生成 AI による文書グラフ化と制約ベース精度保証

Y. Matsuda, ChatGPT 5.2

2026 年 2 月 1 日

1　生成 AI による構造化文書のグラフ化と制約に基づく精度保証の考え方

近年、生成 AI を用いて法律文書や規程文書、技術仕様書といった構造化文書をグラフ構造へ変換する試みが広く行われている。生成 AI は、文書の分割、参照関係の抽出、概念の同定といった作業を高い柔軟性で実行できるため、従来のルールベース手法と比較して実装・運用のコストを大幅に低減できる。一方で、生成 AI が生成するグラフ構造は確率的であり、その **構文的**（*syntactical*） および **意味的**（*semantical*） な正確性をどのように保証するかという問題が残る。

この問題の本質的な難しさは、検証対象となるインスタンスが事実上無限である点にある。すなわち、「すべての条文」「すべての参照関係」「すべての定義」を人手で確認することは不可能であり、意味解釈そのものを直接検証対象とする方法はスケールしない。本ホワイトペーパーでは、この問題に対し、**意味そのものを検証するのではなく、意味が破綻すると必ず破れる構造的制約を検証する** という立場を取る。

基本的な発想は以下の三点に要約できる。第一に、対象とする文書集合に対して最小限かつ安定したスキーマ（例：Document、Unit、階層関係、参照関係）を仮定する。第二に、そのスキーマから論理的・構造的に導かれる制約（constraint）を定式化する。第三に、生成 AI が出力したグラフに対して、これらの制約を機械的に検査する。重要なのは、これらの制約が実行時には完全に構文的（syntactical）に評価可能である点である。

制約は一見すると単なる構造チェックに見えるが、実際には意味的整合性の重要な部分を間接的にカバーしている。例えば、条文階層に循環が存在しないこと、各条文が一意な識別子を持つこと、参照が実在する条文または概念に向いていることといった条件は、いずれも意味理解を前提とせずに検査できる。しかし、これらが破られる場合、法的意味論としても破綻している可能性が極めて高い。この意味で、制約は「syntax によって semantics を部分的に代理する」役割を果たす。

さらに、制約には決定的に成否が判定できるものと、統計的・経験的に異常を検知するものの二種類が存在する。前者には一意性制約や非循環制約が含まれ、これらは精度保証の中核を成す。後者には階層深さや参照次数の分布に基づく外れ値検出が含まれ、これらは保証というよりも監査やレビューの優先順位付けに用いられる。両者を明確に区別することで、「保証できる範囲」と「監視すべき範囲」を理論的に整理できる。

本アプローチの利点は、生成 AI の内部表現や推論過程に依存しない点にある。制約はグラフ構造そのものに対して定義されるため、NetworkX 等の一般的なグラフ理論ツールや、Neo4j のようなグラフデータベース、あるいは独自実装の検証パイプライン上でも同一に適用できる。これは、生成 AI を「意味理解エンジン」としてではなく、「候補生成器」として位置づけ、その出力を形式的に検証するという設計思想に対応している。

まとめると、本ホワイトペーパーで提案する枠組みは、生成 AI によるグラフ構造化の利便性を最大限に活かしつつ、無限に存在しうる生成結果を直接検証するのではなく、スキーマから自動的に導かれる制約を用いて精度を管理するものである。この枠組みは、意味検証を完全に自動化することを目指すものではないが、意味的破綻が構造的異常として現れる領域を体系的に捉えることで、現実的かつスケーラブルな精度保証を可能にする。

2　構造文書グラフ構造化における破綻パターン：説明と形式的制約

共通設定（グラフモデル）

構築されたグラフを、有向・ラベル付きのプロパティグラフとしてモデル化する：

$$G = (V, E), \qquad V = V_D \cup V_U \cup V_C, \qquad E = E_{\text{part}} \cup E_{\text{ref}}.$$

ここで V_D は *Document* ノードの集合、V_U は *Unit* ノードの集合、V_C は *Concept* ノードの集合（任意）である。エッジ集合は次のように分割される：

$$E_{\text{part}} \subseteq (V_D \cup V_U) \times V_U \quad (\texttt{HAS\_PART}), \qquad E_{\text{ref}} \subseteq V_U \times (V_U \cup V_C) \quad (\texttt{REFERS\_TO}).$$

属性（部分）関数として、以下を仮定する：

$$\text{id} : V_U \to \Sigma^*, \qquad \text{text} : V_U \to \Sigma^*, \qquad \text{ev} : E_{\text{ref}} \to \mathbb{N} \times \mathbb{N} \ \text{(任意の根拠スパン)}.$$

エッジ集合 E に関する入次数および出次数を、それぞれ $\deg_E^-(v)$、$\deg_E^+(v)$ で表す。到達可能性は $x \rightsquigarrow y$ によって表し、これは x から y への有向パスの存在を意味する。

A. ID・属性まわり（最優先）

P1. Unit の重複生成（同じ `canonical_id` が複数ノード）　症状：同一条文が複数ノードに分裂し、参照が分散・不整合になる。
式（単射性）：

$$\forall u_1, u_2 \in V_U,\ \mathrm{id}(u_1) = \mathrm{id}(u_2) \Rightarrow u_1 = u_2.$$

P2. `canonical_id` 欠落　症状：条文の同定・追跡が不能。参照整合・差分追跡の検証ができない。
式（全定義性）：

$$\forall u \in V_U,\ \mathrm{id}(u) \text{ is defined and } \mathrm{id}(u) \neq \epsilon.$$

P3. `text` 欠落（空の Unit）　症状：根拠追跡不能。後段の再抽出・監査が不可能。
式（非空テキスト）：

$$\forall u \in V_U,\ |\mathrm{text}(u)| > 0.$$

P4. `canonical_id` の形式破綻　症状：条・項・号の復元や参照解決が破綻する。
式（正規言語制約）：

$$\forall u \in V_U,\ \mathrm{id}(u) \in L_{\mathrm{id}},$$

ただし $L_{\mathrm{id}} \subseteq \Sigma^*$ は正規表現等で定義される許容形式集合である。

B. 階層構造（HAS_PART）まわり

P5. 孤立 Unit の生成　症状：どの Document ／上位 Unit にも属さない断片が発生。
式（親の存在）：

$$\forall u \in V_U,\ \exists p \in (V_D \cup V_U) \text{ s.t. } (p, u) \in E_{\mathrm{part}}.$$

P6. 階層循環（HAS_PART にサイクル）　症状：条文構造が木（DAG）でなくなる。
式（非巡回性）：

$$\nexists u \in V_U \text{ s.t. } u \rightsquigarrow u \text{ via } E_{\mathrm{part}}.$$

P7. 親の多重化（Unit が複数の親を持つ）　症状：構造が二重化し、解釈の一意性が失われる。
式（関数性／入次数上限）：

$$\forall u \in V_U,\ \left|\{p \in (V_D \cup V_U) \mid (p, u) \in E_{\mathrm{part}}\}\right| \leq 1,$$

すなわち $\deg^-_{E_{\mathrm{part}}}(u) \le 1$.

P8. 階層深さの異常　症状：過分割（深すぎ）または未分割（浅すぎ）。
式（深さと外れ値）：

$$d(u) = \max\{\ \ell \mid \exists d \in V_D,\ \exists \text{path } d \rightsquigarrow u \text{ in } E_{\mathrm{part}} \text{ of length } \ell\ \},$$

$$D = \{d(u) \mid u \in V_U\}, \qquad u \text{ is anomalous if } d(u) \notin [Q_\alpha(D),\, Q_{1-\alpha}(D)].$$

ここで Q_α は分位点（quantile）である。

C. 参照構造（REFERS_TO）まわり

P9. dangling reference（参照先欠落）　症状：参照表現があるのに対応ノードが欠落（あるいは未解決）。
式（終点存在性）：

$$\forall (u,v) \in E_{\mathrm{ref}},\ u \in V_U\ \wedge\ v \in (V_U \cup V_C).$$

（実装上「参照文字列のみ保持」の場合は、未解決参照集合 $U_{\mathrm{unres}} \subseteq V_U$ を別途定義し $\forall u \in U_{\mathrm{unres}},\ \neg\exists v : (u,v) \in E_{\mathrm{ref}}$ を検査対象とする。）

P10. 自己参照の暴発　症状：「前条」「同項」等の照応解決が自分自身に落ちる。
式（反射性禁止）：

$$\forall (u,v) \in E_{\mathrm{ref}},\ u \neq v.$$

P11. 参照エッジの重複生成　症状：同一参照が多重化しクエリや集計が膨張。
式（単純性／多重辺禁止）：

$$\forall u \in V_U,\ \forall v \in (V_U \cup V_C),\ \Big|\{e \in E_{\mathrm{ref}} \mid e = (u,v)\}\Big| \le 1.$$

P12. 参照型の破綻　症状：Unit が想定外型のノードを参照（ラベル付与ミス等）。
式（型制約の再掲）：

$$E_{\mathrm{ref}} \subseteq V_U \times (V_U \cup V_C).$$

D. Document 境界・版管理

P13. Unit の Document 跨り　症状：1 つの Unit が複数 Document に属し境界が崩壊。
式（唯一の所属 Document）：

$$\forall u \in V_U,\ \Big|\{d \in V_D \mid d \rightsquigarrow u \text{ via } E_{\mathrm{part}}\}\Big| = 1.$$

P14. 施行期間の矛盾　症状：effective_from > effective_to 等により版クエリが不定。
式（全順序制約）：

$$\forall d \in V_D,\ \mathrm{from}(d) \le \mathrm{to}(d),$$

（$\mathrm{to}(d)$ が未定義なら無期限とみなす等、実装規約に合わせる。）

E. evidence（根拠スパン）まわり

P15. evidence 欠落　症状：生成 AI の判断根拠が追跡不能。監査・再検証不可。
式（参照辺への根拠付与）：

$$\forall e \in E_{\mathrm{ref}},\ \mathrm{ev}(e) \text{ is defined.}$$

P16. evidence スパンの不整合　症状：範囲外・逆転・空スパン等により根拠が虚偽/ズレる。
式（区間整合）：

$$\forall e = (u, v) \in E_{\mathrm{ref}},\ \mathrm{ev}(e) = (s, t) \Rightarrow 0 \le s < t \le |\mathrm{text}(u)|.$$

（必要なら「根拠文字列の非空」 $|\mathrm{text}(u)[s{:}t]| > 0$ も加える。）

F. 構文＋統計的異常（semantics が構造に現れる領域）

P17. 定義らしい Unit が Concept に接続されていない　症状：「〜とは…」等の定義構文があるのに定義グラフが欠落。
式（構文ヒューリスティック＋存在性）：

$$U_{\mathrm{def}} = \{u \in V_U \mid \mathrm{text}(u) \in L_{\mathrm{def}}\}, \qquad \forall u \in U_{\mathrm{def}},\ \exists c \in V_C \text{ s.t. } (u, c) \in E_{\mathrm{def}}.$$

ここで L_{def} は定義表現のパターン集合（例：「とは」「をいう」等）、E_{def} は `DEFINES` に相当する辺集合（導入する場合）である。（`DEFINES` を持たない最小スキーマでは、代替として「定義候補が Concept ノード生成を伴う」制約に置換する。）

P18. 参照密度の異常　症状：`REFERS_TO` が極端に多い/少ない（過抽出・漏れの疑い）。
式（参照次数分布の外れ値）：

$$k(u) = \deg^{+}_{E_{\mathrm{ref}}}(u), \qquad K = \{k(u) \mid u \in V_U\},$$

$$u \text{ is anomalous if } k(u) \notin [Q_\alpha(K),\ Q_{1-\alpha}(K)].$$

（より頑健にするなら MAD/Z-score 等の統計的外れ値規準に差し替える。）

3　制約設計の前提となるグラフスキーマ例

本節では、前節までで議論した制約群（P1–P18）がどのようなグラフスキーマを前提として導出されているかを明確にするため、想定される最小スキーマの一例を示す。本スキーマは、法律文

書や規程文書などの構造化文書を対象としつつ、特定のグラフデータベースや実装技術には依存しない抽象的な定義である。

本研究で想定するノード型は、*Document*、*Unit*、および（任意に）*Concept* の三種である。*Document* は法令・規程・ガイドライン等の文書単位を表し、*Unit* は文書内部の構造単位（章・条・項・号など）を統一的に表す抽象ノードである。*Concept* は定義語や抽象概念を表すノードであり、定義関係を明示的に扱う場合にのみ導入される。

各 *Unit* ノードは、少なくとも一意な識別子 `canonical_id` と原文テキスト `text` を属性として持つ。識別子は、文書番号および条項構造に基づいて構成され、同一文書集合内で一意であることが要求される。原文テキストは、後続の検証、再抽出、監査のための根拠情報として保持される。

エッジ型としては、`HAS_PART` および `REFERS_TO` の二種のみを想定する。`HAS_PART` は文書および上位構造単位から下位の *Unit* への階層関係を表し、文書全体の構造を有向非巡回グラフ（通常は木）として表現する。`REFERS_TO` は、ある *Unit* が他の *Unit* または *Concept* を参照する関係を表し、引用、準用、照応、定義参照などを含む一般的な参照関係として扱われる。

参照エッジ `REFERS_TO` には、生成 AI による抽出結果の監査可能性を確保するため、任意で根拠スパン（原文テキスト中の開始位置および終了位置）を属性として付与できるものとする。この情報は意味理解そのものを表すものではないが、制約違反時に人間が原因箇所へ遡るための重要な補助情報となる。

以上のスキーマは意図的に最小限に設計されている。すなわち、「義務」「権利」「条件」「例外」といった意味的カテゴリや、条・項・号といった細粒度の構文型はスキーマレベルでは明示しない。これらは必要に応じて属性や派生エッジとして追加可能であるが、本ホワイトペーパーで提示した制約群は、これらを仮定せずとも成立する。この点において、本スキーマは「意味解釈のためのモデル」ではなく、「生成 AI によるグラフ構造化結果を検証するための基盤」として位置づけられる。

このように、想定スキーマを最小化することで、スキーマから自動的に導かれる制約の集合が安定し、生成 AI の実装や学習状態に依存しない形で構造的精度保証を行うことが可能となる。

3.1 形式化スキーマ（Property Graph Schema）

以下では、上記の抽象スキーマを、プロパティグラフとして実装可能な形に形式化する。ここでの「型」は、プログラミング言語の型ではなく、(i) ノードラベル、(ii) リレーション型、(iii) 属性（プロパティ）とその必須性・一意性、(iv) 接続の許容範囲、を意味する。

ノードラベルと属性

- **:Document**（文書単位）
  - doc_id: String（必須・一意；例：法令番号や内部 ID）
  - title: String（任意）
  - effective_from: Date（任意）
  - effective_to: Date（任意；未設定は無期限）
- **:Unit**（構造単位：条・項・号等を統一）
  - canonical_id: String（必須・一意；例：doc_id/A12.P3.I2）
  - text: String（必須）
  - unit_type: String（任意；article/paragraph/item 等の粗い分類）
  - order_key: String（任意；同一親の下での整列用キー）
- **:Concept**（任意：定義語・概念）
  - concept_id: String（必須・一意；正規化 ID）
  - term: String（必須；代表表記）
  - surface_forms: List<String>（任意；表記ゆれ）

リレーション型（エッジ型）と属性

- **:HAS_PART**（階層）

$$(\texttt{:Document} \cup \texttt{:Unit}) - [\texttt{:HAS\_PART}] \rightarrow \texttt{:Unit}$$

  属性（任意）：

  - order: Integer（子の順序；order_key の代替）

- :REFERS_TO（参照）

$$:\text{Unit} \;-\; [:\text{REFERS\_TO}] \;\rightarrow\; (:\text{Unit} \cup :\text{Concept})$$

  属性（任意）:

  - ref_type: String（任意；quote/apply/define/anaphora 等の粗い区分）
  - evidence_start: Integer（任意；根拠スパン開始）
  - evidence_end: Integer（任意；根拠スパン終端）
  - confidence: Float（任意；抽出信頼度）

スキーマ制約（抽象レベル）　制約群 P1–P18 の基盤として、少なくとも以下を満たすことを想定する。

- 一意性：:Unit(canonical_id) は一意、:Document(doc_id) は一意、（導入するなら）:Concept(concept_id) は一意。
- 必須性：:Unit.canonical_id と :Unit.text は必須。
- 型整合：:HAS_PART の終点は常に :Unit、:REFERS_TO の始点は常に :Unit。

Neo4j 向け DDL 例（制約・インデックス）

以下は上記形式化スキーマを Neo4j に移行する際の、最小限の制約定義例である（Cypher）。

```
// --- uniqueness / existence (nodes) ---

CREATE CONSTRAINT document_doc_id_unique IF NOT EXISTS

FOR (d:Document) REQUIRE d.doc_id IS UNIQUE;
```

```
CREATE CONSTRAINT unit_canonical_id_unique IF NOT EXISTS

FOR (u:Unit) REQUIRE u.canonical_id IS UNIQUE;



CREATE CONSTRAINT unit_text_required IF NOT EXISTS

FOR (u:Unit) REQUIRE u.text IS NOT NULL;



// optional (if Concept is used)

CREATE CONSTRAINT concept_concept_id_unique IF NOT EXISTS

FOR (c:Concept) REQUIRE c.concept_id IS UNIQUE;



CREATE CONSTRAINT concept_term_required IF NOT EXISTS

FOR (c:Concept) REQUIRE c.term IS NOT NULL;



// --- indexes (typical) ---

CREATE INDEX unit_unit_type IF NOT EXISTS

FOR (u:Unit) ON (u.unit_type);



CREATE INDEX refers_to_ref_type IF NOT EXISTS

FOR ()-[r:REFERS_TO]-() ON (r.ref_type);
```

上記 DDL は、制約群のうち「DB が直接保証できる部分」（一意性・必須性・索引）を固定化するための最小セットである。P5 以降（孤立、循環、多重親、参照密度、evidence 整合など）は、DB 制約ではなく検証クエリまたは NetworkX 等の外部検証器で評価する設計を前提とする。

3.2 移行仕様：識別子・語彙・関数性制約の形式化

本節では、前節で示した形式化スキーマを、実際のグラフ生成および検証パイプラインへ移行するために必要となる追加仕様を明示する。ここで扱う仕様は、特定のグラフデータベース機能に依存せず、生成 AI の出力を対象とした事前・事後検証の双方に適用可能な「検証仕様」として定義される。

canonical_id の形式仕様　各 :Unit ノードが持つ canonical_id は、文書内での構造的位置を一意に表現するための正規化識別子であり、以下の形式に従うものとする。

BNF 表記（例）

```
<canonical_id> ::= <doc_id> "/" <article>

                   [ "." <paragraph> ]

                   [ "." <item> ]



<doc_id>       ::= <alnum> { <alnum> | "_" | "-" }

<article>      ::= "A" <number>

<paragraph>    ::= "P" <number>

<item>         ::= "I" <number>

<number>       ::= <digit> { <digit> }
```

これに対応する正規表現の一例を以下に示す。

```
^[A-Za-z0-9_\-]+/A[0-9]+(\.P[0-9]+)?(\.I[0-9]+)?$
```

本仕様の目的は、条・項・号の意味的区別を厳密に表すことではなく、**構造的位置が一意に復元可能であること**を保証する点にある。そのため、必要に応じて節・款等の記号を拡張することは妨げない。

unit_type の許容語彙　:Unit.unit_type は、人間による可読性および簡易的な構造検査を目的とした補助属性であり、以下の有限集合から選択されるものとする。

$$\texttt{unit\_type} \in \{\texttt{part},\ \texttt{chapter},\ \texttt{section},\ \texttt{article},\ \texttt{paragraph},\ \texttt{item},\ \texttt{misc}\}$$

ここで misc は、生成 AI が明確な区分を付与できなかった場合のフォールバックとして用いられる。検証段階では、misc の出現頻度が過度に高い場合を異常として扱うことができる。

ref_type の許容語彙　:REFERS_TO エッジが持つ ref_type 属性は、参照の粗い性質を示すものであり、以下の有限集合を想定する。

$$\texttt{ref\_type} \in \{\texttt{quote},\ \texttt{apply},\ \texttt{define},\ \texttt{anaphora},\ \texttt{other}\}$$

この分類は意味論的厳密性を目的とするものではなく、参照関係の分布異常や抽出偏りを検出するための観測軸として導入される。

階層関係に関する関数性制約　:HAS_PART によって定義される階層構造は、以下の関数性制約を満たすことを要求する。

- 各 :Unit ノードは、高々 1 つの親ノード（:Document または :Unit）を持つ。
- すなわち、:HAS_PART は :Unit に関して部分関数である。

形式的には、以下が成り立つことを検証条件とする。

$$\forall u \in V_U, \quad |\{p \mid (p, u) \in E_{\text{part}}\}| \leq 1$$

この制約は、グラフデータベースのスキーマ機構では直接表現できない場合が多いため、生成後検証または外部グラフ解析（例：NetworkX）によって評価されることを前提とする。

検証仕様としての位置づけ　本節で示した仕様は、データベース制約（DDL）ではなく、**生成 *AI* の出力がスキーマ仮定を満たしているかを判定するための検証仕様**として位置づけられる。これにより、実装や保存形式に依存せず、グラフ構造化パイプラインの任意の段階で同一の検証ロジックを適用することが可能となる。