi \leftrightarrow (\varphi \wedge (\varphi \vee \psi))) \tag{IV 1.51}$$

Metamath Lemma 52. **pm4.45im** *Conjunction with implication. Compare Theorem *4.45 of [WhiteheadRussell] p. 119. (Contributed by NM, 17-May-1998.)*

$$\vdash (\varphi \leftrightarrow (\varphi \wedge (\psi \rightarrow \varphi))) \tag{IV 1.52}$$

Metamath Lemma 53. **pm4.56** *Theorem *4.56 of [WhiteheadRussell] p. 120. (Contributed by NM, 3-Jan-2005.)*

$$\vdash ((\neg\varphi \wedge \neg\psi) \leftrightarrow \neg(\varphi \vee \psi)) \tag{IV 1.53}$$

Metamath Lemma 54. **simpl** *Elimination of a conjunct. Theorem *3.26 (Simp) of [WhiteheadRussell] p. 112. (Contributed by NM, 5-Aug-1993.) (Proof shortened by Wolf Lammen, 13-Nov-2012.)*

$$\vdash ((\varphi \wedge \psi) \rightarrow \varphi) \tag{IV 1.54}$$

Metamath Lemma 55. **syl** *An inference version of the transitive laws for implication* `imim2` *and* `imim1`*, which Russell and Whitehead call "the principle of the syllogism...because...the syllogism in Barbara is derived from them" (quote after Theorem *2.06 of [WhiteheadRussell] p. 101). Some authors call this law a "hypothetical syllogism."*

(A bit of trivia: this is the most commonly referenced assertion in our database. In second place is `eqid`*, followed by* `syl2anc`*,* `adantr`*,* `syl3anc`*, and* `ax-mp`*. The Metamath program command 'show usage' shows the number of references.) (Contributed by NM, 5-Aug-1993.) (Proof shortened by O'Cat, 20-Oct-2011.) (Proof shortened by Wolf Lammen, 26-Jul-2012.)*

$$\vdash (\varphi \rightarrow \psi) \quad \& \quad \vdash (\psi \rightarrow \chi) \quad \Rightarrow \quad \vdash (\varphi \rightarrow \chi) \tag{IV 1.55}$$

Metamath Lemma 56. **sylbi** *A mixed syllogism inference from a biconditional and an implication. Useful for substituting an antecedent with a definition. (Contributed by NM, 5-Aug-1993.)*

$$\vdash (\varphi \leftrightarrow \psi) \quad \& \quad \vdash (\psi \rightarrow \chi) \quad \Rightarrow \quad \vdash (\varphi \rightarrow \chi) \tag{IV 1.56}$$

Metamath Lemma 57. **sylibr** *A mixed syllogism inference from an implication and a biconditional. Useful for substituting a consequent with a definition. (Contributed by NM, 5-Aug-1993.)*

$$\vdash (\varphi \rightarrow \psi) \quad \& \quad \vdash (\chi \leftrightarrow \psi) \quad \Rightarrow \quad \vdash (\varphi \rightarrow \chi) \tag{IV 1.57}$$

Metamath Lemma 58. **trant** *A true antecedent can be removed. (Contributed by FL, 16-Apr-2012.)*

$$\vdash ((\top \rightarrow \varphi) \leftrightarrow \varphi) \tag{IV 1.58}$$

Chapter IV 2

Metamath Lemmas Created for BLESS

One of the powerful features of Metamath is the ability make definitions to suit your subject. The following subsections hold Metamath definitions, theorems, and their proofs, later used in the soundness proofs for BLESS proof rules.

IV 2.1 True $\top$ and False $\bot$

Metamath Lemma 59. **wtrue** *true is a well-formed formula*

wff $\top$ (IV 2.1)

Metamath Lemma 60. **wfalse** *false is a well-formed formula*

wff $\bot$ (IV 2.2)

Metamath Lemma 61. **df-true** *define true*

$\vdash (\top \leftrightarrow (\varphi \rightarrow \varphi))$ (IV 2.3)

Metamath Lemma 62. **df-false** *define false*

$\vdash (\bot \leftrightarrow \neg\top)$ (IV 2.4)

Metamath Lemma 63. **bl.that** *theorems are true*

$\vdash \varphi \quad \Rightarrow \quad \vdash (\varphi \leftrightarrow \top)$ (IV 2.5)

Proof.

1	$\vdash \varphi$	Hyp that.1
2	$\vdash (\varphi \leftrightarrow (\varphi \rightarrow \varphi))$	1, (a1bi *IV*1.6)
3	$\vdash (\top \leftrightarrow (\varphi \rightarrow \varphi))$	(df-true *IV*2.3)
4	$\vdash (\varphi \leftrightarrow \top)$	2, 3, (bitr4i *IV*1.23)

□

Metamath Lemma 64. **bl.cthaf** *complemented theorems are false*

$\vdash \neg\varphi \quad \Rightarrow \quad \vdash (\varphi \leftrightarrow \bot)$ (IV 2.6)

Proof.

1	$\vdash (\bot \leftrightarrow \neg\top)$	(df-false *IV*2.4)
2	$\vdash \neg\varphi$	Hyp cthaf.1
3	$\vdash (\neg\varphi \leftrightarrow \top)$	2, (bl.that *IV*2.5)
4	$\vdash (\neg\top \leftrightarrow \varphi)$	3, (con1bii *IV*1.25)
5	$\vdash (\varphi \leftrightarrow \bot)$	1, 4, (bitr2i *IV*1.20)

□

Metamath Lemma 65. **bl.antrr** *conjunction with true on right*

$\vdash ((\varphi \wedge \top) \leftrightarrow \varphi)$ (IV 2.7)

Proof.

1	$\vdash (\top \leftrightarrow (\varphi \rightarrow \varphi))$	(df-true *IV*2.3)
2	$\vdash ((\varphi \wedge \top) \leftrightarrow (\varphi \wedge (\varphi \rightarrow \varphi)))$	1, (anbi2i *IV*1.10)
3	$\vdash (\varphi \leftrightarrow (\varphi \wedge (\varphi \rightarrow \varphi)))$	(pm4.45im *IV*1.52)
4	$\vdash ((\varphi \wedge \top) \leftrightarrow \varphi)$	2, 3, (bitr4i *IV*1.23)

□

Metamath Lemma 66. **bl.antrl** *conjunction with true on left*

$\vdash ((\top \wedge \varphi) \leftrightarrow \varphi)$ (IV 2.8)

Proof.

1	$\vdash ((\top \wedge \varphi) \leftrightarrow (\varphi \wedge \top))$	(ancom *IV*1.11)
2	$\vdash ((\varphi \wedge \top) \leftrightarrow \varphi)$	(bl.antrr *IV*2.7)
3	$\vdash ((\top \wedge \varphi) \leftrightarrow \varphi)$	1, 2, (bitri *IV*1.24)

□

Metamath Lemma 67. **bl.anfar** *conjunction with false on right*

$\vdash ((\varphi \wedge \bot) \leftrightarrow \bot)$ (IV 2.9)

Proof.

1	$\vdash ((\varphi \wedge \bot) \leftrightarrow \neg(\varphi \rightarrow \neg\bot))$	(df-an *IV*1.27)
2	$\vdash (\neg(\varphi \wedge \bot) \leftrightarrow \neg\neg(\varphi \rightarrow \neg\bot))$	1, (notbii *IV*1.38)
3	$\vdash ((\varphi \rightarrow \neg\bot) \leftrightarrow \neg\neg(\varphi \rightarrow \neg\bot))$	(notnot *IV*1.39)
4	$\vdash (\neg(\varphi \wedge \bot) \leftrightarrow (\varphi \rightarrow \neg\bot))$	2, 3, (bitr4i *IV*1.23)
5	$\vdash (\varphi \rightarrow (\varphi \rightarrow \varphi))$	(ax-1 *IV*1.14)
6	$\vdash (\top \leftrightarrow (\varphi \rightarrow \varphi))$	(df-true *IV*2.3)
7	$\vdash (\varphi \rightarrow \top)$	5, 6, (sylibr *IV*1.57)
8	$\vdash (\bot \leftrightarrow \neg\top)$	(df-false *IV*2.4)
9	$\vdash (\neg\bot \leftrightarrow \neg\neg\top)$	8, (notbii *IV*1.38)
10	$\vdash (\top \leftrightarrow \neg\neg\top)$	(notnot *IV*1.39)
11	$\vdash (\neg\bot \leftrightarrow \top)$	9, 10, (bitr4i *IV*1.23)
12	$\vdash (\top \rightarrow \neg\bot)$	11, (biimpri *IV*1.19)
13	$\vdash ((\varphi \rightarrow \top) \rightarrow (\varphi \rightarrow \neg\bot))$	12, (imim2i *IV*1.35)
14	$\vdash (\varphi \rightarrow \neg\bot)$	7, 13, (ax-mp *IV*1.15)
15	$\vdash ((\varphi \rightarrow \neg\bot) \leftrightarrow \top)$	14, (bl.that *IV*2.5)
16	$\vdash (\neg(\varphi \wedge \bot) \leftrightarrow \top)$	4, 15, (bitri *IV*1.24)
17	$\vdash (\bot \leftrightarrow \neg\top)$	(df-false *IV*2.4)
18	$\vdash (\neg\bot \leftrightarrow \neg\neg\top)$	17, (notbii *IV*1.38)
19	$\vdash (\top \leftrightarrow \neg\neg\top)$	(notnot *IV*1.39)
20	$\vdash (\neg\bot \leftrightarrow \top)$	18, 19, (bitr4i *IV*1.23)
21	$\vdash (\neg(\varphi \wedge \bot) \leftrightarrow \neg\bot)$	16, 20, (bitr4i *IV*1.23)
22	$\vdash ((\varphi \wedge \bot) \leftrightarrow \bot)$	21, (con4bii *IV*1.26)

□

Metamath Lemma 68. **bl.anfal** *conjunction with false on left*

$\vdash ((\bot \wedge \varphi) \leftrightarrow \bot)$ (IV 2.10)

Proof.

1	$\vdash ((\bot \wedge \varphi) \leftrightarrow (\varphi \wedge \bot))$	(ancom *IV*1.11)
2	$\vdash ((\varphi \wedge \bot) \leftrightarrow \bot)$	(bl.anfar *IV*2.9)
3	$\vdash ((\bot \wedge \varphi) \leftrightarrow \bot)$	1, 2, (bitri *IV*1.24)

□

Metamath Lemma 69. **bl.ortrr** *disjunction with true on right*

$\vdash ((\varphi \vee \top) \leftrightarrow \top)$ (IV 2.11)

Proof.

1	$\vdash ((\varphi \vee \top) \leftrightarrow (\neg\varphi \rightarrow \top))$	(df-or *IV*1.28)
2	$\vdash ((\varphi \vee \top) \rightarrow (\neg\varphi \rightarrow \top))$	1, (biimpi *IV*1.18)
3	$\vdash (\varphi \rightarrow \top)$	(a1tru *IV*1.7)
4	$\vdash ((\varphi \leftrightarrow \varphi) \rightarrow (\varphi \rightarrow \varphi))$	(idd *IV*1.33)
5	$\vdash (\top \leftrightarrow (\varphi \leftrightarrow \varphi))$	(df-tru *IV*1.29)
6	$\vdash (\top \leftrightarrow (\varphi \rightarrow \varphi))$	(df-true *IV*2.3)
7	$\vdash (\top \rightarrow \top)$	4, 5, 6, (3imtr4i *IV*1.5)
8	$\vdash (\varphi \rightarrow \top)$	3, 7, (syl *IV*1.55)
9	$\vdash ((\varphi \rightarrow \top) \rightarrow ((\neg\varphi \rightarrow \top) \rightarrow \top))$	(pm2.61 *IV*1.48)
10	$\vdash ((\neg\varphi \rightarrow \top) \rightarrow \top)$	8, 9, (ax-mp *IV*1.15)
11	$\vdash ((\varphi \vee \top) \rightarrow \top)$	2, 10, (syl *IV*1.55)
12	$\vdash (\top \rightarrow (\varphi \vee \top))$	(olc *IV*1.40)
13	$\vdash ((\varphi \vee \top) \leftrightarrow \top)$	11, 12, (impbii *IV*1.36)

□

Metamath Lemma 70. **bl.ortrl** *disjunction with true on left*

$\vdash ((\top \vee \varphi) \leftrightarrow \top)$ (IV 2.12)

Proof.

1	$\vdash ((\varphi \vee \top) \leftrightarrow (\top \vee \varphi))$	(orcom *IV*1.45)
2	$\vdash ((\varphi \vee \top) \leftrightarrow \top)$	(bl.ortrr *IV*2.11)
3	$\vdash ((\top \vee \varphi) \leftrightarrow \top)$	1, 2, (bitr3i *IV*1.22)

□

Metamath Lemma 71. **bl.orfar** *disjunction with false on right*

$\vdash ((\varphi \vee \bot) \leftrightarrow \varphi)$ (IV 2.13)

Proof.

1	$\vdash ((\neg\varphi \wedge \neg\bot) \leftrightarrow \neg(\varphi \vee \bot))$	(pm4.56 *IV*1.53)
2	$\vdash (\bot \leftrightarrow \neg\top)$	(df-false *IV*2.4)
3	$\vdash (\neg\bot \leftrightarrow \neg\neg\top)$	2, (notbii *IV*1.38)
4	$\vdash (\top \leftrightarrow \neg\neg\top)$	(notnot *IV*1.39)
5	$\vdash (\neg\bot \leftrightarrow \top)$	3, 4, (bitr4i *IV*1.23)
6	$\vdash ((\neg\varphi \wedge \neg\bot) \leftrightarrow (\neg\varphi \wedge \top))$	5, (anbi2i *IV*1.10)
7	$\vdash ((\neg\varphi \wedge \top) \leftrightarrow \neg\varphi)$	(bl.antrr *IV*2.7)
8	$\vdash ((\neg\varphi \wedge \neg\bot) \leftrightarrow \neg\varphi)$	6, 7, (bitri *IV*1.24)
9	$\vdash (\neg(\varphi \vee \bot) \leftrightarrow \neg\varphi)$	1, 8, (bitr3i *IV*1.22)
10	$\vdash ((\varphi \vee \bot) \leftrightarrow \varphi)$	9, (con4bii *IV*1.26)

□

Metamath Lemma 72. **bl.orfal** *disjunction with false on left*

$\vdash ((\bot \vee \varphi) \leftrightarrow \varphi)$ (IV 2.14)

Proof.

1	$\vdash ((\bot \vee \varphi) \leftrightarrow (\varphi \vee \bot))$	(orcom *IV*1.45)
2	$\vdash ((\varphi \vee \bot) \leftrightarrow \varphi)$	(bl.orfar *IV*2.13)
3	$\vdash ((\bot \vee \varphi) \leftrightarrow \varphi)$	1, 2, (bitri *IV*1.24)

□

Metamath Lemma 73. **bl.an2impor2** *bl.an2impor2*

$\vdash ((\varphi \wedge \psi) \rightarrow (\varphi \vee \chi))$ (IV 2.15)

Proof.

1	$\vdash ((\varphi \wedge \psi) \rightarrow \varphi)$	(simpl *IV*1.54)
2	$\vdash (\varphi \rightarrow (\varphi \vee \chi))$	(orc *IV*1.44)
3	$\vdash ((\varphi \wedge \psi) \rightarrow (\varphi \vee \chi))$	1, 2, (syl *IV*1.55)

□

IV 2.2 Well-Formed Formula Substitution

Metamath Lemma 74. **df-sbw** *define substitution of a wff by a wff in a wff*

$\vdash ([\bar{\psi}/\chi]\bar{\varphi} \leftrightarrow ((\chi \leftrightarrow \psi) \rightarrow \varphi))$ (IV 2.16)

Metamath Lemma 75. **bl.sbwid** *Substitution by the Same wff has no Effect*

$\vdash ([\bar{\psi}/\psi]\bar{\varphi} \leftrightarrow \varphi)$ (IV 2.17)

Proof.

1	$\vdash ([\bar{\psi}/\psi]\bar{\varphi} \leftrightarrow ((\psi \leftrightarrow \psi) \rightarrow \varphi))$	(df-sbw *IV*2.16)
2	$\vdash (\psi \leftrightarrow \psi)$	(biid *IV*1.17)
3	$\vdash (\varphi \leftrightarrow ((\psi \leftrightarrow \psi) \rightarrow \varphi))$	2, (a1bi *IV*1.6)
4	$\vdash (((\psi \leftrightarrow \psi) \rightarrow \varphi) \leftrightarrow \varphi)$	3, (bicomi *IV*1.16)
5	$\vdash ([\bar{\psi}/\psi]\bar{\varphi} \leftrightarrow \varphi)$	1, 4, (bitri *IV*1.24)

□

Metamath Lemma 76. **bl.bisbwl** *Equivalence of wff Substitution on Left*

$\vdash (\chi \leftrightarrow \psi) \quad \Rightarrow \quad \vdash ([\bar{\psi}/\chi]\bar{\varphi} \leftrightarrow \varphi)$ (IV 2.18)

Proof.

1	$\vdash ([\bar{\psi}/\chi]\bar{\varphi} \leftrightarrow ((\chi \leftrightarrow \psi) \rightarrow \varphi))$	(df-sbw *IV*2.16)
2	$\vdash (\chi \leftrightarrow \psi)$	Hyp bisbwl.1
3	$\vdash (\varphi \leftrightarrow ((\chi \leftrightarrow \psi) \rightarrow \varphi))$	2, (a1bi *IV*1.6)
4	$\vdash ([\bar{\psi}/\chi]\bar{\varphi} \leftrightarrow \varphi)$	1, 3, (bitr4i *IV*1.23)

□

Metamath Lemma 77. **bl.bisbwr** *Equivalence of wff Substitution on Right*

$\vdash (\chi \leftrightarrow \psi) \quad \Rightarrow \quad \vdash (\varphi \leftrightarrow [\bar{\psi}/\chi]\bar{\varphi})$ (IV 2.19)

Proof.

1	$\vdash (\chi \leftrightarrow \psi)$	Hyp bisbwr.1
2	$\vdash ([\bar{\psi}/\chi]\bar{\varphi} \leftrightarrow \varphi)$	1, (bl.bisbwl *IV*2.18)
3	$\vdash (\varphi \leftrightarrow [\bar{\psi}/\chi]\bar{\varphi})$	2, (bicomi *IV*1.16)

□

Metamath Lemma 78. **bl.sylsbw** *problem with LaTeX*

$\vdash (\varphi \rightarrow \psi) \quad \& \quad \vdash (\chi \leftrightarrow \theta) \quad \Rightarrow \quad \vdash ([\bar{\theta}/\chi]\bar{\varphi} \rightarrow [\bar{\theta}/\chi]\bar{\psi})$ (IV 2.20)

Proof.

1	$\vdash (\varphi \rightarrow \psi)$	Hyp sylsbw.1
2	$\vdash (((\chi \leftrightarrow \theta) \rightarrow \varphi) \rightarrow ((\chi \leftrightarrow \theta) \rightarrow \psi))$	1, (imim2i *IV*1.35)
3	$\vdash ([\bar{\theta}/\chi]\bar{\varphi} \leftrightarrow ((\chi \leftrightarrow \theta) \rightarrow \varphi))$	(df-sbw *IV*2.16)
4	$\vdash ([\bar{\theta}/\chi]\bar{\psi} \leftrightarrow ((\chi \leftrightarrow \theta) \rightarrow \psi))$	(df-sbw *IV*2.16)
5	$\vdash ([\bar{\theta}/\chi]\bar{\varphi} \rightarrow [\bar{\theta}/\chi]\bar{\psi})$	2, 3, 4, (3imtr4i *IV*1.5)

□

Metamath Lemma 79. **bl.sbwsyl** *Syllogism of wff Substitution Elimination from Inference*

$\vdash ([\bar{\theta}/\chi]\bar{\varphi} \rightarrow [\bar{\theta}/\chi]\bar{\psi}) \quad \& \quad \vdash (\chi \leftrightarrow \theta) \quad \Rightarrow \quad \vdash (\varphi \rightarrow \psi)$ (IV 2.21)

Proof.

1	$\vdash (\chi \leftrightarrow \theta)$	Hyp sbwsyl.2
2	$\vdash ([\bar{\theta}/\chi]\bar{\varphi} \rightarrow [\bar{\theta}/\chi]\bar{\psi})$	Hyp sbwsyl.1
3	$\vdash ([\bar{\theta}/\chi]\bar{\varphi} \leftrightarrow ((\chi \leftrightarrow \theta) \rightarrow \varphi))$	(df-sbw *IV*2.16)
4	$\vdash ([\bar{\theta}/\chi]\bar{\psi} \leftrightarrow ((\chi \leftrightarrow \theta) \rightarrow \psi))$	(df-sbw *IV*2.16)
5	$\vdash (((\chi \leftrightarrow \theta) \rightarrow \varphi) \rightarrow ((\chi \leftrightarrow \theta) \rightarrow \psi))$	2, 3, 4, (3imtr3i *IV*1.4)
6	$\vdash ((\chi \leftrightarrow \theta) \rightarrow (\varphi \rightarrow \psi))$	5, (pm2.86i *IV*1.49)
7	$\vdash (\varphi \rightarrow \psi)$	1, 6, (ax-mp *IV*1.15)

□

IV 2.3 Many-Term Conjunction and Disjunction

Metamath Lemma 80. **df-lan0** *no wff in the list*

$\vdash (\bigwedge() \leftrightarrow \top)$ (IV 2.22)

Metamath Lemma 81. **df-lan1** *just one wff in the list*

$\vdash (\bigwedge(\varphi) \leftrightarrow \varphi)$ (IV 2.23)

Metamath Lemma 82. **df-lan2wl** *conjunction of two conjunction wff-list*

$\vdash (\bigwedge(l_1 l_2) \leftrightarrow (\bigwedge(l_1) \wedge \bigwedge(l_2)))$ (IV 2.24)

Metamath Lemma 83. **df-lor0** *no wff in the list*

$\vdash (\bigvee() \leftrightarrow \bot)$ (IV 2.25)

Metamath Lemma 84. **df-lor1** *just one wff in the list*

$$\vdash (\bigvee(\varphi) \leftrightarrow \varphi) \tag{IV 2.26}$$

Metamath Lemma 85. **df-lor2wl** *disjunction of two disjunction wff-lists*

$$\vdash (\bigvee(\mathrm{l}_1\mathrm{l}_2) \leftrightarrow (\bigvee(\mathrm{l}_1) \vee \bigvee(\mathrm{l}_2))) \tag{IV 2.27}$$

Metamath Lemma 86. **bl.an2wl** *two-term list is conjunction*

$$\vdash (\bigwedge(\varphi\psi) \leftrightarrow (\varphi \wedge \psi)) \tag{IV 2.28}$$

Proof.

1	$\vdash (\bigwedge(\varphi\psi) \leftrightarrow (\bigwedge(\varphi) \wedge \bigwedge(\psi)))$	(df-lan2wl *IV*2.24)
2	$\vdash (\bigwedge(\varphi) \leftrightarrow \varphi)$	(df-lan1 *IV*2.23)
3	$\vdash (\bigwedge(\psi) \leftrightarrow \psi)$	(df-lan1 *IV*2.23)
4	$\vdash ((\bigwedge(\varphi) \wedge \bigwedge(\psi)) \leftrightarrow (\varphi \wedge \psi))$	2, 3, (anbi12i *IV*1.8)
5	$\vdash (\bigwedge(\varphi\psi) \leftrightarrow (\varphi \wedge \psi))$	1, 4, (bitri *IV*1.24)

□

Metamath Lemma 87. **bl.an3wl** *three-term conjunction wff-list is triple-conjunction*

$$\vdash (\bigwedge(\varphi\psi\chi) \leftrightarrow (\varphi \wedge \psi \wedge \chi)) \tag{IV 2.29}$$

Proof.

1	$\vdash (\bigwedge(\varphi\psi\chi) \leftrightarrow (\bigwedge(\varphi) \wedge \bigwedge(\psi\chi)))$	(df-lan2wl *IV*2.24)
2	$\vdash (\bigwedge(\varphi) \leftrightarrow \varphi)$	(df-lan1 *IV*2.23)
3	$\vdash (\bigwedge(\psi\chi) \leftrightarrow (\psi \wedge \chi))$	(bl.an2wl *IV*2.28)
4	$\vdash ((\bigwedge(\varphi) \wedge \bigwedge(\psi\chi)) \leftrightarrow (\varphi \wedge (\psi \wedge \chi)))$	2, 3, (anbi12i *IV*1.8)
5	$\vdash (\bigwedge(\varphi\psi\chi) \leftrightarrow (\varphi \wedge (\psi \wedge \chi)))$	1, 4, (bitri *IV*1.24)
6	$\vdash ((\varphi \wedge \psi \wedge \chi) \leftrightarrow (\varphi \wedge (\psi \wedge \chi)))$	(3anass *IV*1.1)
7	$\vdash (\bigwedge(\varphi\psi\chi) \leftrightarrow (\varphi \wedge \psi \wedge \chi))$	5, 6, (bitr4i *IV*1.23)

□

Metamath Lemma 88. **bl.ancomphwl** *commute first and last terms in conjunction wff-list*

$$\vdash (\bigwedge(\varphi\mathrm{l}_1) \leftrightarrow \bigwedge(\mathrm{l}_1\varphi)) \tag{IV 2.30}$$

Proof.

1	$\vdash (\bigwedge(\varphi \mathsf{l}_1) \leftrightarrow (\bigwedge(\varphi) \wedge \bigwedge(\mathsf{l}_1)))$	(df-lan2wl *IV*2.24)
2	$\vdash ((\bigwedge(\varphi) \wedge \bigwedge(\mathsf{l}_1)) \leftrightarrow (\bigwedge(\mathsf{l}_1) \wedge \bigwedge(\varphi)))$	(ancom *IV*1.11)
3	$\vdash (\bigwedge(\mathsf{l}_1 \varphi) \leftrightarrow (\bigwedge(\mathsf{l}_1) \wedge \bigwedge(\varphi)))$	(df-lan2wl *IV*2.24)
4	$\vdash ((\bigwedge(\mathsf{l}_1) \wedge \bigwedge(\varphi)) \leftrightarrow \bigwedge(\mathsf{l}_1 \varphi))$	3, (bicomi *IV*1.16)
5	$\vdash (\bigwedge(\varphi \mathsf{l}_1) \leftrightarrow \bigwedge(\mathsf{l}_1 \varphi))$	1, 2, 4, (3bitri *IV*1.3)

□

Metamath Lemma 89. **bl.ancomwlwl** *commute conjunction wff-lists*

$\vdash (\bigwedge(\mathsf{l}_1 \mathsf{l}_2) \leftrightarrow \bigwedge(\mathsf{l}_2 \mathsf{l}_1))$ (IV 2.31)

Proof.

1	$\vdash (\bigwedge(\mathsf{l}_1 \mathsf{l}_2) \leftrightarrow (\bigwedge(\mathsf{l}_1) \wedge \bigwedge(\mathsf{l}_2)))$	(df-lan2wl *IV*2.24)
2	$\vdash ((\bigwedge(\mathsf{l}_1) \wedge \bigwedge(\mathsf{l}_2)) \leftrightarrow (\bigwedge(\mathsf{l}_2) \wedge \bigwedge(\mathsf{l}_1)))$	(ancom *IV*1.11)
3	$\vdash (\bigwedge(\mathsf{l}_2 \mathsf{l}_1) \leftrightarrow (\bigwedge(\mathsf{l}_2) \wedge \bigwedge(\mathsf{l}_1)))$	(df-lan2wl *IV*2.24)
4	$\vdash ((\bigwedge(\mathsf{l}_2) \wedge \bigwedge(\mathsf{l}_1)) \leftrightarrow \bigwedge(\mathsf{l}_2 \mathsf{l}_1))$	3, (bicomi *IV*1.16)
5	$\vdash (\bigwedge(\mathsf{l}_1 \mathsf{l}_2) \leftrightarrow \bigwedge(\mathsf{l}_2 \mathsf{l}_1))$	1, 2, 4, (3bitri *IV*1.3)

□

Metamath Lemma 90. **bl.ancomphfirst** *make any wff first in conjunction wff-list*

$\vdash (\bigwedge(\mathsf{l}_1 \varphi \mathsf{l}_2) \leftrightarrow \bigwedge(\varphi \mathsf{l}_2 \mathsf{l}_1))$ (IV 2.32)

Proof.

1	$\vdash (\bigwedge(\mathsf{l}_1 \varphi \mathsf{l}_2) \leftrightarrow \bigwedge(\varphi \mathsf{l}_2 \mathsf{l}_1))$	(bl.ancomwlwl *IV*2.31)

□

Metamath Lemma 91. **bl.ancomphlast** *make any wff last in conjunction wff-list*

$\vdash (\bigwedge(\mathsf{l}_1 \varphi \mathsf{l}_2) \leftrightarrow \bigwedge(\mathsf{l}_2 \mathsf{l}_1 \varphi))$ (IV 2.33)

Proof.

1	$\vdash (\bigwedge(\mathsf{l}_1 \varphi \mathsf{l}_2) \leftrightarrow \bigwedge(\mathsf{l}_2 \mathsf{l}_1 \varphi))$	(bl.ancomwlwl *IV*2.31)

□

Metamath Lemma 92. **bl.anpfw** *pull first wff from conjunction wff-list*

$$\vdash (\bigwedge(\varphi \mathrm{l}_1) \leftrightarrow (\varphi \wedge \bigwedge(\mathrm{l}_1))) \qquad \text{(IV 2.34)}$$

Proof.

1	$\vdash (\bigwedge(\varphi \mathrm{l}_1) \leftrightarrow (\bigwedge(\varphi) \wedge \bigwedge(\mathrm{l}_1)))$	(df-lan2wl *IV*2.24)
2	$\vdash (\bigwedge(\varphi) \leftrightarrow \varphi)$	(df-lan1 *IV*2.23)
3	$\vdash ((\bigwedge(\varphi) \wedge \bigwedge(\mathrm{l}_1)) \leftrightarrow (\varphi \wedge \bigwedge(\mathrm{l}_1)))$	2, (anbi1i *IV*1.9)
4	$\vdash (\bigwedge(\varphi \mathrm{l}_1) \leftrightarrow (\varphi \wedge \bigwedge(\mathrm{l}_1)))$	1, 3, (bitri *IV*1.24)

□

Metamath Lemma 93. **bl.anplw** *pull last wff from conjunction wff-list*

$$\vdash (\bigwedge(\mathrm{l}_1 \varphi) \leftrightarrow (\bigwedge(\mathrm{l}_1) \wedge \varphi)) \qquad \text{(IV 2.35)}$$

Proof.

1	$\vdash (\bigwedge(\mathrm{l}_1 \varphi) \leftrightarrow (\bigwedge(\mathrm{l}_1) \wedge \bigwedge(\varphi)))$	(df-lan2wl *IV*2.24)
2	$\vdash (\bigwedge(\varphi) \leftrightarrow \varphi)$	(df-lan1 *IV*2.23)
3	$\vdash ((\bigwedge(\mathrm{l}_1) \wedge \bigwedge(\varphi)) \leftrightarrow (\bigwedge(\mathrm{l}_1) \wedge \varphi))$	2, (anbi2i *IV*1.10)
4	$\vdash (\bigwedge(\mathrm{l}_1 \varphi) \leftrightarrow (\bigwedge(\mathrm{l}_1) \wedge \varphi))$	1, 3, (bitri *IV*1.24)

□

Metamath Lemma 94. **bl.anpmw** *pull middle wff from conjunction wff-list*

$$\vdash (\bigwedge(\mathrm{l}_1 \varphi \mathrm{l}_2) \leftrightarrow (\varphi \wedge \bigwedge(\mathrm{l}_2 \mathrm{l}_1))) \qquad \text{(IV 2.36)}$$

Proof.

1	$\vdash (\bigwedge(\mathrm{l}_1 \varphi \mathrm{l}_2) \leftrightarrow \bigwedge(\varphi \mathrm{l}_2 \mathrm{l}_1))$	(bl.ancomphfirst *IV*2.32)
2	$\vdash (\bigwedge(\varphi \mathrm{l}_2 \mathrm{l}_1) \leftrightarrow (\varphi \wedge \bigwedge(\mathrm{l}_2 \mathrm{l}_1)))$	(bl.anpfw *IV*2.34)
3	$\vdash (\bigwedge(\mathrm{l}_1 \varphi \mathrm{l}_2) \leftrightarrow (\varphi \wedge \bigwedge(\mathrm{l}_2 \mathrm{l}_1)))$	1, 2, (bitri *IV*1.24)

□

Metamath Lemma 95. **bl.anabpf** *absorb first term conjunction wff-list*

$$\vdash (\bigwedge(\bigwedge(\mathrm{l}_1)\mathrm{l}_2) \leftrightarrow \bigwedge(\mathrm{l}_1 \mathrm{l}_2)) \qquad \text{(IV 2.37)}$$

Proof.

1	$\vdash (\bigwedge(\bigwedge(\mathrm{l}_1)\mathrm{l}_2) \leftrightarrow (\bigwedge(\bigwedge(\mathrm{l}_1)) \wedge \bigwedge(\mathrm{l}_2)))$	(df-lan2wl *IV*2.24)
2	$\vdash (\bigwedge(\bigwedge(\mathrm{l}_1)) \leftrightarrow \bigwedge(\mathrm{l}_1))$	(df-lan1 *IV*2.23)
3	$\vdash ((\bigwedge(\bigwedge(\mathrm{l}_1)) \wedge \bigwedge(\mathrm{l}_2)) \leftrightarrow (\bigwedge(\mathrm{l}_1) \wedge \bigwedge(\mathrm{l}_2)))$	2, (anbi1i *IV*1.9)
4	$\vdash (\bigwedge(\mathrm{l}_1\mathrm{l}_2) \leftrightarrow (\bigwedge(\mathrm{l}_1) \wedge \bigwedge(\mathrm{l}_2)))$	(df-lan2wl *IV*2.24)
5	$\vdash ((\bigwedge(\mathrm{l}_1) \wedge \bigwedge(\mathrm{l}_2)) \leftrightarrow \bigwedge(\mathrm{l}_1\mathrm{l}_2))$	4, (bicomi *IV*1.16)
6	$\vdash (\bigwedge(\bigwedge(\mathrm{l}_1)\mathrm{l}_2) \leftrightarrow \bigwedge(\mathrm{l}_1\mathrm{l}_2))$	1, 3, 5, (3bitri *IV*1.3)

□

Metamath Lemma 96. **bl.anabpm** *absorb parentheses, middle term*

$\vdash (\bigwedge(\mathrm{l}_1 \bigwedge(\mathrm{l}_2)\mathrm{l}_3) \leftrightarrow \bigwedge(\mathrm{l}_1\mathrm{l}_2\mathrm{l}_3))$ (IV 2.38)

Proof.

1	$\vdash (\bigwedge(\mathrm{l}_1 \bigwedge(\mathrm{l}_2)\mathrm{l}_3) \leftrightarrow \bigwedge(\bigwedge(\mathrm{l}_2)\mathrm{l}_3\mathrm{l}_1))$	(bl.ancomwlwl *IV*2.31)
2	$\vdash (\bigwedge(\bigwedge(\mathrm{l}_2)\mathrm{l}_3\mathrm{l}_1) \leftrightarrow \bigwedge(\mathrm{l}_2\mathrm{l}_3\mathrm{l}_1))$	(bl.anabpf *IV*2.37)
3	$\vdash (\bigwedge(\mathrm{l}_2\mathrm{l}_3\mathrm{l}_1) \leftrightarrow \bigwedge(\mathrm{l}_1\mathrm{l}_2\mathrm{l}_3))$	(bl.ancomwlwl *IV*2.31)
4	$\vdash (\bigwedge(\mathrm{l}_1 \bigwedge(\mathrm{l}_2)\mathrm{l}_3) \leftrightarrow \bigwedge(\mathrm{l}_1\mathrm{l}_2\mathrm{l}_3))$	1, 2, 3, (3bitri *IV*1.3)

□

Metamath Lemma 97. **bl.anabpl** *absorb parentheses, last term*

$\vdash (\bigwedge(\mathrm{l}_1 \bigwedge(\mathrm{l}_2)) \leftrightarrow \bigwedge(\mathrm{l}_1\mathrm{l}_2))$ (IV 2.39)

Proof.

1	$\vdash (\bigwedge(\mathrm{l}_1 \bigwedge(\mathrm{l}_2)) \leftrightarrow \bigwedge(\bigwedge(\mathrm{l}_2)\mathrm{l}_1))$	(bl.ancomwlwl *IV*2.31)
2	$\vdash (\bigwedge(\bigwedge(\mathrm{l}_2)\mathrm{l}_1) \leftrightarrow \bigwedge(\mathrm{l}_2\mathrm{l}_1))$	(bl.anabpf *IV*2.37)
3	$\vdash (\bigwedge(\mathrm{l}_2\mathrm{l}_1) \leftrightarrow \bigwedge(\mathrm{l}_1\mathrm{l}_2))$	(bl.ancomwlwl *IV*2.31)
4	$\vdash (\bigwedge(\mathrm{l}_1 \bigwedge(\mathrm{l}_2)) \leftrightarrow \bigwedge(\mathrm{l}_1\mathrm{l}_2))$	1, 2, 3, (3bitri *IV*1.3)

□

Metamath Lemma 98. **bl.or2wl** *two-term disjunction wff-list is disjunction*

$\vdash (\bigvee(\varphi\psi) \leftrightarrow (\varphi \vee \psi))$ (IV 2.40)

Proof.

1	$\vdash (\bigvee(\varphi\psi) \leftrightarrow (\bigvee(\varphi) \vee \bigvee(\psi)))$	(df-lor2wl *IV*2.27)
2	$\vdash (\bigvee(\varphi) \leftrightarrow \varphi)$	(df-lor1 *IV*2.26)
3	$\vdash (\bigvee(\psi) \leftrightarrow \psi)$	(df-lor1 *IV*2.26)
4	$\vdash ((\bigvee(\varphi) \vee \bigvee(\psi)) \leftrightarrow (\varphi \vee \psi))$	2, 3, (orbi12i *IV*1.41)
5	$\vdash (\bigvee(\varphi\psi) \leftrightarrow (\varphi \vee \psi))$	1, 4, (bitri *IV*1.24)

□

Metamath Lemma 99. **bl.or3wl** *three-term disjunction wff-list is disjunction*

$\vdash (\bigvee(\varphi\psi\chi) \leftrightarrow (\varphi \vee \psi \vee \chi))$ (IV 2.41)

Proof.

1	$\vdash (\bigvee(\varphi\psi\chi) \leftrightarrow (\bigvee(\varphi) \vee \bigvee(\psi\chi)))$	(df-lor2wl *IV*2.27)
2	$\vdash (\bigvee(\varphi) \leftrightarrow \varphi)$	(df-lor1 *IV*2.26)
3	$\vdash (\bigvee(\psi\chi) \leftrightarrow (\psi \vee \chi))$	(bl.or2wl *IV*2.40)
4	$\vdash ((\bigvee(\varphi) \vee \bigvee(\psi\chi)) \leftrightarrow (\varphi \vee (\psi \vee \chi)))$	2, 3, (orbi12i *IV*1.41)
5	$\vdash (\bigvee(\varphi\psi\chi) \leftrightarrow (\varphi \vee (\psi \vee \chi)))$	1, 4, (bitri *IV*1.24)
6	$\vdash ((\varphi \vee \psi \vee \chi) \leftrightarrow (\varphi \vee (\psi \vee \chi)))$	(3orass *IV*1.2)
7	$\vdash (\bigvee(\varphi\psi\chi) \leftrightarrow (\varphi \vee \psi \vee \chi))$	5, 6, (bitr4i *IV*1.23)

□

Metamath Lemma 100. **bl.orcomphwl** *commute first and last terms of disjunction wff-list*

$\vdash (\bigvee(\varphi l_1) \leftrightarrow \bigvee(l_1\varphi))$ (IV 2.42)

Proof.

1	$\vdash (\bigvee(\varphi l_1) \leftrightarrow (\bigvee(\varphi) \vee \bigvee(l_1)))$	(df-lor2wl *IV*2.27)
2	$\vdash ((\bigvee(\varphi) \vee \bigvee(l_1)) \leftrightarrow (\bigvee(l_1) \vee \bigvee(\varphi)))$	(orcom *IV*1.45)
3	$\vdash (\bigvee(l_1\varphi) \leftrightarrow (\bigvee(l_1) \vee \bigvee(\varphi)))$	(df-lor2wl *IV*2.27)
4	$\vdash ((\bigvee(l_1) \vee \bigvee(\varphi)) \leftrightarrow \bigvee(l_1\varphi))$	3, (bicomi *IV*1.16)
5	$\vdash (\bigvee(\varphi l_1) \leftrightarrow \bigvee(l_1\varphi))$	1, 2, 4, (3bitri *IV*1.3)

□

Metamath Lemma 101. **bl.orcomwlwl** *commute disjunction wff-lists*

$$\vdash (\bigvee(l_1\, l_1) \leftrightarrow \bigvee(l_1\, l_1)) \qquad \text{(IV 2.43)}$$

Proof.

1	$\vdash (\bigvee(l_1\, l_1) \leftrightarrow (\bigvee(l_1) \vee \bigvee(l_1)))$	(df-lor2wl *IV*2.27)
2	$\vdash ((\bigvee(l_1) \vee \bigvee(l_1)) \leftrightarrow (\bigvee(l_1) \vee \bigvee(l_1)))$	(orcom *IV*1.45)
3	$\vdash (\bigvee(l_1\, l_1) \leftrightarrow (\bigvee(l_1) \vee \bigvee(l_1)))$	(df-lor2wl *IV*2.27)
4	$\vdash ((\bigvee(l_1) \vee \bigvee(l_1)) \leftrightarrow \bigvee(l_1\, l_1))$	3, (bicomi *IV*1.16)
5	$\vdash (\bigvee(l_1\, l_1) \leftrightarrow \bigvee(l_1\, l_1))$	1, 2, 4, (3bitri *IV*1.3)

□

Metamath Lemma 102. **bl.orcomphfirst** *make any wff first in disjunction wff-list*

$$\vdash (\bigvee(l_1\varphi l_1) \leftrightarrow \bigvee(\varphi l_1\, l_1)) \qquad \text{(IV 2.44)}$$

Proof.

1	$\vdash (\bigvee(l_1\varphi l_1) \leftrightarrow \bigvee(\varphi l_1\, l_1))$	(bl.orcomwlwl *IV*2.43)

□

Metamath Lemma 103. **bl.orcomphlast** *make any wff last in disjunction wff-list*

$$\vdash (\bigvee(l_1\varphi l_1) \leftrightarrow \bigvee(l_1\, l_1\varphi)) \qquad \text{(IV 2.45)}$$

Proof.

1	$\vdash (\bigvee(l_1\varphi l_1) \leftrightarrow \bigvee(l_1\, l_1\varphi))$	(bl.orcomwlwl *IV*2.43)

□

Metamath Lemma 104. **bl.orpfw** *pull first wff from disjunction wff-list*

$$\vdash (\bigvee(\varphi l_1) \leftrightarrow (\varphi \vee \bigvee(l_1))) \qquad \text{(IV 2.46)}$$

Proof.

1	$\vdash (\bigvee(\varphi l_1) \leftrightarrow (\bigvee(\varphi) \vee \bigvee(l_1)))$	(df-lor2wl *IV*2.27)
2	$\vdash (\bigvee(\varphi) \leftrightarrow \varphi)$	(df-lor1 *IV*2.26)
3	$\vdash ((\bigvee(\varphi) \vee \bigvee(l_1)) \leftrightarrow (\varphi \vee \bigvee(l_1)))$	2, (orbi1i *IV*1.42)
4	$\vdash (\bigvee(\varphi l_1) \leftrightarrow (\varphi \vee \bigvee(l_1)))$	1, 3, (bitri *IV*1.24)

□

Metamath Lemma 105. **bl.orplw** *pull last wff from disjunction wff-list*

$$\vdash (\bigvee(l_1\varphi) \leftrightarrow (\bigvee(l_1) \vee \varphi)) \qquad \text{(IV 2.47)}$$

Proof.

1	$\vdash (\bigvee(l_1\varphi) \leftrightarrow (\bigvee(l_1) \vee \bigvee(\varphi)))$	(df-lor2wl *IV*2.27)
2	$\vdash (\bigvee(\varphi) \leftrightarrow \varphi)$	(df-lor1 *IV*2.26)
3	$\vdash ((\bigvee(l_1) \vee \bigvee(\varphi)) \leftrightarrow (\bigvee(l_1) \vee \varphi))$	2, (orbi2i *IV*1.43)
4	$\vdash (\bigvee(l_1\varphi) \leftrightarrow (\bigvee(l_1) \vee \varphi))$	1, 3, (bitri *IV*1.24)

□

Metamath Lemma 106. **bl.orpmw** *pull middle wff from disjunction wff-list*

$$\vdash (\bigvee(l_1\varphi l_1) \leftrightarrow (\varphi \vee \bigvee(l_1\, l_1))) \qquad \text{(IV 2.48)}$$

Proof.

1	$\vdash (\bigvee(l_1\varphi l_1) \leftrightarrow \bigvee(\varphi l_1\, l_1))$	(bl.orcomphfirst *IV*2.44)
2	$\vdash (\bigvee(\varphi l_1\, l_1) \leftrightarrow (\varphi \vee \bigvee(l_1\, l_1)))$	(bl.orpfw *IV*2.46)
3	$\vdash (\bigvee(l_1\varphi l_1) \leftrightarrow (\varphi \vee \bigvee(l_1\, l_1)))$	1, 2, (bitri *IV*1.24)

□

Metamath Lemma 107. **bl.orabpf** *absorb parentheses, first term in disjunction wff-list*

$$\vdash (\bigvee(\bigvee(l_1)l_1) \leftrightarrow \bigvee(l_1\, l_1)) \qquad \text{(IV 2.49)}$$

Proof.

1	$\vdash (\bigvee(\bigvee(l_1)l_1) \leftrightarrow (\bigvee(\bigvee(l_1)) \vee \bigvee(l_1)))$	(df-lor2wl *IV*2.27)
2	$\vdash (\bigvee(\bigvee(l_1)) \leftrightarrow \bigvee(l_1))$	(df-lor1 *IV*2.26)
3	$\vdash ((\bigvee(\bigvee(l_1)) \vee \bigvee(l_1)) \leftrightarrow (\bigvee(l_1) \vee \bigvee(l_1)))$	2, (orbi1i *IV*1.42)
4	$\vdash (\bigvee(l_1\, l_1) \leftrightarrow (\bigvee(l_1) \vee \bigvee(l_1)))$	(df-lor2wl *IV*2.27)
5	$\vdash ((\bigvee(l_1) \vee \bigvee(l_1)) \leftrightarrow \bigvee(l_1\, l_1))$	4, (bicomi *IV*1.16)
6	$\vdash (\bigvee(\bigvee(l_1)l_1) \leftrightarrow \bigvee(l_1\, l_1))$	1, 3, 5, (3bitri *IV*1.3)

□

Metamath Lemma 108. **bl.orabpm** *absorb parentheses, middle term in disjunction wff-list*

$$\vdash (\bigvee(l_1 \bigvee(l_1)l_3) \leftrightarrow \bigvee(l_1\, l_1\, l_3)) \qquad \text{(IV 2.50)}$$

Proof.

1	$\vdash (\bigvee(l_1 \bigvee(l_1) l_3) \leftrightarrow \bigvee(\bigvee(l_1) l_3\, l_1))$	(bl.orcomwlwl *IV*2.43)
2	$\vdash (\bigvee(\bigvee(l_1) l_3\, l_1) \leftrightarrow \bigvee(l_1\, l_3\, l_1))$	(bl.orabpf *IV*2.49)
3	$\vdash (\bigvee(l_1\, l_3\, l_1) \leftrightarrow \bigvee(l_1\, l_1\, l_3))$	(bl.orcomwlwl *IV*2.43)
4	$\vdash (\bigvee(l_1 \bigvee(l_1) l_3) \leftrightarrow \bigvee(l_1\, l_1\, l_3))$	1, 2, 3, (3bitri *IV*1.3)

□

Metamath Lemma 109. **bl.orabpl** *absorb parentheses, last term in disjunction wff-list*

$\vdash (\bigvee(l_1 \bigvee(l_1)) \leftrightarrow \bigvee(l_1\, l_1))$ (IV 2.51)

Proof.

1	$\vdash (\bigvee(l_1 \bigvee(l_1)) \leftrightarrow \bigvee(\bigvee(l_1) l_1))$	(bl.orcomwlwl *IV*2.43)
2	$\vdash (\bigvee(\bigvee(l_1) l_1) \leftrightarrow \bigvee(l_1\, l_1))$	(bl.orabpf *IV*2.49)
3	$\vdash (\bigvee(l_1\, l_1) \leftrightarrow \bigvee(l_1\, l_1))$	(bl.orcomwlwl *IV*2.43)
4	$\vdash (\bigvee(l_1 \bigvee(l_1)) \leftrightarrow \bigvee(l_1\, l_1))$	1, 2, 3, (3bitri *IV*1.3)

□

Metamath Lemma 110. **bl.dbo2a** *distribution or-over-and two terms with wff-lists*

$\vdash (\bigwedge(l_1(\varphi \vee \psi) l_1) \leftrightarrow ((\varphi \wedge \bigwedge(l_1\, l_1)) \vee (\psi \wedge \bigwedge(l_1\, l_1))))$ (IV 2.52)

Proof.

1	$\vdash (\bigwedge(l_1(\varphi \vee \psi) l_1) \leftrightarrow \bigwedge((\varphi \vee \psi) l_1\, l_1))$	(bl.ancomphfirst *IV*2.32)
2	$\vdash (\bigwedge((\varphi \vee \psi) l_1\, l_1) \leftrightarrow (\bigwedge((\varphi \vee \psi)) \wedge \bigwedge(l_1\, l_1)))$	(df-lan2wl *IV*2.24)
3	$\vdash (\bigwedge(l_1(\varphi \vee \psi) l_1) \leftrightarrow (\bigwedge((\varphi \vee \psi)) \wedge \bigwedge(l_1\, l_1)))$	1, 2, (bitri *IV*1.24)
4	$\vdash (\bigwedge((\varphi \vee \psi)) \leftrightarrow (\varphi \vee \psi))$	(df-lan1 *IV*2.23)
5	$\vdash ((\bigwedge((\varphi \vee \psi)) \wedge \bigwedge(l_1\, l_1)) \leftrightarrow ((\varphi \vee \psi) \wedge \bigwedge(l_1\, l_1)))$	4, (anbi1i *IV*1.9)
6	$\vdash (((\varphi \vee \psi) \wedge \bigwedge(l_1\, l_1)) \leftrightarrow ((\varphi \wedge \bigwedge(l_1\, l_1)) \vee (\psi \wedge \bigwedge(l_1\, l_1))))$	(andir *IV*1.12)
7	$\vdash (\bigwedge(l_1(\varphi \vee \psi) l_1) \leftrightarrow ((\varphi \wedge \bigwedge(l_1\, l_1)) \vee (\psi \wedge \bigwedge(l_1\, l_1))))$	3, 5, 6, (3bitri *IV*1.3)

□

Metamath Lemma 111. **bl.dba2o** *distribution and-over-or two terms with wff-lists*

$\vdash (\bigvee(l_1(\varphi \wedge \psi) l_1) \leftrightarrow ((\varphi \vee \bigvee(l_1\, l_1)) \wedge (\psi \vee \bigvee(l_1\, l_1))))$ (IV 2.53)

Proof.

1	$\vdash (\bigvee(l_1(\varphi \wedge \psi)l_1) \leftrightarrow \bigvee((\varphi \wedge \psi)l_1\, l_1))$	(bl.orcomphfirst *IV*2.44)
2	$\vdash (\bigvee((\varphi \wedge \psi)l_1\, l_1) \leftrightarrow (\bigvee((\varphi \wedge \psi)) \vee \bigvee(l_1\, l_1)))$	(df-lor2wl *IV*2.27)
3	$\vdash (\bigvee(l_1(\varphi \wedge \psi)l_1) \leftrightarrow (\bigvee((\varphi \wedge \psi)) \vee \bigvee(l_1\, l_1)))$	1, 2, (bitri *IV*1.24)
4	$\vdash (\bigvee((\varphi \wedge \psi)) \leftrightarrow (\varphi \wedge \psi))$	(df-lor1 *IV*2.26)
5	$\vdash ((\bigvee((\varphi \wedge \psi)) \vee \bigvee(l_1\, l_1)) \leftrightarrow ((\varphi \wedge \psi) \vee \bigvee(l_1\, l_1)))$	4, (orbi1i *IV*1.42)
6	$\vdash (((\varphi \wedge \psi) \vee \bigvee(l_1\, l_1)) \leftrightarrow ((\varphi \vee \bigvee(l_1\, l_1)) \wedge (\psi \vee \bigvee(l_1\, l_1))))$	(ordir *IV*1.46)
7	$\vdash (\bigvee(l_1(\varphi \wedge \psi)l_1) \leftrightarrow ((\varphi \vee \bigvee(l_1\, l_1)) \wedge (\psi \vee \bigvee(l_1\, l_1))))$	3, 5, 6, (3bitri *IV*1.3)

□

Metamath Lemma 112. **bl.dbo2awl** *distribution or-over-and with wff-lists*

$$\vdash (\bigwedge(l_1 \bigvee(l_1\, l_3)l_4) \leftrightarrow ((\bigvee(l_1) \wedge \bigwedge(l_4\, l_1)) \vee (\bigvee(l_3) \wedge \bigwedge(l_4\, l_1)))) \qquad \text{(IV 2.54)}$$

Proof.

1	$\vdash (\bigwedge(l_1 \bigvee(l_1\, l_3)l_4) \leftrightarrow (\bigvee(l_1\, l_3) \wedge \bigwedge(l_4\, l_1)))$	(bl.anpmw *IV*2.36)
2	$\vdash (\bigvee(l_1\, l_3) \leftrightarrow (\bigvee(l_1) \vee \bigvee(l_3)))$	(df-lor2wl *IV*2.27)
3	$\vdash ((\bigvee(l_1\, l_3) \wedge \bigwedge(l_4\, l_1)) \leftrightarrow ((\bigvee(l_1) \vee \bigvee(l_3)) \wedge \bigwedge(l_4\, l_1)))$	2, (anbi1i *IV*1.9)
4	$\vdash (((\bigvee(l_1) \vee \bigvee(l_3)) \wedge \bigwedge(l_4\, l_1)) \leftrightarrow ((\bigvee(l_1) \wedge \bigwedge(l_4\, l_1)) \vee (\bigvee(l_3) \wedge \bigwedge(l_4\, l_1))))$	(andir *IV*1.12)
5	$\vdash (\bigwedge(l_1 \bigvee(l_1\, l_3)l_4) \leftrightarrow ((\bigvee(l_1) \wedge \bigwedge(l_4\, l_1)) \vee (\bigvee(l_3) \wedge \bigwedge(l_4\, l_1))))$	1, 3, 4, (3bitri *IV*1.3)

□

Metamath Lemma 113. **bl.dba2owl** *distribution and-over-or with wff lists*

$$\vdash (\bigvee(l_1 \bigwedge(l_1\, l_3)l_4) \leftrightarrow ((\bigwedge(l_1) \vee \bigvee(l_4\, l_1)) \wedge (\bigwedge(l_3) \vee \bigvee(l_4\, l_1)))) \qquad \text{(IV 2.55)}$$

Proof.

1	$\vdash (\bigvee(l_1 \bigwedge(l_1\, l_3)l_4) \leftrightarrow (\bigwedge(l_1\, l_3) \vee \bigvee(l_4\, l_1)))$	(bl.orpmw *IV*2.48)
2	$\vdash (\bigwedge(l_1\, l_3) \leftrightarrow (\bigwedge(l_1) \wedge \bigwedge(l_3)))$	(df-lan2wl *IV*2.24)
3	$\vdash ((\bigwedge(l_1\, l_3) \vee \bigvee(l_4\, l_1)) \leftrightarrow ((\bigwedge(l_1) \wedge \bigwedge(l_3)) \vee \bigvee(l_4\, l_1)))$	2, (orbi1i *IV*1.42)
4	$\vdash (((\bigwedge(l_1) \wedge \bigwedge(l_3)) \vee \bigvee(l_4\, l_1)) \leftrightarrow ((\bigwedge(l_1) \vee \bigvee(l_4\, l_1)) \wedge (\bigwedge(l_3) \vee \bigvee(l_4\, l_1))))$	(ordir *IV*1.46)
5	$\vdash (\bigvee(l_1 \bigwedge(l_1\, l_3)l_4) \leftrightarrow ((\bigwedge(l_1) \vee \bigvee(l_4\, l_1)) \wedge (\bigwedge(l_3) \vee \bigvee(l_4\, l_1))))$	1, 3, 4, (3bitri *IV*1.3)

□

Metamath Lemma 114. **bl.an2impor2** *bl.an2impor2*

$$\vdash ((\varphi \wedge \psi) \rightarrow (\varphi \vee \chi)) \qquad \text{(IV 2.56)}$$

Proof.

1	$\vdash ((\varphi \wedge \psi) \rightarrow \varphi)$	(simpl *IV*1.54)
2	$\vdash (\varphi \rightarrow (\varphi \vee \chi))$	(orc *IV*1.44)
3	$\vdash ((\varphi \wedge \psi) \rightarrow (\varphi \vee \chi))$	1, 2, (syl *IV*1.55)

□

Metamath Lemma 115. **bl.ctao** *Common Term Between And-List and Or-List*

$$\vdash (\bigwedge(l_1 \varphi l_1) \rightarrow \bigvee(l_3 \varphi l_4)) \qquad \text{(IV 2.57)}$$

Proof.

1	$\vdash ((\varphi \wedge \bigwedge(l_1\, l_1)) \rightarrow (\varphi \vee \bigvee(l_4\, l_3)))$	(bl.an2impor2 *IV*2.56)
2	$\vdash (\bigwedge(l_1 \varphi l_1) \leftrightarrow (\varphi \wedge \bigwedge(l_1\, l_1)))$	(bl.anpmw *IV*2.36)
3	$\vdash (\bigvee(l_3 \varphi l_4) \leftrightarrow (\varphi \vee \bigvee(l_4\, l_3)))$	(bl.orpmw *IV*2.48)
4	$\vdash (\bigwedge(l_1 \varphi l_1) \rightarrow \bigvee(l_3 \varphi l_4))$	1, 2, 3, (3imtr4i *IV*1.5)

□

Metamath Lemma 116. **bl.animporan** *trouble with LaTeX*

$$\vdash (\bigwedge(l_1\, l_1\, l_3) \rightarrow \bigvee(l_4 \bigwedge(l_1) l_5)) \qquad \text{(IV 2.58)}$$

Proof.

1	$\vdash (\bigwedge(l_1 \bigwedge(l_1) l_3) \leftrightarrow \bigwedge(l_1\, l_1\, l_3))$	(bl.anabpm *IV*2.38)
2	$\vdash (\bigwedge(l_1 \bigwedge(l_1) l_3) \leftrightarrow (\bigwedge(l_1) \wedge \bigwedge(l_3\, l_1)))$	(bl.anpmw *IV*2.36)
3	$\vdash (\bigwedge(l_1\, l_1\, l_3) \leftrightarrow (\bigwedge(l_1) \wedge \bigwedge(l_3\, l_1)))$	1, 2, (bitr3i *IV*1.22)
4	$\vdash ((\bigwedge(l_1) \wedge \bigwedge(l_3\, l_1)) \rightarrow (\bigwedge(l_1) \vee \bigvee(l_5\, l_4)))$	(bl.an2impor2 *IV*2.56)
5	$\vdash (\bigwedge(l_1\, l_1\, l_3) \rightarrow (\bigwedge(l_1) \vee \bigvee(l_5\, l_4)))$	3, 4, (sylbi *IV*1.56)
6	$\vdash (\bigvee(l_4 \bigwedge(l_1) l_5) \leftrightarrow (\bigwedge(l_1) \vee \bigvee(l_5\, l_4)))$	(bl.orpmw *IV*2.48)
7	$\vdash (\bigwedge(l_1\, l_1\, l_3) \rightarrow \bigvee(l_4 \bigwedge(l_1) l_5))$	5, 6, (sylibr *IV*1.57)

□

Metamath Lemma 117. **bl.orcwl** *Or-Introduction Schema of wff*

$$\vdash (\varphi \rightarrow \bigvee(l_1 \varphi l_1)) \qquad \text{(IV 2.59)}$$

Proof.

1	$\vdash (\varphi \rightarrow (\varphi \vee \bigvee(l_1\, l_1)))$	(orc *IV*1.44)
2	$\vdash (\bigvee(l_1 \varphi l_1) \leftrightarrow (\varphi \vee \bigvee(l_1\, l_1)))$	(bl.orpmw *IV*2.48)
3	$\vdash (\varphi \rightarrow \bigvee(l_1 \varphi l_1))$	1, 2, (sylibr *IV*1.57)

□

Metamath Lemma 118. **bl.ais** *And-Introduction Schema of wff*

$\vdash (\bigwedge(l_1 \varphi l_1) \rightarrow \varphi)$ (IV 2.60)

Proof.

1	$\vdash (\bigwedge(l_1 \varphi l_1) \leftrightarrow (\varphi \wedge \bigwedge(l_1\, l_1)))$	(bl.anpmw *IV*2.36)
2	$\vdash ((\varphi \wedge \bigwedge(l_1\, l_1)) \rightarrow \varphi)$	(simpl *IV*1.54)
3	$\vdash (\bigwedge(l_1 \varphi l_1) \rightarrow \varphi)$	1, 2, (sylbi *IV*1.56)

□

Metamath Lemma 119. **bl.aiswl** *And-Introduction Schema of wff-List*

$\vdash (\bigwedge(l_1\, l_1\, l_3) \rightarrow \bigwedge(l_1))$ (IV 2.61)

Proof.

1	$\vdash (\bigwedge(l_1 \bigwedge(l_1) l_3) \leftrightarrow \bigwedge(l_1\, l_1\, l_3))$	(bl.anabpm *IV*2.38)
2	$\vdash (\bigwedge(l_1 \bigwedge(l_1) l_3) \leftrightarrow (\bigwedge(l_1) \wedge \bigwedge(l_3\, l_1)))$	(bl.anpmw *IV*2.36)
3	$\vdash (\bigwedge(l_1\, l_1\, l_3) \leftrightarrow (\bigwedge(l_1) \wedge \bigwedge(l_3\, l_1)))$	1, 2, (bitr3i *IV*1.22)
4	$\vdash ((\bigwedge(l_1) \wedge \bigwedge(l_3\, l_1)) \rightarrow \bigwedge(l_1))$	(simpl *IV*1.54)
5	$\vdash (\bigwedge(l_1\, l_1\, l_3) \rightarrow \bigwedge(l_1))$	3, 4, (sylbi *IV*1.56)

□

Chapter IV 3

Axioms

BLESS proof obligations are dispatched by transforming them into simpler proof obligations until recognized as axiomatic tautologies.

When BLESS→Proof→axioms is chosen from the drop down menu, the proof tools examines all of the currently-unsolved proof obligations to see if any match known tautologies. Testing whether the current batch of proof obligations has any that can be recognized as known tautologies (axioms) may also be invoked by a hot key[1] and the "axioms" keyword in a proof script.

IV 3.1 True Conclusion Schema (tc): P→true is tautology

Any formula implying true is always true.

Example:[2]

```
This Proof Obligation:

[serial 1028]:
P [87] <<PACE(now) and vs@now>>
S [87]->
Q [87] <<true>>
Reason:  True Conclusion Schema (tc): P->true
  What for: Law of Excluded Middle: P or not P is tautology [serial 1026]
Has been solved by True Conclusion Schema (tc): P->true
```

Metamath Lemma 120. **bl.tc** *True Consequent*

$$\vdash (\varphi \rightarrow \top) \tag{IV 3.1}$$

[1] opt-cmd-A on Mac OS
[2] from proof of VVI.aadl

Proof.

1	$\vdash (\varphi \rightarrow (\varphi \rightarrow \varphi))$	(ax-1 *IV*1.14)
2	$\vdash (\top \leftrightarrow (\varphi \rightarrow \varphi))$	(df-true *IV*2.3)
3	$\vdash (\varphi \rightarrow \top)$	1, 2, (sylibr *IV*1.57)

□

IV 3.2 Identity (id): P→P is tautology

That P→P should be obvious, but Metamath has a proof from axioms.

Example:[3]

```
This Proof Obligation:

[serial 1036]:
P [96] <<VP_vvi()>>
S [96]->
Q [14] <<VP_vvi()>>
Reason:  Identity (id):  P->P is tautology
  What for: applied port output <<pre>> -> <<M(vp)>> [serial 1034]
Has been solved by Identity (id):  P->P is tautology
```

P→P is Metamath theorem “id”, Principle of Identity, number 68.

IV 3.3 Or-Introduction Schema (orcwl): B → (C or B or D)

Disjunction can be freely-introduced to the consequent.

Example:

```
  not yet used in BLESS correctness proof
```

Metamath Lemma 121. **bl.orcwl** *Or-Introduction Schema of wff*

$$\vdash (\varphi \rightarrow \bigvee(\mathrm{I}_1\varphi\mathrm{I}_1)) \qquad \text{(IV 3.2)}$$

Proof.

1	$\vdash (\varphi \rightarrow (\varphi \vee \bigvee(\mathrm{I}_1\mathrm{I}_1)))$	(orc *IV*1.44)
2	$\vdash (\bigvee(\mathrm{I}_1\varphi\mathrm{I}_1) \leftrightarrow (\varphi \vee \bigvee(\mathrm{I}_1\mathrm{I}_1)))$	(bl.orpmw *IV*2.48)
3	$\vdash (\varphi \rightarrow \bigvee(\mathrm{I}_1\varphi\mathrm{I}_1))$	1, 2, (sylibr *IV*1.57)

□

[3] from proof of VVI.aadl

IV 3.4 And-Elimination Schema (ais aiswl): (A or B or C) → B

Conjunction can be freely-introduced to the premise.

Example:[4]

```
This Proof Obligation:

[serial 1037]:
P [96] <<(VP_vvi()) and vp@now>>
S [96]->
Q [96] <<vp@now>>
Reason:  And Introduction Schema (aisph):  (X and Y)->X
  What for: applied port output <<pre and vp@now>> -> <<post>> [serial 1034]
Has been solved by And Introduction Schema (aisph):  (X and Y)->X
```

Metamath Lemma 122. **bl.ais** *And-Introduction Schema of wff*

$$\vdash (\bigwedge(\mathsf{l}_1\varphi\mathsf{l}_1) \rightarrow \varphi) \tag{IV 3.3}$$

Proof.

1	$\vdash (\bigwedge(\mathsf{l}_1\varphi\mathsf{l}_1) \leftrightarrow (\varphi \wedge \bigwedge(\mathsf{l}_1\mathsf{l}_1)))$	(bl.anpmw *IV*2.36)
2	$\vdash ((\varphi \wedge \bigwedge(\mathsf{l}_1\mathsf{l}_1)) \rightarrow \varphi)$	(simpl *IV*1.54)
3	$\vdash (\bigwedge(\mathsf{l}_1\varphi\mathsf{l}_1) \rightarrow \varphi)$	1, 2, (sylbi *IV*1.56)

□

Metamath Lemma 123. **bl.aiswl** *And-Introduction Schema of wff-List*

$$\vdash (\bigwedge(\mathsf{l}_1\mathsf{l}_1\mathsf{l}_3) \rightarrow \bigwedge(\mathsf{l}_1)) \tag{IV 3.4}$$

Proof.

1	$\vdash (\bigwedge(\mathsf{l}_1 \bigwedge(\mathsf{l}_1)\mathsf{l}_3) \leftrightarrow \bigwedge(\mathsf{l}_1\mathsf{l}_1\mathsf{l}_3))$	(bl.anabpm *IV*2.38)
2	$\vdash (\bigwedge(\mathsf{l}_1 \bigwedge(\mathsf{l}_1)\mathsf{l}_3) \leftrightarrow (\bigwedge(\mathsf{l}_1) \wedge \bigwedge(\mathsf{l}_3\mathsf{l}_1)))$	(bl.anpmw *IV*2.36)
3	$\vdash (\bigwedge(\mathsf{l}_1\mathsf{l}_1\mathsf{l}_3) \leftrightarrow (\bigwedge(\mathsf{l}_1) \wedge \bigwedge(\mathsf{l}_3\mathsf{l}_1)))$	1, 2, (bitr3i *IV*1.22)
4	$\vdash ((\bigwedge(\mathsf{l}_1) \wedge \bigwedge(\mathsf{l}_3\mathsf{l}_1)) \rightarrow \bigwedge(\mathsf{l}_1))$	(simpl *IV*1.54)
5	$\vdash (\bigwedge(\mathsf{l}_1\mathsf{l}_1\mathsf{l}_3) \rightarrow \bigwedge(\mathsf{l}_1))$	3, 4, (sylbi *IV*1.56)

□

[4] from proof of VVI.aadl

IV 3.5 And-Elimination/Or-Introduction Schema (ctao): (P and Q)→(P or R)

The And-Elimination/Or-Introduction Schema seems to be the most common axiom used to solve proof obligations. One all atomic actions have been reduced to implications, solving those implications is (mostly) beating into normal form of a conjuntion implying a disjunction.

The Metamath represents the term (a wff) in common as φ, and possibly-empty lists of wffs as $l_1 l_2 l_3$ and l_4. Because the wff-lists are nullable, the common may occur at the beginning, end, or in the middle of the conjunction $\bigwedge$ and disjunction $\bigvee$. Regardless of the number of terms a common term between a conjunction and a disjunction is enough to assure that the implication of the disjunction, by the conjuntion, will always be a tautology.[5]
Metamath Lemma 124. **bl.ctao** *Common Term Between And-List and Or-List*

$$\vdash \left(\bigwedge (l_1 \varphi l_1) \rightarrow \bigvee (l_3 \varphi l_4)\right) \qquad \text{(IV 3.5)}$$

Proof.

1	$\vdash ((\varphi \wedge \bigwedge (l_1 l_1)) \rightarrow (\varphi \vee \bigvee (l_4 l_3)))$	(bl.an2impor2 *IV*2.56)
2	$\vdash (\bigwedge (l_1 \varphi l_1) \leftrightarrow (\varphi \wedge \bigwedge (l_1 l_1)))$	(bl.anpmw *IV*2.36)
3	$\vdash (\bigvee (l_3 \varphi l_4) \leftrightarrow (\varphi \vee \bigvee (l_4 l_3)))$	(bl.orpmw *IV*2.48)
4	$\vdash \left(\bigwedge (l_1 \varphi l_1) \rightarrow \bigvee (l_3 \varphi l_4)\right)$	1, 2, 3, (3imtr4i *IV*1.5)

□

IV 3.6 Premise Has All Terms of Conjunction within Disjunction (animporan): (A and B and C and D) → (E or (B and C) or F)

An extension of ctao allows a term of the consequent disjunction to be a conjunction of terms from a premise's conjunction.
Metamath Lemma 125. **bl.animporan** *Premise Has All Terms of Conjunction within Disjunction*

$$\vdash \left(\bigwedge (l_1 l_1 l_3) \rightarrow \bigvee (l_4 \bigwedge (l_1) l_5)\right) \qquad \text{(IV 3.6)}$$

[5] Before I extended Metamath for arbitrary-length logical operators, the most number of terms of conjuntion needed by any proof thus far was 3. Now nice things can be proved about expressions and relations that are unrestricted in number of terms.

Proof.

1	$\vdash (\bigwedge(l_1 \bigwedge(l_1)l_3) \leftrightarrow \bigwedge(l_1 l_1 l_3))$	(bl.anabpm *IV*2.38)
2	$\vdash (\bigwedge(l_1 \bigwedge(l_1)l_3) \leftrightarrow (\bigwedge(l_1) \wedge \bigwedge(l_3 l_1)))$	(bl.anpmw *IV*2.36)
3	$\vdash (\bigwedge(l_1 l_1 l_3) \leftrightarrow (\bigwedge(l_1) \wedge \bigwedge(l_3 l_1)))$	1, 2, (bitr3i *IV*1.22)
4	$\vdash ((\bigwedge(l_1) \wedge \bigwedge(l_3 l_1)) \rightarrow (\bigwedge(l_1) \vee \bigvee(l_5 l_4)))$	(bl.an2impor2 *IV*2.56)
5	$\vdash (\bigwedge(l_1 l_1 l_3) \rightarrow (\bigwedge(l_1) \vee \bigvee(l_5 l_4)))$	3, 4, (sylbi *IV*1.56)
6	$\vdash (\bigvee(l_4 \bigwedge(l_1)l_5) \leftrightarrow (\bigwedge(l_1) \vee \bigvee(l_5 l_4)))$	(bl.orpmw *IV*2.48)
7	$\vdash (\bigwedge(l_1 l_1 l_3) \rightarrow \bigvee(l_4 \bigwedge(l_1)l_5))$	5, 6, (sylibr *IV*1.57)

□

Axiom in the general case where the premise is conjunction of any number of terms, and the conclusion is disjunction also of any number of terms, having (at least) one term in common:

Chapter IV 4

Laws of Logic

Laws manipulate logical formulas with simple, obvious tautologies.

IV 4.1 Laws of Conjunction

IV 4.1.1 Law of Contradiction: P and not P is false [pm3.24]

See (**??**).

IV 4.1.2 Law of And-Simplification: P and P is P [andim]

See (**??**).

IV 4.1.3 Law of And-Simplification: P and true is P [bl.antrr]

See (IV 2.7).

IV 4.1.4 Law of And-Simplification: P and false is false [bl.anfar

See (IV 2.9).

IV 4.1.5 Law of And-Simplification: P and (Q or P) is P [bl.PandQorPisP]

Metamath Lemma 126. **bl.PandQorPisP** *Law of And-Simplification: P and (Q or P) is P*

$$\vdash ((\varphi \wedge (\psi \vee \varphi)) \leftrightarrow \varphi) \tag{IV 4.1}$$

Proof.

1	$\vdash (\varphi \leftrightarrow (\varphi \wedge (\varphi \vee \psi)))$	(pm4.45 *IV*1.51)
2	$\vdash ((\varphi \vee \psi) \leftrightarrow (\psi \vee \varphi))$	(orcom *IV*1.45)
3	$\vdash ((\varphi \wedge (\varphi \vee \psi)) \leftrightarrow (\varphi \wedge (\psi \vee \varphi)))$	2, (anbi2i *IV*1.10)
4	$\vdash (\varphi \leftrightarrow (\varphi \wedge (\psi \vee \varphi)))$	1, 3, (bitri *IV*1.24)
5	$\vdash ((\varphi \wedge (\psi \vee \varphi)) \leftrightarrow \varphi)$	4, (bicomi *IV*1.16)

□

IV 4.2 Laws of Disjunction

IV 4.2.1 Law of Excluded Middle: P or not P is tautology [exmid]

See exmid (IV 1.31).

IV 4.2.2 Law of Or-Simplification: P or P is P [oridm]

See oridm (IV 1.47).

IV 4.2.3 Law of Or-Simplification: P or true is tautology [bl.ortrr,bl.ortrl

See bl.ortrr (IV 2.11) and bl.ortrl (IV 2.12).

IV 4.2.4 Law of Or-Simplification: P or false is P [bl.orfar,bl.orfal]

See bl.orfar (IV 2.13) and bl.orfal (IV 2.14).

IV 4.2.5 Law of Or-Simplification: P or (Q and P) is P [bl.PorQandPisP, bl.PorPandQisP, bl.PisPorPandQ, bl.PisPorQandP]

Metamath Lemma 127. **bl.PorQandPisP** *Disjunction of Self Conjunction*

$$\vdash ((\varphi \vee (\psi \wedge \varphi)) \leftrightarrow \varphi) \tag{IV 4.2}$$

Proof.

1	$\vdash (\varphi \leftrightarrow (\varphi \vee (\varphi \wedge \psi)))$	(pm4.44 *IV*1.50)
2	$\vdash ((\varphi \wedge \psi) \leftrightarrow (\psi \wedge \varphi))$	(ancom *IV*1.11)
3	$\vdash ((\varphi \vee (\varphi \wedge \psi)) \leftrightarrow (\varphi \vee (\psi \wedge \varphi)))$	2, (orbi2i *IV*1.43)
4	$\vdash (\varphi \leftrightarrow (\varphi \vee (\psi \wedge \varphi)))$	1, 3, (bitri *IV*1.24)
5	$\vdash ((\varphi \vee (\psi \wedge \varphi)) \leftrightarrow \varphi)$	4, (bicomi *IV*1.16)

□

IV 4.2.6 Implication Law 1: false implies P is tautology [falim]

See falim (IV 1.32).

IV 4.2.7 Implication Law 2: true implies P is P [trant]

See trant (IV 1.58).

IV 4.2.8 Implication Law 3: P implies false is not P [bl.pifinp]

IV 4.2.9 Implication Law 4: P implies true is tautology [a1tru]

See a1tru (IV 1.7).

IV 4.3 Equality Laws

IV 4.3.1 Expression Equality: a=a [eqid]

See eqid (IV 1.30).

IV 4.3.2 Logical Equivalence: P ↔ P [biid]

See biid (IV 1.17).

IV 4.3.3 Superfluity of Equivalence: P ↔ true is P

IV 4.4 Inequality Laws

IV 4.4.1 Total Order Law: a<a is false [lntr]

See lntr (**??**).

IV 4.4.2 Partial Order Law 1: a≤a [leid]

See leid (IV 1.37).

IV 4.4.3 Partial Order Law 2: 1+a≤b is a<b

IV 4.4.4 Partial Order Law 3: a≤b-1 is a<b

IV 4.4.5 At Most Is Not Less Than: (a¡=b) = not(b¡a)

IV 4.5 Quantification Laws

IV 4.5.1 Empty Range Law: all a:t in false are V is tautology

IV 4.5.2 Empty Range Law: exists a:t in false that V is false

IV 4.5.3 Empty Range Law: (sum a:t in false of V) = 0

IV 4.5.4 Solitary Range Law: all a:t in j..j are V is V[j/a]

IV 4.5.5 Solitary Range Law: exists a:t in j..j that V is V[j/a]

IV 4.5.6 Solitary Range Law: (sum a:t in j..j of V) = V[j/a]

IV 4.5.7 Assumed Range Law: (all a:t in R are V) iff (all a:t in R are (R and V))

IV 4.5.8 Assumed Range Law: (exists a:t in R that V) iff (exists a:t in R that (R and V))

IV 4.5.9 True Body Law: all a:t in R are true is tautology

IV 4.5.10 True Body Law: exists a:t in R that true is tautology

IV 4.5.11 False Body Law: all a:t in R are false is false

IV 4.5.12 False Body Law: exists a:t in R that false is false

IV 4.5.13 Solitary Open Left Range Law: exists a:t in j,.j that V is false

IV 4.5.14 Solitary Open Right Range Law: exists a:t in j.,j that V is false

IV 4.5.15 Solitary Open Range Law: exists a:t in j,,j that V is false

IV 4.5.16 Introduction of (unused) Universal Quantification

IV 4.5.17 Introduction of (unused) Existential Quantification

IV 4.5.18 Introduction of Existential Quantification

IV 4.5.19 Replacement of Quantified Variables with #1, #2, etc.

IV 4.5.20 Moving Range Between Bound and Body

Chapter IV 5

Action Composition

IV 5.0.1 Sequential Composition Rule

The sequential composition rule implements inference rule [SCk] in I 8.5.

```
<<P1>> S1 <<Q1 and P2>>
<<Q1 and P2>> S2 <<Q2 and P3>>
. . .
<<Qk-1 and Pk>> Sk <<Qk>>
_____________________________________
<<P1>> S1 <<Q1>> ; <<P2>> S2 <<Q2>> ; . . . ; <<Pk>> Sk <<Qk>>
```

Metamath Theorem 1. **bl.sck** *Sequential Composition*

(IV 5.1)

Proof.

(IV 5.2)

□

IV 5.0.2 Concurrent Composition Rule

The concurrent composition rule implements inference rule [CCk] in I 8.6.

```
<<P>> -> <<P1>>,  <<P>> -> <<P2>>, . . . , <<P>> -> <<Pn>>
<<P1>> S1 <<Q1>>, <<P2>> S2 <<Q2>>, . . . , <<Pn>> Sn <<Qn>>
<<Q1 and Q2 and  . . . and Qn>> -> <<Q>>
-----------------------------------------
      <<P>> S <<Q>> where S is
<<P1>> S1 <<Q1>> & <<P2>> S2 <<Q2>> & . . . & <<Pn>> Sn <<Qn>>
```

Metamath Theorem 2. **bl.cck** *Concurrent Composition*

(IV 5.3)

Proof.

(IV 5.4)

□

IV 5.0.3 Alternative Rule

The alternative rule implements inference rule [IF] in I 8.7.

```
   <<P and B1>> -> <<P1>>, <<P and B2>> -> <<P2>>, . . . , <<P and Bn>> -> <<Pn>>,
   <<B1 and P1>> S1 <<Q1>>, <<B2 and P2>> S2 <<Q2>>, . . . , <<Bn and Pn>> Sn <<Qn>>,
     <<Q1>> -> <<Q>>, <<Q2>> -> <<Q>>, . . . <<Qn>> -> <<Q>>
     <<P>> -> <<B1 or B2 or ... or Bn>>
 ------------------------------------------------------------------------
<<P>> if B1-><<P1>>S1<<Q1>> [] B2-><<P2>>S2<<Q2>> [] . . . [] Bn-><<Pn>>Sn<<Qn>> fi <<Q>>
```

Metamath Theorem 3. **bl.iffi** *Alternative*

(IV 5.5)

Proof.

(IV 5.6)

□

IV 5.0.4 Iterative Rule

The iterative rule implements inference rule [LOOP] in I 8.10.

```
       P -> I
     <<I>> S <<I>>
   (I and not E) -> Q
      B>0 -> E
     B(i)>B(i+1)
 ---------------------------------------------------
<<P>> while (E) invariant <<I>> bound B { S } <<Q>>
```

Metamath Theorem 4. **bl.loop** *Iterative*

(IV 5.7)

IV 5.0.5 Do-Until Rule

The do-until rule implements inference rule [UNTIL] in I 8.10, and is defined in terms of an equivalent while loop.

```
?P? S <<I>>; while (not E) invariant <<I>> bound B \{S\} ?Q?
 ------------------------------------------------
 ?P? do invariant <<I>> bound B S until (E) ?Q?
```

Metamath Theorem 5. **bl.until** *Do-Until*

(IV 5.8)

IV 5.0.6 For-Loop Rule

The for-loop rule implements inference rule [FOR] in I 8.10, and is defined in terms of an equivalent while loop. A for-loop is nicer to prove because the bound function `ub-a` and guard `a<=ub` are made for you.

```
      lb<=ub
    <<P>>
variables a:Ideal::integer:=lb;\{while (a<=ub) invariant <<I>> bound ub-a \{S(a);a:=a+1\}\}
      <<Q>>
  ------------------------------------------------
  <<P>> for (a in lb..ub) invariant <<I>> \{S(a)\}  <<Q>>
```

Metamath Theorem 6. **bl.for** *For-Loop*

(IV 5.9)

IV 5.0.7 Existential Lattice Quantification Rule

The existential lattice quantification rule implements inference rule [ELQ] in I 8.8.

```
      <<P and x=e>> -> <<A>>
        <<A>> T <<B>>
      <<exists x:t that B>> -> <<Q>>
  ------------------------------------------------
  <<P>> declare x:t:=e; { <<A>> T <<B>> } <<Q>>
```

Metamath Theorem 7. **bl.elq** *Existential Lattice Quantification*

(IV 5.10)

IV 5.0.8 Universal Lattice Quantification Rule

The universal lattice quantification rule implements inference rule [ULQ] in **??**.

```
                    <<A>> T <<B>>
      <<P and (z in r) and (x=e) and FAT>> -> <<wp(forall,Q)>>
 ------------------------------------------------------------------------
<<P>> forall z:i in r declare variable x:t:=e; begin <<A>> T <<B>> end <<Q>>
  where wp = Q[<<all z:i in r are B>>/<<all z:i in r are A>>]
    (in postcondition Q, replace all occurances of
    \"all z:i in r are B\" with \"all z:i in r are A\"
  and FAT are \"fetch-add terms\} if any
```

Metamath Theorem 8. **bl.ulq** *Universal Lattice Quantification*

(IV 5.11)

IV 5.0.9 Asserted Action Rule

An action may have a precondition, postcondition, both, or neither. Sometimes a proof obligation of the form <<P>>S <<Q>> will have an asserted action for S= <<P1>>S1 <<Q1>>:

```
   <<P>>
<<P1>> S1 <<Q1>>
  <<Q>>
```

To write this, phantom braces will be used to show grouping:

<<P>>{<<P1>>S1 <<Q1>>} <<Q>>

Four cases:

```
   <<P>> -> <<P1>>
<<P1>> S1 <<Q1>>
  <<Q1>> -> <<Q>>
-----------------------------------
 <<P>> {<<P1>> S1 <<Q1>>} <<Q>>
```

Metamath Theorem 9. **bl.aab** *Asserted Action (both)*

(IV 5.12)

```
   <<P>> -> <<P1>>
<<P1>> S1 <<Q>>
-----------------------------------
 <<P>> {<<P1>> S1 } <<Q>>
```

Metamath Theorem 10. **bl.aapre** *Asserted Action (pre)*

(IV 5.13)

```
   <<P>>  S1 <<Q1>>
  <<Q1>> -> <<Q>>
-------------------------------
 <<P>> { S1 <<Q1>>} <<Q>>
```

Metamath Theorem 11. **bl.aapost** *Asserted Action (post)*

(IV 5.14)

```
   <<P>>  S1 <<<Q>>
-------------------------------
 <<P>> { <<Q1>>} <<Q>>
```

Metamath Theorem 12. **bl.aanone** *Asserted Action (none)*

(IV 5.15)

Chapter IV 6

Actions

IV 6.0.1 Skip Rule

Skip is implication. Skip semantics are defined by [S] in I 11.2.

$$\frac{\texttt{<<P>>} \rightarrow \texttt{<<Q>}}{\texttt{<<P>> skip <<Q>>}}$$

Metamath Theorem 13. **bl.skip** *Skip*

(IV 6.1)

IV 6.0.2 Assignment Rule

Assignment is substitution. Assignment semantics are defined by [A] in I 11.2.

$$\frac{\texttt{<<P>>} \rightarrow \texttt{<<wp(x:=e,Q)>> which is <<Q[x/e]>>}}{\texttt{<<P>> x:=e <<Q>>}}$$

Metamath Theorem 14. **bl.a** *Assignment*

(IV 6.2)

IV 6.0.3 Fetch-Add Rule

Fetch-add combines additions, and returns intermediate values, useful for interference-free shared data structures. Fetch-add semantics are defined in I 8.13.1.

```
      <<P and b in c..a-1>> -> <<Q>>
(+1)-------------------------------
      <<P and c<=a>> fetchadd(a,1,b) <<Q>>
```

Metamath Theorem 15. **bl.fap1** *Fetch-Add +1*

(IV 6.3)

```
      <<P and  b in a+1..c>> -> <<Q>>
(-1)--------------------------------
     <<P and a<=c>> fetchadd(a,-1,b) <<Q>>
```

Metamath Theorem 16. **bl.fam1** *Fetch-Add -1*

(IV 6.4)

```
      <<P and a+=e>> -> <<Q>>
(+e)-----------------------
     <<P>> fetchadd(a,e,b) <<Q>>
```

Metamath Theorem 17. **bl.fae** *Fetch-Add e*

(IV 6.5)

IV 6.0.4 Subprogram Invocation

Subprogram invocation becomes term substitution, `Q[post/pre]`. Subprogram invocation semantics [SI] are defined in **??**.

```
 <<P>>-> wp(C(e1,e2,...),Q)
 --------------------------
  <<P>> C(e1,e2,...) <<Q>>
where C is the name of a procedure;
e1,e2,... are expressions;
procedure C(p1:t1,p2:t2,...) declare x begin <<pre>> F <<post>> end
 where wp(C(e1,e2,...),Q) = Q[post/pre]
where pre and post are the precondition and postcondition of C
having substituted actuals (e1,. . .) for formals (p1,...)
```

Metamath Theorem 18. **bl.si** *Subprogram Invocation*

(IV 6.6)

IV 6.0.5 Port Output

Port output causes an event to be issued by an AADL out port.

```
(A and p@now) => B
-----------------------------
   <<A>> p! <<B>>}
```

Metamath Theorem 19. **bl.poe** *Port Output Event*

(IV 6.7)

```
(A and (p=e)@now) => B
-----------------------------
   <<A>> p!(e) <<B>>}
```

Metamath Theorem 20. **bl.pov** *Port Output Value*

(IV 6.8)

IV 6.0.6 Port Input

Needs semantics: **??**
Metamath Theorem 21. **bl.pi** *Port Input*

(IV 6.9)

IV 6.0.7 Asserted Action

Chapter IV 7

Modus Ponens

IV 7.0.1 Modus Ponens

```
X and P and (P implies Q)
-------------------------
        Q and X
```

IV 7.0.2 Modus Ponens weakening precondition [MODUS_PONENS]

```
<<P and (P implies Q)>> -> <<R>>
--------------------------------
   " <<Q and (P implies Q)>> -> <<R>>
```

IV 7.0.3 Definition of implication: A→B = not A or B

IV 7.0.4 Sequent Composition: if A→B and A→C and A→D then A→(B and C and D)

Chapter IV 8

Equality and Inequality

IV 8.0.1 Total Order Law: a<a is false

IV 8.0.2 Partial Order Law 1: a≤a by definition

IV 8.0.3 Partial Order Law 2: 1+a≤b is a<b

IV 8.0.4 Partial Order Law 3: a≤b-1 is a<b

IV 8.0.5 Equality Law: a=a by definition

IV 8.0.6 At Most Is Not Less Than: (a¡=b) = not(b¡a)

IV 8.0.7 Equality of iff: (a iff a) is tautology

IV 8.0.8 Superfluisity of iff: a iff true is a

IV 8.0.9 Superfluisity of iff: true iff a is a

Chapter IV 9

Substitution

IV 9.0.1 Substitution of Assertion Labels

IV 9.0.2 Substitution of Equals (top level)

```
<<P[a/b] and a=a>> S <<Q>>
-------------------------
 <<P and a=b>> S <<Q>>}
```

IV 9.0.3 Substitution of Equals (anywhere)

```
<<...(P[a/b] and a=a)>> S <<Q>>
-------------------------
 <<...(P and a=b)>> S <<Q>>}
```

Chapter IV 10

Combining

IV 10.0.1 Concurrent Fetchadd Rule: (all z in r are fa+=e) iff (fa=sum z in r of e)

Chapter IV 11

Algebra

IV 11.0.1 Algebra

Must discriminate everything called “algebra”.

IV 11.0.2 Subtraction of Zero: a-0 is a

IV 11.0.3 Subtraction of Added Value: (a+b)-a is b

IV 11.0.4 Remove Equivalent Term: P(a) and P(b) and a=b is P(a) and a=b

IV 11.0.5 Add Unnecessary Parentheses For No Good Reason: a = (a)

IV 11.0.6 Add Unnecessary Parentheses to Range Bound For No Good Reason: a..b = (a)..b

Chapter IV 12

Assertion Introduction

IV 12.0.1 Introduction of an Assertion to Postcondition

```
     <<P>> S <<Q>> , <<A>>
     ---------------------------------
     <<P>> S <<Q and A>>
```

IV 12.0.2 User-Defined Rule as Assertion

IV 12.0.3 Modus Ponens using Assertion on Premise

```
A implies B,    B and C implies D,
-----------------------------------
     A and C implies D}\
```

IV 12.0.4 Modus Ponens using Assertion on Consequence

```
A implies B,    C implies D and A
---------------------------------
   C implies D and B
```

IV 12.0.5 Introduction of an Assertion to Precondition

```
     <<P>> S <<Q>>
     ---------------"+
     <<P and rule>> S <<Q>>
```

IV 12.0.6 Introduction a of Term Common to Pre- and Postcondtions

```
<<P>> S <<Q>>
------------
<<P and X>> S <<Q and X>>
```

Chapter IV 13

Discrete Time

IV 13.0.1 Eternal Truth: trueˆx is true

IV 13.0.2 Eternal Falsity: falseˆx is false

IV 13.0.3 Zero Ticks Is Now: xˆ0 is x

IV 13.0.4 One-Tick Rule: xˆ1 is x'

IV 13.0.5 Double Negation: not not A is A

IV 13.0.6 Previous Tick Rule: A'ˆ-1 is A

IV 13.0.7 Moving ïotïnto ˆ: not (xˆe) is (not x)ˆe

IV 13.0.8 Eternal Number: numberˆe is number

IV 13.0.9 Hoist Caret: aˆe op bˆe is (a op b)ˆe

Chapter IV 14

Reflexivity and Associativity

IV 14.0.1 Reflexivity of Addition: a+b=b+a

IV 14.0.2 Reflexivity of Multiplication: a*b=b*a

IV 14.0.3 Reflexivity of Equality: (a=b) = (b=a)

IV 14.0.4 Reflexivity of Inequality: (a!=b) = (b!=a)

IV 14.0.5 Irreflexivity of Greater Than: (a¿b) = (b¡a)

IV 14.0.6 Irreflexivity of At Least: (a¿=b) = (b¡=a)

IV 14.0.7 Equivlence of negation and subtraction: (a-b) = (a + (-b))

IV 14.0.8 Reflexivity of Conjunction: (m and k) = (k and m)

IV 14.0.9 Reflexivity of Disjunction: (m or k) = (k or m)

IV 14.0.10 Reflexivity of Exclusive-Disjunction: (m xor k) = (k xor m)

IV 14.0.11 Associativity: (b.c).a = a.b.c

Chapter IV 15

DeMorgan´s Laws

IV 15.0.1 DeMorgan´s Law: not (A and B) = (not A) or (not B)

IV 15.0.2 DeMorgan´s Law: not (A or B) = (not A) and (not B)

IV 15.0.3 DeMorgan´s Law: not exists x:t in l..h that p = all x:t in l..h are not p

IV 15.0.4 DeMorgan´s Law: not all x:t in l..h are p = exists x:t in l..h that not p

Appendix A

Alphabetized Grammar

```
action ::=
  basic_action
  | behavior_action_block
  | alternative | for_loop
  | forall_action
  | while_loop
  | do_until_loop
  | locking_action                                          §I 8.3 p93

actual_assertion_parameter ::=
  formal_identifier : actual_assertion_expression           §I 5.4.6 p70

actual_assertion_parameter_list ::=
  actual_assertion_parameter
    { , actual_assertion_parameter }*                       §I 5.3.5 p62

actual_parameter ::= target | expression                    §I 9.8 p122

alternative ::=
  if guarded_action { [] guarded_action }+ fi
  |
  if ( boolean_expression_or_relation ) behavior_actions
  { elsif ( boolean_expression_or_relation )
    behavior_actions }*
  [ else behavior_actions ]
  end if                                                    §I 8.7 p100

array_range_list ::= natural_range { , natural_range }*     §I 4.7 p48

array_size ::= [ natural_value_constant  ]                  §I 6.3 p78

array_type ::= array [ array_range_list ] of type           §I 4.7 p48
```

```
asserted_action ::=
  [ precondition_assertion ]
  action
  [ postcondition_assertion ]
```
§1 8.2 p92

```
assertion ::=
  << ( assertion_predicate
  | assertion_function
  | assertion_enumeration
  | assertion_enumeration_invocation ) >>
```
§1 5.2 p55

```
assertion_annex_library ::=
  annex Assertion {** { assertion }+ **} ;
```
§1 5.1 p54

```
assertion_enumeration ::=
  asserion_enumeration_label_identifier :
    parameter_identifier +=>
  enumeration_pair { , enumeration_pair }*
```
§1 5.2.4 p57

```
assertion_enumeration_invocation ::=
  +=> assertion_enumeration_label_identifier
  ( actual_assertion_parameter )
```
§1 5.4.7 p71

```
assertion_expression ::=
  assertion_subexpression
    [ { + assertion_subexpression }+
    | { * assertion_subexpression }+
    | - assertion_subexpression
    | / assertion_subexpression
    | ** assertion_subexpression
    | mod assertion_subexpression
    | rem assertion_subexpression ]
  | sum logic_variables [ logic_variable_domain ]
      of assertion_expression
  | product logic_variables [ logic_variable_domain ]
      of assertion_expression
  | numberof logic_variables [ logic_variable_domain ]
      that subpredicate
```
§1 5.4 p66

```
assertion_function ::=
  [ label_identifier : [ formal_assertion_parameter_list ] ]
   := ( assertion_expression | conditional_assertion_function )
```
§1 5.2.3 p57

```
assertion_function_invocation ::=
  assertion_function_identifier
    ( [ assertion_expression |
    actual_assertion_parameter
    { , actual_assertion_parameter }* ] )
```
§1 5.4.6 p70

```
assertion_predicate ::=
  [ label_identifier : [ formal_assertion_parameter_list ] : ]
  predicate

assertion_range ::=
  assertion_subexpression range_symbol assertion_subexpression

assertion_subexpression ::=
  [ - | abs ]  timed_expression
  | assertion_type_conversion

assertion_type_conversion ::=
  ( natural | integer | rational | real | complex | time )
  parenthesized_assertion_expression

assertion_value ::=
  now | tops | timeout
  | value_constant
  | variable_name
  | assertion_function_invocation
  | port_value

assignment ::=
  variable_name [ ' ] := ( expression | record_term | any )

(for subprograms)
basic_action ::=
  skip | assignment | simultaneous_assignment | when_throw
  | subprogram_invocation

(for threads)
basic_action ::=
  skip
  | assignment
  | simultaneous_assignment
  | communication_action
  | timed_action
  | when_throw
  | combinable_operation
  | issue_exception
  | computation_action

behavior_action_block ::=
  [ quantified_variables ] { [ behavior_actions ] }
  [ timeout behavior_time ]  [ catch_clause ]

behavior_actions ::=
  asserted_action
  | sequential_composition
  | concurrent_composition
```

§I 5.2.2 p56 §I 5.3.6 p63 §I 5.4 p66 §I 5.4 p67 §I 5.4.3 p68 §I 8.4.2 p95 §I 11.2 p135 §I 8.4 p94 §I 8.8 p102 §I 8.1 p92

```
behavior_annex ::=
  [ assert { assertion }+ ]
  [ invariant assertion ]
  [ variables ]
  states { behavior_state }+
  [ transitions ]
```
§I 6.1 p74

```
behavior_state ::=
  behavior_state_identifier
     : [initial] [complete] [final] state [ assertion ] ;
```
§I 6.2 p75

```
behavior_time ::=  integer_expression unit_identifier
```
§I 8.4.4 p96

```
behavior_transition ::=
    [ behavior_transition_label : ]
    source_state_identifier { , source_state_identifier }*
    -[ [ transition_condition ] ]-> destination_state_identifier
   [ { [ behavior_actions ] } ] [ assertion ] ;
```
§I 6.4 p80

```
behavior_transition_label ::=
    transition_identifier [ [ priority_natural_literal ] ]
```
§I 6.4 p80

```
behavior_variable ::=
  local_variable_declarator { , local_variable_declarator }*
  : [ modifier ] type [ := value_constant ] [ assertion ] ;
```
§I 6.3 p78

```
case_choice ::=
  ( boolean_expression_or_relation  ) -> expression
```
§I 10.7 p130

```
case_expression ::=
  ( case_choice { , case_choice }+ )
```
§I 10.7 p130

```
catch_clause ::=
  catch ( ( exception_label : basic_action ) )+
```
§I 8.11 p108

```
combinable_operation ::=
 fetchadd
 ( target_variable_name ,
   arithmetic_expression [, result_identifier] )
 | ( fetchor | fetchand | fetchxor )
 ( target_variable_name , boolean_expression
   [, result_identifier] )
 | swap
 ( target_variable_name , reference_variable_name
   , result_identifier )
```
§I 8.13 p110

```
communication_action ::=
  subprogram_invocation
  | output_port_name ! [ ( expression ) ]
  | input_port_name ? ( target )
  | frozen_input_port_name >>
```
§I 9.1 p114

```
completion_relative_timeout_catch ::= timeout behavior_time
```
§I 7.2 p88

```
component_element_reference ::=
  subcomponent_identifier | bound_prototype_identifier
  | feature_identifier | self

computation_action ::=
  computation ( behavior_time [ .. behavior_time ] )
  [ in binding ( processor_unique_component_classifier_reference
  { , processor_unique_component_classifier_reference }+ )]

concurrent_composition ::=
  asserted_action { & asserted_action }+

conditional_assertion_expression ::=
  ( predicate ?? assertion_expression
    : assertion_expression )

conditional_assertion_function ::=
    ( condition_value_pair { , condition_value_pair }* )

conditional_expression ::=
  ( boolean_expression_or_relation ??
    expression : expression )
  |
  ( if boolean_expression_or_relation then
    expression else expression )

condition_value_pair ::=
    parenthesized_predicate ->  assertion_expression

constant_number_range ::=
  [ [-] numeric_constant .. [-] numeric_constant  ]

data_component_name ::=
  { package_identifier :: }* data_component_identifier
   [ . implementation_identifier ]

declarator ::= identifier { array_size }*

dispatch_condition ::=
  on dispatch [ dispatch_expression ] [ frozen frozen_ports ]

dispatch_conjunction ::=
   dispatch_trigger  { and dispatch_trigger }*

dispatch_expression ::=
  dispatch_conjunction  { or dispatch_conjunction }*
  | stop
  | dispatch_relative_timeout_catch
  | completion_relative_timeout_catch
  | provides_subprogram_access_identifier

dispatch_relative_timeout_catch ::= timeout

dispatch_trigger ::= in_event_port_name | in_event_data_port_name
  | port_event_timeout_catch
```

§I 10.2.2 p126 · §I 8.4.4 p96 · §I 8.6 p98 · §I 5.4.4 p69 · §I 5.4.5 p69 · §I 10.6 p129 · §I 5.4.5 p69 · §I 4.6 p46 · §I 4.4 p45 · §I 6.3 p78 · §I 7.1 p86 · §I 7.1 p86 · §I 7.1 p86 · §I 7.2 p88 · §I 7.1 p86

```
do_until_loop ::=
  do
  [ invariant assertion ]
  [ bound integer_expression ]
  behavior_actions
  until ( boolean_expression_or_relation )
```
§I 8.10.3 p107

```
enumeration_pair ::= enumeration_literal_identifier -> predicate
```
§I 5.2.4 p57

```
enumeration_type ::=
  enumeration
  ( defining_enumeration_literal_identifier
  { , defining_enumeration_literal_identifier }* )
```
§I 4.5 p45

```
event ::= < port_variable_or_state_identifier >
```
§I 5.3.10 p65

```
event_expression ::=
  [not] event
  | event_subexpression (and event_subexpression)+
  | event_subexpression (or event_subexpression)+
  | event - event
```
§I 5.3.10 p65

```
event_subexpression ::=
  [ always | never ] ( event_expression ) | event
```
§I 5.3.10 p65

```
event_trigger ::=
  in_event_subcomponent_port_reference
  | in_event_data_subcomponent_port_reference
  | ( trigger_logical_expression )
```
§I 6.7 p83

```
exception_label ::= ( exception_identifier )+ | all
```
§I 8.11 p108

```
execute_condition ::=
  boolean_expression_or_relation | timeout | otherwise
```
§I 6.5 p82

```
existential_quantification ::=
   exists logic_variables logic_variable_domain
   that predicate
```
§I 5.3.9 p64

```
expression ::=
  subexpression
  [ { + numeric_subexpression }+
  | { * numeric_subexpression }+
  |   - numeric_subexpression
  |   / numeric_subexpression
  |   mod natural_subexpression
  |   rem integer_subexpression
  |   ** numeric_subexpression
  | { and boolean_subexpression }+
  | { or boolean_subexpression }+
  | { xor boolean_subexpression }+
  |   and then boolean_subexpression
  |   or else boolean_subexpression  ]
```
§I 10.4 p128

```
expression_or_relation ::=
  subexpression [ relation_symbol subexpression ]
```
§I 10.5 p129

```
for_loop ::=
  for integer_identifier
    in integer_expression .. integer_expression
  [ invariant assertion ]
  { asserted_action }
```
§I 8.10.2 p106

```
forall_action ::=
  forall variable_identifier { , variable_identifier }*
    in integer_expression .. integer_expression
    behavior_action_block
```
§?? p??

```
formal_assertion_parameter ::=
  parameter_identifier [ ~ type_name ]
```
§I 5.2.1 p55

```
formal_assertion_parameter_list ::=
  formal_assertion_parameter
  { , formal_assertion_parameter }*
```
§I 5.2.1 p55

```
formal_expression_pair ::=
  formal_identifier => actual_expression
```
§I 10.8 p131

```
frozen_ports ::= in_port_name { , in_port_name }*
```
§I 7.1 p86

```
function_call ::=
  { package_identifier :: }*
  function_identifier ( [ function_parameters ] )
```
§I 10.8 p131

```
function_parameters ::=
  formal_expression_pair { , formal_expression_pair }*
```
§I 10.8 p131

```
guarded_action ::=
  ( boolean_expression_or_relation ) ~> behavior_actions
```
§I 8.7 p100

```
index_expression_or_range ::=
    integer_expression [ ..  integer_expression ]
```
§I 10.3 p126

```
integer_expression ::=
  [ - ]
  ( integer_assertion_value
  | ( integer_expression - integer_expression )
  | ( integer_expression / integer_expression )
  | ( integer_expression { + integer_expression }+ )
  | ( integer_expression { * integer_expression }+ ) )
```
§I 5.3.4 p61

```
internal_condition ::=
  on internal internal_port_name { or internal_port_name }*
```
§I 6.6 p83

```
issue_exception ::=
  exception
  ( [ exception_state_identifier , ] message_string_literal )
```
§I 8.4.5 p97

```
locking_action ::=
  *!< | *!>
  | required_data_access_name !<
  | required_data_access_name !>
```
§I 8.12 p109

```
logic_variable_domain ::=
  in ( assertion_expression range_symbol assertion_expression
    | predicate )
```
§I 5.3.8 p64

```
logic_variables ::=
  logic_variable_identifier { , logic_variable_identifier }*
  : type
```
§I 5.3.8 p64

```
logical_operator ::=
  and | or | xor | and then | or else
```
§I 6.7 p83

```
mode_condition ::=  on trigger_logical_expression
```
§I 6.7 p83

```
modifier ::= nonvolatile | constant |  shared |  spread | final
```
§I 6.3 p78

```
name ::=
  root_identifier { [ index_expression_or_range ] }*
    { . field_identifier { [ index_expression_or_range ] }* }*
```
§I 10.3 p126

```
natural_number ::=
 natural_integer_literal
 | natural_constant_identifier
 | natural_property
```
§I 4.7 p48

```
natural_range ::= natural_number [ .. natural_number ]
```
§I 4.7 p48

```
number_type ::=
  ( natural | integer | rational | real | complex | time )
  [ constant_number_range ]
  [ units aadl_unit_literal_identifier ]
```
§I 4.6 p46

```
numeric_constant ::= numeric_literal | numeric_property
```
§I 4.6 p46

```
parameter ::= [ formal_parameter_identifier : ] actual_parameter
```
§I 9.8 p122

§I 9.8 p122
```
parameter_list ::= parameter { , parameter }*
```

```
parenthesized_assertion_expression ::=
  ( assertion_expression )
  | conditional_assertion_expression
```
§I 5.4.2 p68
```
  | record_term
```

§I 5.3.7 p63
```
parenthesized_predicate ::= ( predicate )
```

```
port_name ::=
  { subcomponent_identifier . }* port_identifier
```
§I 9.1 p115
```
    [ [ natural_literal ] ]
```

```
port_event_timeout_catch ::=
  timeout ( port_identifier { [ or ] port_identifier }* )
```
§I 7.2 p89
```
    behavior_time
```

```
port_value ::=
```
§I 10.9 p132
```
  in_port_name ( ? | 'count | 'fresh | 'updated )
```

```
predicate ::=
  universal_quantification
  | existential_quantification
  | subpredicate
    [ { and subpredicate }+
    | { or subpredicate }+
    | { xor subpredicate }+
    | implies subpredicate
    | iff subpredicate
```
§I 5.3 p58
```
    | -> subpredicate ]
```

```
predicate_invocation ::=
  assertion_identifier
```
§I 5.3.5 p62
```
    ( [ assertion_expression | actual_assertion_parameter_list ] )
```

```
predicate_relation ::=
  assertion_subexpression relation_symbol assertion_subexpression
  | assertion_subexpression in assertion_range
```
§I 5.3.6 p62
```
  | shared_integer_name += assertion_subexpression
```

```
property ::=
```
§I 10.2.1 p125
```
  property_constant | property_reference
```

```
property_constant ::=
```
§I 10.2.1 p125
```
  property_set_identifier :: property_constant_identifier
```

```
property_field ::=
  [ integer_value ]
  | . field_identifier
  | . upper_bound
```
§I 10.2.2 p126
```
  | . lower_bound
```

§I 10.2.2 p126
```
property_name ::= property_identifier { property_field }*
```

```
property_reference ::=
  ( # [ property_set_identifier :: ]
  | component_element_reference #
  | unique_component_classifier_reference #
  | self #  )
  property_name
```
§I 10.2.2 p125

```
quantified_variables ::= declare { behavior_variable }+
```
§I 8.8 p102

```
range_symbol ::= .. | ,. | ., | ,,
```
§I 5.3.6 p63

```
record_field ::= defining_field_identifier : type ;
```
§I 4.8 p49

```
record_term ::= ( { record_value }+ )
```
§I 8.4.2 p95

```
record_type ::= record ( { record_field }+ )
```
§I 4.8 p49

```
record_value ::= field_identifier => value ;
```
§I 8.4.2 p95

```
relation_symbol ::= = | < | > | <= | >= | != | <>
```
§I 5.3.6 p63

```
sequential_composition ::=
  asserted_action { ; asserted_action }+
```
§I 8.5 p97

```
simultaneous_assignment ::=
  ( variable_name [ ' ] { , variable_name [ ' ] }+
  :=
  ( expression | record_term | any )
    { , ( expression | record_term | any ) }+ )
```
§I 8.4.3 p96

```
subcomponent_port_reference ::=
  subcomponent_identifier { . subcomponent_identifier }*
  . port_identifier
```
§I 6.7 p83

```
subexpression ::=
  [ - | not  | abs ]
  ( value | ( expression_or_relation )
  | conditional_expression | case_expression )
```
§I 10.5 p129

```
subpredicate ::=
  [ not ]
  ( true | false | stop
  | predicate_relation
  | timed_predicate
  | event_expression
  | def logic_variable_identifier )
```
§I 5.3.1 p59

```
subprogram_annex_subclause ::=
  annex Action {** subprogram_behavior **} ;
```
§I 11 p133

```
subprogram_behavior ::=
  [ assert { assertion }+ ]
  [ pre assertion ]
  [ post assertion ]
  [ invariant assertion ]
§I 11.1 p133  behavior_action_block

subprogram_invocation ::=
§I 9.8 p122  subprogram_name ( [parameter_list] )

subprogram_name ::=
  subprogram_prototype_name
  | required_subprogram_access_name
  | subprogram_subcomponent_name
  | subprogram_unique_component_classifier_reference
  | required_data_access_name . provided_subprogram_access_name
§I 9.8 p122  | local_variable_name . provided_subprogram_access_name

§I 9.1 p115  target ::= local_variable_name | output_port_name

time_expression ::=
  time_subexpression
  | time_subexpression - time_subexpression
  | time_subexpression / time_subexpression
  | time_subexpression { + time_subexpression }+
§I 5.3.3 p60  | time_subexpression { * time_subexpression }+

time_subexpression ::= [ - ]
  ( time_assertion_value
  | ( time_expression )
§I 5.3.3 p60  | assertion_function_invocation )

timed_expression ::=
    ( assertion_value
      | parenthesized_assertion_expression
      | predicate_invocation )
§I 5.4.1 p67    [ ' | ^ integer_expression | @ time_expression ]

timed_predicate ::=
  ( name | parenthesized_predicate | predicate_invocation )
§I 5.3.2 p59  [ ' | @ time_expression | ^ integer_expression ]

transition_condition ::=
  dispatch_condition
  | execute_condition
  | mode_condition
§I 6.4 p80  | internal_condition

§I 6.4 p80  transitions ::= transitions { behavior_transition }+

trigger_logical_expression ::=
§I 6.7 p83  event_trigger { logical_operator event_trigger }*
```

```
type ::=
  type_name
  | number_type
  | enumeration_type
  | array_type
  | record_type
  | variant_type
  | boolean
  | string
```
§I 4.3 p44

```
type_name ::=
  { package_identifier :: }* data_component_identifier
    [ . implementation_identifier ]
  | natural | integer | rational | real
  | complex | time | string
```
§I 5.2.1 p56

```
unique_component_classifier_reference ::=
  { package_identifier :: }* component_type_identifier
  [ . component_implementation_identifier ]
```
§I 10.2.2 p126

```
universal_quantification ::=
  all logic_variables logic_variable_domain
  are predicate
```
§I 5.3.8 p64

(for subprograms)

```
value ::=
  variable_name | value_constant | function_call
  | incoming_subprogram_parameter_identifier | null
```
§I 11.3 p135

(for threads)

```
value ::=
  now | tops | timeout | null |
  | value_constant | in mode ( { mode_identifier }+ )
  | variable_name | function_call | port_value
```
§I 10.1 p124

```
value_constant ::=
  true | false | numeric_literal | string_literal
  | property_constant | property_reference
```
§I 10.2 p125

```
variables ::= variables { behavior_variable }+
```
§I 6.3 p78

```
variant_type ::=
  variant [ discriminant_identifier ]
  ( { record_field }+ )
```
§I 4.9 p50

```
when_throw ::=
  when ( boolean_expression ) throw exception_identifier
```
§I 8.11 p108

```
while_loop ::=
  while ( boolean_expression_or_relation )
  [ invariant assertion ]
  [ bound integer_expression ]
  behavior_action_block
```
§I 8.10.1 p105

Bibliography

[1] J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, Cambridge, UK, 1996.

[2] J Barnes. *High Integrity Ada: The SPARK Approach*. Addison Wesley Longman, Reading, UK, 1997.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceedings of the Construction and Analysis of Safe, Secure, and Interoperable Smart devices CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*, pages 47–69, NewYork, NY, 2005. Springer-Verlag.

[4] M. Barnett and W. Schulte. Runtime verification of .NET contracts. *Systems and Software*, 65:199–208, 2003.

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. *The Essence of Computation: Complexity, Analysis, Transformation*, 2566:85–108, 1990.

[6] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *Proceedings International Symposium on Software Testing and Analysis*, ISSTA02, pages 123–133, NewYork, NY, 1981. ACM.

[7] R. Cartwright. Formal program testing. In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, pages 125–132, NewYork, NY, 1981. ACM.

[8] J. Chang and D. J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Software Engineering ESEC/FSE 99*, volume 1687 of *Lecture Notes in Computer Science*, pages 285–302, NewYork, NY, 1999. Springer-Verlag.

[9] Y. Cheon and G. T. Leavens. A runtime assertion checker for the java modeling language (JML). In *Proceedings of the International Conference on Software Engineering Research and Practice*, SERP02, pages 322–328. CSREA Press, 2002.

[10] A. Cimatti, M. Dorigatti, and S. Tonetta. Ocra: A tool for checking the refinement of temporal contracts. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 702–705, Nov 2013.

[11] A. Cimatti and S. Tonetta. A property-based proof system for contract-based design. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pages 21–28, Sept 2012.

[12] Darren Cofer, Andrew Gacek, Steven Miller, Michael W. Whalen, Brian LaValley, and Lui Sha. Compositional verification of architectural models. In *Proceedings of the 4th International Conference on NASA Formal Methods*, NFM'12, pages 126–140, Berlin, Heidelberg, 2012. Springer-Verlag.

[13] et.al. Cohen, E. Vcc: A practical system for verifying concurrent c. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5764 of *Lecture Notes in Computer Science*, pages 23–42, Berlin, 2009. Springer-Verlag.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI02, pages 234–245, NewYork, NY, 2002. ACM.

[15] R.W. Floyd. Assigning meanings to programs. *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, 19:19–31, 1967.

[16] Yann Glouche, Paul Le Guernic, Jean-Pierre Talpin, and Thierry Gautier. A Boolean algebra of contracts for logical assume-guarantee reasoning. Research Report RR-6570, INRIA, 2008.

[17] D. Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, NewYork, NY, 1981.

[18] David Gries. *The Science of Programming*. Springer, 1981.

[19] J. Hatcliff, G.T. Leavens, K. Leino, Rustan M., P. Müller, and M. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, 2012.

[20] E.C.R. Hehner. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, NewYork, NY, 1996.

[21] R. M. Hierons and et.al. Using formal specifications to support testing. *Computing Surveys*, 41:9:1–76, 2002.

[22] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[23] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, MA, 2006.

[24] J. Jacky, M. Veanes, C. Campbell, and W Schulte. *Model-Based Software Testing and Analysis with C#*. Formal approaches of computing and information technology. Cambridge University Press, Cambridge, UK, 2008.

[25] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5:596–619, 1983.

[26] C.B. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1990.

[27] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, NewYork, NY, 2002.

[28] L. Lamport. How to write a 21st century proof. `http://research.microsoft.com/en-us/um/people/lamport/pubs/proof.pdf`, 2012.

[29] B.R. Larson. Formal semantics for the pacemaker system specification. In *High Integrity Language Technology, HILT'14*. ACM, 2014.

[30] B.R. Larson. Safety-critical software behavior specification. In *Proceedings of the 2016 High Integrity Language Technology Workshop*, volume tbd of *tbd*, page tbd. ACM, 2015.

[31] B.R. Larson. Proving correctness of safety-critical software compositions. In *Proceedings of the 2015 Formal Techniques for Safety-Critical Systems Conference*, volume tbd of *tbd*, page tbd. ACM, 2016.

[32] B.R. Larson. Transforming safety-critical software proof outlines into proofs. In *Proceedings of the 2015 Formal Techniques for Safety-Critical Systems Conference*, volume tbd of *tbd*, page tbd. ACM, 2016.

[33] B.R. Larson, Y. Zhang, S.Cc Barrett, J. Hatcliff, and P.L. Jones. Enabling safe interoperation by medical device virtual integration. *IEEE Design and Test*, October 2015.

[34] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7:417–426, July 1981.

[35] C. Morgan. *Programming from Specifications: Second Edition.* Prentice Hall International, Hempstead, UK, 1994.

[36] C. Morgan and T. Vickers. *On the refinement calculus*. Formal approaches of computing and information technology. Springer-Verlag, NewYork, NY, 1990.

[37] J. B. Morris. Programming by successive refinement of data abstractions. *Software–Practice & Experience*, 10:241–298, 1980.

[38] B. Moszkowski. The programming language tempura. *J. Symb. Comput.*, 22(5-6):730–733, November 1996.

[39] H. Partsch and R. Steinbrggen. Program transformation systems. *Computing Surveys*, 15:199–236, 1983.

[40] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[41] SAE International. *SAE AS5506B. Architecture Analysis & Design Language (AADL)*, 2009.

[42] Boston Scientific. PACEMAKER system specification. `http://sqrl.mcmaster.ca/_SQRLDocuments/PACEMAKER.pdf`, 2007.

[43] C. Zhou and M. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. Monographs in Theoretical Computer Science. Springer-Verlag, NewYork, NY, 2004.

Index