SATX

Oscar Riveros

6 de febrero de 2026

Índice general

Parte I

SATX

CAPÍTULO 1

REFERENCE

1.1. Portada

- Nombre: SATX Full Reference.
- Versión del paquete: `0.5.8` (ver `satx/__init__.py` y `satx/stdlib.py`).
- Alcance: referencia completa y formal de API, contratos, invariantes y límites de SATX.
- Fuente de verdad: `docs/satx_technical_reference.tex` (este documento).
- Principio normativo: el código (`satx/`) implementa esta especificación; si hay discrepancia, prevalece este documento.
- Documentos complementarios (semántica y taxonomía):
- Semántica formal de SAT/#SAT, patrones y anti-patrones verificables.
- Integración simbólica (CAS/SIR).
- Reglas formales de anchos de bits, coerción y casts.
- $H10_b$
- Clasificación semántica HARD→TRANSCENDENTAL y mapeo a ejemplos del repositorio.

1.2. Convenciones y notación

- Conjuntos numéricos: $\mathbb{N}$ = 0,1,2,..., $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ y $\mathbb{B}$ = 0,1.
- Anchura de bits: w $\in \mathbb{N}^+$.
- Dominio unsigned: U_w = 0, ..., 2^w - 1.
- Dominio signed (two's complement): S_w = -2^{w-1}, ..., 2^{w-1} - 1.
- Variables SATX:

- `Unit` representa un entero en U_w o S_w según `csp.signed`.
- `Fixed` representa un racional exacto `raw/scale`, con `raw` ∈ U_w o S_w y `scale` ∈ $\mathbb{N}^+$.
- Restricción (constraint): predicado C: D → $\mathbb{B}$. Una instancia es ⟨V, D, C⟩ con V variables, D dominios y C restricciones.
- Modelo/solución: asignación M: V → D tal que $\forall C\_i \in$ C, C_i(M) = True. SAT si ∃M; UNSAT si no existe.
- CNF/DIMACS: cláusulas (∨) de literales; CNF = ∧ de cláusulas. Un literal es una variable booleana o su negación (en SATX: un entero con signo).
- Reificación: una restricción se representa por un literal ℓ tal que ℓ es verdadero ⇔ la restricción se satisface.
- Backends externos (abstractos): en ejemplos se usan los placeholders `SAT_CMD` (SAT), `COUNT_CMD` (#SAT), `MAXSAT_CMD` (MaxSAT) y `MIP_CMD` (MIP). SATX no incluye solvers.

1.3. Arquitectura del paquete (visión modular)

Mapa de módulos

- `satx/__init__.py`: fachada; re-exporta `satx.stdlib` y `satx.fixed`; expone `__version__`, `current_engine()` y submódulos `satx.ir` / `satx.sym` / `satx.solve` / `satx.explain` / `satx.maxsat_lex` / `satx.plan` / `satx.core`.
- `satx/stdlib.py`: DSL principal y utilidades de solve/#SAT.
- `satx/alu.py`: constructor CNF (ALU) y puertas lógicas/bit-vector.
- `satx/unit.py`: enteros bit-vector (`Unit`) y operadores.
- `satx/constraint.py`: objetos `Constraint` y reificación.
- `satx/fixed.py`: punto fijo (`Fixed`) y utilidades de escala.
- `satx/rational.py`, `satx/gaussian.py`: tipos auxiliares con restricciones por efecto colateral.
- `satx/gcc.py`: Global Constraint Catalog (restricciones globales de alto nivel).
- `satx/h10_b.py`: H10_b operacional (solubilidad diofántica relativizada a recursos) y compilación a SAT.
- `satx/balanced_cnf.py`: CNFs balanceadas y aritmética exacta (SAT equation).
- `satx/agency_lr.py`: utilidades numéricas de bounds tipo Lieb–Robinson (agency/blast radius).
- `satx/theory.py`: álgebra de teorías (ontología pública/privada) y conteo proyectado $\#_{Y_0}$.
- `satx/inverse.py`: preimagen y unrolling acotado (planning inverso) sobre teorías.

- `satx/core/theory.py`: teorías como objetos portables con interfaz **PUBLIC/PRIVATE**.
- `satx/ir/cnf.py`, `satx/ir/trace.py`: IR CNF enriquecida con trazabilidad (DIMACS + sidecar).
- `satx/solve/api.py`: interfaz formal para SAT/#SAT/MaxSAT/MIP (comandos abstractos) y verificación.
- `satx/explain/core.py`: explicación auditables (cores, depuración por deltas) sobre IR trazable.
- `satx/maxsat_lex.py`: frontend de MaxSAT lexicográfico (tradeoffs por niveles).
- `satx/plan/transition.py`, `satx/plan/queries.py`: planning inverso/preimagen con trazas verificables.
- `satx/ad.py`: diseño/validación/síntesis de algoritmos como ejecuciones acotadas (unrolling), propiedades temporales finitas, decoders y post-hoc check.
- `satx/lms.py`: DSL mínimo de Layered Metric Space (LMS) como teoría por capas.
- `satx/sym/`: API simbólica pública (SIR) sobre `satx.cas.ast.Atom`: constructores, `Context` y `parse`.
- `satx/cas/`: backend CAS (AST, parser y `Engine`) sin dependencias externas.

Diagrama textual de dependencias

- `satx` → `satx.stdlib` → `satx.alu` → (`satx.unit`, `satx.constraint`)
- `satx.stdlib` → (`satx.fixed`, `satx.rational`, `satx.gaussian`)
- `satx.gcc` → `satx` (API de `satx.stdlib`)
- `satx` → `satx.sym` → `satx.cas` (simbólico: SIR + CAS)

Separación de APIs

- Pública: `satx` (raíz), `satx.stdlib`, `satx.fixed`, `satx.gcc`, `satx.sym`.
- Semi-pública (avanzado): `satx.unit`, `satx.constraint`, `satx.alu`, `satx.rational`, `satx.gaussian`, `satx.cas`.
- Internos relevantes: funciones con prefijo `_` (documentadas cuando afectan contratos).

1.4. satx.ad: Algorithm Design & Adversarial Analysis

Propósito: `satx.ad` extiende SATX con infraestructura genérica para diseñar, analizar y sintetizar algoritmos discretos como *ejecuciones acotadas* y propiedades temporales finitas (sin lógicas temporales).

Estado, acción y transición

- **Estado**: registro tipado alocado por tiempo: S_t. Tipos base: booleanos como enteros 0/1 y enteros acotados (`Unit`).
- **Acción**: decisiones por paso: A_t. Puede ser un schedule determinista (constantes) o variables para síntesis.
- **Transición**: relación $T(S_t, A_t, S_{t+1})$ compilada a restricciones SATX.

Unrolling acotado

Un horizonte H se representa como:

$$\exists S_0..S_H, A_0..A_{H-1} \quad \bigwedge_{t<H} T(S_t, A_t, S_{t+1})$$

`satx.ad.unroll(...)` crea un `Run` con `run.states[t]` y `run.actions[t]` y agrega T por paso.

Propiedades temporales finitas

- `ad.always(run, P)`: seguridad (invariante acotado) $\forall t \leq H : P(S_t)$.
- `ad.eventually(run, Q)`: liveness acotado $\exists t \leq H : Q(S_t)$ (disyunción finita).
- `ad.never(run, Q)`: azúcar para `always(not Q)`.

Best-so-far

Muchos reclamos algorítmicos dependen del mejor valor alcanzado, no solo del estado final:

$$\texttt{best}[t] := \max_{u \leq t} \texttt{score}(S_u)$$

`run.best_so_far(score_fn)` construye esta señal con comparaciones y mux (`Unit.iff`) sin asumir un backend específico.

Meta vs instancia (emisión y post-hoc)

Separación crítica:

- El *meta-modelo* es la CNF/ILP generada por SATX (variables de diseño, trazas, selectores, etc.).
- La *instancia real* (CNF DIMACS, grafo, política, etc.) se obtiene *decodificando* el modelo a un objeto externo.

`Run.emit_instance(decoder, out_path)` resuelve, decodifica y emite el artefacto. `ad.posthoc_check` ejecuta un checker Python externo sobre el archivo emitido.

Simetrías: quotient oracle

`ad.quotient_oracle(oracle_fn, transforms, agg="min"/"max")` agrega un oráculo sobre una órbita de transformaciones $S \mapsto S'$ (por ejemplo, para factorizar simetrías) sin codificar nada específico del dominio.

Ejemplo mínimo (toy)

```
import satx
import satx.ad as ad

satx.engine(bits=8, strict=True)
H = 4
spec = ad.StateSpec(fields={"x": ad.Int(), "flag": ad.Bool()}, namespace="toy")

def init(S0):
    return [S0.x == 0, S0.flag == 0]

def step(S, A, Sp):
    return [Sp.x == S.x + 1, Sp.flag == S.flag]

run = ad.unroll(init, step, ad.steps(H), H, spec=spec)
run.always(lambda S: (S.x >= 0) & (S.x <= H))

# Para ejemplos chicos: backend interno (no recomendado para instancias grandes).
sat = satx.satisfy(solver="python -m satx.dpll_backend")
print(sat)
```

Nota (case study: HESS)

`satx.ad` es completamente genérico. Un posible caso de estudio es el análisis adversarial de HESS, pero la API no depende de HESS ni lo importa.

1.5. Reference

Módulo ‘satx‘

Propósito formal: fachada del paquete; re-exporta la API pública de `satx.stdlib` y `satx.fixed`.

‘___version___‘

- Identificador exacto: `satx.__version__`.
- Dominio/codominio: `str`.
- Semántica formal: constante equivalente a `satx.stdlib.VERSION`.
- Errores/excepciones: no aplica.
- Complejidad: O(1).

- Ejemplo:

```
import satx
print(satx.__version__)
```

‘current_engine()‘

- Identificador exacto: `satx.current_engine`.
- Firma: `current_engine() -> satx.alu.ALU | None`.
- Dominio/codominio: devuelve el motor global (`csp`) o `None`.
- Semántica formal:
- Precondiciones: ninguna.
- Postcondiciones: no modifica el estado; solo devuelve la referencia actual.
- Errores/excepciones: no aplica.
- Complejidad: O(1).
- Ejemplos:

```
import satx
print(satx.current_engine())  # None si no hay motor
satx.engine(bits=8)
print(satx.current_engine())  # instancia de ALU
```

```
import satx
satx.engine(bits=4)
eng = satx.current_engine()
print(eng.bits)
```

‘csp‘ (atributo dinámico)

- Identificador exacto: `satx.csp` (vía `__getattr__`).
- Dominio/codominio: `satx.alu.ALU | None`.
- Semántica formal: alias dinámico de `satx.stdlib.csp`.
- Nota de compatibilidad: `satx.__getattr__` también delega a `satx.stdlib` para símbolos no re-exportados explícitamente (incluyendo helpers con prefijo `_`).
- Errores/excepciones: `AttributeError` sólo si el símbolo no existe en `satx.stdlib`.
- Ejemplo:

```
import satx
satx.engine(bits=8)
print(satx.csp.bits)
```

Re-exports

- `satx` re-exporta todos los símbolos públicos de `satx.stdlib` y los símbolos de `satx.fixed` listados en `satx/__init__.py`.
- El contrato de cada símbolo se documenta en sus módulos correspondientes.

Módulo ‘satx.stdlib‘

Propósito formal: DSL principal para construir restricciones, generar CNF, resolver SAT/#SAT y utilidades auxiliares.

Nota global

- La mayoría de funciones llaman `check_engine()`; por compatibilidad, el comportamiento por defecto ante motor no inicializado es imprimir un mensaje y terminar el proceso (`exit(0)`).
- En modo estricto (`engine(..., strict=True)` o `Engine(strict=True)`), `check_engine(raise_=True)` y `require_engine()` lanzan `EngineNotInitializedError`.
- Mezclar objetos de motores distintos (`ALU`) dispara errores (ver funciones específicas).

Re-exports en este módulo

- `ALU`, `Unit`, `apply_constraint`, `Gaussian`, `Rational` se importan a `satx.stdlib` y forman parte del namespace.
- Sus contratos se documentan en `satx.alu`, `satx.unit`, `satx.constraint`, `satx.gaussian`, `satx.rational`.

‘version()‘

- Identificador exacto: `satx.stdlib.version`.
- Firma: `version() -> str`.
- Semántica formal: imprime información del sistema y retorna `VERSION`.
- Errores/excepciones: no aplica.
- Complejidad: O(1).
- Ejemplos:

```
import satx
v = satx.version()
print(v)
```

```
import satx
# No requiere motor
print(satx.version())
```

‘check_engine(*, raise_=None)‘

- Identificador exacto: `satx.stdlib.check_engine`.
- Firma: `check_engine(*, raise_=None) -> None`.
- Semántica formal: si `csp` es `None`, emite un mensaje de error y:
- (modo compatible) termina el proceso con código 0; o
- (modo estricto) lanza `EngineNotInitializedError`.
- El modo estricto se activa con `engine(..., strict=True)` / `Engine(strict=True)` o forzando `raise_=True`.
- Errores/excepciones: `EngineNotInitializedError` si el motor es requerido en modo estricto.
- Complejidad: O(1).
- Ejemplos:

```
import satx
satx.engine(bits=8)
satx.check_engine()  # no hace nada
```

```
import satx
try:
    satx.check_engine(raise_=True)
except satx.EngineNotInitializedError:
    print("sin motor")
```

‘EngineNotInitializedError‘

- Identificador exacto: `satx.stdlib.EngineNotInitializedError`.
- Tipo: subclase de `RuntimeError`.
- Semántica formal: se lanza cuando una operación requiere un motor inicializado y el motor no existe (modo estricto).

‘require_engine()‘

- Identificador exacto: `satx.stdlib.require_engine`.
- Firma: `require_engine() -> satx.alu.ALU`.
- Semántica formal: equivalente a `check_engine(raise_=True)` y retorna el motor global actual.
- Errores/excepciones: `EngineNotInitializedError` si no hay motor.
- Complejidad: O(1).

‘engine_ctx(*, engine=None, **kwargs)‘

- Identificador exacto: `satx.stdlib.engine_ctx`.
- Firma: `engine_ctx(*, engine=None, **kwargs) -> contextmanager`.
- Semántica formal: establece temporalmente el motor global (`csp`) dentro de un `with`.
- Si `engine=None`, crea un `Engine(**kwargs)` y lo cierra al salir del bloque.
- Si `engine` es un `Engine`, activa su `csp` y su modo estricto.
- Complejidad: O(1).
- Ejemplos:

```
import satx
with satx.engine_ctx(bits=8, strict=True):
    x = satx.integer()
    satx.add(x == 1)
```

```
import satx
eng = satx.Engine(bits=8)
with satx.engine_ctx(engine=eng):
    x = satx.integer()
```

‘class Engine‘

- Identificador exacto: `satx.stdlib.Engine`.
- Semántica formal: motor explícito (multi-engine) que encapsula un `ALU` y parámetros de compilación.
- `Engine(...)` puede usarse como context manager (`with Engine(...): ...`) o con `engine_ctx(engine=...)`.
- Invariante: está prohibido combinar valores/restricciones de motores distintos; se lanza `ValueError` al mezclar.

‘CNFPolicy‘

- Identificador exacto: `satx.stdlib.CNFPolicy`.
- Firma: `CNFPolicy(mode='none', directory=None, name=None, overwrite=True, keep_last=True, suffix='', timestamp=False, last_path=None)`.
- Dominio/codominio: estructura de configuración de artefactos CNF.
- Semántica formal:
- `resolve_path(kind: str) -> str` crea un path de salida con reglas de `mode`, `directory`, `name`, `suffix`, `timestamp`, `overwrite` y `keep_last`.

- Si `overwrite=False`, el path se vuelve único añadiendo sufijos numéricos.
- Si `keep_last=False`, elimina el último archivo generado distinto al nuevo.
- Errores/excepciones: puede propagar errores de filesystem.
- Complejidad: O(1) + operaciones de filesystem.
- Ejemplos:

```
from satx.stdlib import CNFPolicy
p = CNFPolicy(directory='cnf', name='job', timestamp=True)
path = p.resolve_path('solve')
print(path)
```

```
from satx.stdlib import CNFPolicy
p = CNFPolicy(directory='cnf', name='job', overwrite=False)
_ = p.resolve_path('solve')
_ = p.resolve_path('solve')  # genera otro nombre si ya existe
```

‘current_cnf_path()‘

- Identificador exacto: `satx.stdlib.current_cnf_path`.
- Firma: `current_cnf_path() -> str | None`.
- Semántica formal: retorna `csp.cnf` si el motor existe, o `None`.
- Complejidad: O(1).
- Ejemplo:

```
import satx
satx.engine(bits=8, cnf_path='problem.cnf')
print(satx.current_cnf_path())
```

‘last_cnf_path()‘

- Identificador exacto: `satx.stdlib.last_cnf_path`.
- Firma: `last_cnf_path() -> str | None`.
- Semántica formal: retorna la última ruta generada por la política CNF.
- Complejidad: O(1).
- Ejemplo:

```
import satx

BITS = 32

satx.engine(bits=BITS, cnf_mode='on_solve', cnf_dir='cnf')
_ = satx.satisfy(solver=SAT_CMD)
print(satx.last_cnf_path())
```

‘to_cnf(path, *, simplify=False)‘

- Identificador exacto: `satx.stdlib.to_cnf`.
- Firma: `to_cnf(path, *, simplify=False) -> None`.
- Dominio/codominio: escribe el CNF actual en `path` (DIMACS).
- Semántica formal:
- Precondición: motor inicializado.
- Postcondición: el archivo contiene todas las cláusulas activas (registro + CNF interno).
- Parámetros:
- `path`: ruta destino (str/PathLike).
- `simplify`: si `True`, aplica preprocesado CNF determinista antes de escribir (normalización, propagación de unidades y subsunción limitada).
- Errores/excepciones: `TypeError`/`ValueError` si `path` inválido; errores de filesystem.
- Complejidad: O(n_cláusulas).
- Ejemplos:

```
import satx
satx.engine(bits=4)
x = satx.integer()
satx.add(x == 1)
satx.to_cnf('out.cnf')
```

```
import satx
satx.engine(bits=4)
try:
    satx.to_cnf('')
except ValueError:
    print('path vacío')
```

‘engine(...)‘

- Identificador exacto: `satx.stdlib.engine`.
- Firma:

`engine(bits=None, deep=None, info=False, cnf_path=None, signed=False, simplify=True, fixed_default=False, fixed_scale=1, max_fixed_pow=8, width_policy="promote", width_warn=True, cnf_mode='none', cnf_dir=None, cnf_name=None, cnf_overwrite=True, cnf_keep_last=True, cnf_suffix=", cnf_timestamp=False, const_policy="strict", strict=False) -> None`.

- Dominio/codominio: inicializa el motor global `csp`.
- Semántica formal:
- Precondición: ninguna.
- Postcondición: `csp` es una nueva instancia `ALU` con parámetros configurados.
- Invariantes: `bits`, `deep`, `fixed_scale`, `max_fixed_pow` deben ser enteros positivos.
- `width_policy` ∈ {`"promote"`, `"strict"`, `"legacy"`}.
- `width_warn` es booleano.
- `const_policy` ∈ {`"strict"`, `"warn"`, `"wrap"`} (default: `"strict"`).
- `const_policy` controla qué ocurre cuando un literal no cabe en el ancho con el que se codifica: `strict` lanza `ValueError`; `warn` hace wrap y emite `UserWarning`; `wrap` hace wrap silencioso (módulo 2^w).
- `bits` fija el ancho por defecto; no es un maximo. Los `Unit` pueden definir un ancho distinto via `bits=...`.
- Errores/excepciones: `TypeError`/`ValueError` para parámetros inválidos.
- Complejidad: O(1).
- Ejemplos:

```
import satx
satx.engine(bits=8, signed=False, fixed_default=False)
```

```
import satx
try:
    satx.engine(bits=0)
except ValueError:
    print('bits inválido')
```

```
import satx
satx.engine(bits=10, const_policy="strict")
x = satx.integer()
# Error: 3007 no cabe en 10 bits (wrap a 959 si se permitiera)
satx.add(x == 3007)
```

‘integer(bits=None, scale=None, force_int=False, label=None)‘

- Identificador exacto: `satx.stdlib.integer`.
- Firma: `integer(bits=None, scale=None, force_int=False, label=None) -> satx.Unit | satx.Fixed`.
- Dominio/codominio: crea una variable entera (Unit) o fixed-point (Fixed).
- Semántica formal:
- Precondición: motor inicializado.
- Si `scale` es None y `fixed_default=True`, devuelve `Fixed` con escala por defecto.
- Si `scale` se define, devuelve `Fixed(raw, scale)`.
- `label` es un identificador humano opcional (`str | None`) para trazabilidad y debugging.
- `label` no afecta la lÃ³gica ni los encodings (CNF/WCNF/ILP) y no participa en restricciones u operaciones.
- `label` se almacena como atributo opcional `.label` en el objeto retornado (y en `raw` para `Fixed`).
- Errores/excepciones: `ValueError` si `force_int=True` y `scale` no es None.
- Complejidad: O(bits).
- Ejemplos:

```
import satx
satx.engine(bits=8)
x = satx.integer()
y = satx.integer(scale=10)
```

```
import satx
satx.engine(bits=8)
x = satx.integer(bits=8, label="counter_t3")
y = satx.integer(bits=1, label="flag_accept")
```

```
import satx
satx.engine(bits=8)
try:
    satx.integer(scale=10, force_int=True)
except ValueError:
    print('incompatible')
```

‘constant(value, bits=None, scale=None)‘

- Identificador exacto: `satx.stdlib.constant`.
- Firma: `constant(value, bits=None, scale=None) -> satx.Unit | satx.Fixed`.
- Semántica formal: crea una constante; si `scale` se define, usa `fixed_const`.
- Errores/excepciones: pueden propagarse desde `fixed_const` (p.ej. rango o tipo).
- Complejidad: O(bits).
- Ejemplos:

```
import satx
satx.engine(bits=8)
c = satx.constant(7)
f = satx.constant(1.25, scale=100)
```

```
import satx
satx.engine(bits=4, signed=False)
try:
    satx.constant(-1)
except ValueError:
    print('constante negativa en engine unsigned')
```

‘z0()‘ / ‘z1()‘ / ‘ssum(xs, start=None)‘

- Identificadores exactos: `satx.stdlib.z0`, `satx.stdlib.z1`, `satx.stdlib.ssum`.
- Firmas:
- `z0() -> satx.Unit` (constante 0 tipada).
- `z1() -> satx.Unit` (constante 1 tipada).
- `ssum(xs, start=None) -> satx.Unit | satx.Fixed`.
- Propósito: utilidades de modelado para evitar patrones repetidos y errores de tipos al sumar.
- Semántica formal:
- Precondición: motor inicializado (usa `satx.constant` internamente si `start=None`).
- `ssum` acepta cualquier iterable (`list`, `tuple`, generadores).
- `start` por defecto es `z0()` y actúa como neutro tipado.
- Si `xs` está vacío, retorna `start`.
- Errores/excepciones: `TypeError` si `xs is None`.
- Complejidad: O(n) sumas, n = longitud de `xs`.

- Ejemplo:

```
import satx
satx.engine(bits=8)
x = satx.integer()
y = satx.integer()
satx.add(satx.ssum([x, y]) >= 1)
```

‘in_range(x, lo, hi, label=None, kind="hard")‘ / ‘bin01(x, label=None, kind="hard")‘

- Identificadores exactos: `satx.stdlib.in_range`, `satx.stdlib.bin01`.
- Firmas:
- `in_range(x, lo, hi, label=None, kind="hard") -> list[ConstraintRecord]`.
- `bin01(x, label=None, kind="hard") -> list[ConstraintRecord]`.
- Semántica formal:
- `in_range` registra dos restricciones: $lo \leq x$ y $x \leq hi$.
- No crea variables nuevas: delega en `satx.add` y retorna los `ConstraintRecord` generados.
- Si `label` no es None, se generan etiquetas derivadas con sufijos `_ge<lo>` y `_le<hi>`.
- `kind` se propaga a `satx.add` y permite clasificar restricciones (por defecto `"hard"`).
- `bin01` es un caso particular: `in_range(x, 0, 1, ...)`.
- Regla de diseño: en SATX, un binario es un entero acotado por rango; no se recomienda modelar binarios como enteros de 1 bit (`bits=1`).
- Ejemplo:

```
import satx
satx.engine(bits=8)
b = satx.integer()
satx.bin01(b, label="b")
```

‘unit_le_fixed(u, x)‘ / ‘unit_ge_fixed(u, x)‘ / ‘unit_eq_fixed(u, x)‘

- Identificador exacto: `satx.stdlib.unit_le_fixed`, `satx.stdlib.unit_ge_fixed`, `satx.stdlib.unit_eq_fixed`.
- Firma: `(u: Unit, x: Fixed) -> Unit | bool`.
- Dominio/codominio: compara `u` con `x` (tras escalar `u` por `x.scale`).
- Semántica formal:

- Precondición: `u` y `x` pertenecen al mismo motor.
- Si ambos valores están asignados, retorna `bool`.
- Si no, agrega la restricción correspondiente y retorna `u` (efecto colateral).
- Errores/excepciones: `TypeError` si tipos incorrectos; `ValueError` si motores distintos.
- Complejidad: O(bits).
- Ejemplos:

```
import satx
satx.engine(bits=8)
u = satx.integer()
x = satx.fixed(scale=10)
satx.unit_le_fixed(u, x)
```

```
import satx
satx.engine(bits=8)
try:
    satx.unit_eq_fixed(satx.integer(), satx.fixed(scale=10))
except ValueError:
    print('motor distinto')
```

‘subsets(lst, k=None, complement=False)‘

- Identificador exacto: `satx.stdlib.subsets`.
- Firma: `subsets(lst, k=None, complement=False) -> (Unit, list[Unit]) | (Unit, list[Unit], list[Unit])`.
- Dominio/codominio:
- Entrada `lst`: secuencia de valores (cada elemento puede ser `Unit` o literal).
- Salida `bits`: `Unit` de longitud `len(lst)`.
- Salida `subset_`: lista de `Unit` con elementos condicionados por `bits`.
- Salida `complement_` opcional si `complement=True`.
- Semántica formal:
- Precondiciones: motor inicializado; `len(lst)` define el ancho del selector.
- Postcondiciones: para cada i, `subset_[i] = 0` si `bits[i]=0` y `subset_[i]=lst[i]` si `bits[i]=1` (codificado vía `iff`).
- Si `k` se provee: agrega la restricción $\sum_i sel_i = k$, donde $sel_i \in \{0, 1\}$ son los bits de `bits`.
- Si `complement=True`: `complement_[i]=lst[i]` cuando `bits[i]=0` y `0` cuando `bits[i]=1`.

- Errores/excepciones: no valida tipos explícitamente; errores pueden propagarse desde operaciones internas (comparaciones/sumas).
- Complejidad: O(n) variables y restricciones (n = `len(lst)`).
- Ejemplos:

```
import satx
satx.engine(bits=4)
bits, subset = satx.subsets([1, 2, 3], k=2)
```

```
import satx
satx.engine(bits=4)
bits, subset, comp = satx.subsets([10, 20], complement=True)
```

‘subset(lst, k, empty=None, complement=False)‘

- Identificador exacto: `satx.stdlib.subset`.
- Firma: `subset(lst, k, empty=None, complement=False) -> list[Unit] | tuple[list[Unit], list[Unit]]`.
- Dominio/codominio: `lst` secuencia de valores; `k` entero; retorna lista de longitud `len(lst)` con selección “a lo más k”.
- Semántica formal:
- Precondiciones: motor inicializado.
- Postcondiciones: se crean variables internas para seleccionar hasta `k` elementos de `lst`.
- Los elementos no seleccionados se igualan a `empty` (por defecto `0` en el dominio de la variable).
- Si `complement=True`, retorna además la selección complementaria (mismas posiciones con semántica inversa).
- Comportamiento inferido: el límite “a lo más k” se implementa con cláusulas de cardinalidad en `ALU.at_most_k`.
- Errores/excepciones: no valida tipos explícitamente; puede fallar si `k` no es entero o si los elementos no son compatibles.
- Complejidad: O(n choose k+1) cláusulas por `at_most_k` (n = `len(lst)`), más O(n) restricciones adicionales.
- Ejemplos:

```
import satx
satx.engine(bits=8)
subset = satx.subset([5, 6, 7, 8], k=2)
```

```
import satx
satx.engine(bits=8)
subset, comp = satx.subset([1, 2, 3], k=1, complement=True)
```

‘vector(bits=None, size=None, is_gaussian=False, is_rational=False, *, fixed=None, scale=None)‘

- Identificador exacto: `satx.stdlib.vector`.
- Firma: `vector(bits=None, size=None, is_gaussian=False, is_rational=False, *, fixed=None, scale=None)`.
- Dominio/codominio:
- Si `is_rational=True`: lista de `Rational` de longitud `size`.
- Si `is_gaussian=True`: lista de `Gaussian` de longitud `size`.
- En otro caso: lista de `Unit` o `Fixed` según `fixed/scale`.
- Semántica formal:
- Precondiciones: motor inicializado; `size` define longitud.
- Postcondiciones: cada elemento es una variable nueva (o estructura compuesta con variables nuevas).
- `fixed/scale` se resuelven vía `_resolve_fixed_params`; si `scale` se define, fuerza `Fixed`.
- Errores/excepciones: errores de tipo si `fixed/scale` inválidos (ver `_resolve_fixed_params`); puede fallar si `size` es `None`.
- Complejidad: O(size) variables y restricciones.
- Ejemplos:

```
import satx
satx.engine(bits=8)
v = satx.vector(size=3)
```

```
import satx
satx.engine(bits=8)
v = satx.vector(size=2, fixed=True, scale=10)
```

‘matrix(bits=None, dimensions=None, is_gaussian=False, is_rational=False, *, fixed=None, scale=None)‘

- Identificador exacto: `satx.stdlib.matrix`.
- Firma: `matrix(bits=None, dimensions=None, is_gaussian=False, is_rational=False, *, fixed=None, scale=None)`.
- Dominio/codominio: retorna una lista de listas con forma `dimensions=(n,m)`; cada celda es `Unit`, `Fixed`, `Rational` o `Gaussian`.
- Semántica formal:

- Precondiciones: motor inicializado; `dimensions` debe ser una tupla `(n, m)`.
- Postcondiciones: crea `n*m` variables nuevas o estructuras compuestas según flags.
- `fixed/scale` se aplican solo cuando no es racional/gaussiana.
- Errores/excepciones: no valida `dimensions` explícitamente; errores de tipo si no es indexable.
- Complejidad: O(n · m) variables y restricciones.
- Ejemplos:

```
import satx
satx.engine(bits=8)
mtx = satx.matrix(dimensions=(2, 3))
```

```
import satx
satx.engine(bits=8)
mtx = satx.matrix(dimensions=(2, 2), is_rational=True)
```

‘matrix_permutation(lst, n)‘

- Identificador exacto: `satx.stdlib.matrix_permutation`.
- Firma: `matrix_permutation(lst, n) -> tuple[list[Unit], list[Unit]]`.
- Dominio/codominio: `lst` es una matriz n×n aplanada; retorna `(xs, ys)` de longitud `n`.
- Semántica formal:
- Precondiciones: motor inicializado; `lst` debe tener longitud `n*n`.
- Postcondiciones: `xs` es una permutación de índices `0..n-1` (se imponen rango y all-different).
- Comportamiento inferido: `ys` selecciona valores de `lst` mediante `ALU.indexing` (índices basados en pares consecutivos de `xs`).
- Errores/excepciones: no valida longitud ni tipos explícitamente.
- Complejidad: O(n^2) restricciones más las de `indexing`.
- Ejemplos:

```
import satx
satx.engine(bits=4)
xs, ys = satx.matrix_permutation(list(range(9)), n=3)
```

‘permutations(lst, n)‘

- Identificador exacto: `satx.stdlib.permutations`.
- Firma: `permutations(lst, n) -> tuple[list[Unit], list[Unit]]`.
- Semántica formal:
- Precondiciones: motor inicializado.
- Postcondiciones: `xs` es una permutación de `0..n-1`; `ys[i]` es el valor de `lst` indexado por `xs[i]`.
- Errores/excepciones: no valida tipos de `lst` o `n` explícitamente.
- Complejidad: O(n^2) restricciones (all-different y elementos).
- Ejemplos:

```
import satx
satx.engine(bits=4)
xs, ys = satx.permutations([10, 20, 30], n=3)
```

‘combinations(lst, n)‘

- Identificador exacto: `satx.stdlib.combinations`.
- Firma: `combinations(lst, n) -> tuple[list[Unit], list[Unit]]`.
- Semántica formal:
- Precondiciones: motor inicializado.
- Postcondiciones: `ys[i]` selecciona un elemento de `lst` vía índice `xs[i]` sin imponer unicidad sobre `xs`.
- Errores/excepciones: no valida tipos explícitamente.
- Complejidad: O(n · |lst|) restricciones asociadas a `element`.
- Ejemplos:

```
import satx
satx.engine(bits=4)
xs, ys = satx.combinations([1, 2, 3], n=2)
```

‘all_binaries(lst)‘

- Identificador exacto: `satx.stdlib.all_binaries`.
- Firma: `all_binaries(lst) -> None`.
- Semántica formal:
- Para cada elemento `x`:
- Si `x` es `Fixed`, agrega $x.raw \in \{0, scale\}$.
- En otro caso, agrega $0 \leq x \leq 1$.
- Errores/excepciones: no valida tipos; errores pueden propagarse desde comparaciones.
- Complejidad: O(n).
- Ejemplos:

```
import satx
satx.engine(bits=4)
xs = satx.vector(size=3)
satx.all_binaries(xs)
```

‘switch(x, ith, neg=False)‘

- Identificador exacto: `satx.stdlib.switch`.
- Firma: `switch(x, ith, neg=False) -> Unit`.
- Semántica formal:
- Retorna un selector booleano codificado como `Unit` de 1 bit:
- `1` si el bit `ith` de `x` es verdadero (o falso si `neg=True`).
- `0` en caso contrario.
- Comportamiento inferido: usa `iff` para construir un mux sin fijar `x`.
- Errores/excepciones: no valida rango de `ith`.
- Complejidad: O(1) en número de cláusulas por bit.
- Ejemplos:

```
import satx
satx.engine(bits=8)
x = satx.integer()
bit0 = satx.switch(x, 0)
```

‘one_of(lst)‘

- Identificador exacto: `satx.stdlib.one_of`.
- Firma: `one_of(lst) -> Unit`.
- Semántica formal:
- Crea un vector de selección one-hot `bits` y fuerza $\sum_i bits_i = 1$.
- Retorna la suma seleccionada $\sum_i bits_i \cdot lst[i]$ (codificada con `iff`).
- Errores/excepciones: no valida tipos ni longitud vacía.
- Complejidad: O(n).
- Ejemplos:

```
import satx
satx.engine(bits=4)
choice = satx.one_of([3, 5, 7])
```

‘factorial(x)‘

- Identificador exacto: `satx.stdlib.factorial`.
- Firma: `factorial(x) -> Unit`.
- Semántica formal (comportamiento inferido):
- Crea un selector interno `sub` con $\sum_i sub_i = 1$ y $\sum_i sub_i \cdot i = x$.
- Retorna $\sum_i sub_i \cdot \prod_{j=0}^{i-1}(x - j)$ para `i` en `[1, bits-1]`.
- Errores/excepciones: no valida rango de `x` más allá del selector.
- Complejidad: O(bits^2) en términos de `csp.bits`.
- Ejemplos:

```
import satx
satx.engine(bits=6)
x = satx.integer()
fx = satx.factorial(x)
```

‘sigma(f, i, n)‘

- Identificador exacto: `satx.stdlib.sigma`.
- Firma: `sigma(f, i, n) -> Unit`.
- Semántica formal (comportamiento inferido):
- Usa un selector `sub` tal que $\sum_j sub_j = 1$ y $\sum_j sub_j \cdot j = n + 1$.
- Retorna $\sum_j sub_j \cdot \sum_{t=i}^{j-1} f(t)$.
- Errores/excepciones: no valida tipos ni rango de `n`.
- Complejidad: O(bits^2) en términos de `csp.bits`.
- Ejemplos:

```
import satx
satx.engine(bits=6)
xs = satx.vector(size=6)
res = satx.sigma(lambda t: xs[t], 0, xs[0])
```

‘pi(f, i, n)‘

- Identificador exacto: `satx.stdlib.pi`.
- Firma: `pi(f, i, n) -> Unit`.
- Semántica formal (comportamiento inferido):
- Selector `sub` con $\sum_j sub_j = 1$ y $\sum_j sub_j \cdot j = n + 1$.
- Retorna $\sum_j sub_j \cdot \prod_{t=i}^{j-1} f(t)$.
- Errores/excepciones: no valida tipos ni rango de `n`.
- Complejidad: O(bits^2) en términos de `csp.bits`.
- Ejemplos:

```
import satx
satx.engine(bits=6)
xs = satx.vector(size=6)
res = satx.pi(lambda t: xs[t], 0, xs[0])
```

‘dot(xs, ys)‘

- Identificador exacto: `satx.stdlib.dot`.
- Firma: `dot(xs, ys) -> Unit | Fixed | Rational | Gaussian`.
- Semántica formal: retorna el producto punto $\sum_i xs[i] \cdot ys[i]$ usando el motor actual.
- Errores/excepciones: no valida longitudes ni tipos; errores pueden propagarse desde multiplicaciones.
- Complejidad: O(n).
- Ejemplos:

```
import satx
satx.engine(bits=8)
a = satx.vector(size=3)
b = satx.vector(size=3)
prod = satx.dot(a, b)
```

‘mul(xs, ys)‘

- Identificador exacto: `satx.stdlib.mul`.
- Firma: `mul(xs, ys) -> list`.
- Semántica formal: retorna el producto elemento a elemento `[xs[i]*ys[i]]`.
- Errores/excepciones: no valida longitudes.
- Complejidad: O(n).
- Ejemplos:

```
import satx
satx.engine(bits=8)
xs = satx.vector(size=2)
ys = satx.vector(size=2)
prod = satx.mul(xs, ys)
```

‘values(lst, cleaner=None)‘

- Identificador exacto: `satx.stdlib.values`.
- Firma: `values(lst, cleaner=None) -> list`.
- Semántica formal: retorna `[x.value for x in lst]`; si `cleaner` se provee, filtra con `cleaner`.
- Errores/excepciones: no valida tipos; `x.value` puede ser `None` si no se ha resuelto.

- Complejidad: O(n).
- Ejemplos:

```
import satx
satx.engine(bits=4)
xs = satx.vector(size=2)
vals = satx.values(xs)
```

‘apply_single(lst, f, indexed=False)‘

- Identificador exacto: `satx.stdlib.apply_single`.
- Firma: `apply_single(lst, f, indexed=False) -> None`.
- Semántica formal:
- Si `indexed=False`: aplica `f(x)` para cada `x` en `lst`.
- Si `indexed=True`: aplica `f(i, x)`.
- Si `f` retorna `Constraint` o lista de `Constraint`, se llaman `.apply()`.
- Errores/excepciones: no valida que `f` sea callable.
- Complejidad: O(n).
- Ejemplos:

```
import satx
satx.engine(bits=4)
xs = satx.vector(size=3)
satx.apply_single(xs, lambda x: x >= 0)
```

‘apply_dual(lst, f, indexed=False)‘

- Identificador exacto: `satx.stdlib.apply_dual`.
- Firma: `apply_dual(lst, f, indexed=False) -> None`.
- Semántica formal: aplica `f(a,b)` a cada par con `i<j`; versión indexada usa `f(i,j,a,b)`.
- Errores/excepciones: no valida `f`.
- Complejidad: O(n^2).
- Ejemplos:

```
import satx
satx.engine(bits=4)
xs = satx.vector(size=3)
satx.apply_dual(xs, lambda a, b: a != b)
```

‘apply_different(lst, f, indexed=False)‘

- Identificador exacto: `satx.stdlib.apply_different`.
- Firma: `apply_different(lst, f, indexed=False) -> None`.
- Semántica formal: aplica `f(a,b)` a cada par con `i != j`; versión indexada usa `f(i,j,a,b)`.
- Errores/excepciones: no valida `f`.
- Complejidad: O(n^2).
- Ejemplos:

```
import satx
satx.engine(bits=4)
xs = satx.vector(size=3)
satx.apply_different(xs, lambda a, b: a <= b)
```

‘all_different(args)‘

- Identificador exacto: `satx.stdlib.all_different`.
- Firma: `all_different(args) -> None`.
- Semántica formal: agrega `x != y` para todo par `x,y` en `args` (i<j).
- Errores/excepciones: no valida tipos.
- Complejidad: O(n^2).
- Ejemplos:

```
import satx
satx.engine(bits=4)
xs = satx.vector(size=4)
satx.all_different(xs)
```

‘all_out(args, values)‘

- Identificador exacto: `satx.stdlib.all_out`.
- Firma: `all_out(args, values) -> None`.
- Semántica formal: agrega `x != v` para cada `x` en `args` y cada `v` en `values`.
- Errores/excepciones: no valida tipos.
- Complejidad: O(n · m).
- Ejemplos:

```
import satx
satx.engine(bits=4)
xs = satx.vector(size=3)
satx.all_out(xs, values=[0, 1])
```

‘all_in(args, values)‘

- Identificador exacto: `satx.stdlib.all_in`.
- Firma: `all_in(args, values) -> None`.
- Semántica formal: agrega `x == one_of(values)` para cada `x` en `args`.
- Errores/excepciones: no valida tipos.
- Complejidad: O(n · m).
- Ejemplos:

```
import satx
satx.engine(bits=4)
xs = satx.vector(size=3)
satx.all_in(xs, values=[0, 2, 3])
```

‘add(expr, label=None, kind="hard", soft=False, weight=1)‘ (alias ‘require‘, ‘assume‘)

- Identificador exacto: `satx.stdlib.add`, `satx.stdlib.require`, `satx.stdlib.assume`.
- Firma: `add(expr, label=None, kind="hard", soft=False, weight=1) -> ConstraintRecord | list[ConstraintRecord] | None`.
- Dominio/codominio:
- `expr` puede ser `Constraint`, `bool`, `Unit`, `Fixed`, lista/tupla o callable.
- `soft` marca la restriccion como soft para MaxSAT (default `False`).
- `weight` es el peso (int positivo) usado en MaxSAT para restricciones soft (default `1`).
- `expr` no acepta `int`/`float` crudos como restricciones (p.ej. `satx.add(1)`); use comparaciones (`x == 1`, `x >= 0`) o conectivos (`satx.any`, `satx.all`).
- Retorna `ConstraintRecord` para un único `expr`, o lista si `expr` es secuencia; `None` si `expr` es vacío.
- Semántica formal:
- Normaliza `expr` (aplana secuencias y evalúa callables).
- Cada restricción se reifica con un literal de activación `a`; se añade la cláusula $\neg a \vee \ell$ (equiv. $a \rightarrow \ell$), donde ℓ es el literal reificado de la restricción.

- Si `soft=True`, registra la restriccion como soft con su `weight` para MaxSAT. Por defecto SAT/#SAT siguen tratandola como hard (compatibilidad); para chequear solo factibilidad hard, use `satisfy(ignore_soft=True)` (o `satisfy_hard_only`), y para conteo hard-only use `counting(ignore_soft=True)`.
- Registra `ConstraintRecord` con campos `{id, label, kind, constraint, activation, soft, weight}`.
- Errores/excepciones:
- `TypeError`/`ValueError` si `label`/`kind` no son compatibles con la cantidad de restricciones.
- `TypeError` si la restricción no se puede convertir con `as_constraint`.
- Complejidad: O(m) para m restricciones (más el coste de reificación).
- Ejemplos:

```
import satx
satx.engine(bits=4)
x = satx.integer()
rec = satx.add(x >= 1, label="cota", kind="hard")
```

```
import satx
satx.engine(bits=4)
xs = satx.vector(size=2)
recs = satx.add([xs[0] != xs[1], xs[0] >= 0], soft=[False, True], weight=[1, 3])
```

‘soft(expr, weight=1, label=None, kind="soft")‘

- Identificador exacto: `satx.stdlib.soft`.
- Firma: `soft(expr, weight=1, label=None, kind="soft") -> ConstraintRecord | list[ConstraintRecord] | None`.
- Semántica formal: alias de `add(expr, label=..., kind=..., soft=True, weight=weight)`.
- Ejemplos:

```
import satx
satx.engine(bits=4)
x = satx.integer(bits=1)
satx.soft(x == 1, weight=2, label="prefer_x")
```

‘block_current(vars)‘

- Identificador exacto: `satx.stdlib.block_current`.
- Firma: `block_current(vars) -> Constraint`.
- Semántica formal:
- Precondición: cada variable en `vars` debe tener `value` asignado.
- Postcondición: agrega una cláusula que bloquea la asignación actual (disyunción de desigualdades).
- Uso típico: enumeración SAT; tras cada modelo, llamar `block_current(vars)` para excluir esa asignación en la siguiente llamada al solver.
- Anti-patrón: construir bloqueos manuales con comparaciones cuando `Unit.value` ya está asignado puede colapsar a `bool/int` y romper `satx.add`; usar `block_current`.
- Errores/excepciones: `TypeError` si `vars` no es lista/tupla o contiene elementos no SATX; `ValueError` si faltan valores.
- Complejidad: O(n).
- Ejemplos:

```
import satx

BITS = 32

satx.engine(bits=BITS)
x = satx.integer()
satx.add(x == 1)
if satx.satisfy(solver=SAT_CMD):
    satx.block_current([x])
```

‘implies(condition, constraint)‘

- Identificador exacto: `satx.stdlib.implies`.
- Firma: `implies(condition, constraint) -> Constraint`.
- Semántica formal: codifica $\neg condition \vee constraint$, reifica y aplica las cláusulas.
- Errores/excepciones: propagadas desde `as_constraint` si los tipos no son soportados.
- Complejidad: O(1) más el coste de reificación.
- Ejemplos:

```
import satx
satx.engine(bits=4)
x = satx.integer()
expr = satx.implies(x == 0, x + 1 == 1)
```

‘any(exprs)‘

- Identificador exacto: `satx.stdlib.any`.
- Firma: `any(exprs) -> Constraint`.
- Semántica formal:
- Coerción: `coerce(e) = as_constraint(e, alu=csp)`.
- Si `exprs` es iterable (y no es `str`/`bytes`), se materializa como lista finita `[e1, ..., en]`; si no, se interpreta como `[exprs]`.
- Requiere `n >= 1`.
- Retorna `OrConstraint([coerce(e1), ..., coerce(en)])`.
- No registra ni aplica cláusulas automáticamente; para imponerla usar `satx.add(...)` o `Constraint.apply()`.
- Errores/excepciones: `ValueError` si no hay operandos; `TypeError` si algún elemento no es coercible.
- Complejidad: O(n) en número de operandos, más el coste de reificación al aplicar.
- Ejemplos:

```
import satx
satx.engine(bits=4)
x = satx.integer()
y = satx.integer()
cover = satx.any([x == 1, y == 2, x + y == 3])
satx.add(cover)
```

‘all(exprs)‘

- Identificador exacto: `satx.stdlib.all`.
- Firma: `all(exprs) -> Constraint`.
- Semántica formal:
- Coerción: `coerce(e) = as_constraint(e, alu=csp)`.
- Si `exprs` es iterable (y no es `str`/`bytes`), se materializa como lista finita `[e1, ..., en]`; si no, se interpreta como `[exprs]`.
- Requiere `n >= 1`.
- Retorna `AndConstraint([coerce(e1), ..., coerce(en)])`.
- No registra ni aplica cláusulas automáticamente; para imponerla usar `satx.add(...)` o `Constraint.apply()`.

- Errores/excepciones: `ValueError` si no hay operandos; `TypeError` si algún elemento no es coercible.
- Complejidad: O(n) en número de operandos, más el coste de reificación al aplicar.
- Ejemplos:

```
import satx
satx.engine(bits=4)
x = satx.integer()
y = satx.integer()
both = satx.all([x >= 0, y >= 0, x + y <= 3])
satx.add(both)
```

‘flatten(mtx)‘

- Identificador exacto: `satx.stdlib.flatten`.
- Firma: `flatten(mtx) -> list`.
- Semántica formal: retorna una lista plana con los elementos de `mtx` (delegado a `ALU.flatten`).
- Errores/excepciones: no valida tipos ni estructura.
- Complejidad: O(n) en número total de elementos.
- Ejemplos:

```
import satx
satx.engine(bits=4)
mtx = [[satx.integer(), satx.integer()], [satx.integer()]]
flat = satx.flatten(mtx)
```

‘bits()‘

- Identificador exacto: `satx.stdlib.bits`.
- Firma: `bits() -> int`.
- Semántica formal: retorna el ancho por defecto del motor (no limita `Unit` con `bits` explícitos).
- Errores/excepciones: termina el proceso si no hay motor.
- Complejidad: O(1).
- Ejemplos:

```
import satx
satx.engine(bits=8)
print(satx.bits())
```

‘oo()‘

- Identificador exacto: `satx.stdlib.oo`.
- Firma: `oo() -> int`.
- Semántica formal: retorna el máximo representable del motor actual.
- Signed: $2^{bits-1} - 1$.
- Unsigned: $2^{bits} - 1$.
- Errores/excepciones: termina el proceso si no hay motor.
- Complejidad: O(1).
- Ejemplos:

```
import satx
satx.engine(bits=4, signed=False)
print(satx.oo())  # 15
```

‘element(item, data, *, encoding="auto", assume_in_range=True)‘

- Identificador exacto: `satx.stdlib.element`.
- Firma: `element(item, data, *, encoding="auto", assume_in_range=True) -> Unit`.
- Semántica formal:
- Crea un índice `ith` y agrega restricciones para que `item` sea igual a `data[ith]`.
- Retorna el índice `ith`.
- `encoding` selecciona el encoding interno: `"onehot"` (selector) o `"binary"` (mux por bits). `"auto"` elige `onehot` para listas pequeñas y `binary` para listas grandes.
- `assume_in_range=True` fuerza $0 \leq ith < |data|$ cuando es posible; si se viola, la teoría queda UNSAT.
- Errores/excepciones: `ValueError` si `data` está vacío o si `encoding` es inválido; `TypeError` si `assume_in_range` no es booleano.
- Complejidad: O(n) en tamaño de la lista (con constantes distintas según el encoding).
- Ejemplos:

```
import satx
satx.engine(bits=4)
x = satx.integer()
idx = satx.element(x, [10, 20, 30], encoding="binary")
```

‘index(ith, data, *, encoding="auto", assume_in_range=True)‘

- Identificador exacto: `satx.stdlib.index`.
- Firma: `index(ith, data, *, encoding="auto", assume_in_range=True) -> Unit`.
- Semántica formal:
- Crea un valor `item` y agrega restricciones para que `item == data[ith]`.
- Retorna `item`.
- `encoding`/`assume_in_range`: mismos significados que en `element`.
- Errores/excepciones: `ValueError` si `data` está vacío o si `encoding` es inválido.
- Complejidad: O(n) con n = `len(data)`.
- Ejemplos:

```
import satx
satx.engine(bits=4)
ith = satx.integer()
val = satx.index(ith, [3, 5, 7], encoding="onehot")
```

‘gaussian(x=None, y=None)‘

- Identificador exacto: `satx.stdlib.gaussian`.
- Firma: `gaussian(x=None, y=None) -> satx.Gaussian`.
- Semántica formal:
- Si `x` e `y` son `None`, crea dos `Unit` nuevos y retorna `Gaussian(x, y)`.
- Si se proporcionan, los usa directamente.
- Errores/excepciones: no valida tipos explícitamente.
- Complejidad: O(1).
- Ejemplos:

```
import satx
satx.engine(bits=4)
z = satx.gaussian()
```

‘rational(x=None, y=None)‘

- Identificador exacto: `satx.stdlib.rational`.
- Firma: `rational(x=None, y=None) -> satx.Rational`.
- Semántica formal:
- Si `x` e `y` son `None`, crea dos `Unit` nuevos y retorna `Rational(x, y)`.
- Si se proporcionan, los usa directamente.
- Errores/excepciones: no valida tipos explícitamente.
- Complejidad: O(1).
- Ejemplos:

```
import satx
satx.engine(bits=4)
r = satx.rational()
```

‘exact_div(x, y)‘

- Identificador exacto: `satx.stdlib.exact_div`.
- Firma: `exact_div(x, y) -> Unit`.
- Semántica formal:
- Convierte `x` y `y` a `Unit` si son literales.
- Alinea anchos con `resize` a $W = \max(w_x, w_y)$ usando extensión según `signed`.
- Agrega `y != 0` y construye un cociente `q` tal que `x == q * y` y el residuo es 0.
- Errores/excepciones: `TypeError`/`ValueError` pueden surgir si los tipos no son compatibles.
- Complejidad: O(bits^2) por división binaria.
- Ejemplos:

```
import satx
satx.engine(bits=6)
x = satx.integer()
q = satx.exact_div(x, 3)
```

‘at_most_k(x, k, *, encoding="auto")‘

- Identificador exacto: `satx.stdlib.at_most_k`.
- Firma: `at_most_k(x, k, *, encoding="auto") -> Unit`.
- Semántica formal: restricción de cardinalidad sobre el bit-vector `x` en el estilo legado de SATX:
- `activated`(b) se interpreta como $b = 0$ (bit desactivado) y el constraint impone:
- (i) a lo más `k` bits activados, y (ii) al menos un bit activado (selección no vacía).
- `encoding` selecciona una familia de encodings: `"auto"`, `"seq"`, `"totalizer"`, `"network"`, `"binary"`.
- Errores/excepciones: `TypeError`/`ValueError` si parámetros inválidos.
- Complejidad: depende del encoding; evita el blow-up combinatorio del encoding por combinaciones.
- Ejemplos:

```
import satx
satx.engine(bits=6)
x = satx.integer()
satx.at_most_k(x, 2, encoding="seq")
```

‘sqrt(x)‘

- Identificador exacto: `satx.stdlib.sqrt`.
- Firma: `sqrt(x) -> Unit`.
- Semántica formal: crea `y` y agrega `x == y ** 2`; retorna `y`.
- Errores/excepciones: no valida tipos explícitamente.
- Complejidad: depende de la codificación de potencia.
- Ejemplos:

```
import satx
satx.engine(bits=6)
x = satx.integer()
y = satx.sqrt(x)
```

‘hess_sequence(n, oracle, fast=False, cycles=1, target=0, seq=None)‘

- Identificador exacto: `satx.stdlib.hess_sequence`.
- Firma: `hess_sequence(n, oracle, fast=False, cycles=1, target=0, seq=None) -> list[int]`.
- Semántica formal (comportamiento inferido): heurística de búsqueda sobre permutaciones de longitud `n` que intenta minimizar `oracle(seq)`.
- Errores/excepciones: no valida tipos de `oracle` ni de `seq`.
- Complejidad: potencialmente O(cycles $\cdot n^3$) según la exploración; no está acotada estrictamente.
- Ejemplos:

```
import satx
seq = satx.hess_sequence(4, lambda s: sum(s))
```

‘hess_binary(n, oracle, fast=False, cycles=1, target=0, seq=None)‘

- Identificador exacto: `satx.stdlib.hess_binary`.
- Firma: `hess_binary(n, oracle, fast=False, cycles=1, target=0, seq=None) -> list[bool]`.
- Semántica formal (comportamiento inferido): heurística sobre bit-vectores de longitud `n` que minimiza `oracle(bits)`.
- Errores/excepciones: no valida tipos.
- Complejidad: potencialmente O(cycles $\cdot n^2$).
- Ejemplos:

```
import satx
vec = satx.hess_binary(5, lambda b: sum(b))
```

‘hess_abstract(xs, oracle, f, g, log=None, fast=False, cycles=1, target=0)‘

- Identificador exacto: `satx.stdlib.hess_abstract`.
- Firma: `hess_abstract(xs, oracle, f, g, log=None, fast=False, cycles=1, target=0) -> list`.
- Semántica formal (comportamiento inferido):
- Búsqueda heurística sobre `xs` usando transformaciones `f` y `g` (aplicadas en pares `(i,j)`).
- Minimiza `oracle(xs)`; puede llamar `log(top, opt)` si se provee.

- Errores/excepciones: no valida tipos; depende de que `f`/`g` modifiquen `xs` in-place.
- Complejidad: no acotada estrictamente; depende de `cycles` y `len(xs)`.
- Ejemplos:

```
import satx
xs = [0, 1, 2]
res = satx.hess_abstract(xs, lambda v: sum(v), lambda i,j,v: None, lambda i,j,v: None
    )
```

‘hyper_loop(n, m)‘

- Identificador exacto: `satx.stdlib.hyper_loop`.
- Firma: `hyper_loop(n, m) -> iterator[list[int]]`.
- Semántica formal: genera todas las tuplas de longitud `n` con valores en `{0,1,...,m-1}`.
- Errores/excepciones: no valida tipos de `n` y `m`.
- Complejidad: O(m^n) elementos generados.
- Ejemplos:

```
import satx
for v in satx.hyper_loop(2, 3):
    print(v)
```

‘reshape(lst, dimensions)‘

- Identificador exacto: `satx.stdlib.reshape`.
- Firma: `reshape(lst, dimensions) -> list`.
- Semántica formal: reestructura `lst` a la forma `dimensions` usando `ALU.reshape`.
- Errores/excepciones: no valida compatibilidad entre longitud y dimensiones.
- Complejidad: O(n).
- Ejemplos:

```
import satx
satx.engine(bits=4)
xs = list(range(6))
mtx = satx.reshape(xs, (2, 3))
```

‘tensor(dimensions)‘

- Identificador exacto: `satx.stdlib.tensor`.
- Firma: `tensor(dimensions) -> Unit`.
- Semántica formal: crea un `Unit` con estructura multidimensional (`deep=dimensions`) y lo registra en el motor.
- Errores/excepciones: no valida `dimensions` explícitamente.
- Complejidad: O(prod(dimensions)) para crear bits.
- Ejemplos:

```
import satx
satx.engine(bits=4)
t = satx.tensor(dimensions=(2, 2))
```

‘clear(lst)‘

- Identificador exacto: `satx.stdlib.clear`.
- Firma: `clear(lst) -> None`.
- Semántica formal: asigna `value=None` a cada variable en `lst` (soporta `Fixed` vía `raw`).
- Errores/excepciones: no valida tipos.
- Complejidad: O(n).
- Ejemplos:

```
import satx
satx.engine(bits=4)
x = satx.integer()
satx.clear([x])
```

‘rotate(x, k)‘

- Identificador exacto: `satx.stdlib.rotate`.
- Firma: `rotate(x, k) -> Unit`.
- Semántica formal: retorna un nuevo `Unit` con bits de `x` rotados `k` posiciones (mod w); $w = x.bits$ si `x` es `Unit`, o $w =$ `bits()` si es literal.
- Errores/excepciones: no valida rango de `k`.
- Complejidad: O(bits).
- Ejemplos:

```
import satx
satx.engine(bits=4)
x = satx.integer()
y = satx.rotate(x, 1)
```

‘is_probable_prime_base2(p)‘

- Identificador exacto: `satx.stdlib.is_probable_prime_base2`.
- Firma: `is_probable_prime_base2(p) -> None`.
- Semántica formal (comportamiento inferido): agrega la restricción de pseudoprimalidad 2^p mód $p = 2$ (test de Fermat base 2).
- Nota: no es una prueba de primalidad; puede aceptar pseudoprimos.
- Errores/excepciones: no valida tipos ni condiciones de coprimalidad explícitamente.
- Complejidad: depende de la codificación de potencia y módulo.
- Ejemplos:

```
import satx
satx.engine(bits=6)
p = satx.integer()
satx.is_probable_prime_base2(p)
```

‘is_probable_prime(p, *, bases=(2,3,5,7,11))‘

- Identificador exacto: `satx.stdlib.is_probable_prime`.
- Firma: `is_probable_prime(p, *, bases=(2,3,5,7,11)) -> None`.
- Semántica formal: agrega tests de Fermat para múltiples bases a: a^p mód $p = a$, para cada $a \in bases$.
- Nota: es más fuerte que base-2, pero sigue siendo un test probabilístico (no certifica primalidad).
- Errores/excepciones: `TypeError`/`ValueError` si `bases` es inválido.

```
import satx
satx.engine(bits=6)
p = satx.integer()
satx.is_probable_prime(p, bases=(2, 3, 5))
```

‘is_prime(p)‘

- Identificador exacto: `satx.stdlib.is_prime`.
- Firma: `is_prime(p) -> None`.
- Semántica formal: alias compatible hacia atrás de `is_probable_prime_base2(p)`.
- Ejemplo:

```
import satx
satx.engine(bits=6)
p = satx.integer()
satx.is_prime(p)
```

‘is_not_prime(p)‘

- Identificador exacto: `satx.stdlib.is_not_prime`.
- Firma: `is_not_prime(p) -> None`.
- Semántica formal (comportamiento inferido): agrega `p != 2` y 2^p mód $p \neq 2$.
- Errores/excepciones: no valida tipos.
- Complejidad: similar a `is_prime`.
- Ejemplos:

```
import satx
satx.engine(bits=6)
p = satx.integer()
satx.is_not_prime(p)
```

‘has_solver(name)‘

- Identificador exacto: `satx.stdlib.has_solver`.
- Firma: `has_solver(name) -> bool`.
- Semántica formal: retorna `True` si `name` se encuentra en el PATH (vía `shutil.which`).
- Errores/excepciones: `TypeError` si `name` no es `str`.
- Complejidad: O(longitud PATH).
- Ejemplos:

```
import satx
print(satx.has_solver("wsl"))
```

`satisfy(solver="", params="", log=False, *, debug=False, ignore_soft=False)`

- Identificador exacto: `satx.stdlib.satisfy`.
- Firma: `satisfy(solver="", params="", log=False, *, debug=False, ignore_soft=False) -> bool`.
- Semántica formal:
- Sea Φ una fórmula proposicional en CNF: $\Phi = \bigwedge_{j=1}^{m} C_j$, donde $C_j = \bigvee_{k=1}^{r_j} \ell_{jk}$ y cada literal ℓ_{jk} es una variable booleana x_i o su negación $\neg x_i$.
- SATX exporta Φ a DIMACS CNF y ejecuta el comando externo `solver` (concatenando `params`).
- Modos de entrada:
- (archivo) si `solver` no termina con `<` y no contiene `{cnf}`, SATX agrega la ruta CNF como último argumento.
- (plantilla) si `solver` contiene `{cnf}`, SATX sustituye `{cnf}` por la ruta CNF.
- (stdin) si `solver` termina con `<`, SATX envía el CNF por entrada estándar.
- Contrato de salida (DIMACS): el solver debe imprimir un estado `s SATISFIABLE` o `s UNSATISFIABLE`; si el estado es SAT, además debe imprimir un modelo mediante líneas `v ... 0`.
- Si el estado es SAT y hay modelo, SATX asigna valores a `csp.variables` (`Unit.value`), bloquea el modelo actual para llamadas subsecuentes, y retorna `True`.
- Si el estado es UNSAT, retorna `False`.
- Diagnóstico: si el proceso no puede ejecutarse, si la salida es vacía/no interpretable, o si reporta SAT sin modelo, lanza `RuntimeError` (no se confunde con UNSAT).
- `log=True` imprime la salida del solver en tiempo real.
- `ignore_soft=True` ignora restricciones registradas con `soft=True` (no se incluyen como activaciones hard en SAT y se fijan sus activaciones a `False`). Esto permite chequear factibilidad de las restricciones hard sin que preferencias soft causen UNSAT.
- Atajo: `satisfy_hard_only(...)` es equivalente a `satisfy(..., ignore_soft=True)`.
- Si existen restricciones soft y `ignore_soft=False`, `satisfy` emite un `UserWarning` indicando que las soft se trataran como hard para SAT.
- `debug=True` preserva los artefactos temporales (CNF y captura `.mod`); la última ruta CNF queda en `satx.csp.last_cnf_path`.
- Errores/excepciones: `TypeError`/`ValueError` en parámetros inválidos; `RuntimeError` en fallo operacional del solver.
- Complejidad: dependiente del solver externo.

- Ejemplos:

```
import satx

BITS = 32

satx.engine(bits=BITS)
x = satx.integer()
satx.add(x == 3)
if satx.satisfy(solver=SAT_CMD, debug=True, ignore_soft=True):
    print(x.value)
```

‘counting(solver="", cnf_path=None, keep_cnf=False, _path=None, extra_args=None, *, active_records=None, extra_clauses=None, ignore_soft=False)‘

- Identificador exacto: `satx.stdlib.counting`.
- Firma: `counting(solver="", cnf_path=None, keep_cnf=False, _path=None, extra_args=None, *, active_records=None, extra_clauses=None, ignore_soft=False) -> int`.
- Semántica formal:
- Genera CNF y ejecuta el backend externo provisto en `COUNT_CMD`.
- Precondición: si `_path` es `None`, entonces `solver` debe ser `str` no vacío (en caso contrario lanza `ValueError`).
- Modos de entrada (compatibles con `satisfy`):
- (archivo) si `solver` no termina con `<` y no contiene `{cnf}`, SATX agrega la ruta CNF como último argumento (después de `extra_args`).
- (plantilla) si `solver` contiene `{cnf}`, SATX sustituye `{cnf}` por la ruta CNF.
- (stdin) si `solver` termina con `<`, SATX envía el CNF por stdin (remueve el `<` antes de ejecutar) y no agrega `cnf_path` como argumento.
- Los modos `<` y `{cnf}` son excluyentes.
- Interpreta la salida DIMACS (`s mc <count>` u otras variantes).
- Si el resultado es UNSAT, retorna `0`; si hay conteo, retorna `count`.
- `ignore_soft=True` ignora restricciones soft (no se incluyen como hard) y fija sus activaciones a `False` para evitar multiplicidad en el conteo debida a activaciones libres.
- Si no se puede interpretar, lanza `RuntimeError`.
- Errores/excepciones:

- `TypeError/ValueError` en parámetros inválidos.
- `RuntimeError` si el contador no reporta el conteo.
- Complejidad: dependiente del solver externo.
- Ejemplos:

```
import satx

BITS = 32

satx.engine(bits=BITS)
x = satx.integer()
satx.add(x == 1)
count = satx.counting(solver=COUNT_CMD)
```

‘maxsat(vars=None, *, constraints=None, exclude=None, kinds=None, solver="", wcnf_path=None, keep_wcnf=False, _path=None, extra_args=None)‘

- Identificador exacto: `satx.stdlib.maxsat`.
- Firma: `maxsat(vars=None, *, constraints=None, exclude=None, kinds=None, solver="", wcnf_path=None, keep_wcnf=False, _path=None, extra_args=None) -> dict`.
- Semántica formal:
- Exporta WCNF (DIMACS weighted) y ejecuta un solver MaxSAT externo.
- Las restricciones son hard por defecto; las soft se registran con `soft=True` o `satx.soft(...)`.
- Para cada restricción soft, SATX usa su literal de activación como relajación: agrega una cláusula soft `(activation)` con peso `weight`.
- `opt_cost` es la suma de pesos de las soft violadas (equivalente a minimizar el peso total pagado por desactivar activaciones).
- Unweighted MaxSAT es el caso particular `weight=1` (p.ej. `satx.soft(expr)`).
- Los hard se escriben con peso `top`, donde `top = sum(weights) + 1`.
- Modos de entrada (WCNF):
- (archivo) si `solver` no termina con `<` y no contiene `{wcnf}`/`{cnf}`, SATX agrega la ruta WCNF como último argumento (después de `extra_args`).
- (plantilla) si `solver` contiene `{wcnf}` (o `{cnf}`), SATX sustituye el marcador por la ruta WCNF.
- (stdin) si `solver` termina con `<`, SATX envía el WCNF por stdin (remueve el `<` antes de ejecutar) y no agrega `wcnf_path` como argumento.

- Los modos `<` y `{wcnf}`/`{cnf}` son excluyentes.
- Parsea la salida tipo MaxSAT (`s`, `o`, `v`) y retorna `{status, opt_cost, model}`.
- `status` toma valores `OPTIMUM`, `UNSAT` o `UNKNOWN`.
- `model` devuelve valores en el mismo formato que `synthesize` para `vars`.
- Si `vars` es `None`, retorna valores para todas las variables registradas en el motor.
- Errores/excepciones: `TypeError`/`ValueError` en parámetros inválidos; si la salida no es interpretable retorna `status="UNKNOWN"`.
- Ejemplos:

```
import satx

BITS = 32

satx.engine(bits=BITS)
x = satx.integer(bits=1)
satx.add(x == 1, soft=True, weight=2)
result = satx.maxsat(x, solver=MAXSAT_CMD)
print(result)
```

MIP en SATX: exportación (LP / MPS): ‘to_mip(...)‘ (alias ‘export_mip(...)‘)

- Identificadores exactos: `satx.stdlib.to_mip` (alias `satx.stdlib.export_mip`).
- Firma: `to_mip(*, format="lp", objective=None, sense="min", path=None) -> str | path`.
- Principio de diseño: SATX no resuelve MIP; solo exporta un modelo portable (no importa ni ejecuta solvers).
- Implementación interna (para uso avanzado): el submódulo `satx.mip` centraliza la IR (variables, restricciones, objetivo), los exportadores (LP/MPS) y el parser genérico de salida de solvers.
- Salida: modelo determinista en formato LP o MPS (estándar) con secciones completas (objetivo, restricciones, bounds, variables binarias/enteras).
- Convención: los booleanos se modelan como enteros con bounds $[0, 1]$; si el modelo impone $[0, 1]$, se exportan como `Binary`.
- Subset soportado (por ahora):
- Variables creadas con `satx.integer()`.
- Expresiones lineales: suma de coeficientes enteros por variables + constante (`+`, `-`, negación, y `*` solo con constante entera).

- Restricciones lineales: `<=`, `>=`, `==`.
- Conectivos (fragmento lineal): `satx.all` (AND) y `satx.any` (OR) cuando sus operandos son comparaciones lineales (o ANDs de comparaciones).
- Negación (fragmento lineal): `(lhs == rhs)` se lineariza como disyunción $(lhs \leq rhs - 1) \vee (lhs \geq rhs + 1)$.
- Implicación (fragmento lineal): `(cmp -> conj)` cuando el antecedente es una comparación `<=/>=` y el consecuente es una conjunción de comparaciones.
- Objetivo lineal opcional (`sense="min"` o `"max"`); si `objective=None`, exporta factibilidad (objetivo 0).
- No soportado (lanza `UnsupportedForMIPError` con certificado): `element`, `among`, `in_/is_in`, desigualdades estrictas `</>`, `!=` directo (`x != y`), y expresiones no lineales.
- Semántica de `path`: si `path=None`, retorna el string del modelo; si no, escribe el archivo y retorna el path.
- Ejemplo mínimo:

```
import satx
satx.engine(bits=4)
x = satx.integer()
y = satx.integer()
satx.add(0 <= x); satx.add(x <= 1)
satx.add(0 <= y); satx.add(y <= 1)
satx.add(x + y <= 1)
lp = satx.to_mip(format="lp", objective=x + 2*y, sense="min")
print(lp)
satx.to_mip(format="mps", objective=x + 2*y, sense="min", path="model.mps")
```

MIP en SATX: integración genérica 'mip(...)'

- Identificador exacto: `satx.stdlib.mip`.
- Firma: `mip(vars, *, objective=None, sense="min", format="mps", solver=None, workdir=None, keep_files=False, timeout=None) -> dict`.
- Semántica formal:
- Exporta el modelo MIP (LP/MPS) con el mismo subset lineal soportado por `to_mip`.
- Si `solver=None`, no ejecuta nada y retorna el modelo exportado en `raw_output`.
- Si `solver` se provee, ejecuta el comando externo de forma genérica (SATX no depende ni asume ningún solver MIP concreto) y captura stdout/stderr.
- Modos de ejecución:
- STDIN MODE: si `solver` termina en `<`, SATX envía el modelo por stdin (remueve el `<` antes de ejecutar).

- FILE MODE: si `solver` contiene `{model_file}`, SATX escribe el modelo a un archivo temporal y reemplaza el marcador por la ruta real (use comillas en `solver` si la ruta puede contener espacios).
- Parser (fase 1): interpreta la salida buscando:
- `"Value of objective function: <number>"` (objetivo opcional),
- `"Actual values of the variables:"` seguido de líneas `<var_name> <value>`.
- Retorno (compatible hacia atrás): diccionario con campos mínimos `{status, objective, model, raw_output}` y campos adicionales para alineación estable:
- `status`: `"ok"|"infeasible"|"error"`.
- `objective`: número o `None`.
- `model`: `dict {nombre_exportado: valor}` filtrado a las variables solicitadas en `vars` (p.ej. `x0000001`).
- `raw_output`: stdout/stderr combinado (o el modelo exportado si `solver=None`).
- `vars`: lista de variables solicitadas (en el mismo orden provisto por el usuario).
- `var_names`: lista de nombres exportados (LP/MPS) en el mismo orden que `vars`; si `status != "ok"`, es `None`.
- `model_list`: lista de valores alineada 1:1 con `vars`; para cada i, corresponde a `var_names[i]`. Si el solver no reporta un valor, se usa `None`. Si `status != "ok"`, es `None`.
- `model_map`: diccionario alineado `{str(vars[i]): model_list[i]}`; para variables MIP (creadas con `satx.integer()`), `str(var)` coincide con el nombre exportado (`x0000001`, etc.). Si `status != "ok"`, es `None`.
- Si `status="ok"` y el parser entrega valores enteros para una variable solicitada, SATX asigna `Unit.value` (p.ej. para poder inspeccionar `[xi.value for xi in vars]`).
- Errores/excepciones: `UnsupportedForMIPError` si el modelo usa constructos no compatibles; `ValueError` si `solver` no especifica un modo (`<` o `{model_file}`).
- Ejemplo mínimo (FILE MODE):

```
import satx

BITS = 32

satx.engine(bits=BITS)
x = satx.integer()
satx.add(0 <= x); satx.add(x <= 1)
result = satx.mip([x], objective=x, sense="min", format="mps", solver=MIP_CMD)
print(result["status"], result["objective"], result["model"])
```

‘explain_unsat(constraints=None, *, kinds=None, labels=None, solver="", params="", log=False)‘

- Identificador exacto: `satx.stdlib.explain_unsat`.
- Firma: `explain_unsat(constraints=None, *, kinds=None, labels=None, solver="", params="", log=False) -> list[ConstraintRecord] | None`.
- Semántica formal:
- Si el conjunto es SAT, retorna `None`.
- Si es UNSAT, retorna un subconjunto reducido de `ConstraintRecord` (core) por eliminación greedy.
- No garantiza minimalidad.
- `labels` (opcional) filtra candidatas por etiqueta antes de extraer el core.
- Errores/excepciones: propagados desde `_resolve_constraint_records` y `_solve_sat`.
- Complejidad: O(m^2) llamadas al solver en el peor caso (m = número de restricciones candidatas).
- Ejemplos:

```
import satx

BITS = 32

satx.engine(bits=BITS)
x = satx.integer()
ca = satx.add(x == 0, label="a")
cb = satx.add(x == 1, label="b")
core = satx.explain_unsat(constraints=[ca, cb], solver=SAT_CMD)
```

‘unsat_core(constraints=None, *, kinds=None, labels=None, solver="", params="", log=False)‘

- Identificador exacto: `satx.stdlib.unsat_core`.
- Firma: `unsat_core(constraints=None, *, kinds=None, labels=None, solver="", params="", log=False) -> list[ConstraintRecord] | None`.
- Semántica formal: alias de `explain_unsat(...)`.

‘relax(constraints=None, *, kinds=None, solver="", params="", log=False, limit=None, objective="count", kind_priority=None)‘

- Identificador exacto: `satx.stdlib.relax`.
- Firma: `relax(constraints=None, *, kinds=None, solver="", params="", log=False, limit=None, objective="count", kind_priority=None) -> list[ConstraintRecord]`.
- Semántica formal: elimina restricciones de forma greedy (core-guided) hasta obtener SAT; retorna las eliminadas.
- `objective` controla qué candidato se elimina primero desde un core: `"count"` (por id), `"weight"` (por peso), o `"kind"` (prioridad por tipo con `kind_priority`).
- Errores/excepciones: `TypeError`/`ValueError` si `limit` es inválido.
- Complejidad: O(m^2) en el peor caso (m = restricciones candidatas).
- Ejemplos:

```
import satx

BITS = 32

satx.engine(bits=BITS)
x = satx.integer()
ca = satx.add(x == 0)
cb = satx.add(x == 1)
dropped = satx.relax(solver=SAT_CMD)
```

‘suggest_fixes(...)‘

- Identificador exacto: `satx.stdlib.suggest_fixes`.
- Firma: `suggest_fixes(constraints=None, *, kinds=None, solver="", params="", log=False, limit=None) -> list[ConstraintRecord]`.
- Semántica formal: alias de `relax`.
- Ejemplos:

```
import satx

BITS = 32

satx.engine(bits=BITS)
x = satx.integer()
fixes = satx.suggest_fixes(solver=SAT_CMD)
```

‘synthesize(vars, *, constraints=None, exclude=None, kinds=None, solver="", params="", log=False, block=False)‘

- Identificador exacto: `satx.stdlib.synthesize`.
- Firma: `synthesize(vars, *, constraints=None, exclude=None, kinds=None, solver="", params="", log=False, block=False) -> Any | list[Any] | None`.
- Semántica formal:
- Resuelve con el backend provisto en `SAT_CMD` y devuelve los valores de `vars`.
- `constraints` puede seleccionar subconjuntos por id/label/record.
- `exclude` elimina restricciones de la búsqueda.
- Si `block=True`, bloquea el modelo obtenido.
- Retorna `None` si UNSAT.
- Errores/excepciones: `TypeError` si `vars` contiene tipos no soportados.
- Complejidad: dependiente del solver externo.
- Ejemplos:

```
import satx

BITS = 32

satx.engine(bits=BITS)
x = satx.integer()
satx.add(x == 2)
val = satx.synthesize(x, solver=SAT_CMD)
```

‘exposure(event, *, given=None, constraints=None, kinds=None, solver="", extra_args=None, normalize=True, project=None, internal_max_vars=128, approx=False, pivot=16, trials=9, seed=0)‘

- Identificador exacto: `satx.stdlib.exposure`.
- Firma: `exposure(event, *, given=None, constraints=None, kinds=None, solver="", extra_args=None, normalize=True, project=None, internal_max_vars=128, approx=False, pivot=16, trials=9, seed=0) -> dict`.
- Semántica formal:
- Sea Φ la CNF inducida por las restricciones activas (filtradas por `constraints`/`kinds`).
- Sea `G(given)` el conjunto de cláusulas unitarias que fijan los bits de cada `Unit`/`Fixed` en `given` a sus valores actuales (`value`).
- Sea `E` el literal de reificación del evento (`E = reify(event)`).

- Si `project is None`, computa $total = \#(\Phi \wedge G)$ y $count = \#(\Phi \wedge G \wedge E)$ (conteo completo) vía un contador externo provisto en `solver`.
- Si `project` se provee, computa conteos proyectados $\#_P$ sobre los bits de `project` usando un backend interno (solver-free) con límite `internal_max_vars`.
- `approx=True` habilita una aproximación por hashing XOR determinista (controlada por `pivot`/`trials`/`seed`); en este caso se retorna `meta` con trazas del conteo.
- Si `normalize=True`, retorna `ratio = count / total` (y `0` si `total=0`).
- `given` implementa fijación de valores, no condicionamiento lógico: no acepta `Constraint`; para condicionar por una restricción `C`, registrar `C` en Φ con `satx.add(C)` y usar `exposure` sin `given` (o además fijar valores si aplica).
- Nota: el conteo completo incluye variables auxiliares del encoding; el modo `project=` implementa una proyección explícita sobre variables de alto nivel.
- Errores/excepciones: `TypeError` si `normalize` no es `bool`; errores de `counting` y de fijación (`given`) pueden propagarse.
- Complejidad: depende del contador externo o es exponencial en `internal_max_vars` en el backend interno.
- Ejemplos:

```
import satx

satx.engine(bits=1, strict=True)
x = satx.integer(bits=1, force_int=True)
z = satx.integer(bits=1, force_int=True)
satx.implies(x == 1, z == 0)
info = satx.exposure(x == 1, project=x)
```

‘distribution(expr, values=None, *, given=None, constraints=None, kinds=None, solver="", extra_args=None, normalize=True, project=None, internal_max_vars=128, approx=False, pivot=16, trials=9, seed=0)‘

- Identificador exacto: `satx.stdlib.distribution`.
- Firma: `distribution(expr, values=None, *, given=None, constraints=None, kinds=None, solver="", extra_args=None, normalize=True, project=None, internal_max_vars=128, approx=False, pivot=16, trials=9, seed=0) -> dict`.
- Semántica formal:
- Sea Φ la CNF inducida por las restricciones activas (filtradas por `constraints`/`kinds`) y sea $G(given)$ la fijación de valores (como en `exposure`).
- Si `expr` es booleano (`Constraint` o `bool`), cuenta para cada $v \in values$ (por defecto $\{True, False\}$):

- $count[v] = \#(\Phi \wedge G \wedge (expr = v))$.
- Si `expr` es no-booleano, requiere `values` explícito y cuenta:
- $count[v] = \#(\Phi \wedge G \wedge (expr == v))$.
- Retorna `{"total": total, "counts": dist}` y, si `normalize=True`, agrega `ratios[v] = count[v]/total` (o `0` si `total=0`).
- Errores/excepciones: `ValueError` si `values` no se provee para expresiones no booleanas.
- Si `project is None`, el conteo depende de un contador externo; si `project` se provee, usa conteo proyectado interno (limitado por `internal_max_vars`).
- Ejemplos:

```
import satx

satx.engine(bits=1, strict=True)
x = satx.integer(bits=1, force_int=True)
dist = satx.distribution(x, values=[0, 1], project=x)
```

‘wmc(weights, *, constraints=None, kinds=None, given=None, internal_max_vars=128, preprocess=True)‘

- Identificador exacto: `satx.stdlib.wmc`.
- Firma: `wmc(weights, *, constraints=None, kinds=None, given=None, internal_max_vars=128, preprocess=True) -> Fraction`.
- Semántica formal: conteo pesado (WMC) exacto usando un backend interno (DPLL) sobre la CNF activa; retorna un `fractions.Fraction`.
- `weights` es un diccionario $\{var\_id \mapsto (w_F, w_T)\}$ sobre variables DIMACS; pesos faltantes se tratan como $(1, 1)$.
- Límite: si la CNF excede `internal_max_vars`, lanza `RuntimeError` para evitar explosión combinatoria.

‘check_witness(model, *, constraints=None, kinds=None, labels=None)‘

- Identificador exacto: `satx.stdlib.check_witness`.
- Firma: `check_witness(model, *, constraints=None, kinds=None, labels=None) -> dict`.
- Semántica formal: valida un testigo en formato DIMACS (lista de literales con signo) contra las restricciones registradas a nivel alto en el motor actual.
- Retorna `{"ok": bool, "failures": [...] }`; cada falla incluye `id`, `label` y `kind` (y `error` si hubo excepción al evaluar).

- Uso: auditoría post-solución y verificación determinista de invariantes del modelo.

```
import satx
from satx.cnf_utils import dpll_solve

with satx.Engine(bits=2, signed=False, strict=True):
    x = satx.integer(bits=2, force_int=True)
    satx.add(x <= 2, label="bnd")
    ir = satx.ir.from_engine(preprocess=False)
    model = dpll_solve(int(ir.num_vars), [list(c) for c in ir.clauses])
    chk = satx.check_witness(model)
    assert chk["ok"]
```

‘reset()‘

- Identificador exacto: `satx.stdlib.reset`.
- Firma: `reset() -> None`.
- Semántica formal:
- Cierra el archivo CNF si está abierto y elimina el archivo en disco si existe.
- Reinicia el estado `render`.
- Errores/excepciones: ignora errores de borrado.
- Complejidad: O(1) + coste de E/S.
- Ejemplos:

```
import satx
satx.engine(bits=4)
satx.reset()
```

Módulo ‘satx.sym‘

Propósito formal: exponer el Symbolic IR (SIR) de SATX como un único tipo (`SymExpr := satx.cas.ast.Atom`) y un contexto de evaluación sin estado global (`Context`), delegando las transformaciones al `satx.cas.core.Engine`.

‘SymExpr‘

- Identificador exacto: `satx.sym.SymExpr`.
- Dominio/codominio: alias de `satx.cas.ast.Atom`.
- Semántica formal: no hay wrappers ni tipos paralelos; el SIR es exactamente el AST del CAS.

Constructores (‘var‘, ‘const‘, ‘app‘, ‘tensor‘, ‘parse‘)

- `var(name: str) -> SymExpr`: símbolo `Sym(name)`.
- `const(value) -> SymExpr`: constante/coerción a átomo del CAS (por ejemplo `int -> Rational`, `float -> Double`, `Fraction -> Rational`).
- `app(op: str, *args) -> SymExpr`: aplicación prefijo `(op arg1 arg2 ...)`.
- `tensor(shape, elems) -> Tensor`: tensor n-D con `elem` plano row-major y `len(elem)=prod(shape)`.
- `parse(src: str) -> list[SymExpr]`: re-export del parser del CAS (`satx.cas.parser.parse`).

‘Context‘

- Identificador exacto: `satx.sym.Context`.
- Propósito: contenedor aislado de un `satx.cas.Engine`; no usa el `satx.stdlib.csp` global.
- Campo: `engine`.
- `bindings` (propiedad): mapa `name -> Atom` con bindings actuales de símbolos de usuario (excluye keywords del CAS).
- Métodos: `eval`, `simplify`, `float`, `subst`, `derivative`, `integral`, `defint`, `taylor`, `get_component`, `set_component`.
- Gestión de estado: `set(name, value)` y `with_bindings(...)`.

Conversión (‘to_python‘, ‘as_float‘)

- `to_python(expr)`: baja `Double/Rational/Tensor/Sym` a valores de Python (best-effort).
- `as_float(expr)`: convierte `Double/Rational` a `float`.

Puente CAS→FD (‘compile_fd‘, ‘constraint‘, ‘hash_sir‘)

- `hash_sir(expr)`: hash estable (SHA-256) basado en prefijo canA3nico, para caches/memoizaciA3n.
- `compile_fd(expr, Gamma, bits_default=..., policy=...)` → `(term, side_constraints, report)`.
- `constraint(expr, Gamma, simplify=True, policy=...)` → `(Constraint, side_constraints, report)` y aplica `side_constraints`.
- Tipos en `Gamma`: `Bool`, `Unit(bits, signed, arith)`, `Fixed(bits, scale, signed, arith)`, `Tensor(shape, base_type)`.
- Fallos: retorna `Unsupported` (no excepciA3n) fuera de Σ_{FD}.

- Trazas: `Context.capture_trace()` y `policy={"trace": True}` (ver capA-tulo SIMBA"LICO).

```
from fractions import Fraction

from satx.sym import Context, app, const, parse, tensor, to_python, var

ctx = Context()
x = var("x")

expr = app("+", app("^", x, const(2)), app("*", const(3), x), const(1))
assert ctx.engine.to_prefix(expr) == "(+ (^ x 2) (* 3 x) 1)"

ast = parse("-x + 1/2")[0]
assert ctx.engine.to_prefix(ast) == "(+ (* -1 x) (* 1 (^ 2 -1)))"

t = tensor((2, 2), [const(1), const(2), const(3), const(4)])
assert to_python(ctx.float(t)) == ((1.0, 2.0), (3.0, 4.0))
```

Módulo ‘satx.cas‘

Propósito formal: backend CAS embebido en SATX (stdlib-only), basado en un AST Lisp-like (`satx.cas.ast`) y un evaluador único con dispatch por keywords (`satx.cas.core.Engine`).

‘Engine‘

- Identificador exacto: `satx.cas.Engine`.
- Semántica formal: sesión de evaluación con tablas globales de bindings/funciones (modelo de “REPL” acumulativo).
- API clave:
- `run(src) -> str`: evalúa un programa y retorna texto de salida.
- `run_buf(src) -> None`: evalúa sin recolectar salida.
- `eval(expr) -> Atom`: evalúa un átomo AST en el ambiente actual.
- `to_prefix(expr) -> str`: imprime una forma prefijo canónica.
- `get_component/set_component`: indexación tensorial 1-based (ver keyword `[`).
- Keywords: tabla en `satx.cas.core.KEYWORD_TO_METHOD` (incluye `simplify`, `float`, `derivative`, `integral`, `defint`, `taylor`, `[`, `factorial`).
- Plugins: `register_keyword(...)` permite registrar keywords adicionales con metadatos (arity/purity/partiality/doc) sin modificar el nA§cleo.

‘CASError‘

- Identificador exacto: `satx.cas.CASError`.
- Semántica formal: error semántico del motor (por ejemplo `defint: unsolved integral`).

CLI (`python -m satx.cas`)

- Entrada: `python -m satx.cas` (ver `satx/cas/cli.py`).
- Semántica: evalúa expresiones, archivos `.em` o stdin; incluye REPL simple.

Módulo ‘satx.ir‘

Propósito formal: IR auditable y estable para snapshots de CNF (DIMACS) y validación determinista de testigos, sin depender de backends externos.

‘IRVar‘

- Identificador exacto: `satx.ir.IRVar`.
- Semántica formal: variable de alto nivel (nombre + bits + signed) representada por un bloque de literales DIMACS (LSB-first).

‘CNFIR‘

- Identificador exacto: `satx.ir.CNFIR`.
- Campos: `num_vars`, `clauses`, `vars`, `metadata`.
- Métodos clave: `to_dimacs()/from_dimacs()`, `to_json()/from_json()`, `semantic_hash()`, `validate_dimacs_model(model)`, `decode_units(model)`.

‘from_engine(engine=None, *, include_activations=True, preprocess=False, include_encoding_metadata=False)‘

- Identificador exacto: `satx.ir.from_engine`.
- Semántica formal: snapshot del motor actual a una `CNFIR` (CNF + mapeo canónico best-effort para `Unit`).

```
import satx

with satx.Engine(bits=2, signed=False, strict=True):
    x = satx.integer(bits=2, force_int=True)
    satx.add(x <= 2)
    ir = satx.ir.from_engine(preprocess=True)
    dimacs = ir.to_dimacs()
    ir2 = satx.CNFIR.from_dimacs(dimacs)
    assert ir2.num_vars == ir.num_vars
```

IR trazable (CNF + sidecar)

La clase `CNFIR` es un snapshot mínimo (CNF + mapeo best-effort de `Unit`). Para auditoría y explicación se define un IR enriquecido y determinista que preserva:

- un *VarMap* explícito (símbolo ↔ id DIMACS ↔ metadatos),
- *TaggedClause* (cláusula + tags + origen),
- serialización dual: DIMACS estándar + *sidecar* JSON mínimo para trazas,
- hash semántico determinista del IR (independiente del backend).

'VarMap'

- Identificador exacto: `satx.ir.VarMap`.
- Semántica formal: estructura $\mathcal{V}$ que define una biyección parcial entre símbolos y variables booleanas DIMACS.
- Interpretación: un *símbolo* es una cadena estable (`"x0[0]"`, `"x0[1]"`, ...) y un *id* es un entero $i \in \{1, \ldots, n\}$ correspondiente a una variable DIMACS.
- Invariantes:
  - (i) si $\mathcal{V}(s) = i$ y $\mathcal{V}(s') = i$ entonces $s = s'$ (inyectividad),
  - (ii) si $\mathcal{V}^{-1}(i) = s$ entonces $\mathcal{V}(s) = i$ (coherencia),
  - (iii) todo i referenciado por el *sidecar* satisface $1 \leq i \leq n$.
- Metadatos: $\mathcal{V}$ puede adjuntar información JSON-serializable por símbolo (por ejemplo: rol **PUBLIC/PRIVATE**, nombre de variable de alto nivel, índice de bit, etc.).

'TaggedClause'

- Identificador exacto: `satx.ir.TaggedClause`.
- Semántica formal: una cláusula etiquetada es un triple

$$K = (C,\ \text{tags}(K),\ \text{origin}(K)),$$

  donde C es una cláusula DIMACS (lista finita de literales distintos de 0), $\text{tags}(K)$ es un conjunto finito de etiquetas (strings) y $\text{origin}(K)$ es un objeto JSON que identifica el origen (por ejemplo label de restricción, paso temporal, etc.).
- Invariante DIMACS: cada literal $\ell \in C$ cumple $\ell \neq 0$ y $|\ell| \leq n$.

‘CNF‘ (IR trazable)

- Identificador exacto: `satx.ir.CNF`.
- Campos: `num_vars`, `clauses` (tupla de `TaggedClause`), `varmap` (`VarMap`), `metadata`.
- Semántica formal: $\Phi = \bigwedge_{j=1}^{m} C_j$ donde cada C_j es la proyección DIMACS de `clauses[j]` ignorando tags/origen. Tags y origen son *no lógicos*.
- Serialización:
  - DIMACS: `CNF.to_dimacs()` escribe únicamente Φ.
  - Sidecar: `CNF.to_sidecar_json()` escribe un JSON canónico con (i) `varmap`, (ii) la lista de cláusulas con `lits`/`tags`/`origin`, y (iii) `metadata`.
- Hash determinista: `CNF.semantic_hash()` es SHA-256 sobre el JSON canónico de una forma normal:
  - literales normalizados por cláusula (sin 0; orden canónico),
  - cláusulas ordenadas lexicográficamente por (`lits`, `tags`, `origin`),
  - tags ordenadas,
  - `metadata` excluida por defecto (incluirla es opcional y explícito).

‘from_engine_trace(engine=None, *, preprocess=False)‘

- Identificador exacto: `satx.ir.from_engine_trace`.
- Semántica formal: snapshot del motor actual a un `CNF` trazable, incluyendo tags/orígenes mínimos por cláusula cuando estén disponibles.
- Límite: si no existe información de origen/cláusula en el motor, el IR resultante puede contener tags vacíos; esto no afecta satisfacibilidad, pero limita explicación.

Compatibilidad

- `CNFIR` permanece como IR mínimo y estable (legacy).
- `CNF` es el IR trazable recomendado para auditoría/explicación.

Módulo ‘satx.core.theory‘

Propósito formal: teoría como objeto matemático portable con interfaz **PUBLIC/PRIVATE** y hashing determinista.

‘Theory‘

- Identificador exacto: `satx.core.theory.Theory` (alias de `satx.Theory`).
- Semántica formal: ver sección `FDAS` (semántica pública $\mathrm{Mod}_{\mathrm{pub}}$, proyección π_{pub} y operadores).
- Inmutabilidad: `Theory` es un snapshot inmutable; toda operación retorna una nueva teoría.

Operaciones (álgebra)

- Identificadores exactos: `satx.meet`, `satx.join`, `satx.exists` (`hide`), `satx.project`, `satx.rename`, `satx.lift`, `satx.not_full`, `satx.not_public`.
- Higiene: toda composición respeta α-renaming de variables privadas para evitar colisiones.

Módulo ‘satx.solve.api‘

Propósito formal: interfaz solver-agnostic para resolver/verificar instancias CNF con salidas auditables (testigos y, cuando existan, certificados).

‘check_model(cnf, assignment)‘

- Identificador exacto: `satx.solve.api.check_model`.
- Semántica formal: retorna `True` sii $\mu \models \Phi$, donde μ es una asignación booleana total/parcial representada como lista de literales o mapping id→bool.
- Garantía: determinista; no usa backends externos.

‘solve_sat(cnf, *, SAT_CMD=None, want_proof=True, ...)‘

- Identificador exacto: `satx.solve.api.solve_sat`.
- Semántica formal: intenta decidir SAT/UNSAT para Φ. Si SAT, retorna un testigo μ verificable con `check_model`.
- Certificados: si `want_proof=True`, el resultado puede incluir un certificado UNSAT (si el backend lo produce) o un certificado interno (sólo para instancias pequeñas).
- No-garantía: si el backend externo no soporta pruebas y la instancia excede el límite interno, SATX puede retornar `UNSAT` sin certificado verificable (esto debe estar explicitado en el artefacto/evidencia).

‘verify_unsat_proof(proof, cnf)‘

- Identificador exacto: `satx.solve.api.verify_unsat_proof`.
- Semántica formal: retorna `True` sii el certificado `proof` implica Φ UNSAT bajo el formato soportado.
- Límite: si el formato de prueba no es soportado por SATX, retorna `False` (no se inventan verificaciones).

‘wmc(cnf, weights, *, COUNT_CMD=None, ...)‘

- Identificador exacto: `satx.solve.api.wmc`.
- Semántica formal (WMC): dado un mapeo de pesos $w_i(0), w_i(1)$ por variable booleana x_i, define
$$\mathrm{WMC}(\Phi) = \sum_{\mu \models \Phi} \prod_{i=1}^{n} w_i(\mu(x_i)).$$
- Garantía: para instancias pequeñas, SATX puede computar WMC exacto con backend interno (enumeración/DPLL); para instancias grandes, la interfaz permite delegación abstracta a `COUNT_CMD` si existe un backend compatible.

Módulo ‘satx.explain.core‘

Propósito formal: explicar inconsistencia en términos auditables (*cores*) usando tags de IR.

‘unsat_core(cnf, tags)‘

- Identificador exacto: `satx.explain.core.unsat_core`.
- Entrada: una CNF trazable `CNF` y un conjunto opcional de tags candidatos.
- Semántica formal: retorna un subconjunto $U \subseteq$ tags tal que la sub-CNF inducida por U (más cláusulas sin tags) es UNSAT.
- Fallback (instancias pequeñas): si no hay soporte de core en backend, SATX puede usar *delta debugging* (eliminación) sobre tags, usando un SAT interno, limitado por tamaño ($\leq$ mundo finito configurable).
- Salida: core como lista determinista de tags; el mapeo tag→origen se obtiene del sidecar.

Módulo ‘satx.maxsat_lex‘

Propósito formal: MaxSAT lexicográfico como lenguaje nativo de tradeoffs (sin fijar solver).

Definición (hard/soft y niveles)

Sea Φ_H una CNF *hard*. Sea $\mathcal{S}$ un multiconjunto de cláusulas *soft* con nivel y peso:

$$\mathcal{S} = \{(\ell, w, C)\}, \quad \ell \in \mathbb{N},\ w \in \mathbb{N}^+,\ C \text{ cláusula}.$$

Para un modelo μ de Φ_H, definimos el costo por nivel:

$$\mathrm{cost}_\ell(\mu) = \sum_{(\ell,w,C)\in\mathcal{S}} w \cdot \mathbf{1}[\mu \not\models C].$$

El orden lexicográfico minimiza el vector $(\mathrm{cost}_0, \mathrm{cost}_1, \dots)$.

‘lex_optimize(hard, soft, *, MAXSAT_CMD=None, ...)‘

- Identificador exacto: `satx.maxsat_lex.lex_optimize`.
- Semántica formal: retorna un modelo μ que satisface Φ_H y es mínimo en el orden lexicográfico inducido por cost_ℓ.
- Reducción (cuando aplica): se puede reducir a Weighted MaxSAT por escalamiento determinista de pesos:
  - Sea $W_\ell = \sum w$ el máximo costo posible en el nivel ℓ.
  - Defina factores $K_{L-1} = 1$ y $K_\ell = (W_{\ell+1} + 1) \cdot K_{\ell+1}$ para $\ell = L-2, \ldots, 0$.
  - Reemplace cada soft (ℓ, w, C) por peso $w \cdot K_\ell$.
- Salida auditable: además de μ, retorna violaciones por nivel (cláusulas/tags rotas) y costos $\text{cost}_\ell(\mu)$.
- Fallback: para instancias pequeñas, se permite enumeración exhaustiva de modelos (sin backend externo).

Módulo ‘satx.plan.transition‘

Propósito formal: representar transiciones finitas $T(s, a, s')$ como teorías y habilitar unrolling con trazas verificables.

Definición (Transición)

Una transición es una relación booleana finita:

$$T : \text{Assign}(S) \times \text{Assign}(A) \times \text{Assign}(S) \to \{0, 1\},$$

representada por un objeto `Transition` que fija los nombres de variables de estado S, acción A y estado siguiente S', junto con una teoría relacional $\varphi(S, A, S')$.

Unrolling

Para horizonte K, el unrolling es:

$$\exists S_0..S_K, A_0..A_{K-1} \bigwedge_{t<K} \varphi(S_t, A_t, S_{t+1}).$$

Módulo ‘satx.plan.queries‘

Propósito formal: queries canónicas de planning inverso (reachable/count/preimage/best) sobre transiciones finitas.

‘preimage(T, G)‘

- Semántica formal (1 paso):

$$\mathrm{Pre}(G)(s) = \exists a, s'.\ T(s, a, s') \wedge G(s').$$

- Salida: teoría sobre S (posiblemente con privadas auxiliares) que caracteriza estados iniciales que alcanzan G en 1 paso.

‘reachable(T, I, G, K)‘

- Semántica formal: retorna SAT/UNSAT de la fórmula de unrolling con condición inicial $I(S_0)$ y meta $G(S_K)$.
- Testigo: si SAT, retorna una traza $(S_0, A_0, S_1, \ldots, S_K)$ y un checker paso a paso.

‘count_plans(T, I, G, K)‘

- Semántica formal: interfaz para conteo de planes (exacto o abstracto) sobre el conjunto de secuencias $A_0..A_{K-1}$ que admiten alguna ejecución satisfaciendo I y G.
- Backend: puede delegar a conteo proyectado sobre variables de acción.

‘best_plan(T, I, G, K, costs)‘

- Semántica formal: optimización sobre el espacio de planes alcanzables, expresada como MaxSAT lexicográfico (costos por nivel).
- Salida: traza óptima + desglose auditable de violaciones/costos por nivel.

Módulo ‘satx.architecture‘

Propósito formal: capa de alto nivel para integración determinista tipo “pirámide” sobre teorías finitas, con:

- bloques con frontera **PUBLIC/PRIVATE**,
- adaptadores (renombrado + glue) para compatibilidad de interfaces,
- gates verificables (`SAT`, ∀, # proyectado, robustez) sin fijar solvers concretos,
- políticas CI (reglas) y reportes auditables,
- store FDAS por filesystem (cache + provenance) indexado por `semantic_hash()`.

Tipos finitos: ‘FiniteType‘

Dominio finito. Un tipo finito fija una codificación por bits:

$$\text{FiniteType} = (b, \text{signed}), \qquad b \in \mathbb{N}^+,\ \text{signed} \in \{0, 1\}.$$

Su dominio $\mathcal{D}$ es:

$$\mathcal{D} = \begin{cases} U_b = \{0, \ldots, 2^b - 1\} & \text{si unsigned,} \\ S_b = \{-2^{b-1}, \ldots, 2^{b-1} - 1\} & \text{si signed (two's complement).} \end{cases}$$

Contrato público: ‘InterfaceSpec‘

Una interfaz fija el conjunto de variables observables Y y su tipado finito:

$$I = (\text{name}, \text{version}, Y, \mathcal{D}, \iota),$$

donde $\iota(Y)$ es una fórmula (invariante) *sólo* sobre Y.

- Identificador exacto: `satx.architecture.core.InterfaceSpec`.
- Invariante: `types` define $\mathcal{D}(y)$ para todo $y \in Y$ (dominios finitos obligatorios).
- Restricción: `invariants` (la fórmula ι) no puede referenciar variables fuera de Y.

Bloque: ‘Block‘

Un bloque es una teoría con frontera:

$$B = (\text{name}, I, Z, \varphi, \text{requires}, \text{guarantees}),$$

donde Z es el conjunto de privadas y $\varphi(Y, Z)$ es el cuerpo del bloque.

- Identificador exacto: `satx.architecture.core.Block`.
- Convención: `requires` y `guarantees` son fórmulas *sólo* sobre Y.
- Teoría asociada: $\Phi_B = (X, \varphi \wedge \iota \wedge \text{requires}, Y, Z)$ con $X = Y \uplus Z$.
- Sellado (`seal`): la interfaz pública del bloque es su semántica observacional:

$$\text{Mod}_{\text{pub}}(B) \;:=\; \{\, \nu : Y \to \mathcal{D} \;:\; \exists \omega : Z \to \mathcal{D} \;\; (\nu \uplus \omega) \models \varphi \wedge \iota \wedge \text{requires} \,\}.$$

Adaptador: ‘Adapter‘

Un adaptador transforma un bloque fuente (interfaz I_s) a una interfaz destino I_t:

$$A = (\text{name}, I_s, I_t, \rho, \psi),$$

donde ρ es un renombrado $\rho : Y_s \to Y_t$ (total sobre Y_s) y $\psi(Y_t)$ es el *glue*.

- Identificador exacto: `satx.architecture.core.Adapter`.
- Totalidad: ρ debe cubrir todas las variables públicas de I_s.
- Compatibilidad de tipos: para cada $y \in Y_s$, $\mathcal{D}_s(y)$ y $\mathcal{D}_t(\rho(y))$ deben coincidir.
- Semántica: adapt(B, A) renombra el bloque y conjoin ψ (más sellado a I_t).

Capa y pirámide: ‘Layer‘ y ‘Pyramid‘

- `Layer`: colección finita de bloques + adaptadores, con modo `meet` o `join`.
- `Pyramid`: composición secuencial de capas sobre una interfaz base I_0.
- `compile()`: retorna una teoría compuesta Φ sobre Y_0, respetando higiene (α-renaming) de privadas.

Composición segura

- `meet`: conjunción semántica (intersección de mundos públicos), con α-renaming automático de privadas para evitar colisiones.
- `join`: disyunción semántica (unión de mundos públicos), también con higiene.
- Compatibilidad: dos teorías son compatibles si comparten el mismo conjunto tipado de públicas (o existe un adaptador explícito).

Gates verificables

Sean Φ, Ψ teorías sobre la misma interfaz pública Y.

- `gate_sat`: decide si existe un modelo completo $\mu \models \Phi$ (retorna testigo público cuando SAT).
- `gate_projected_count`: computa $\#_{Y_0}(\Phi)$ para $Y_0 \subseteq Y$ (exacto en mundos finitos pequeños; interfaz abstracta para backends).
- `gate_robustness`: robustez $r = \#_{Y_0}(\Phi)/|\mathrm{Assign}(Y_0)|$ y falla si $r < \tau$.
- `gate_forall`: para una propiedad $P(Y)$, verifica $\forall y \in \mathrm{Assign}(Y)\ (\iota(y) \Rightarrow P(y))$. Equivalentemente, UNSAT de $\iota \wedge \neg P$.
- `gate_no_public_growth`: controla regresión pública:

$$\mathrm{Mod}_{\mathrm{pub}}(\Psi) \subseteq \mathrm{Mod}_{\mathrm{pub}}(\Phi) \quad \Longleftrightarrow \quad \Psi \wedge \neg_{\mathrm{pub}}\Phi \text{ es UNSAT.}$$

En todos los casos, SATX no fija solvers: cualquier backend externo se proporciona vía placeholders (`SAT_CMD`, `COUNT_CMD`, `MAXSAT_CMD`, `MIP_CMD`) o se usa un backend interno sólo para instancias pequeñas (auditoría/tests).

Políticas CI

Una política Π es una función determinista que inspecciona una pirámide y retorna un resultado:

$$\Pi : \mathrm{Pyramid} \to (\mathrm{ok}, \mathrm{severity}, \mathrm{message}, \mathrm{evidence}).$$

Incluye, entre otras: consistencia SAT global, no crecimiento público, robustez mínima en APIs críticas, invariantes de interfaz, y totalidad de adaptadores.

Store FDAS por filesystem: ‘FDASStore‘

- Identificador exacto: `satx.architecture.fdas_store.FDASStore`.
- Claves: `Theory.semantic_hash()`, `Query.semantic_hash()`, `Result.semantic_hash()`.
- Persistencia: serialización JSON canónica (ordenada) y cache por hash.
- Provenance: metadatos que vinculan teorías a bloques/adaptadores/políticas que las produjeron.

Módulo ‘satx.fixed‘

Propósito formal: representación de decimales de punto fijo mediante `value = raw / scale`, donde `raw` es `Unit`.

‘class Fixed‘

- Identificador exacto: `satx.fixed.Fixed`.
- Firma: `Fixed(raw: Unit, scale: int)` (dataclass inmutable).
- Dominio/codominio:
- `raw`: `Unit` del motor actual.
- `scale`: entero positivo ($scale \in \mathbb{N}^+,\ scale > 0$).
- Semántica formal:
- Invariante: `scale` es positivo y representable como entero en el motor (`_check_scale_fits_unit`).
- `value` (propiedad): retorna `Fraction(raw.value, scale)` si hay modelo, en otro caso `None`.
- Operaciones aritméticas usan `raw` y preservan escala; no hay redondeo implícito.
- Comparaciones retornan `Constraint` si no hay valores, o `bool` si hay valores.
- Errores/excepciones:
- `TypeError` si `raw` no es `Unit` o si `scale` no es `int` positivo.
- `ValueError` en operaciones con escalas o motores incompatibles.
- Complejidad: O(1) para construcción; operaciones dependen del tamaño de `raw`.
- Métodos relevantes:
- `debug_repr()`: representación detallada (`Fixed(raw=..., scale=...)` o `Fixed(value=...)`).
- `__str__`: si `scale` es potencia de 10, imprime decimal; si no, imprime fracción; si no hay valor, `<?>`.
- Ejemplos:

```
import satx
satx.engine(bits=8)
x = satx.fixed(scale=100)
print(x)  # "<?>" antes de resolver
```

```
import satx
satx.engine(bits=8)
x = satx.fixed_const(1.25, scale=100)
print(x.debug_repr())
```

‘fixed(*, scale=100, bits=None)‘

- Identificador exacto: `satx.fixed.fixed`.
- Firma: `fixed(*, scale: int = 100, bits: int | None = None) -> Fixed`.
- Semántica formal: crea un `Unit` nuevo y lo envuelve en `Fixed` con escala `scale`.
- Errores/excepciones: `TypeError`/`ValueError` si `scale` no es positivo.
- Complejidad: O(bits).
- Ejemplos:

```
import satx
satx.engine(bits=8)
f = satx.fixed(scale=10)
```

‘fixed_const(x, *, scale=100)‘

- Identificador exacto: `satx.fixed.fixed_const`.
- Firma: `fixed_const(x: NumberLike, *, scale: int = 100) -> Fixed`.
- Semántica formal:
- Codifica `raw = round(x * scale)` (o exacto si `x` es `int`/`Fraction`).
- Verifica que `raw` cabe en el motor actual.
- Errores/excepciones:
- `TypeError` si `x` es `bool` o tipo no soportado.
- `ValueError` si `Fraction` no es representable exactamente o si `raw` no cabe en el motor.
- Complejidad: O(1).
- Ejemplos:

```
import satx
satx.engine(bits=8)
f = satx.fixed_const(1.25, scale=100)
```

```
import satx
from fractions import Fraction
satx.engine(bits=8)
try:
    satx.fixed_const(Fraction(1, 3), scale=10)
except ValueError:
    print("no representable")
```

‘fixed_lcm_scale(a, b)‘

- Identificador exacto: `satx.fixed.fixed_lcm_scale`.
- Firma: `fixed_lcm_scale(a: Fixed, b: Fixed) -> tuple[int, int, int]`.
- Semántica formal: retorna `(lcm, lcm/a.scale, lcm/b.scale)`.
- Errores/excepciones: `TypeError` si no son `Fixed`; `ValueError` si motores distintos.
- Complejidad: O(1).
- Ejemplos:

```
import satx
satx.engine(bits=8)
a = satx.fixed(scale=10)
b = satx.fixed(scale=25)
print(satx.fixed_lcm_scale(a, b))
```

‘fixed_rescale_to(x, target_scale)‘

- Identificador exacto: `satx.fixed.fixed_rescale_to`.
- Firma: `fixed_rescale_to(x: Fixed, target_scale: int) -> Fixed`.
- Semántica formal:
- Si `target_scale` es múltiplo de `x.scale`, multiplica el `raw` por el factor.
- Si `x.scale` es múltiplo de `target_scale`, requiere divisibilidad exacta (restricción) y divide.
- Si ninguno es múltiplo del otro, lanza `ValueError`.
- Errores/excepciones: `TypeError` si `x` no es `Fixed`; `ValueError` en incompatibilidad o división no exacta.
- Complejidad: O(bits).
- Ejemplos:

```
import satx
satx.engine(bits=8)
x = satx.fixed(scale=10)
y = satx.fixed_rescale_to(x, 100)
```

‘fixed__add__rescaled(a, b, *, target__scale=None)‘

- Identificador exacto: `satx.fixed.fixed_add_rescaled`.
- Firma: `fixed_add_rescaled(a: Fixed, b: Fixed, *, target_scale: int | None = None) -> Fixed`.
- Semántica formal: suma `a` y `b` tras reescalar a `target_scale` (o al `lcm` de escalas).
- Errores/excepciones: `TypeError` si no son `Fixed`; `ValueError` si motores distintos.
- Complejidad: O(bits).
- Ejemplos:

```
import satx
satx.engine(bits=8)
a = satx.fixed(scale=10)
b = satx.fixed(scale=25)
res = satx.fixed_add_rescaled(a, b)
```

‘as__fixed(u, scale)‘

- Identificador exacto: `satx.fixed.as_fixed`.
- Firma: `as_fixed(u: Unit, scale: int) -> Fixed`.
- Semántica formal: envuelve un `Unit` existente con escala explícita.
- Errores/excepciones: `TypeError`/`ValueError` si `scale` no es válido.
- Nota: `scale` es obligatorio; no hay valor por defecto. Para envolver un entero sin escala, usa `as_fixed(u, 1)`.
- Complejidad: O(1).
- Ejemplos:

```
import satx
satx.engine(bits=8)
u = satx.integer()
f = satx.as_fixed(u, 10)
```

‘to__rational(f)‘

- Identificador exacto: `satx.fixed.to_rational`.
- Firma: `to_rational(f: Fixed) -> satx.Rational`.
- Semántica formal: retorna `Rational(f.raw, scale)` usando un `Unit` constante como denominador.

- Errores/excepciones: `TypeError` si `f` no es `Fixed`.
- Complejidad: O(1).
- Ejemplos:

```
import satx
satx.engine(bits=8)
f = satx.fixed(scale=10)
r = satx.to_rational(f)
```

‘fixed_div_exact(a, b, *, scale=None)‘

- Identificador exacto: `satx.fixed.fixed_div_exact`.
- Firma: `fixed_div_exact(a: Fixed, b: Fixed, *, scale: int | None = None) -> satx.Rational | Fixed`.
- Semántica formal:
- Requiere `a.scale == b.scale`.
- Si `scale is None`: retorna `Rational(a.raw, b.raw)`.
- Si `scale` se define: crea `r` con `a.raw * scale == r * b.raw` y retorna `Fixed(r, scale)`.
- Errores/excepciones: `TypeError` si no son `Fixed`; `ValueError` si escalas o motores no coinciden.
- Nota: con `scale=None` el retorno es `Rational` (sin `.raw`/`.scale`); usa `scale=...` si necesitas un `Fixed`.
- Complejidad: O(bits^2) en la ecuación de multiplicación.
- Ejemplos:

```
import satx
satx.engine(bits=8)
a = satx.fixed(scale=10)
b = satx.fixed(scale=10)
r = satx.fixed_div_exact(a, b)
```

‘as_int(f, policy="exact")‘

- Identificador exacto: `satx.fixed.as_int`.
- Firma: `as_int(f: Fixed, policy: str = "exact") -> Unit | int`.
- Semántica formal:
- Convierte `f` a entero según `policy`:

- `exact`: requiere `raw % scale == 0`.
- `floor`, `ceil`, `round` (round-half-up).
- Si `f` tiene valor asignado, retorna `int`.
- Errores/excepciones: `TypeError` si `f` no es `Fixed` o `policy` no es `str`; `ValueError` si `policy` inválido o no representable en modo `exact`.
- Complejidad: O(bits).
- Ejemplos:

```
import satx
satx.engine(bits=8)
f = satx.fixed(scale=10)
q = satx.as_int(f, policy="floor")
```

‘fixed__mul__floor(a, b)‘

- Identificador exacto: `satx.fixed.fixed_mul_floor`.
- Firma: `fixed_mul_floor(a: Fixed, b: Fixed) -> Fixed`.
- Semántica formal: multiplica y reescala a la misma escala usando `floor`.
- Errores/excepciones: `TypeError` si no son `Fixed`; `ValueError` si escalas/motores no coinciden.
- Complejidad: O(bits^2).
- Ejemplos:

```
import satx
satx.engine(bits=8)
a = satx.fixed(scale=10)
b = satx.fixed(scale=10)
res = satx.fixed_mul_floor(a, b)
```

‘fixed__mul__round(a, b)‘

- Identificador exacto: `satx.fixed.fixed_mul_round`.
- Firma: `fixed_mul_round(a: Fixed, b: Fixed) -> Fixed`.
- Semántica formal: multiplica y reescala a la misma escala usando redondeo half-up.
- Errores/excepciones: `TypeError` si no son `Fixed`; `ValueError` si escalas/motores no coinciden.
- Complejidad: O(bits^2).
- Ejemplos:

```
import satx
satx.engine(bits=8)
a = satx.fixed(scale=10)
b = satx.fixed(scale=10)
res = satx.fixed_mul_round(a, b)
```

‘fixed_bounds(expr_or_fixed, assumptions=None)‘

- Identificador exacto: `satx.fixed.fixed_bounds`.
- Firma: `fixed_bounds(expr_or_fixed, assumptions=None) -> tuple[Fraction, Fraction]`.
- Semántica formal:
- Si es `Fixed` o `Unit`, calcula el rango en función de bits y signed.
- Si es valor numérico, retorna `(value, value)`.
- `assumptions` puede ser `{"min":..., "max":...}` o `(min, max)`.
- Errores/excepciones: `TypeError` si la entrada no es compatible; `ValueError` si las asunciones son inconsistentes.
- Complejidad: O(1).
- Ejemplos:

```
import satx
from fractions import Fraction
satx.engine(bits=4, signed=False)
x = satx.fixed(scale=10)
lo, hi = satx.fixed_bounds(x, assumptions=(Fraction(0), Fraction(1, 2)))
```

‘fixed_advice(*vars, degree=2, expected_max=None, explain=False)‘

- Identificador exacto: `satx.fixed.fixed_advice`.
- Firma: `fixed_advice(*vars, degree=2, expected_max=None, explain=False) -> dict`.
- Semántica formal:
- Calcula información de escala y rangos seguros:
- `resolution = 1/scale`.
- `value_max_abs`, `safe_mul_value_max`, `safe_pow_value_max`.
- Si `expected_max` se provee, sugiere `bits_suggested`.

- Errores/excepciones: `TypeError`/`ValueError` si `degree` o `expected_max` inválidos.
- Complejidad: O(k) con k número de escalas.
- Ejemplos:

```
import satx
satx.engine(bits=8)
info = satx.fixed_advice(degree=2, explain=True)
```

‘block_fixed_solution(vars)‘

- Identificador exacto: `satx.fixed.block_fixed_solution`.
- Firma: `block_fixed_solution(vars) -> list[int]`.
- Semántica formal:
- Toma valores actuales y añade una cláusula que bloquea esa asignación.
- Acepta `Fixed` o `Unit` (lista o elemento).
- Errores/excepciones: `ValueError` si variables no tienen valor; `TypeError` si tipos no soportados.
- Complejidad: O(n · bits).
- Ejemplos:

```
import satx

BITS = 32

satx.engine(bits=BITS)
x = satx.integer()
satx.add(x == 1)
if satx.satisfy(solver=SAT_CMD):
    satx.block_fixed_solution(x)
```

‘enumerate_models_fixed(vars, *, limit=None, solver="sat_solver", params="", log=False)‘

- Identificador exacto: `satx.fixed.enumerate_models_fixed`.
- Firma: `enumerate_models_fixed(vars, *, limit=None, solver="sat_solver", params="", log=False) -> iterator[list]`.
- Semántica formal:
- Itera modelos SAT, devolviendo una lista de valores para `vars`.

- Tras cada modelo, bloquea la solución actual.
- Si `limit` se define, se detiene en ese número de modelos.
- Errores/excepciones: `TypeError` si `vars` contiene tipos no soportados o `limit` inválido.
- Complejidad: dependiente del backend `SAT_CMD` (parámetro `solver`).
- Ejemplos:

```
import satx

BITS = 32

satx.engine(bits=BITS)
x = satx.integer()
for model in satx.enumerate_models_fixed([x], limit=2, solver=SAT_CMD):
    print(model)
```

Módulo ‘satx.unit‘

Propósito formal: entero bit-vector (`Unit`) con operadores aritméticos, lógicos y comparaciones reificadas en CNF.

‘class Unit‘

- Identificador exacto: `satx.unit.Unit`.
- Firma: `Unit(alu, key=None, block=None, value=None, bits=None, deep=None)`.
- Dominio/codominio:
- `alu`: instancia de `satx.alu.ALU`.
- `block`: lista de literales CNF (opcional).
- `bits`: anchura de bit-vector.
- `deep`: forma multidimensional (producto = `bits`).
- Semántica formal:
- Si `value` se provee, crea un bloque constante; el atributo `value` se llena cuando se aplica un modelo.
- Si `block` es `None`, crea nuevas variables con `alu.create_variable`.
- Si `deep` se define, `data` se organiza con `alu.reshape`.
- Invariante: `len(block) == bits` y `Unit.alu` es el motor de referencia.
- Errores/excepciones: errores pueden surgir si `alu` no es válido o `bits/deep` no son coherentes.

- Complejidad: O(bits) para construcción de variables.

Métodos y operadores principales (resumen)

- Membresía:
- `is_in(item)` / `is_not_in(item)`: retorna `bool` si `value` está asignado; en otro caso retorna `InConstraint`.
- Aritmética:
- `__add__`, `__sub__`, `__mul__`: retornan `Unit` (o `Fixed` si se combina con punto fijo) y agregan restricciones; si `value` está asignado, retornan `int`.
- `__pow__`: potencia con exponente `int` o `Unit` (si es `Unit`, usa selector interno); no implementa shift binario.
- `__floordiv__`, `__mod__`: división entera y módulo con restricciones; si `value` está asignado y divisor 0, lanza `ZeroDivisionError`.
- `__truediv__`: retorna `Rational`.
- Comparaciones:
- `__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, `__ge__`: retornan `Constraint` o `bool` si ambos valores están fijados.
- Bitwise:
- `__and__`, `__or__`, `__xor__`: retornan `Unit` con puertas bit a bit.
- Casts de ancho:
- `zext(W)`, `sext(W)`, `trunc(W)`, `resize(W, mode=...)`.
- Desplazamientos:
- `__lshift__`, `__rshift__`: comportamiento inferido: `x « k` codifica `x * (2**k)` y `x » k` codifica `x // (2**k)`.
- Lógica adicional:
- `implies(constraint)`: delega a `satx.implies`.
- `iff(bit, other)`: selecciona entre `self` y `other` según `bit` (literal/Constraint/Unit/bool).
- `__getitem__`: si `item` es int retorna el literal de bit; si es iterable, retorna una función de selección.
- `binary` (property): devuelve los bits del `value` (requiere modelo).
- `clear()`: asigna `value=None`.
- `reverse(copy=False)`: invierte el orden de bits (in-place o copia).

Notas de implementación relevantes

- Operaciones con `Fixed` se detectan vía `_fixed_info`; se prohíbe mezclar motores distintos.
- Alineación de anchos: `width_policy` define si se extiende, trunca o falla; `width_warn` controla warnings.
- En modo `signed`, algunas operaciones agregan restricciones de overflow (p.ej. suma y resta).
- `__abs__` agrega una restricción `x >= 0` en modo signed.

Ejemplos

```
import satx
satx.engine(bits=4)
x = satx.integer()
y = satx.integer()
expr = (x + y) >= 3
satx.add(expr)
```

```
import satx
satx.engine(bits=4)
x = satx.integer()
sel = satx.switch(x, 0)
choice = x.iff(sel, 0)
```

Módulo 'satx.constraint'

Propósito formal: abstracciones de restricciones con reificación a literales CNF.

'class Constraint'

- Identificador exacto: `satx.constraint.Constraint`.
- Firma: `Constraint(alu)` (clase base abstracta).
- Semántica formal:
- `reify()` produce un literal `l` tal que $l \leftrightarrow C$, donde C es la restricción.
- `apply()` agrega la cláusula del literal al CNF; es idempotente.
- Operadores lógicos `&`, `|`, construyen restricciones compuestas.
- `__bool__` lanza `TypeError` (no tiene valor booleano directo).
- Errores/excepciones: `NotImplementedError` en `reify` base.
- Complejidad: depende de la reificación concreta.
- Ejemplos:

```
import satx
satx.engine(bits=4)
x = satx.integer()
cons = (x > 0)
cons.apply()
```

‘ConstraintRecord‘

- Identificador exacto: `satx.constraint.ConstraintRecord`.
- Firma: dataclass `{id: int, label: str|None, kind: str, constraint: Constraint, activation: int, soft: bool, weight: int}`.
- Semántica formal: estructura de metadatos usada por `satx.add`.
- `soft`/`weight` se usan para MaxSAT; por defecto SAT/#SAT tratan soft como hard (ver `satisfy(ignore_soft=True)` / `counting(ignore_soft=True)` para hard-only).
- Ejemplo:

```
import satx
satx.engine(bits=4)
x = satx.integer()
rec = satx.add(x > 0, label="c1")
print(rec.id, rec.label, rec.kind)
```

‘ConstantConstraint(alu, value)‘

- Semántica formal: representa una constante booleana; reifica a literal constante del motor.
- Errores/excepciones: no aplica.
- Ejemplo:

```
import satx
from satx.constraint import ConstantConstraint
satx.engine(bits=4)
cc = ConstantConstraint(satx.current_engine(), True)
```

‘ComparisonConstraint(lhs, rhs, op)‘

- Semántica formal:
- Reifica comparaciones `==`, `!=`, `<`, `<=`, `>`, `>=` entre `lhs` y `rhs`.
- Usa comparadores signed/unsigned según `alu.signed`.
- Alinea `lhs` y `rhs` según `width_policy`; en `strict` lanza `ValueError` y puede emitir `UserWarning` si `width_warn=True`.

- Errores/excepciones: `ValueError` si `op` no es soportado.
- Ejemplos:

```
import satx
from satx.constraint import ComparisonConstraint
satx.engine(bits=4)
x = satx.integer()
cc = ComparisonConstraint(x, 1, ">=")
cc.apply()
```

‘InConstraint(lhs, values, negate=False)‘

- Semántica formal:
- Reifica `lhs in values` (o `lhs not in values` si `negate=True`).
- Cada valor se alinea con `lhs` según `width_policy`; en `strict` lanza `ValueError` y puede emitir `UserWarning` si `width_warn=True`.
- Si `values` está vacío: retorna constante verdadera/falsa según `negate`.
- Errores/excepciones: `ValueError` si hay valores de otros motores.
- Ejemplos:

```
import satx
from satx.constraint import InConstraint
satx.engine(bits=4)
x = satx.integer()
cons = InConstraint(x, [0, 2])
cons.apply()
```

‘ImplicationConstraint(condition, consequence)‘

- Semántica formal: reifica $\neg condition \vee consequence$.
- Errores/excepciones: `ValueError` si motores difieren.
- Ejemplos:

```
import satx
from satx.constraint import ImplicationConstraint
satx.engine(bits=4)
x = satx.integer()
cons = ImplicationConstraint(x > 0, x < 3)
cons.apply()
```

‘NotConstraint(operand)‘ / ‘AndConstraint(operands)‘ / ‘OrConstraint(operands)‘

- Semántica formal: composición lógica de restricciones con simplificación constante.
- Errores/excepciones: `ValueError` si la lista está vacía o motores difieren.
- Ejemplos:

```
import satx
from satx.constraint import AndConstraint
satx.engine(bits=4)
x = satx.integer()
cons = AndConstraint([x > 0, x < 3])
cons.apply()
```

‘compile_bool(expr, *, alu=None)‘

- Identificador exacto: `satx.constraint.compile_bool`.
- Firma: `compile_bool(expr, *, alu=None) -> int`.
- Semántica formal: reifica `expr` a un literal CNF.
- Errores/excepciones: propagados desde `as_constraint`.
- Ejemplo:

```
import satx
from satx.constraint import compile_bool
satx.engine(bits=4)
x = satx.integer()
lit = compile_bool(x > 0, alu=satx.current_engine())
```

‘as_constraint(value, *, alu=None)‘

- Identificador exacto: `satx.constraint.as_constraint`.
- Firma: `as_constraint(value, *, alu=None) -> Constraint`.
- Semántica formal:
- Si `value` es `Constraint`, retorna `value`.
- Si `value` es `bool`, requiere `alu` y retorna `ConstantConstraint`.
- Si `value` tiene `alu` y `block`, retorna `ComparisonConstraint(value, 0, "!=")`.
- Errores/excepciones: `TypeError` si `value` no es convertible.
- Ejemplo:

```
import satx
from satx.constraint import as_constraint
satx.engine(bits=4)
x = satx.integer()
cons = as_constraint(x)
```

‘apply_constraint(value, *, alu=None)‘

- Identificador exacto: `satx.constraint.apply_constraint`.
- Firma: `apply_constraint(value, *, alu=None) -> Constraint | None`.
- Semántica formal:
- Aplana listas/tuplas y aplica cada restricción.
- Si `value` es callable, se evalúa.
- Errores/excepciones: propagados desde `as_constraint`.
- Ejemplo:

```
import satx
from satx.constraint import apply_constraint
satx.engine(bits=4)
x = satx.integer()
apply_constraint(x > 0)
```

Módulo ‘satx.rational‘

Propósito formal: racional simbólico `x / y` con restricciones por efecto colateral.

‘class Rational‘

- Identificador exacto: `satx.rational.Rational`.
- Firma: `Rational(x, y)`.
- Dominio/codominio: `x` (numerador) y `y` (denominador) pueden ser `Unit` u objetos compatibles.
- Semántica formal:
- Si `y` es `Unit`, agrega `y != 0`.
- Operaciones aritméticas retornan nuevos `Rational` construidos por productos/cruces.
- Comparaciones (`==, !=, <, <=, >, >=`) agregan restricciones y retornan `True`.
- No simplifica ni reduce fracciones; no hay normalización automática.

- Errores/excepciones: no valida tipos explícitamente; errores pueden propagarse desde operaciones internas.
- Complejidad: depende de la complejidad de las operaciones de `Unit`.
- Ejemplos:

```
import satx
satx.engine(bits=4)
x = satx.integer()
y = satx.integer()
r = satx.Rational(x, y)
```

```
import satx
satx.engine(bits=4)
x = satx.integer()
r = satx.Rational(x, satx.integer())
satx.add(r != satx.Rational(1, satx.integer()))
```

Módulo ‘satx.gaussian‘

Propósito formal: enteros gaussianos `a + bj` construidos con `Unit`.

‘class Gaussian‘

- Identificador exacto: `satx.gaussian.Gaussian`.
- Firma: `Gaussian(x, y)` con `x` real, `y` imaginario.
- Semántica formal:
- Operaciones aritméticas retornan nuevos `Gaussian` (o combinaciones con `Rational` en división).
- `__eq__` / `__ne__` agregan restricciones y retornan `True`.
- `conjugate()` retorna `(x, -y)`.
- `__abs__` retorna un `Gaussian` con parte imaginaria 0 y módulo en la parte real.
- Errores/excepciones: no valida tipos explícitamente.
- Complejidad: depende de operaciones en `Unit`.
- Ejemplos:

```
import satx
satx.engine(bits=4)
z = satx.Gaussian(satx.integer(), satx.integer())
```

```
import satx
satx.engine(bits=4)
a = satx.gaussian()
b = satx.gaussian()
expr = a * b
```

Módulo ‘satx.gcc‘

Propósito formal: implementación de Global Constraint Catalog (GCC) como macros sobre `satx.stdlib`.

Convención: estas funciones agregan restricciones y no retornan valor significativo (salvo indicación).

‘abs_val(x, y)‘

- Identificador exacto: `satx.gcc.abs_val`.
- Firma: `abs_val(x, y) -> None`.
- Semántica formal: agrega `y >= 0` y `abs(x) == y`.
- Errores/excepciones: no valida tipos.
- Ejemplos:

```
import satx
from satx import gcc
satx.engine(bits=4)
x = satx.integer()
y = satx.integer()
gcc.abs_val(x, y)
```

‘all_differ_from_at_least_k_pos(k, lst)‘

- Semántica formal: para cada par de vectores en `lst`, difieren en al menos `k` posiciones.
- Complejidad: O($|lst|^2$ * len(vector)).
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
V = [satx.vector(size=3), satx.vector(size=3)]
gcc.all_differ_from_at_least_k_pos(2, V)
```

‘all_differ_from_at_most_k_pos(k, lst)‘

- Semántica formal: para cada par de vectores en `lst`, difieren en a lo más `k` posiciones.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
V = [satx.vector(size=3), satx.vector(size=3)]
gcc.all_differ_from_at_most_k_pos(1, V)
```

‘all_differ_from_exactly_k_pos(k, lst)‘

- Semántica formal: combina las dos anteriores para imponer diferencia exacta.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
V = [satx.vector(size=3), satx.vector(size=3)]
gcc.all_differ_from_exactly_k_pos(1, V)
```

‘all_equal(lst)‘

- Semántica formal: agrega `x == y` para todo par en `lst`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
xs = satx.vector(size=3)
gcc.all_equal(xs)
```

‘all_equal_except_0(lst)‘

- Semántica formal: todos los elementos distintos de 0 deben ser iguales entre sí.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
xs = satx.vector(size=3)
gcc.all_equal_except_0(xs)
```

‘all_different(lst)‘

- Semántica formal: alias de `satx.all_different`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
xs = satx.vector(size=3)
gcc.all_different(xs)
```

‘element(idx, lst, val)‘

- Semántica formal: impone `val == lst[idx]`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
idx = satx.integer()
val = satx.integer()
gcc.element(idx, [1,2,3], val)
```

‘in_(x, domain)‘

- Semántica formal:
- Si `domain=(lb, ub)`, agrega `lb <= x <= ub`.
- Si es conjunto/lista, agrega `x == one_of(values)`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
x = satx.integer()
gcc.in_(x, (0, 3))
```

‘not_in(x, values)‘

- Semántica formal: agrega `x != v` para todo `v` en `values`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
x = satx.integer()
gcc.not_in(x, [0, 1])
```

‘among(n, lst, values)‘ / ‘among_var(n, lst, values)‘

- Semántica formal: impone que `n` sea el número de elementos de `lst` en `values`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
xs = satx.vector(size=3)
n = satx.integer()
gcc.among(n, xs, [0, 1])
```

‘at_least(k, lst, val)‘ / ‘at_most(k, lst, val)‘ / ‘exactly(k, lst, val)‘

- Semántica formal: cardinalidad de ocurrencias de `val` en `lst`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
xs = satx.vector(size=3)
gcc.exactly(2, xs, 1)
```

‘minimum(mn, lst)‘ / ‘maximum(mx, lst)‘

- Semántica formal: `mn` (resp. `mx`) es el mínimo (resp. máximo) de `lst`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
xs = satx.vector(size=3)
mn = satx.integer()
gcc.minimum(mn, xs)
```

‘sum(lst, rel, rhs)‘

- Semántica formal: agrega `rel(sum(lst), rhs)`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
xs = satx.vector(size=3)
gcc.sum(xs, lambda a, b: a <= b, 5)
```

‘scalar_product(coeffs, lst, rel, rhs)‘

- Semántica formal: agrega $rel(\sum_i coeffs[i] \cdot lst[i], rhs)$.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
xs = satx.vector(size=3)
gcc.scalar_product([1,2,3], xs, lambda a,b: a == b, 5)
```

‘nvalue(n, lst)‘

- Semántica formal: impone que `n` sea el número de valores distintos en `lst`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
xs = satx.vector(size=3)
n = satx.integer()
gcc.nvalue(n, xs)
```

‘lex_less(a, b)‘ / ‘lex_lesseq(a, b)‘ / ‘lex_greater(a, b)‘ / ‘lex_greatereq(a, b)‘

- Semántica formal: orden lexicográfico estricto o no estricto.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
a = satx.vector(size=3)
b = satx.vector(size=3)
gcc.lex_lesseq(a, b)
```

‘lex_chain_less(vectors)‘ / ‘lex_chain_lesseq(vectors)‘ / ‘lex_chain_greater(vectors)‘ / ‘lex_chain_greatereq(vectors)‘

- Semántica formal: aplica restricciones lexicográficas entre vectores consecutivos.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
vs = [satx.vector(size=2) for _ in range(3)]
gcc.lex_chain_lesseq(vs)
```

‘inverse(fwd, inv)‘

- Semántica formal: `inv` es la inversa de `fwd` sobre `0..n-1`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
perm = satx.vector(size=3)
inv = satx.vector(size=3)
gcc.inverse(perm, inv)
```

‘circuit(succ)‘

- Semántica formal: `succ` define un único circuito Hamiltoniano sobre `0..n-1`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
succ = satx.vector(size=4)
gcc.circuit(succ)
```

‘bin_packing(load, bin, size)‘

- Semántica formal: `load[b]` es la suma de `size[i]` para ítems con `bin[i]==b`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
load = satx.vector(size=2)
bin = satx.vector(size=3)
size = [1, 2, 1]
gcc.bin_packing(load, bin, size)
```

‘bin_packing_capa(load, bin, size, capa)‘

- Semántica formal: `bin_packing` + `load[b] <= capa[b]`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
load = satx.vector(size=2)
bin = satx.vector(size=3)
size = [1, 2, 1]
capa = satx.vector(size=2)
gcc.bin_packing_capa(load, bin, size, capa)
```

‘diffn(x, y, dx, dy)‘

- Semántica formal: rectángulos `(x[i], y[i], dx[i], dy[i])` no se solapan por pares.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
x = satx.vector(size=2)
y = satx.vector(size=2)
dx = satx.vector(size=2)
dy = satx.vector(size=2)
gcc.diffn(x, y, dx, dy)
```

‘cumulative(start, duration, demand, limit, horizon)‘

- Semántica formal: recurso acumulativo en tiempo discreto `t in [0, horizon)`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
start = satx.vector(size=2)
duration = [1, 2]
demand = satx.vector(size=2)
limit = satx.integer()
gcc.cumulative(start, duration, demand, limit, horizon=4)
```

‘gcd(x, y, z)‘

- Semántica formal (comportamiento inferido): impone condiciones para que `z` sea un divisor común con restricciones adicionales basadas en módulo.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=6)
x = satx.integer()
y = satx.integer()
z = satx.integer()
gcc.gcd(x, y, z)
```

‘sort(lst1, lst2)‘

- Semántica formal: `lst2` es una permutación ordenada no-decreciente de `lst1`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
xs = satx.vector(size=3)
ys = satx.vector(size=3)
gcc.sort(xs, ys)
```

‘sort_permutation(lst_from, lst_per, lst_to)‘

- Semántica formal: `lst_to` es una permutación ordenada de `lst_from` controlada por `lst_per`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
xs = satx.vector(size=3)
perm = satx.vector(size=3)
ys = satx.vector(size=3)
gcc.sort_permutation(xs, perm, ys)
```

‘count(val, lst, rel, lim)‘

- Semántica formal: sea `N` el número de elementos iguales a `val`; agrega `rel(N, lim)`.
- Ejemplo:

```
import satx
from satx import gcc
satx.engine(bits=4)
xs = satx.vector(size=3)
gcc.count(1, xs, lambda a,b: a <= b, 2)
```

Módulo ‘satx.alu‘

Propósito formal: motor CNF de bajo nivel (ALU) para construir puertas y bit-vectors.

‘class ALU‘

- Identificador exacto: `satx.alu.ALU`.
- Firma: `ALU(bits=None, deep=None, cnf="")`.
- Semántica formal:
- Inicializa el motor con ancho `bits`, profundidad `deep` y modo de CNF (`cnf==""` en memoria, si no, archivo).
- Mantiene contadores `number_of_variables`, `number_of_clauses`.
- Define literales constantes `true` y `false` mediante una variable fija (ver nota).
- Errores/excepciones: errores de E/S si `cnf` no es escribible.
- Complejidad: O(1) en inicialización (más E/S si se abre archivo).

Notas de implementación relevantes

- Constantes: `true` se crea como variable y se fuerza a falso; `false` es su negación (literal constante verdadero en CNF). Esto es utilizado por los comparadores y reificación.
- Si `cnf` es ruta no vacía, las cláusulas se emiten a disco; en otro caso se almacenan en `_cnf_clauses`.

API de construcción y estado

- `zero` / `one`: constantes `Unit` cacheadas de 0 y 1.
- `add_variable() -> int`: reserva un literal nuevo.
- `add_block(clause) -> list[int]`: añade una cláusula; con `simplify=True` elimina literales triviales y, si queda vacía, emite la cláusula vacía (`0`).
- `mapping(key, value)`: registra el mapeo `key -> block`.
- `create_block(size=None) -> list[int]`: crea un bloque de bits.
- `new_key() -> str`: genera un identificador.

- `create_variable(key=None, size=None) -> (key, block)`: crea variables nuevas.
- `create_constant(value, size=None) -> list[int]`: crea un bloque constante para `value`.

Puertas booleanas (literales)

- `binary_or((a,b), ol=None)` / `binary_and((a,b), ol=None)`.
- `binary_xor_gate([a,b], ol=None)` / `binary_xnor_gate([a,b], ol=None)`.
- `binary_mux_gate([sel,lhs,rhs], ol=None)`.
- `or_gate(il, ol=None)` / `and_gate(il, ol=None)`.
- `fas_gate([lhs,rhs,c_in], ol=None)` (sumador completo – suma).
- `fac_gate([lhs,rhs,c_in], ol=None)` (sumador completo – acarreo).

Puertas bit-vector

- `gate_vector(bge, lhs_il, rhs_il, ol=None)`.
- `bv_and_gate`, `bv_or_gate`, `bv_xor_gate`, `bv_xnor_gate`.
- `bv_rca_gate(lhs_il, rhs_il, carry_in_lit=None, ol=None, carry_out_lit=None)`.
- `bv_rcs_gate(lhs_il, rhs_il, ol=None)` (resta vía complemento a 2).
- `bv_pm_gate(lhs_il, rhs_il, ol=None, ow_lit=None)` (producto parcial).
- `bv_ule_gate` / `bv_sle_gate` / `bv_eq_gate`.
- `bv_mux_gate(lhs_il, rhs_il, s_lhs_lit=None, ol=None)`.
- `bv_lud_gate(lhs_il, rhs_il, ol=None, remainder_ol=None)` (división con cociente y residuo).
- `bv_lur_gate(lhs_il, rhs_il, ol=None)` (residuo solamente).
- `stg_or_gate(il, ol=None)` (OR acumulado por sufijos).

Constructores y utilidades

- `int(key=None, block=None, value=None, size=None, deep=None) -> Unit`.
- `array(dimension, size=None, signed=True) -> list[Unit]`.
- `reshape(lst, shape) -> list`.
- `mul(xs, ys)`, `dot(xs, ys)`, `values(xs, cleaner=None)`, `flatten(xs)`.
- `apply(xs, single=None, dual=None, different=None)` y `apply_indexed(...)`.

Restricciones combinatorias

- `element(x, lst, y)`.
- `indexing(xs, ys, lst)` / `sequencing(xs, ys, lst)`.
- `permutations(xs, lst)` / `combinations(xs, lst)`.
- `factorial(x)`, `sigma(f, i, n)`, `pi(f, i, n)`.
- `sqrt(x)`.
- `at_most_k(x, k)`.
- `subset(k, data, empty=None, complement=False)`.

Ejemplo

```
import satx
satx.engine(bits=4)
alu = satx.current_engine()
a = alu.int()
b = alu.int()
res_block = alu.bv_rca_gate(a.block, b.block)
res = satx.Unit(alu, block=res_block)
```

1.6. stdlib y construcciones declarativas

Modelo declarativo (DSL)

- Variables: `Unit` y `Fixed` representan enteros y racionales exactos (raw/scale). En notación, $x \in U_w$ o $x \in S_w$ y $f = raw/scale$.
- Restricciones: se crean con operadores (`==, !=, <, <=, >, >=`), combinadores lógicos (, `&`, `|`) y funciones (`all_different`, `one_of`, etc.).
- Registro: `satx.add(expr)` reifica `expr` en un literal de activación y lo incorpora al CNF.
- Evaluación: `Constraint` no tiene verdad booleana; debe aplicarse o registrarse.

Composición formal

- Reificación: para toda restricción `R`, existe literal `l_R` tal que $l_R \leftrightarrow R$.
- Conjunción/disyunción: `R1 & R2` reifica $l = l_1 \wedge l_2$; `R1 | R2` reifica $l = l_1 \vee l_2$.
- Implicación: `implies(A, B)` codifica $\neg A \vee B$.
- Selección condicional: `Unit.iff(bit, other)` crea un mux de bit-vector controlado por `bit`.

Construcciones declarativas frecuentes

- Selección: `one_of(values)` crea un selector one-hot.
- Cardinalidad: `all_binaries`, `at_most_k`, `subset`, `subsets`.

- Arreglos: `vector`, `matrix`, `tensor`, `reshape`, `flatten`.
- Iteración declarativa: `apply_single`, `apply_dual`, `apply_different`.

Ejemplo compuesto

```
import satx
satx.engine(bits=4)
xs = satx.vector(size=3)
satx.all_different(xs)
satx.add(sum(xs) <= 5)
```

1.7. Fixed-point / Decimales / Escalas

Definición formal

- `Fixed(raw, scale)` representa `value = raw / scale`.
- `raw` es `Unit` con ancho `w`, por lo que:
- Unsigned: $raw \in [0, 2^w - 1]$.
- Signed: $raw \in [-2^{w-1}, 2^{w-1} - 1]$.
- Resolución: `resolution = 1/scale`.
- Rango de valores: $value \in [\text{mín}(raw)/scale, \text{máx}(raw)/scale]$.

Invariantes relevantes

- `scale > 0` y representable en el motor (ver `_check_scale_fits_unit`).
- No hay redondeo implícito en `Fixed.__mul__`; la reescala es exacta.
- `Fixed.__truediv__` está prohibido; use `fixed_div_exact`.

Consejos de escala (fixed_advice)

- $raw\_max\_abs = 2^{w-1} - 1$ en signed o $raw\_max\_abs = 2^w - 1$ en unsigned.
- $safe\_mul\_value\_max \approx \sqrt{raw\_max\_abs}/scale$.
- $safe\_pow\_value\_max \approx (raw\_max\_abs/(scale^2))^{1/degree}$.
- `bits_suggested` puede inferirse de `expected_max`.

Ejemplo

```
import satx
satx.engine(bits=8)
f = satx.fixed(scale=100)
info = satx.fixed_advice(f, degree=2)
```

1.8. Engine / Solving / Resultados

Motor global

- `satx.engine(...)` inicializa `csp` como instancia de `ALU` y configura parámetros (`bits`, `signed`, `cnf_mode`, `fixed_default`, `max_fixed_pow`).
- `check_engine()` por defecto termina el proceso si el motor no está inicializado; en modo estricto puede lanzar `EngineNotInitializedError`.
- Multi-engine: `Engine` y `engine_ctx` permiten aislar modelos sin reusar el motor global (no es thread-safe).

CNF y artefactos

- `add()` registra restricciones con literal de activación (útil para `explain_unsat`, `relax`, `synthesize`).
- `to_cnf(path)` exporta DIMACS con todas las restricciones activas.
- `cnf_mode`: `none`, `on_solve`, `always` (controla si se persisten CNF al resolver).

SAT (satisfy)

- `satisfy(solver=SAT_CMD, params=..., debug=...)` ejecuta un backend externo sobre la CNF exportada y parsea salida DIMACS (`s ...`, `v ...`).
- Retorna `True` si SAT y `False` si UNSAT; si la ejecución falla o la salida no es interpretable, lanza `RuntimeError`.
- Si hay modelo, asigna `Unit.value` por decodificación bit a bit (two's complement si `signed=True`).
- Bloquea el modelo actual automáticamente para permitir enumeración.

#SAT (counting)

- `counting(solver=COUNT_CMD, ...)` invoca un contador de modelos externo y parsea la salida.
- Si UNSAT, retorna 0; si hay conteo explícito, retorna el entero reportado.

Resultados

- `Unit.value`: `int` asignado tras resolver.
- `Fixed.value`: `Fraction(raw.value, scale)` si hay modelo.
- `Rational` y `Gaussian` no materializan valores; sus componentes pueden tener `value`.

1.9. Compatibilidad, límites y no-garantías

- No incluye solvers: el usuario debe proveer backends compatibles en `SAT_CMD` y `COUNT_CMD` (y opcionalmente `MAXSAT_CMD`/`MIP_CMD`).
- `check_engine()` por defecto finaliza el proceso si no hay motor; en modo estricto puede lanzar `EngineNotInitializedError`.
- `Unit.__lshift__`/`__rshift__` no implementan shift binario clásico; para shift bit-a-bit use `shl`/`shr`/`ashr` (ver sección `Unit`).
- Primalidad: `is_prime` es un test de Fermat base 2 (pseudoprimalidad); `is_probable_prime` aplica múltiples bases, pero no certifica primalidad.
- `Rational`/`Gaussian` no normalizan ni simplifican valores; son expresiones simbólicas.
- Overflow: operaciones de `Unit`/`Fixed` pueden generar UNSAT; no hay saturación ni wrap-around en operaciones. Nota: los literales se codifican a ancho fijo y por defecto (`const_policy="strict"`) se rechazan si no caben; use `const_policy="warn"` o `"wrap"` si quiere permitir wrap.
- Estado global: `csp` es un default global; para aislamiento use `Engine`/`engine_ctx`. No es thread-safe.
- `hess_*` son heurísticas; no garantizan optimalidad.

1.10. Índice de símbolos (Symbol index)

‘satx‘

- `__version__` → versión del paquete → `satx`
- `current_engine` → retorna el motor global actual → `satx`
- `csp` → alias dinámico del motor global → `satx`

‘satx.stdlib‘

- `version` → imprime info del sistema y retorna versión → `satx.stdlib`
- `check_engine` → valida motor global (exit si no) → `satx.stdlib`
- `CNFPolicy` → política de artefactos CNF → `satx.stdlib`
- `current_cnf_path` → ruta CNF actual → `satx.stdlib`
- `last_cnf_path` → última ruta CNF generada → `satx.stdlib`
- `to_cnf` → exporta DIMACS CNF → `satx.stdlib`
- `engine` → inicializa motor → `satx.stdlib`
- `integer` → crea `Unit` o `Fixed` → `satx.stdlib`

- `constant` → constante `Unit` o `Fixed` → `satx.stdlib`
- `unit_le_fixed` → `Unit <= Fixed` → `satx.stdlib`
- `unit_ge_fixed` → `Unit >= Fixed` → `satx.stdlib`
- `unit_eq_fixed` → `Unit == Fixed` → `satx.stdlib`
- `subsets` → subconjuntos con selector → `satx.stdlib`
- `subset` → subconjunto de a lo más k → `satx.stdlib`
- `vector` → vector de variables → `satx.stdlib`
- `matrix` → matriz de variables → `satx.stdlib`
- `matrix_permutation` → permutación de matriz → `satx.stdlib`
- `permutations` → permutaciones de lista → `satx.stdlib`
- `combinations` → combinaciones con repetición → `satx.stdlib`
- `all_binaries` → fuerza valores binarios → `satx.stdlib`
- `switch` → selector por bit → `satx.stdlib`
- `one_of` → selección one-hot → `satx.stdlib`
- `factorial` → factorial simbólico → `satx.stdlib`
- `sigma` → sumatoria simbólica → `satx.stdlib`
- `pi` → productoria simbólica → `satx.stdlib`
- `dot` → producto punto → `satx.stdlib`
- `mul` → producto elemento a elemento → `satx.stdlib`
- `values` → extrae valores → `satx.stdlib`
- `apply_single` → aplica unary sobre vector → `satx.stdlib`
- `apply_dual` → aplica sobre pares i<j → `satx.stdlib`
- `apply_different` → aplica sobre pares i≠j → `satx.stdlib`
- `all_different` → all-different → `satx.stdlib`
- `all_out` → exclusión de valores → `satx.stdlib`
- `all_in` → inclusión en conjunto → `satx.stdlib`
- `add` → registra restricciones → `satx.stdlib`
- `require` → alias de `add` → `satx.stdlib`
- `assume` → alias de `add` → `satx.stdlib`

- `block_current` → bloquea modelo actual → `satx.stdlib`
- `implies` → implicación booleana → `satx.stdlib`
- `flatten` → aplana listas → `satx.stdlib`
- `bits` → ancho de bits → `satx.stdlib`
- `oo` → máximo representable → `satx.stdlib`
- `element` → índice de elemento → `satx.stdlib`
- `index` → elemento por índice → `satx.stdlib`
- `gaussian` → constructor gaussiano → `satx.stdlib`
- `rational` → constructor racional → `satx.stdlib`
- `exact_div` → división exacta → `satx.stdlib`
- `at_most_k` → cardinalidad de bits → `satx.stdlib`
- `sqrt` → raíz cuadrada simbólica → `satx.stdlib`
- `hess_sequence` → heurística de secuencias → `satx.stdlib`
- `hess_binary` → heurística binaria → `satx.stdlib`
- `hess_abstract` → heurística abstracta → `satx.stdlib`
- `hyper_loop` → generador de tuplas → `satx.stdlib`
- `reshape` → reestructura lista → `satx.stdlib`
- `tensor` → tensor multidimensional → `satx.stdlib`
- `clear` → limpia valores → `satx.stdlib`
- `rotate` → rotación de bits → `satx.stdlib`
- `is_prime` → test Fermat base 2 → `satx.stdlib`
- `is_not_prime` → negación del test → `satx.stdlib`
- `has_solver` → verifica ejecutable → `satx.stdlib`
- `satisfy` → SAT con solver externo → `satx.stdlib`
- `counting` → #SAT con contador externo → `satx.stdlib`
- `explain_unsat` → core UNSAT → `satx.stdlib`
- `relax` → relaja restricciones → `satx.stdlib`
- `suggest_fixes` → alias de `relax` → `satx.stdlib`
- `synthesize` → sintetiza valores → `satx.stdlib`

- `exposure` → exposición de evento → `satx.stdlib`
- `distribution` → distribución por conteo → `satx.stdlib`
- `reset` → resetea el motor → `satx.stdlib`

‘satx.sym‘

- `SymExpr` → alias del SIR (`Atom`) → `satx.sym`
- `Context` → contexto de evaluación CAS → `satx.sym`
- `var` → constructor de símbolo → `satx.sym`
- `const` → constructor de constante → `satx.sym`
- `app` → aplicación prefijo → `satx.sym`
- `tensor` → constructor de tensor → `satx.sym`
- `parse` → parser del CAS (re-export) → `satx.sym`
- `to_python` → baja a valores Python → `satx.sym`
- `as_float` → conversión numérica → `satx.sym`

‘satx.cas‘

- `Engine` → motor CAS → `satx.cas`
- `CASError` → excepción del CAS → `satx.cas`

‘satx.fixed‘

- `Fixed` → decimal de punto fijo → `satx.fixed`
- `fixed` → variable `Fixed` → `satx.fixed`
- `fixed_const` → constante `Fixed` → `satx.fixed`
- `fixed_lcm_scale` → LCM de escalas → `satx.fixed`
- `fixed_rescale_to` → reescala exacta → `satx.fixed`
- `fixed_add_rescaled` → suma con reescala → `satx.fixed`
- `as_fixed` → envuelve `Unit` → `satx.fixed`
- `to_rational` → convierte a `Rational` → `satx.fixed`
- `fixed_div_exact` → división exacta → `satx.fixed`
- `as_int` → conversión a entero con política → `satx.fixed`
- `fixed_mul_floor` → multiplicación con floor → `satx.fixed`

- `fixed_mul_round` → multiplicación con round → `satx.fixed`
- `fixed_bounds` → límites de rango → `satx.fixed`
- `fixed_advice` → guía de escala → `satx.fixed`
- `block_fixed_solution` → bloquea solución → `satx.fixed`
- `enumerate_models_fixed` → enumera modelos → `satx.fixed`

‘satx.unit‘

- `Unit` → entero bit-vector → `satx.unit`

‘satx.constraint‘

- `Constraint` → base de restricciones → `satx.constraint`
- `ConstraintRecord` → metadatos de restricción → `satx.constraint`
- `ConstantConstraint` → constante booleana → `satx.constraint`
- `ComparisonConstraint` → comparaciones → `satx.constraint`
- `InConstraint` → pertenencia a conjunto → `satx.constraint`
- `ImplicationConstraint` → implicación → `satx.constraint`
- `NotConstraint` → negación → `satx.constraint`
- `AndConstraint` → conjunción → `satx.constraint`
- `OrConstraint` → disyunción → `satx.constraint`
- `compile_bool` → reifica a literal → `satx.constraint`
- `as_constraint` → convierte a Constraint → `satx.constraint`
- `apply_constraint` → aplica restricciones → `satx.constraint`

‘satx.rational‘

- `Rational` → racional simbólico → `satx.rational`

‘satx.gaussian‘

- `Gaussian` → entero gaussiano simbólico → `satx.gaussian`

‘satx.gcc‘

- `abs_val` → valor absoluto → `satx.gcc`
- `all_differ_from_at_least_k_pos` → diferencia mínima → `satx.gcc`
- `all_differ_from_at_most_k_pos` → diferencia máxima → `satx.gcc`
- `all_differ_from_exactly_k_pos` → diferencia exacta → `satx.gcc`
- `all_equal` → igualdad global → `satx.gcc`
- `all_equal_except_0` → igualdad excepto 0 → `satx.gcc`
- `all_different` → all-different → `satx.gcc`
- `element` → elemento por índice → `satx.gcc`
- `in_` → pertenencia a dominio → `satx.gcc`
- `not_in` → exclusión de dominio → `satx.gcc`
- `among` → conteo de valores → `satx.gcc`
- `among_var` → conteo variable → `satx.gcc`
- `at_least` → al menos k ocurrencias → `satx.gcc`
- `at_most` → a lo más k ocurrencias → `satx.gcc`
- `exactly` → exactamente k ocurrencias → `satx.gcc`
- `minimum` → mínimo de lista → `satx.gcc`
- `maximum` → máximo de lista → `satx.gcc`
- `sum` → suma con relación → `satx.gcc`
- `scalar_product` → producto escalar → `satx.gcc`
- `nvalue` → número de valores distintos → `satx.gcc`
- `lex_less` → lexicográfico estricto → `satx.gcc`
- `lex_lesseq` → lexicográfico no estricto → `satx.gcc`
- `lex_greater` → lexicográfico estricto $>$ → `satx.gcc`
- `lex_greatereq` → lexicográfico no estricto $>$ → `satx.gcc`
- `lex_chain_less` → cadena lexicográfica $<$ → `satx.gcc`
- `lex_chain_lesseq` → cadena lexicográfica $\leq$ → `satx.gcc`
- `lex_chain_greater` → cadena lexicográfica $>$ → `satx.gcc`
- `lex_chain_greatereq` → cadena lexicográfica $\geq$ → `satx.gcc`

‘satx.alu‘

Parte II

SEMÁNTICA

CAPÍTULO 2

SEMÁNTICA FORMAL Y PATRONES DE USO

Este documento complementa con una especificación **semántica** y **matemática** del uso de SATX como compilador a CNF con:

- **SAT exacto** (existencia y síntesis).
- **#SAT exacto** (conteo de modelos y cocientes de conteo como operador semántico).

Fuente de verdad: implementación en `satx/` (principalmente `satx/stdlib.py`, `satx/constraint.py`, `satx/unit.py`, `satx/alu.py`).

2.1. Núcleo semántico: de 'Unit' a CNF

Universo booleano y CNF

Sea B el conjunto de variables booleanas internas de la CNF (en DIMACS, enteros positivos). Una asignación total es $\mu \in \{0,1\}^B$.

Una fórmula CNF Φ es una conjunción finita de cláusulas; cada cláusula es una disyunción de literales. SATX construye Φ incrementalmente mediante puertas y restricciones bit-vector.

'Unit' como entero bit-vector

Un `Unit` representa un entero mediante un vector de bits booleanos de longitud `w`:

- En modo unsigned (`csp.signed=False`): valores en $U_w = \{0, \ldots, 2^w - 1\}$.
- En modo signed (`csp.signed=True`): valores en complemento a dos $S_w = \{-2^{w-1}, \ldots, 2^{w-1} - 1\}$.

En el código:

- `satx.engine(bits=w, signed=...)` fija el ancho global por defecto.

- `satx.integer()` crea un `Unit` de ancho `w` (salvo que se especifique `bits=...`).

  Un `Unit` posee:

- `block`: lista de literales DIMACS (uno por bit).
- `value`: valor entero asignado luego de resolver (`satx.satisfy` / `satx.synthesize` con `apply_model`).

Semántica formal de anchos de bits

- Un `Unit` tiene ancho fijo $w = $ `len(block)`; ese ancho define su dominio.
- El dominio signed/unsigned es global y lo define `engine.signed`.
- `satx.engine(bits=...)` fija el ancho por defecto; se usa solo cuando no se especifica `bits` en variables o constantes.
- El ancho del motor no es un máximo: un `Unit` puede tener `bits` distinto sin coerción implícita.

Políticas de coerción de ancho (`width_policy`)

- Cuando dos `Unit` difieren en ancho, SATX alinea los operandos antes de operar.
- `promote` (default): $W = \max(w_1, w_2)$; se extiende con ceros si unsigned o con signo si signed.
- `strict`: error inmediato si $w_1 \neq w_2$ (`ValueError`).
- `legacy`: $W = \min(w_1, w_2)$; trunca bits altos (solo compatibilidad).
- Aritmética y bitwise retornan ancho W; comparaciones usan W para alinear.
- Si un operando no es `Unit`, se crea una constante con el ancho del otro; si ambos son literales, usa `bits()`.

```
import satx
satx.engine(bits=8, width_policy="promote")
a = satx.integer(bits=3)
b = satx.integer(bits=5)
s = a + b        # ancho 5
```

Overflow y truncación

- `ADD`/`SUB`/`MUL` no permiten overflow; si el resultado no cabe en el ancho resultante, la CNF queda UNSAT.
- No hay wrap-around ni saturación implícita en operaciones aritméticas.
- Literales fuera de rango: al crear una constante de ancho W, con `const_policy="strict"` (default) se lanza `ValueError`; con `"warn"` se hace wrap (módulo 2^W) y se emite `UserWarning`; con `"wrap"` se hace wrap silenciosamente.

- La truncación solo ocurre vía `trunc()` o `resize(..., mode="trunc")`, o en `legacy` al alinear anchos.

```
import satx
satx.engine(bits=3)
x = satx.constant(7, bits=3)
y = satx.constant(1, bits=3)
z = x + y        # overflow -> UNSAT
```

```
import satx
satx.engine(bits=8)
a = satx.integer(bits=3)
b = satx.integer(bits=5)
t = (a + b).trunc(3)  # truncación explícita
```

Casts explícitos de ancho

- `x.zext(W)`: extiende a W con ceros.
- `x.sext(W)`: extiende a W replicando el bit más significativo.
- `x.trunc(W)`: conserva bits bajos y descarta los altos.
- `x.resize(W, mode=...)`: si $W > w$, extiende (por defecto según `signed`); si $W < w$, requiere `mode="trunc"`.

```
import satx
satx.engine(bits=8, signed=False)
x = satx.integer(bits=3)
x5 = x.zext(5)
x2 = x.resize(2, mode="trunc")
```

```
import satx
satx.engine(bits=8)
x = satx.integer(bits=4)
x.resize(2)  # ValueError: resize would truncate
```

Comparaciones y coerción

- `==, !=, <, <=, >, >=` alinean operandos según `width_policy` antes de comparar.
- La semántica signed/unsigned viene de `engine.signed`.

```
import satx
satx.engine(bits=8, signed=False, width_policy="promote")
a = satx.integer(bits=1)
b = satx.integer(bits=2)
satx.add(a == b)   # fuerza b[1] == 0
```

```
import satx
satx.engine(bits=8, width_policy="legacy")
a = satx.integer(bits=1)
b = satx.integer(bits=2)
satx.add(a == b)   # trunca b a 1 bit (compatibilidad)
```

Desplazamientos: ‘«‘/‘»‘ vs ‘shl‘/‘shr‘/‘ashr‘

- Por compatibilidad histórica, `Unit.__lshift__` (`x « k`) y `Unit.__rshift__` (`x » k`) no implementan shift bit-a-bit clásico.
- En SATX, `x « k` se modela como escalamiento aritmético `x * (2*k)` y `x » k` como `x // (2*k)`.
- Para shift bit-a-bit dentro del ancho fijo (descarta overflow, rellena con ceros o con signo), use:
- `x.shl(k)`: shift lógico a la izquierda (fill=0).
- `x.shr(k)`: shift lógico a la derecha (fill=0).
- `x.ashr(k)`: shift aritmético a la derecha (sign-extend si `signed=True`).
- Si `k` es una `Unit`, SATX soporta shift variable; en modo signed impone `k >= 0`.

```
import satx
satx.engine(bits=8, signed=False)
x = satx.integer(bits=8)
y = x.shl(2)
z = x.shr(1)
```

Traducción a CNF (visión conceptual)

- Zero-extension: los bits nuevos se fijan a la constante 0.
- Sign-extension: los bits nuevos copian el MSB.
- Truncación: descarta bits altos sin agregar cláusulas.
- `resize` combina estas operaciones; preserva semántica bit a bit.
- Con `simplify=True`, una cláusula vacía se emite como `0`, y `_block_model` reescribe el header DIMACS al final.

Warnings y modos seguros

- `width_warn=True` emite `UserWarning` cuando se alinea un mismatch de anchos.
- `width_policy="strict"` + casts explícitos es el modo recomendado para modelos de producción.
- Diferencia clave: un modelo puede ser válido pero sorprendente si hay coerción implícita.

‘Constraint‘ y reificación

Una restricción C (objeto `Constraint`) representa un predicado sobre B. La reificación (`Constraint.reify()`) produce un literal ℓ tal que:

$\ell = 1 \Leftrightarrow C$ (en el sentido de satisfacibilidad bajo Φ).

SATX implementa conectivos a nivel de restricciones:

- Negación: `~C`
- Conjunción: `C1 & C2`
- Disyunción: `C1 | C2`

Véase `satx/constraint.py` (`NotConstraint`, `AndConstraint`, `OrConstraint`, `ComparisonConstraint`).

Registro de restricciones: ‘satx.add‘ / ‘satx.require‘

El procedimiento `satx.add(expr, ...)` registra restricciones en el motor actual.

Hechos verificables en `satx/stdlib.py`:

- `expr` se normaliza: callables se evalúan; secuencias se aplanan.
- Cada restricción se registra con un literal de activación (para habilitar selección por `kind`/`label`).
- `expr` debe ser convertible con `satx.constraint.as_constraint`.

Consecuencia práctica:

- `satx.add(1)` es inválido: `int` no es un tipo de restricción.
- Para expresar una condición booleana debe construirse una `Constraint` (p.ej. `x == 1`, `x >= 0`, `satx.any([...])`).

Validación y garantías

- La semántica de anchos fue validada con enumeración exhaustiva en dominios pequeños.
- Validación registrada en `validate_bit_width.py`: evaluador bit-vector en Python y verificación cruzada de conjuntos SAT.
- Los casos cubren operaciones con anchos distintos, casts explícitos y comparaciones alineadas.
- Se eliminan clases de errores previos: truncación silenciosa fuera de `legacy` y CNF corrupta por cláusulas vacías.
- SATX no garantiza proyección #SAT ni corrección semántica si se usa `legacy` o se ignoran warnings.

Limitaciones conocidas

- Algunas utilidades internas usan el ancho del motor (p.ej. `ALU.sqrt`); revisar si se espera otro ancho explícito.
- `ALU.element` y rutinas derivadas asumen estructuras basadas en el ancho del motor y tamaños de listas.
- Las constantes fuera de rango se truncan bit a bit (no hay validación de rango).
- En streaming CNF, el header DIMACS se escribe al final; el archivo es consistente solo tras el cierre/reescritura.

2.2. SAT exacto: existencia, síntesis y enumeración

‘satx.satisfy‘ y asignación de valores

Sea Φ la CNF acumulada en el motor.

- Si Φ es satisfacible, `satx.satisfy(solver=...)` obtiene un modelo μ y asigna a cada `Unit` un `value` coherente con sus bits.
- Si es insatisfacible, retorna `False`.
- Si el solver externo falla o la salida no cumple el contrato DIMACS, lanza `RuntimeError` (no se confunde con UNSAT).

La asignación ocurre por decodificación del modelo del solver externo (vía `_apply_model_values` en `satx/stdlib.py`).

‘satx.synthesize‘

`satx.synthesize(vars, ...)` resuelve SAT y devuelve los valores de `vars` en algún modelo.

Sea $V = [v_1, \ldots, v_k]$ el conjunto de variables consultadas; el retorno es una tupla (o lista) de valores enteros / racionales (para `Fixed`).

Enumeración de soluciones (patrón)

Para enumerar asignaciones distintas de un conjunto de variables `V` sin #SAT proyectado:
1) Resolver y obtener un modelo. 2) Añadir una cláusula de bloqueo que excluya esa asignación sobre `V`. 3) Repetir hasta UNSAT.

SATX provee el bloqueador estándar:

- `satx.block_current(V)` en `satx/stdlib.py`.

Formalmente, si el modelo actual fija V a valores $\alpha = (\alpha_1, \ldots, \alpha_k)$, entonces se agrega:
$(v_1 \neq \alpha_1) \vee (v_2 \neq \alpha_2) \vee \cdots \vee (v_k \neq \alpha_k)$.

Esta operación requiere que cada `vi` tenga `value` asignado (i.e. provenga de un modelo ya aplicado).

2.3. #SAT exacto: conteo, exposición y distribución

‘satx.counting‘

Sea Φ la CNF del motor (más cláusulas extra, si se especifican). `satx.counting(solver=...)` retorna:

$\#(\Phi) = |\{\mu \in \{0,1\}^B : \Phi(\mu) = 1\}|$.

SATX no incluye contador; `solver` referencia un ejecutable externo que debe emitir un conteo en un formato DIMACS compatible (ver parser en `satx/stdlib.py:_parse_dimacs_counter_output`).

‘satx.exposure‘

`satx.exposure(event, given=..., ...)` computa un cociente de conteos.

Formalización:

- Sea Φ la CNF base.
- Sea `G` el conjunto de cláusulas unitarias inducidas por `given`.
- Sea `E` el literal de reificación del evento (`E = reify(event)`).

Entonces:

- $total = \#(\Phi \wedge G)$
- $count = \#(\Phi \wedge G \wedge E)$
- `ratio = count / total` (si `normalize=True`).

Hechos verificables:

- `given` se procesa por `_fixing_clauses` (`satx/stdlib.py`): acepta solo `Unit` o `Fixed` con valores ya asignados (se fijan sus bits al modelo actual). No acepta restricciones.
- Para condicionar por una restricción lógica `C`, debe registrarse `C` en Φ mediante `satx.add(C)` y luego invocar `exposure` sin `given` (o además fijar asignaciones si se requiere).

‘satx.distribution‘

`satx.distribution(expr, values=[...], given=..., ...)` computa una distribución discreta por conteos:

- Para cada valor $v \in values$, se cuenta $\#(\Phi \wedge G \wedge (expr = v))$.
- El resultado puede normalizarse por el total.

Si `expr` es booleano (`Constraint`), el conjunto de valores por defecto es `{True, False}`.

2.4. Cocientes #SAT como semántica de “probabilidad” (con convenio explícito)

SATX no impone una interpretación probabilística; sin embargo, es común inducirla mediante variables exógenas uniformes.

Variables exógenas uniformes

Sea `U` el conjunto de bits (o `Unit` discretos) introducidos como micro-azar, típicamente:

- unconstrained, o
- con restricciones de rango simples (p.ej. $0 \leq u \leq p\_den - 1$).

Si el modelo asegura correspondencia uniforme entre elecciones de `U` y modelos completos, entonces:
$P(E \mid \Phi) := \#(\Phi \wedge E)/\#(\Phi)$.

Advertencia estructural: multiplicidad por variables auxiliares

El conteo #SAT en SATX cuenta asignaciones sobre **todas** las variables booleanas en CNF, incluyendo auxiliares del encoding.

Para que un cociente sea interpretable como probabilidad sobre un conjunto explícito de elecciones, debe evitarse que existan variables auxiliares “libres” (no determinadas) que introduzcan factores multiplicativos no deseados.

En términos formales, si $B = U \uplus W$ y la proyección de $Sol(\Phi)$ sobre U no es una biyección, entonces $\#(\Phi)$ no coincide con $2^{|U|}$ y el cociente no se interpreta como probabilidad uniforme sobre U sin corrección adicional.

2.5. Teorías vs trayectorias: conteo proyectado y enumeración

En problemas “teoría-nivel” (TRANSCENDENTAL), interesa distinguir:

- τ: variables que codifican la teoría (leyes, observadores, estructura).
- `x`: variables de trayectoria/estado.

SATX provee #SAT **sin proyección**: `counting/exposure` cuentan modelos completos $(\tau, x, \dots)$.

Consecuencia

El valor:
$\#\{(\tau, x) : \Phi(\tau, x) \wedge O(x)\}$
cuenta “explicaciones” (teoría+trayectoria), no “teorías” distintas.

Patrón para contar teorías distintas (sin #SAT proyectado)

Para obtener:
$|\{\tau : \exists x,\ \Phi(\tau, x) \wedge O(x)\}|$
se usa enumeración SAT sobre τ:
1) `sol = synthesize(tau)` 2) registrar `tau` en un conjunto 3) `block_current(tau)` 4) repetir hasta UNSAT.

Este patrón es el implementado en `categories/transcendental/Transcendental_III.py`.

2.6. Patrones correctos y anti-patrones verificables

Patrones correctos

- **Selección condicional**: usar `satx.implies(A, B)` o `Unit.iff(bit, other)`; no usar `if` de Python sobre restricciones.
- **Bloqueo de modelos**: usar `satx.block_current(vars)` en loops de enumeración.
- **Disyunciones/Conjunciones**: usar `satx.any([...])` / `satx.all([...])` o `OrConstraint` / `AndConstraint`.
- **Conteo condicionado**:
- condición lógica: `satx.add(C)` y luego `satx.exposure(E, ...)`;
- fijación de valores (modelo ya resuelto): `satx.exposure(E, given=[...], ...)`.

Anti-patrones (errores comunes)

- **Evaluar una restricción como booleano**:
- `if (x == 1): ...` lanza `TypeError` (`Constraint.__bool__` no está definido).
- **Bloqueo manual con variables ya evaluadas**:
- después de resolver, expresiones como `(x == 3)` pueden colapsar a `bool`; combinarlas con `&`/`|` puede producir `int` y romper `satx.add`.
- usar `satx.block_current([x, ...])`.
- **Olvidar `solver` en #SAT**:
- `satx.counting()` / `satx.exposure()` requieren un comando de contador no vacío (`solver != ""`).
- **Usar `given` como si fuera una restricción**:
- `given` fija valores actuales de `Unit`/`Fixed`; no acepta `Constraint`.
- **Comparaciones con anchos distintos sin control explícito**:
- en `promote` se extiende a $W = \max(w_1, w_2)$; en `legacy` se trunca; en `strict` lanza error.

- práctica recomendada: usar casts explícitos o activar `width_policy="strict"`.
- **Uso incorrecto de `as_fixed` / `fixed_div_exact`**:
- `as_fixed(u, scale)` requiere `scale` explícito (para envolver enteros, usar `scale=1`).
- `fixed_div_exact(a, b, scale=None)` retorna `Rational` cuando `scale is None`; si se necesita un `Fixed` (con `.raw`/`.scale`), proveer `scale=...`.

CAPÍTULO 3

$H10_b$

3.1. Motivación

Todo procedimiento computable corre sobre estados físicos finitos; por tanto, la noción relevante de *existencia de una solución* para un agente físico es la existencia **representable y verificable**. El objeto operacional natural no es el H10 clásico sobre $\mathbb{Z}$ infinito, sino una familia **tipada por recursos** (ancho de bits y semántica aritmética) cuya satisfacibilidad se decide con **certificados finitos** (SAT).

3.2. Tipado semántico: dos cuantificadores, dos mundos

Sea $p(\vec{x}) \in \mathbb{Z}[\vec{x}]$ un polinomio diofántico en n variables.

H10 ideal (clásico)

$$H10_\infty(p) \ : \ \exists \vec{x} \in \mathbb{Z}^n \ \ p(\vec{x}) = 0.$$

Este predicado es sobre una estructura infinita. Es un objeto metamatemático legítimo, pero no es directamente verificable por un agente finito.

$H10_b$

Fijamos un ancho $b \in \mathbb{N}^+$ y una semántica aritmética σ. Definimos un dominio finito:

$$D_{b,\sigma} \subseteq \{0,1\}^b,$$

interpretado como *unsigned* ($U_b = \{0, \ldots, 2^b - 1\}$), *signed* ($S_b = \{-2^{b-1}, \ldots, 2^{b-1} - 1\}$) o como un dominio modular $\{0, \ldots, m-1\}$ cuando $\sigma = \texttt{mod\_m}$. Entonces:

$$H10_{b,\sigma}(p) \ : \ \exists \vec{x} \in D_{b,\sigma}^n \ \ p(\vec{x}) = 0.$$

Aquí $D_{b,\sigma}$ es finito y el predicado es decidible por enumeración finita; de forma más útil, es **compilable** a SAT con certificado.

Relación exacta (torre hacia el ideal)

Bajo la incrustación estándar "una solución entera finita cabe en algún ancho b", se cumple:

$$H10_{\infty}(p) \iff \exists b\, H10_{b,\sigma^*}(p),$$

para una elección razonable σ^* que emule enteros (p. ej. `no_overflow`). Esta equivalencia expresa una relación semántica correcta, pero no produce un decisor total: el cuantificador $\exists b$ introduce un límite ideal.

3.3. Por qué el límite $b \to \infty$ no produce un decisor total

La familia $H10_{b,\sigma}(p)$ es, en el sentido natural de incrustación de dominios, monótona en b: si existe una solución en algún ancho, entonces existe en anchos mayores. Por tanto:

- Si existe una solución ideal, algún b finito eventualmente la encontrará (semidecidibilidad).
- Si no existe solución ideal, ningún prefijo finito de cotas puede certificar "nunca aparecerá" sin información adicional.

Colapsar la torre

$$H10_{\infty}(p) = \bigvee_{b \geq 1} H10_{b,\sigma}(p)$$

en un decisor total requeriría un módulo efectivo de convergencia, es decir, una función computable $B(p)$ tal que:

$$\Big(\exists \vec{x} \in \mathbb{Z}^n :\ p(\vec{x}) = 0\Big) \Rightarrow \Big(\exists \vec{x} \in D^n_{B(p),\sigma} :\ p(\vec{x}) = 0\Big),$$

y tal B universal no existe en general.

3.4. SAT como interfaz para la verdad operacional

Para cada b y σ, el sistema construye una instancia SAT $\Phi_{p,b,\sigma}$ tal que:

$$\Phi_{p,b,\sigma} \text{ es SAT} \iff H10_{b,\sigma}(p).$$

Un resultado SAT proporciona un **testigo** (asignación a las variables). Un resultado UNSAT puede ir acompañado de **pruebas** (DRAT/FRAT) cuando el pipeline externo lo soporta.

3.5. Consecuencia epistémica

- $H10_{\infty}$ es un objeto semántico ideal: bien definido en teoría, pero no totalmente verificable por un agente físico.
- $H10_{b,\sigma}$ es verdad operacional: testigo representable y certificado verificable.

Esto no "refuta" resultados clásicos; los reubica: pasan a ser teoremas sobre el ideal ∞, no obstáculos para la verificación finita.

3.6. Implicación práctica: resolver no significa colapsar lo ideal

El objetivo no es proclamar "H10 resuelto", sino establecer:

1. Un objeto computacional real: $H10_{b,\sigma}$ con semántica explícita.

2. Un método verificable: compilación exacta y certificados SAT.

3. Una frontera limpia: el cuantificador $\exists b$ no produce un decisor total sin una cota efectiva general.

3.7. Programa de investigación

1. **Clases con cotas efectivas**: identificar fragmentos de p que admitan un $B(p)$ computable y útil.

2. **Semánticas aritméticas**: comparar `no_overflow`, `wraparound`, saturación y `mod_m` como axiomas operacionales.

3. **Certificados fuertes**: registro de pruebas, minimización de núcleos y trazabilidad reproducible.

4. **Fases de emergencia del testigo**: cómo aparecen soluciones cuando crece b (estructura, no "magia").

5. **Numerales no enteros**: semánticas discretas adicionales (punto fijo/racional) preservando verificación finita.

3.8. Módulo `satx.h10_b`

Tipos

- `H10BSpec{bits,signed,arith,fixed_point?}`: recursos (ancho b) y disciplina aritmética. En el core actual de SATX, `arith="no_overflow"` coincide con la semántica nativa de `Unit`/`Fixed` (overflow ⇒ UNSAT).

- `Quantified`: objeto cuantificado con `quantifier` ∈ `{exists, forall}` y un `body`.

Constructores

- `exists(spec, vars, body) -> Quantified`.

- `forall(spec, vars, body) -> Quantified`.

- `body` es una función que recibe `dict[str, Unit|Fixed]` y retorna una restricción SATX (o una lista de restricciones, interpretadas como conjunción).

Ejecución

- solve(q, mode="SAT"|"COUNT"|"BOTH", backend=..., count_backend=..., want_certificate=...).
- mode="SAT" decide el cuantificador (para ∀ usa la reducción a contraejemplo).
- mode="COUNT" retorna count, total y robustness = count/total.
- want_certificate existe como hook; la API estándar de SATX no expone certificados UNSAT, por lo que el campo se retorna como None.

Exploración

- search_b(problem, b_start, b_max, step=...): barre anchos y retorna el primer b que produce SAT.

Ejemplos

```
import satx

# EXISTS: existe x en D_2 tal que x == 3
spec = satx.h10_b.H10BSpec(
    bits=2,
    signed=False,
    arith="no_overflow",
)
q = satx.h10_b.exists(spec, ["x"], lambda v: v["x"] == 3)
res = satx.h10_b.solve(q, mode="SAT")
assert res["status"] == "SAT"
assert res["witness"]["x"] == 3

# Robustez (#SAT/total) del predicado (aqui: 1/4)
rob = satx.h10_b.robustness(q)
assert abs(rob - 0.25) < 1e-9

# FORALL: para todo x en D_2, x - x == 0
q2 = satx.h10_b.forall(
    spec,
    ["x"],
    lambda v: (v["x"] - v["x"]) == 0,
)
res2 = satx.h10_b.solve(q2, mode="SAT")
assert res2["status"] == "SAT"
```

3.9. Módulo satx.balanced_cnf

Propósito

balanced_cnf implementa un fast-path exacto para un fragmento particular de CNF: las *CNFs balanceadas*, donde cada cláusula contiene exactamente un literal por variable. En este

caso (y sin cláusulas repetidas) existe una correspondencia directa entre cláusulas y asignaciones falsificadoras, lo que permite:

- computar un entero S_F (SAT equation) que codifica qué asignaciones fallan;
- obtener el conteo SAT en forma cerrada: $\#SAT(F) = 2^n - |F|$;
- extraer un testigo eligiendo un índice k con bit $b_k = 0$.

API

- `is_balanced_clause(clause, vars) -> bool`.
- `is_balanced_cnf(cnf) -> (bool, detail)`.
- `sat_equation_SF(F) -> int` (falla si hay cláusulas repetidas).
- `sat_count(F) -> int` y `find_witness(F) -> model|None`.

3.10. Módulo satx.agency_lr

Propósito

`agency_lr` implementa utilidades numéricas (sin simulación) para bounds tipo Lieb–Robinson: una cota de agencia Ag, y funciones derivadas como `blast_radius` y `dmax`.

API

- `LRConstants(mu, vLR, KA)`: constantes del bound.
- `agency_bound(d, T, deltaH_L1, const, sizeC, sizeR) -> float`.
- `blast_radius(T, deltaH_L1, delta_threshold, const, sizeC=?, sizeR=?) -> float`.
- `dmax(...)`: alias de `blast_radius`.

3.11. Módulo satx.theory

Propósito

`satx.theory` implementa una capa semántica para tratar una teoría como un subconjunto de asignaciones públicas sobre un dominio finito tipado por recursos (vía `H10BSpec`):

$$\mathrm{Mod}_\Sigma(A) \subseteq \mathrm{Assign}_\Sigma(Y),$$

con variables privadas existenciales internas y operaciones de composición.

Firma (ontología)

Una firma (u ontología) Σ fija:

- un orden de variables públicas $Y = (y_1, \ldots, y_p)$ (`Signature.public_vars`);
- una especificación de dominio por variable (`Signature.spec`), con semántica aritmética:
- (uniforme) un `H10BSpec` compartido por todas las variables; o
- (heterogéneo) un diccionario $\{name \mapsto H10BSpec\}$ con `Signature.spec_for(name)`.

La API `signature(...)` soporta ambos estilos preservando compatibilidad hacia atrás.

AST `TheoryExpr`

Nodos principales:

- `Atom(signature, builder, priv_vars=?)`: hoja compilada a restricciones SATX.
- `BalancedBadSet(signature, clauses)`: hoja CNF balanceada sobre booleans ($b = 1$), sin cláusulas duplicadas.
- `Meet(children)` y `Join(children)`: meet/join semánticos (intersección/unión en Y).
- `NotFull(child)`: negación a nivel de fórmula (antes de proyectar privadas).
- `NotPublic(child)`: complemento público: $\neg_{\text{pub}}$.
- `Exists(vars, child)` y `Forall(vars, child)`: binders sobre dominios finitos.
- `Lift(child, to_signature)` y `Rename(child, mapping)`: unificación/higiene de ontologías.

Higiene de privadas (α-renaming)

Al combinar subárboles hermanos, `meet` y `join` renombra variables privadas para mantenerlas disjuntas. Esto garantiza que las leyes a nivel de conjuntos de mundos públicos no se rompan por colisiones accidentales de testigos internos.

Conteo proyectado

Para $Y_0 \subseteq Y$, `count(expr, project=Y0)` implementa:

$$\#_{Y_0}(A) = \left|\{y_0 : \exists y_{Y \setminus Y_0}, z; \varphi_A(y, z)\}\right|.$$

La respuesta incluye:

- `count`: cardinalidad proyectada;
- `total` $= |D_b|^{|Y_0|}$;
- `robustness`=`count`/`total`.

Planner de backends (MVP)

- **Balanced**: si la teoría es un `BalancedBadSet` (CNF balanceada) se usa conteo por bloques.
- **Bitset**: si $|\text{Assign}(Y_0)|$ es pequeño, se enumera exactamente y se cuenta por conjunto/bitset.
- **Blocking**: fallback exacto que itera modelos SAT y bloquea sólo la proyección sobre Y_0.
- **Components**: cuando el CNF se separa en componentes independientes, el conteo se factoriza como producto.
- **XOR-hash (aprox)**: con `approx=True`, el conteo proyectado puede aproximarse por hashing XOR determinista (controlado por `hash_pivot`/`hash_trials`/`hash_seed`).

Cuando existen cuantificadores universales (`Forall`/`NotPublic`), `solve(...)` intenta un esquema CEGAR para fragmentos soportados y cae a enumeración completa como fallback. `count(...)` enumera el espacio público en el caso universal.

API

- `signature(public_vars, spec) -> Signature` (uniforme) y `signature({name: spec, ...}, order=[...])` (heterogéneo).
- `atom(signature, builder, priv_vars=?)`.
- `meet(...)`, `join(...)`, `not_full(...)`, `not_public(...)`, `exists(...)`, `forall(...)`.
- `solve(expr, project=?)` y `count(expr, project=?)`.
- `robustness(expr, project=?)`.

Ejemplo

```
import satx
th = satx.theory
spec = satx.h10_b.H10BSpec(bits=2, signed=False, arith="no_overflow")
sig = th.signature(["x", "y"], spec)

# phi(x,y) := (x==1) or (y==2)
phi = th.join(
  th.atom(sig, lambda v: v["x"] == 1),
  th.atom(sig, lambda v: v["y"] == 2),
)

# #_x(phi) = 4, pues para todo x existe y=2.
res = th.count(phi, project=["x"])
assert res["count"] == 4 and res["total"] == 4
```

3.12. FDAS (Theory/Query/Result/Algebra/Planner)

Propósito formal: FDAS introduce una separación portable (solver-free) entre:

- `Theory`: snapshot inmutable/serializable del modelo (variables + restricciones + partición pública/privada).
- `Query`: intención tipada (p.ej. SAT, COUNT, PROJECTED_COUNT) con `semantic_hash()` estable.
- `Result`: resultado tipado con `status` y `evidence`.
- `Algebra`: composición cerrada (`meet/join/hide/exists/rename/project/not_public/not_full`), con `hide` como alias de `exists`.
- `Planner`: reglas deterministas + trazabilidad + cache por hash semántico.

Semántica matemática de `Theory` (PUBLIC/PRIVATE)

Firma finita. Una teoría FDAS fija un conjunto finito de variables X con dominios finitos:

$$\Sigma = (X, \mathcal{D}), \qquad X = \{x_1, \ldots, x_n\}, \quad \mathcal{D}(x_i) \text{ finito.}$$

La clase `satx.Theory` particiona X en observables **PUBLIC** Y y ocultas **PRIVATE** Z, con $X = Y \uplus Z$.

Restricciones. Las restricciones (`constraints`) denotan una fórmula finita $\varphi(X)$ (conjunción o AST booleano sobre átomos).

Formato portable de `ConstraintRef.expr` (FDAS expr)

En FDAS, cada átomo `ConstraintRef` contiene un campo `expr: str` que es una representación *portable* y *parseable* de una restricción. Esta representación es deliberadamente restringida para permitir:

- reconstrucción de un engine interno cuando `Theory` no está alineada con el engine activo,
- serialización determinista (hash semántico) y trazabilidad.

Gramática (normativa). El lenguaje de `expr` se define por:

$$\begin{array}{rcl} e & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{null} \mid n \mid x \mid s \mid [e, \ldots, e] \mid (e\ op\ e) \mid f(e, \ldots, e) \\ op & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\ f & ::= & \texttt{not} \mid \texttt{and} \mid \texttt{or} \mid \texttt{implies} \mid \texttt{in} \mid \texttt{notin} \end{array}$$

donde n es un entero decimal, x un identificador de variable, y s un string JSON (con comillas dobles y escapes JSON).

Invariante sintáctico clave. Todas las comparaciones $(e\ op\ e)$ deben estar *parentizadas*. Por ejemplo, se permite `(x == 1)` y **no** se permite `x == 1`.

Notas. `and(...)` y `or(...)` aceptan cero argumentos, interpretándose como $\top$ y $\bot$ respectivamente. Tokens opacos (p.ej. `<?>` o `<...>`) indican snapshots *lossy* y no son reconstruibles; en esos casos `Theory.metadata["lossy_constraints"]=True`.

Modelos completos. El conjunto de modelos completos es:

$$\mathrm{Mod}_{\mathrm{full}}(\varphi) = \{\, \mu : X \to \mathcal{D} \;:\; \mu \models \varphi \,\}.$$

Proyección pública. La proyección pública es $\pi_{\mathrm{pub}}(\mu) = \mu|_Y$. El contenido público de una teoría es el conjunto:

$$\mathrm{Mod}_{\mathrm{pub}}(\varphi; Y, Z) = \{\, \nu : Y \to \mathcal{D} \;:\; \exists \omega : Z \to \mathcal{D} \;\; (\nu \uplus \omega) \models \varphi \,\}.$$

Esta es la semántica observacional usada por conteo proyectado, explicación y planning.

Operadores fundamentales

Sea $\Phi = (\Sigma, \varphi, Y, Z)$ y $\Psi = (\Sigma', \psi, Y', Z')$ (con firmas compatibles tras unificación). Las operaciones FDAS se interpretan por semántica pública:

- **Meet** $\Phi \wedge \Psi$: conjunción semántica. Garantía operacional: preserva higiene por α-renaming de privadas para evitar colisiones entre componentes.
- **Join** $\Phi \vee \Psi$: disyunción semántica (AST).
- **exists/hide** (ocultar): mover variables de Y a Z (existenciales en la interfaz), sin cambiar la fórmula base.
- **exists_private** ($\exists Z$): operador de proyección existencial explícita a la interfaz pública:

  $$\exists_{\mathrm{priv}}\, \Phi \;\equiv\; \exists Z.\, \varphi(Y, Z),$$

  produciendo una teoría equivalente sobre Y (sin privadas) cuando es computable en mundos finitos pequeños.
- **not_full** ($\neg$): negación clásica de la fórmula completa φ.
- **not_public** ($\neg(\exists Z\, \varphi)$): complemento en la interfaz pública:

  $$\neg_{\mathrm{pub}}\, \Phi \;:=\; \neg(\exists Z\, \varphi) \;\equiv\; \forall Z\, \neg\varphi.$$

- **rename_private** (α-renaming): renombrado inyectivo de variables privadas que preserva $\mathrm{Mod}_{\mathrm{pub}}$ y evita capturas al componer.

Hash/freeze determinista

`Theory.semantic_hash()` se define como SHA-256 de una serialización JSON canónica de:

- firma normalizada (lista ordenada por nombre con dominios normalizados),
- restricciones normalizadas,
- partición (Y, Z) (listas ordenadas),
- versión de SATX (para evitar colisiones inter-versión),

excluyendo `metadata` por defecto (incluirla es explícito).

API mínima

- `Phi = satx.Theory.from_engine()` captura el modelo actual (portabilidad).
- `q = satx.Query(kind=..., theory=Phi, params={...})` representa una consulta.
- `r = satx.execute(q, policy={...})` ejecuta una consulta y retorna `Result`.
- `r = satx.ask(...)` construye plan + ejecuta; con `PlannerPolicy(explain=True)` retorna también `PlannerTrace`.

Cache y trazas del planner

- Cache en memoria controlado por `PlannerPolicy.cache` y `PlannerPolicy.cache_max`.
- Clave: `query.semantic_hash()` + política de ejecución (determinista).
- La última traza queda disponible vía `satx.last_planner_trace()` (y se limpia con `satx.clear_planner_cache()`).

PROJECTED_COUNT en FDAS

- `QueryKind.PROJECTED_COUNT` está soportado por `execute` usando un backend interno solver-free para instancias pequeñas.
- Política: `internal_max_vars` limita el tamaño; si se excede, el resultado es `ERROR` con diagnóstico genérico (no se inventan conteos).

```
import satx

with satx.Engine(bits=1, signed=False, strict=True):
    x = satx.integer(bits=1, force_int=True)
    z = satx.integer(bits=1, force_int=True)
    satx.implies(x == 1, z == 0)

    Phi = satx.Theory.from_engine()
    x_name = Phi.signature[0].name
    q = satx.Query(kind=satx.QueryKind.PROJECTED_COUNT, theory=Phi, params={"project"
    : (x_name,)})
    r = satx.execute(q)
    assert r.status == satx.ResultStatus.COUNT
```

3.13. Módulo satx.inverse

Propósito

`inverse` implementa preimagen y unrolling acotado sobre transiciones finitas expresadas como teorías.

Transición

Una transición es una relación:

$$T(s, a, s') \in \{0, 1\},$$

representada por `Transition(spec, state_vars, action_vars, next_state_vars, relation)`.

Preimagen

`preimage(T, G)` construye:

$$\mathrm{Pre}(G)(s) = \exists a, s'.\ T(s, a, s') \wedge G(s').$$

Unrolling y orígenes

`unroll(T,k)` crea copias renombradas $s_0, \ldots, s_k$, $a_0, \ldots, a_{k-1}$ y retorna $\bigwedge_{t<k} T_t$. `origins_k(T, goal, k, *, limits=None, project=None, state_prefix="s", action_prefix="a")` construye el unrolling, renombra `goal` hacia s_k, puede recibir el keyword `goal=` y proyecta sólo las variables mantenidas por `project`. `origins_k_projected(..., rename_state0=("s0",))` es la versión operación-por-defecto que mantiene s_0 como variable pública nombrada `s0` y renombra los demás estados con el prefijo temporal si se requieren alias personalizados.

3.14. Módulo satx.lms

Propósito

`lms` define un DSL mínimo para Layered Metric Space (LMS) como teoría por capas sobre un grafo.

Grafo

`Graph(V,E)` representa un grafo con vértices y aristas.

Teoría por capas (MVP)

Para $k = 0, \ldots, L-1$ y $e \in E$, se introducen variables $\ell_k(e) \in D_b$. En el MVP:

- (opcional) positividad: $\ell_k(e) > 0$;
- (opcional) monotonicidad inter-capa: $\ell_{k+1}(e) \geq \ell_k(e)$;
- (opcional) acción L_1 acotada:

$$\sum_k \sum_e (\ell_{k+1}(e) - \ell_k(e)) \leq B,$$

con monotonicidad para evitar $|\cdot|$ en la primera versión.

La función `lms_theory(graph, layers, spec, ...)` construye una `TheoryExpr` consultable vía `satx.theory.solve/count`.

3.15. Álgebra de Teorías, Conteo Proyectado y Matemática Inversa

Resumen

Este documento complementario formaliza el núcleo conceptual implementado en SATX:

- una capa *semántica* (independiente de representación) para componer modelos como teorías sobre una ontología finita y tipada;
- cuantificación operacional (existenciales/universales) con higiene de variables;
- conteo proyectado como cardinalidad de proyecciones existenciales;
- matemática inversa: preimagen y *unrolling* temporal acotado;
- un puente operacional hacia Layered Metric Space (LMS) como dinámica discreta por capas.

El marco es *operacional*: trabaja sobre dominios finitos tipados por recursos (ancho b) y compila a SAT con testigos (y, si el pipeline lo soporta, certificados).

Palabras clave

- SAT, modelos finitos, cuantificadores operacionales, conteo proyectado, planificación inversa, LMS, verificación con certificados.

Fundamento operacional: torre tipada por recursos $H10_b$

Dos cuantificadores, dos mundos

Sea $p(\vec{x}) \in \mathbb{Z}[\vec{x}]$. Distinguimos:

- **Ideal (clásico)**:
$$H10_\infty(p) \ : \ \exists \vec{x} \in \mathbb{Z}^n \ \ p(\vec{x}) = 0.$$
- **Operacional (relativizado por recursos)**: fijando ancho b y semántica aritmética declarada sobre un dominio finito D_b:
$$H10_b(p) \ : \ \exists \vec{x} \in D_b^n \ \ p(\vec{x}) = 0.$$

Relación con el ideal y frontera no algorítmica

Se cumple la equivalencia (meta-teórica):

$$H10_\infty(p) \iff \exists b \, H10_b(p),$$

pero no induce un decisor total: el cuantificador $\exists b$ no produce una cota efectiva universal.

SAT como interfaz de verdad operacional

Para cada b, $H10_b(p)$ se compila a una instancia SAT $\Phi_{p,b}$ con equivalencia semántica. SAT produce un testigo; UNSAT puede acompañarse de prueba (DRAT/FRAT) si el pipeline lo soporta, haciendo la verdad auditable.

Ontología y teoría: el objeto semántico

Ontología (firma)

Una *ontología* (o firma) Σ fija el universo sobre el cual hablan los modelos:

$$\Sigma = (Y, \mathcal{D}, \preceq),$$

donde $Y = (y_1, \ldots, y_p)$ son variables públicas (observables), $\mathcal{D}$ fija el dominio por variable, y $\preceq$ fija una indexación de asignaciones (relevante para representaciones compactas). Las variables privadas Z existen como auxiliares de compilación, testigos internos o variables de Tseitin.

Teoría como subconjunto de mundos públicos

Una teoría T denota un conjunto de mundos públicos:

$$\mathrm{Mod}_\Sigma(T) \subseteq \mathrm{Assign}_\Sigma(Y).$$

Si T se representa por una fórmula completa $\varphi(Y, Z)$, su contenido público es la proyección existencial:

$$[\![\varphi]\!]_{\mathrm{pub}} \;:=\; \{\, y \in \mathrm{Assign}(Y) \;:\; \exists z\, \varphi(y, z) \,\}.$$

Álgebra de teorías (semántica independiente de representación)

Sea Σ una ontología fija. Para teorías A, B sobre Σ:

Operaciones fundamentales

- **Meet** (conjunción semántica): $\mathrm{Mod}(A \wedge B) = \mathrm{Mod}(A) \cap \mathrm{Mod}(B)$.
- **Join** (disyunción semántica): $\mathrm{Mod}(A \vee B) = \mathrm{Mod}(A) \cup \mathrm{Mod}(B)$.
- **Complemento público** (operador estructural): si A está representada por $\varphi(Y, Z)$, definimos

$$\neg_{\mathrm{pub}} A \;:=\; \neg(\exists Z\, \varphi) \;\equiv\; \forall Z\, \neg\varphi.$$

Leyes booleanas (a nivel de conjuntos)

Estas leyes valen por semántica (conjuntos de mundos públicos): conmutatividad, asociatividad e idempotencia, absorción $A \wedge (A \vee B) = A$, $A \vee (A \wedge B) = A$, y De Morgan público:

$$\neg_{\mathrm{pub}}(A \wedge B) = \neg_{\mathrm{pub}} A \vee \neg_{\mathrm{pub}} B, \qquad \neg_{\mathrm{pub}}(A \vee B) = \neg_{\mathrm{pub}} A \wedge \neg_{\mathrm{pub}} B.$$

Higiene de privadas (α-renaming)

Para preservar equivalencias a nivel público bajo composición, al combinar subárboles (especialmente en `Join`, Tseitin y *unrolling* temporal) se realiza α-renaming de variables privadas para mantenerlas disjuntas entre componentes.

Conteo proyectado: cardinalidad de proyecciones

Definición

Sea $\varphi(Y, Z)$ una teoría completa y sea $Y_0 \subseteq Y$. El conteo proyectado se define por:

$$\#_{Y_0}(\varphi) \;:=\; \Big|\Big\{y_0 \in \mathrm{Assign}(Y_0) \;:\; \exists y_{Y\setminus Y_0},\, z\; \varphi(y, z)\Big\}\Big|.$$

Equivalente:

$$\#_{Y_0}(\varphi) = \#(\exists((Y \setminus Y_0) \cup Z)\; \varphi).$$

Inclusión–exclusión proyectada

Sea $S_A = \{y : \exists z_A\, A(y, z_A)\}$ y $S_B = \{y : \exists z_B\, B(y, z_B)\}$, con privadas disjuntas. Entonces:

$$\#_Y(A \vee B) = |S_A \cup S_B| = \#_Y(A) + \#_Y(B) - \#_Y(A \wedge B).$$

Dualidad con el complemento público

Si el universo público tiene tamaño $|\mathrm{Assign}(Y)|$, entonces:

$$\#_Y(\neg_{\mathrm{pub}} A) = |\mathrm{Assign}(Y)| - \#_Y(A).$$

Representaciones y backends: CNF balanceada y SAT-equation

En el caso de CNFs balanceadas, la relación de satisfacibilidad puede codificarse de forma exacta como un entero/máscara de falsificaciones, útil para verificación y operaciones algebraicas. En SATX, una teoría puede materializarse (según el planner) como:

- `CNF`: general, delega a solver/$\#SAT$;
- `BalancedBadSet / SAT-equation`: conjunto/máscara de falsificaciones;
- `Bitset`: exacto cuando $|\mathrm{Assign}(Y)|$ es pequeño;
- `Lazy(Tseitin)`: AST que se compila al final.

Matemática inversa: preimagen y planificación inversa

Relación de transición

Sea una dinámica como relación:

$$T(s, a, s') \in \{0, 1\},$$

donde s es estado actual, a una acción/decisión (opcional), y s' el estado siguiente.

Preimagen (un paso)

Dado un objetivo $G(s')$, el conjunto de predecesores se define por:

$$\mathrm{Pre}(G)(s) \;:=\; \exists a, s'\; (T(s, a, s') \wedge G(s')).$$

Inversa acotada a k pasos (*unrolling*)

Con variables $(s_0, \dots, s_k)$ y $(a_0, \dots, a_{k-1})$:

$$\Phi_k = \Big(\bigwedge_{t=0}^{k-1} T(s_t, a_t, s_{t+1}) \Big) \wedge G(s_k) \wedge \text{Limits}.$$

Definimos los orígenes posibles:

$$\text{Origins}_k(s_0) = \exists s_1 \dots s_k,\ a_0 \dots a_{k-1}\ \Phi_k,$$

y el conteo proyectado de orígenes $\#_{s_0}(\Phi_k)$.

LMS como instancia natural de matemática inversa

Sea un grafo $G = (V, E)$ y longitudes por capa $\ell_k : E \to \mathbb{R}_{>0}$. En SATX, operacionalmente se trabaja con $\ell_k(e) \in D_b$ (bit-vector o punto fijo) bajo la filosofía $H10_b$.

Variables de estado por capa

Definimos el estado de la capa k como:

$$s_k := \{\ell_k(e)\}_{e \in E}.$$

Cinemática discreta (derivadas inter-capa)

Para cada arista $e \in E$:

- *strain* inter-capa:
$$\sigma_k(e) = \ell_{k+1}(e) - \ell_k(e),$$
- curvatura inter-capa:
$$R_k(e) = \ell_{k+1}(e) - 2\ell_k(e) + \ell_{k-1}(e).$$

(La curvatura intra-capa puede definirse sobre ciclos y añadirse como término adicional.)

Acciones variacionales como restricciones/objetivos

Se consideran acciones como:

$$S^{(1)}_{\text{LMS}}[\ell] = \alpha \sum_k \sum_{c \in C_k} K_k(c)^2 + \beta \sum_k \sum_{e \in E} \sigma_k(e)^2,$$

$$S^{(2)}_{\text{LMS}}[\ell] = \sum_k \sum_{e \in E} \sigma_k(e)^2 + \mu \sum_k \sum_{(e,e') \in \mathcal{N}} (\ell_k(e) - \ell_k(e'))^2.$$

En SATX, estas acciones se materializan como:

- **factibilidad acotada**: $S \le B$ (SAT + búsqueda en B);
- **optimización**: PB/MaxSAT/MIP o iteración de *bounds*.

Materialización (fase discreta)

La materialización introduce kernels $q_k^{(\lambda)} : V \times V \to [0,1]$ que saturan a $\{0,1\}$ en un subgrafo, formando un *backbone*. En SATX se expresa con booleanos auxiliares y cuantificadores operacionales para robustez/estabilidad.

Límites de agencia como constraints y poda (módulo D)

Los bounds tipo Lieb–Robinson permiten acotar influencia efectiva y, en ingeniería, obtener un *blast radius* determinista:

$$R_{\text{blast}} = v'_{\text{LR}} T + \frac{1}{\mu} \log \left(\frac{K'_A \, |C| \, |R| \, \int_0^T \|\Delta H(s)\| \, ds}{\delta} \right).$$

En SATX/LMS inverso, estos bounds se usan como:

- restricciones que recortan historias imposibles;
- heurísticas de poda para planificación inversa (acotando regiones que pueden influir en k pasos).

Curvatura epistémica como control de discretización (módulo B)

Se introduce una interfaz métrica que cuantifica el error de representación:

$$\text{err}(\sigma) = \delta(e(\sigma), j(\iota(\sigma))), \qquad \kappa_S = \inf_{\sigma \in L} \text{err}(\sigma).$$

Bajo el principio DRP (refinamiento no expansivo), $\kappa_S > 0$ indica incompletitud semántica persistente: el refinamiento no colapsa automáticamente la brecha entre representación y semántica. En SATX, esto sugiere un criterio para:

- incrementar b (más resolución);
- cambiar semántica aritmética;
- enriquecer constraints;

sin pretender colapsar el ideal.

Motor operacional: normalización y selección de backend

SATX opera sobre un AST conceptual `TheoryExpr` con nodos:

$$\{\texttt{Meet}, \texttt{Join}, \texttt{NotPublic}, \texttt{Exists}, \texttt{Forall}, \texttt{Lift}, \texttt{Rename}, \texttt{Atom(repr)}\}.$$

Pases típicos:

1. α-renaming de privadas (higiene).
2. Unificación de ontología (lift/rename al supremo).
3. NNF público: $\neg_{\text{pub}} A \rightsquigarrow \forall Z \, \neg\varphi$.

4. Normalización de proyección: `count(project=Y0)` $\rightsquigarrow$ `count_total(Exists(hidden, Ψ))`.

Planner de conteo proyectado (exacto primero):

- `Bitset` si $|\text{Assign}(Y_0)|$ cabe.
- `BalancedBadSet` si aplica (conteo por bloques).
- Factorización por componentes si el grafo de dependencias separa.
- Compilación (BDD/ZDD/d-DNNF) si está disponible.
- Fallback: projected model counting con solver o aproximación por hashing XOR (si se habilita).

Síntesis: componentes del núcleo

1. Tipado operacional de cuantificación por recursos $H10_b$ y frontera explícita con el ideal.
2. Álgebra de teorías sobre una ontología (meet/join/complemento público), con higiene de privadas.
3. Conteo proyectado como primitivo semántico (cardinalidad de proyección).
4. Matemática inversa como preimagen/*unrolling* + $\exists/\forall$ + proyección + conteo.
5. LMS como dinámica por capas expresable en el marco (acciones + materialización).
6. Límites de agencia como bounds operacionales para constraints/poda.
7. CNF balanceada / SAT-equation como backend de compresión semántica y operaciones aritméticas.

Apéndice: notación mínima de API (abstracta)

- `Theory(Σ, repr)` denota un subconjunto de $\text{Assign}_\Sigma(Y)$.
- `meet(A,B)`, `join(A,B)`, `not_public(A)`.
- `exists(V,A)` / `exists(A,V)`: binder sobre dominios finitos que acepta list/tuple de nombres y normaliza internamente a `tuple[str]`. Cualquier orden es válido, el binder también usa `A.signature.public_vars` después de quitar los ocultos.
- `forall(V,A)` / `forall(A,V)`: equivalente dual que acepta los mismos estilos de llamada y mantiene la higiene de privadas.
- `count(project=Y0)` implementa $\#_{Y_0}$; `project` debe ser un subconjunto de `expr.signature.public_vars` y se observa directamente en el `signature`.
- `preimage(T,G)` implementa $\exists a, s'.\, T \wedge G$.

- `origins_k(T, goal, k, *, limits=None, project=None, state_prefix="s", action_prefix="a")`: genera el unrolling $\bigwedge_{t<k} T_t$, une una meta renombrada y proyecta opcionalmente. Ya acepta `goal=` como alias y deja las variables temporales visibles hasta que se proyectan.
- `origins_k_projected(..., rename_state0=("s0",))`: proyección por defecto sobre s_0 con alias normalizados; útil cuando se desea un identificador canónico para `project=("s0",)` y se puede sobreescribir por componente.
- `unroll(T,k)` construye $\bigwedge_{t<k} T_t$.

Una regla de oro para conteos y soluciones es que `project` debe ser un subconjunto de `expr.signature.public_vars`; inspecciona `expr.signature.public_vars` y usa `exists(vars, expr)` (o `not_public`) para ocultar variables antes de contar o proyectar.

Ejemplo reproducible: Álgebra y matemática inversa en dominio finito

La siguiente receta pone en acción `exists`, `not_public`, `count`, `preimage` y la nueva `origins_k_projected` en un dominio de 2 bits para ilustrar la regla de oro del `project`:

```
spec2 = satx.h10_b.H10BSpec(bits=2, signed=False, arith="no_overflow")
sig_xy = th.signature(["x", "y"], spec2)

A = th.atom(sig_xy, lambda v: v["x"] == 1)
B = th.atom(sig_xy, lambda v: v["y"] == 2)

phi = th.join(A, B)
psi = th.meet(A, B)

res_phi = th.count(phi, project=("x",))
res_not_public = th.count(th.not_public(phi), project=("x",))
res_exists = th.count(th.exists(("y",), phi), project=("x",))

sig_sasp = th.signature(["s", "a", "sp"], spec2)
R = th.atom(sig_sasp, lambda v: [
    v["a"] >= 0,
    v["a"] <= 1,
    v["sp"] == v["s"] + v["a"],
])

T = inv.Transition(
    spec=spec2,
    state_vars=("s",),
    action_vars=("a",),
    next_state_vars=("sp",),
    relation=R,
)

G = th.atom(th.signature(("sp",), spec2), lambda v: v["sp"] == 3)
PreG = inv.preimage(T, G)
origins = inv.origins_k_projected(T, goal=G, k=2)
res_origins = th.count(origins, project=("s0",))
```

```
core, states, _ = inv.unroll(T, k=2)
goal_renamed = th.rename(G, { "sp": states[2][0] })
manual = th.meet(core, goal_renamed)
hidden = [name for name in manual.signature.public_vars if name not in set(states[0])
    ]
manual = th.exists(hidden, manual)
manual = th.rename(manual, {states[0][0]: "s0"})
res_manual = th.count(manual, project=("s0",))
```

Los conteos anteriores muestran cómo se pueden ocultar variables públicas con `exists` antes de fijar `project=("x",)`, cómo se renombra automáticamente `origins_k_projected` para exponer `s0` y cómo se puede reproducir el proceso manualmente con `inv.unroll`, `th.rename` y otro `exists` sobre los nombres temporales de la ejecución.

3.16. Nota final

Mucho ruido proviene de usar el mismo verbo "existe" para dos objetos tipológicamente distintos:

- existe$_{\infty}$: ideal.
- existe$_{b,\sigma}$: operacional.

Una vez que el lenguaje se tipa por recursos, el conflicto desaparece: se están discutiendo problemas distintos bajo criterios de verdad diferentes.

3.17. Version notes / compat

- `exists` y `forall` admiten ahora ambos órdenes y list/tuple de nombres, evitando errores en llamadas heredadas.
- `origins_k` acepta `goal=` y conserva el alias antiguo `goal_on_s_k`, mientras que `origins_k_projected(..., rename_state0)` expone `s0` y normaliza alias para el primer estado.
- `count(project=...)` requiere un `project` contenido en `signature.public_vars`; para mantener miembros públicos adicionales aplica `exists` antes de contar.

CAPÍTULO 4

SIMBÓLICO: CAS/SIR

SATX incluye un backend CAS propio (`satx.cas`) y una capa pública (`satx.sym`) que expone un IR simbólico explícito (SIR). En esta sección resume los contratos semánticos mínimos.

4.1. Representación (SIR)

- Tipo único: `SymExpr := satx.cas.ast.Atom` (alias público `satx.sym.SymExpr`).
- Subtipos relevantes: `Sym`, `Rational`, `Double`, `Str`, `Cons`, `Tensor`.
- Listas propias: se representan con `Cons` y terminan en `NIL = Sym("nil")`.

4.2. Invariantes canónicos

- Forma prefijo: una llamada/operación es una lista `(op a b ...)` donde `op` es `Sym(op)`.
- Producto canónico: `(* ...)` se aplana, agrupa factores iguales y combina potencias con la misma base sumando exponentes (p.ej. $a^u a^v \to a^{u+v}$).
- Tensores: `Tensor.dim` es una tupla de dimensiones, `Tensor.ndim == len(dim)` y `Tensor.elem` es plano row-major con `len(elem) == prod(dim)`.
- Indexación 1-based: los índices empiezan en 1 (keyword `[`, `Engine.get_component`/`Engine.set_component`).

4.3. Normalizaciones del parser (`satx.cas.parser`)

El parser produce AST en prefijo y normaliza:

- Negación unaria: `-x` → `(* -1 x)`.
- División: `a/b` → `(* a (^ b -1))`.

- Comparaciones: `==, <=, >=, <, >` → `testeq/testle/testge/testlt/testgt`.
- Indexación: `A[1,2]` → `([ A 1 2)`.
- Factorial: `x!` → `(factorial x)`.
- Entrada prefijo: ademas de infix, `parse(src)` acepta el prefijo canonico emitido por `Engine.to_prefix` (lector S-expression). Esto permite round-trip: `to_prefix(parse(to_prefix(e)))` = `to_prefix(e)` (para expresiones sin tensores literales).

4.4. Delegación al Engine

Las transformaciones simbólicas se delegan construyendo formas prefijo y evaluándolas con `Engine.eval`:

- `Context.simplify(e)` evalúa `(simplify e)`.
- `Context.float(e)` evalúa `(float e)` (best-effort).
- `Context.derivative(e, ...)` evalúa `(derivative e ...)`.
- `Context.integral(e, ...)` evalúa `(integral e ...)`.
- `Context.defint(e, ...)` evalúa `(defint e ...)`.
- `Context.taylor(e, ...)` evalúa `(taylor e ...)`.

Nota: el conjunto de keywords del CAS está registrado en `satx.cas.core.KEYWORD_TO_METHOD`.

4.5. Simplify y comparaciones

- `simplify(x)` aplica una heurística de simplificación (incluye transformaciones trigonométricas en forma exponencial) y retorna una expresión equivalente con menor complejidad sintáctica.
- Identidades trigonométricas típicas se simplifican automáticamente, por ejemplo: `simplify(sin(x)^2 + cos(x)^2)` → `1` y `simplify(tan(x) - sin(x)/cos(x))` → `0`.
- El operador `==` (AST: `testeq`) implementa una prueba de equivalencia por simplificación: en efecto, `A == B` se evalúa como `simplify(A-B) == 0`.

```
from satx.cas.core import Engine

e = Engine()
assert e.run("simplify(sin(x)^2 + cos(x)^2)").strip() == "1"
assert e.run("simplify(tan(x) - sin(x)/cos(x))").strip() == "0"
assert e.run("sin(x)^2 + cos(x)^2 == 1").strip() == "1"
```

4.6. Límites / modos

- `integral`: si no se resuelve, retorna una forma simbólica `(integral ...)` sin error.
- `defint`: lanza `satx.cas.CASError` si la integral queda sin resolver.
- Funciones no definidas: se preservan como formas prefijo (no se lanza error).

```
from fractions import Fraction

from satx.cas.core import CASError
from satx.sym import Context, app, const, parse, tensor, var

ctx = Context()
x = var("x")

# Normalizaciones del parser
ast = parse("-x + 1/2")[0]
assert ctx.engine.to_prefix(ast) == "(+ (* -1 x) (* 1 (^ 2 -1)))"

# defint falla si no hay primitiva cerrada
sinx = app("sin", x)
assert ctx.engine.to_prefix(ctx.integral(sinx, x)).startswith("(integral")
try:
    ctx.defint(sinx, x, const(0), const(1))
    assert False, "expected CASError"
except CASError:
    pass

# Tensores e indexación 1-based
t = tensor((2, 2), [const(1), const(2), const(3), const(4)])
assert ctx.get_component(t, 2, 1).value == Fraction(3, 1)
```

4.7. CAS→FD compilation

SATX expone un puente operacional entre el IR simbólico (SIR) y expresiones de dominio finito (FD) compilables a CNF: `satx.sym.compile_fd` y `satx.sym.constraint`.

Gamma (ambiente tipado)

- `Gamma: dict[name -> type]` tipa símbolos del SIR sin dependencias externas.
- Tipos soportados (descriptores en `satx.sym`): `Bool`, `Unit(bits, signed, arith)`, `Fixed(bits, scale, signed, arith)`, `Tensor(shape, base_type)`.
- También se permite pasar valores SATX directamente en `Gamma` (por ejemplo, un `satx.Unit` o `satx.Fixed` ya creado).
- Tensores: `Tensor(shape=(d1,...,dk), ...)` usa layout row-major e indexación 1-based alineada con el CAS (keyword `[`).

Subconjunto Σ_{FD} (mínimo)

- Aritmética: `+`, `*`, `^` con exponente entero pequeño (política `max_pow`). Exponentes negativos sólo para bases constantes (patrón del parser para división: `(^ q -1)`).
- Comparaciones: `testeq`, `testle`, `testlt`, `testge`, `testgt` (el parser normaliza `==,<=,<,>=,>` a estos keywords).
- Indexación: `[` con índices enteros literales 1-based.
- Factorial: `(factorial n)` para `n` entero literal acotado (`max_factorial`); otros casos retornan `Unsupported`.
- Racionales: se detectan denominadores en el AST (incluyendo `(^ q -1)`) e inferimos `scale = lcm(D)` para compilar a `Fixed`.

Salida: term, side_constraints, report

- `term`: un `satx.Unit`, `satx.Fixed` o `satx.Constraint` reificable.
- `side_constraints`: restricciones adicionales (p.ej. precondiciones de no-overflow cuando `arith="no_overflow"` en motor unsigned).
- `report`: dict simple con:
  - `supported_ops` / `unsupported_ops` encontrados,
  - `scale` con denominadores y `lcm(D)` (acotado por `S_max`),
  - `sir_hash` (hash estable por prefijo canónico),
  - `rewrite_trace` (si se habilita traza).

Ejemplo mínimo

```
import satx
from satx.sym import Unit, Tensor, compile_fd, constraint, parse

with satx.engine_ctx(bits=8, signed=False):
    expr = parse("x + 1 == 3")[0]
    Gamma = {"x": Unit(bits=8, signed=False, arith="no_overflow")}

    term, side, report = compile_fd(expr, Gamma, bits_default=8, policy={"S_max":
    10000})
    # term es un satx.Constraint; side contiene precondiciones (si aplica).

    cons, side2, report2 = constraint(expr, Gamma, simplify=True, policy={"trace":
    True})
    satx.add(cons)  # side2 ya fue aplicado
```

4.8. Engine keyword plugins

El CAS permite extender keywords sin modificar el núcleo.

- API: `Engine.register_keyword(name, handler, arity=..., purity=..., partiality=..., doc=...)`.
- `handler` puede tener firma `handler(call)` o `handler(engine, call)` y debe retornar un `Atom`.
- Metadatos:
  - `arity`: entero o `"variadic"`.
  - `purity`: `pure` | `effectful` (informativo).
  - `partiality`: `total` | `partial` (partial puede lanzar `CASError`).

```
from fractions import Fraction
from satx.cas.ast import Rational, sym
from satx.cas.core import Engine, cadr, list_of

def doubleit(engine: Engine, call):
    x = engine.eval(cadr(call))
    return engine.eval(list_of(sym("*"), Rational(Fraction(2, 1)), x))

eng = Engine()
eng.register_keyword("doubleit", doubleit, arity=1, purity="pure", partiality="total"
    , doc="x -> 2x")
print(eng.run("doubleit(3)"))  # 6
```

4.9. Trazas de reescritura y hashing SIR

- `hash_sir(expr)`: hash estable (SHA-256) basado en prefijo canónico (evita el hash no determinista de Python).
- `Context.capture_trace()`: captura una lista de pasos `{op, before_prefix, after_prefix, cached}` para llamadas por keyword (simplify/derivative/...).
- `satx.sym.constraint(..., policy={"trace": True})` agrega la traza al `report`, registra un record en `satx.encoding_metadata()` y puede snapshotearse dentro de `CNFIR.metadata` con `satx.ir.from_engine(include_encoding_metadata=True)`.

Parte III

TAXONOMY

APÉNDICE A

SIMBÓLICO / ALGEBRAICO

Capacidades donde SATX se usa como **runtime simbólico**: un backend CAS embebido (`satx.cas`) y un IR simbólico público (SIR) en `satx.sym`.

- Implementación: `satx/cas/` (`ast.py`, `parser.py`, `core.py`, `integral_tables.py`), `satx/sym/` (`types.py`, `builders.py`, `context.py`, `numeric.py`).

Capacidades / contratos

- Parser infix→prefijo con normalizaciones (negación unaria, división como potencia -1, indexación, factorial).
- Transformaciones simbólicas por keywords (`simplify`, `derivative`, `integral`, `defint`, `taylor`) delegadas a `Engine.eval`.
- Puente simbólico↔numérico: `float` (best-effort) + `to_python`.
- Tensores: `elem` plano row-major + indexación 1-based (`get_component`/`set_component`).

1. Normalización y construcción de SIR

Objetivo:

- Parsear expresiones (infix) a SIR prefijo canónico para serializar/compilar/inspeccionar.
- Validar invariantes (listas con `NIL`, tensores planos, keywords).

2. Cálculo simbólico y modos de fallo

Objetivo:

- Usar `integral` como operador simbólico total (puede quedar sin resolver).
- Usar `defint` como operador parcial (debe fallar si no hay primitiva cerrada).

3. Tensores e indexación

Objetivo:

- Representar vectores/matrices como tensores n-D en SIR.
- Acceso 1-based consistente con el CAS (keyword `[`).

4. Puente CAS ↔ SATX (dominio finito)

Objetivo:

- Compilar un prefijo del CAS a una expresión SATX (por ejemplo, aritmética racional escalada).
- Verificar propiedades por conteo proyectado con `satx.theory.count(project=...)`.

APÉNDICE B

ESTRUCTURAL / CONFIGURACIONAL

Problemas donde:

- El universo de variables está fijo
- Las leyes son duras
- El conteo mide **robustez estructural**

SATX se usa como:

Compilador exacto de configuraciones

1. Red de sensores con fallos correlacionados

Una red de sensores cubre una región crítica. Los fallos no son independientes, sino correlacionados espacialmente.

Objetivo:

- Verificar existencia de cobertura total
- Medir la fracción de configuraciones resilientes

→ SAT: existencia → #SAT: robustez topológica

---

2. Planificación logística con ventanas temporales inciertas

Acciones discretas en el tiempo con retardos acotados.

Objetivo:

- Verificar si existe un plan factible
- Medir cuántos escenarios temporales lo sostienen

→ SAT: planificación factible → #SAT: sensibilidad temporal

---

3. Sistemas de votación con abstención estratégica

Preferencias discretas con opciones latentes.
Objetivo:

- Verificar equilibrio ganador
- Medir estabilidad del resultado

→ SAT: equilibrio → #SAT: estabilidad institucional

APÉNDICE C

DINÁMICO / POLÍTICO

Aquí aparecen:

- Evolución temporal explícita
- Estados latentes
- Sesgos normativos C

SATX se convierte en:

Motor de mundos posibles condicionados

4. Control autónomo con fallos latentes

Decisiones que activan riesgos diferidos.
Objetivo:

- Existencia de política segura
- Probabilidad estructural de catástrofe

→ SAT: política → #SAT: riesgo latente

---

5. Redes causales con intervenciones parciales

Causalidad lógica con variables intervenibles.
Objetivo:

- Garantizar efectos
- Medir robustez causal

→ SAT: suficiencia → #SAT: robustez causal

---

6. Sistemas con fuga de información

Protocolos donde el estado interno se filtra parcialmente.
Objetivo:

- Verificar ataque completo
- Medir superficie de fuga

→ SAT: exploit → #SAT: superficie de ataque

---

7. Sistemas multiagente con coaliciones dinámicas

Coaliciones que emergen, colapsan o se estabilizan.
Objetivo:

- Existencia de coalición estable
- Fragilidad temporal

→ SAT: estabilidad → #SAT: fragilidad estructural

APÉNDICE D

ONTOLÓGICO / INTERPRETATIVO

Aquí:

- Las leyes pueden ser blandas
- El conteo define plausibilidad
- Las trayectorias compiten como explicaciones

SATX opera como:

Compilador de ontologías discretas

8. Sistemas físicos con leyes aproximadas

Leyes impuestas vía conteo, no como axiomas duros.
→ SAT: existencia física → #SAT: plausibilidad ontológica

---

9. Verificación de alineamiento

Utilidades internas no observables.
→ SAT: alineamiento posible → #SAT: ambigüedad explicativa

---

10. Identidad persistente bajo observaciones parciales

Estados internos no observables.
→ SAT: coherencia → #SAT: indistinguibilidad

---

11. Mundos posibles con leyes emergentes

Reglas locales inducen regularidades globales.
→ SAT: emergencia → #SAT: inevitabilidad

12. Libre albedrío en sistemas deterministas

Variables ocultas vs determinismo completo.
→ SAT: determinismo → #SAT: necesidad de variables latentes

APÉNDICE E

TRANSCENDENTAL / META-SEMANTIC

Este nivel es **cualitativamente distinto**.
No se cuentan estados. No se cuentan trayectorias. No se cuentan configuraciones.

Se cuentan teorías.

SATX deja de resolver problemas:

- El modelo es una variable
- Las leyes son hipótesis
- El lenguaje es parte del espacio de búsqueda

E.1. Definición

Un problema es **Transcendental** si cumple al menos dos:

1. Las leyes están parametrizadas o latentes
2. El conjunto de restricciones es una variable
3. El conteo compara teorías
4. SAT sintetiza ontologías
5. El resultado es inevitabilidad / identificabilidad

E.2. Ejemplos

1. Identificación de leyes físicas mínimas

SAT sintetiza leyes. #SAT mide competencia entre teorías.

2. Observador-dependencia

SAT: realidad compartida #SAT: subdeterminación ontológica

3. Emergencia de causalidad

SAT: causalidad explícita #SAT: acausalidad explicativa

4. Autoverificación formal

SAT: interpretación consistente #SAT: no-equivalencia semántica

---

E.3. Qué exige en SATX

- Variables que representan reglas
- Restricciones sobre restricciones
- Conteo de familias de modelos
- Exposición condicional como medida de inevitabilidad

SATX aquí no es un solver. Es un **compilador de mundos posibles**.

APÉNDICE F

BACKENDS / COMANDOS

F.1. Nombres genéricos de backends

En esta referencia, las variables `SAT_CMD`, `COUNT_CMD`, `MAXSAT_CMD` y `MIP_CMD` denotan **comandos** (strings) provistos por el usuario para ejecutar backends externos. SATX no fija ni menciona solvers concretos.

```
SAT    = "sat_solver"
COUNT  = "sharp_sat_solver"
MAXSAT = "maxsat_solver"
MIP    = "mip_solver"
```

- `SAT`: resuelve SAT sobre una instancia CNF (DIMACS); reporta SAT/UNSAT y, si es SAT, entrega un modelo.
- `COUNT`: ejecuta #SAT sobre una instancia CNF (DIMACS) y retorna el conteo de modelos.
- `MAXSAT`: resuelve (Weighted/Partial) MaxSAT sobre una instancia WCNF; retorna un estado de óptimo y un costo, y opcionalmente un modelo.
- `MIP`: resuelve un modelo MIP exportado por SATX (LP/MPS) y retorna objetivo/valores cuando aplica.

Parte IV

satx.cas