ChatGPT

Design Constraint Validator (DCV) – System Overview

Core Purpose and Capabilities

Design Constraint Validator (DCV) is a tool for enforcing **mathematical design constraints** on design tokens, ensuring consistency, accessibility, and logical coherence in design systems [1]. Unlike a typical schema linter (which only checks if tokens exist or follow naming conventions), DCV validates the **relationships and values** across tokens (it's a "reasoning validator" for design logic [2]). Key capabilities include:

- **Accessibility rules:** Checking color contrast against WCAG standards (e.g. ensuring text/background color pairs meet AA/AAA contrast ratios) and enforcing perceptual lightness thresholds for readability [3]. For example, DCV can flag insufficient text contrast or require a minimum lightness for text on dark surfaces.
- **Order & Monotonicity:** Enforcing that token values follow a defined order or scale. This covers things like typography sizes (e.g. `h1 ≥ h2 ≥ body`) and spacing scales (large spacings should be $\geq$ smaller spacings) [3]. DCV ensures such **monotonic sequences** hold true, catching out-of-order values in a style hierarchy.
- **Thresholds & Policies:** Applying absolute minimums/maximums or policy-based limits. For instance, DCV can require that a touch target size is at least 44px (per accessibility guidelines) or that font sizes never drop below a certain floor [3]. These threshold checks help enforce organizational or universal design standards (like WCAG or Material Design specs).
- **Cross-axis constraints:** Validating relationships *across different token properties*. This means DCV can express rules like *"If condition X on token A is true, then token B must satisfy condition Y."* For example, *"if a font weight is light, then the font size must be above a certain minimum"* is a cross-axis rule. This capability allows **multi-property conditional checks** beyond single-token rules [4], enabling sophisticated design system governance (e.g. tying component size, font weight, and contrast requirements together).
- **Graph intelligence & explanations:** DCV builds a graph of token relationships and can export it (e.g. as a Hasse diagram representing the partial order of tokens) for visualization [3]. The tool not only reports that a constraint failed but also **explains "why"**, by highlighting the chain of token values or references that led to the violation. In other words, it leverages the dependency graph to trace which tokens are implicated in a rule failure, offering insight into how to fix it. *(For example, DCV might report that "Heading H2 is out of order: expected $\leq$ H1 (32px) but found 34px," pointing to the tokens that broke the monotonic size rule.)*

Architecture and Key Abstractions

DCV is implemented as a **validation engine** with a structured, multi-phase workflow [5]. It processes design tokens through several conceptual stages to ensure all constraints are checked methodically:

1. **Token Parsing & Normalization:** The input design tokens (provided as JSON, which can be flat or nested) are first normalized into a consistent internal format. DCV includes adapters to handle common design token JSON formats (Style Dictionary, Tokens Studio, W3C formats, etc.), flattening nested token objects into dot-notated keys and resolving any token references/aliases in the values [6] [7]. The result is a uniform list of token entries (each with an `id` like `"typography.size.h1"` and a concrete value) ready for validation.

2. **Dependency Graph Construction:** DCV then builds a **directed acyclic graph (DAG)** representing all tokens and their dependencies [5]. In this graph model, **nodes** are individual tokens and **edges** represent relationships – either references (one token's value uses another) or constraint relations (like an order requirement between two tokens) [8] [9]. This graph is essentially a mathematical **partial order (poset)** of the tokens under the defined constraints. By constructing the graph, DCV can detect problems like circular references early (e.g. token A referencing B and B referencing A would form a cycle, which DCV flags as an error). The graph structure is central to DCV's reasoning: it enables traversal of token relationships and underpins the explanation and visualization features (e.g., exporting a Hasse diagram of the token poset).

3. **Constraint Validation Engine (Plugins):** Once the token graph is ready, DCV runs the **constraint validation engine**. This engine uses a **plugin-based architecture** – each type of design constraint is implemented as a plugin module that knows how to check a particular kind of rule across the graph [10]. DCV comes with built-in plugins for all the major constraint categories (Monotonic order, WCAG contrast, Threshold checks, Lightness progression, Cross-axis rules) [10]. During this phase, each plugin scans through relevant tokens or token pairs in the graph and produces any **violations** (errors or warnings) if a rule is broken. For example, the Monotonic plugin will iterate through configured token pairs that should be in non-decreasing order and create a violation if it finds a pair out of order, while the WCAG plugin will calculate contrast ratios for specified foreground/background token pairs and flag any that fall below the required minimum [5] [10]. The plugin architecture not only organizes the code but also means the system is extensible – new constraint types can be added as additional plugins following the same interface.

4. **Violation Reporting & Explanations:** In the final phase, DCV aggregates results from all plugins and prepares **human-readable reports** of any constraint violations. Each violation includes a descriptive message and context about which token(s) failed and how. DCV puts emphasis on **provenance tracing** – it can tell you which tokens (nodes) and relationships (edges) in the graph led to a failure [3]. The CLI can display these results in various formats (plain text, JSON, tables) and even output a visualization of the token graph highlighting where constraints were violated. This helps developers quickly understand not just *what* failed, but *why* (for instance, seeing a chain of dependencies or an ordering graph that pinpoints the break in a sequence).

Key Abstractions: At a high level, the DCV system is built on a few core abstractions:

- **Token graph and Engine:** The in-memory model of the design tokens is a graph (nodes/edges as described). The **Engine** is the core class that manages this graph and provides APIs to query tokens (`engine.get(tokenId)`), register constraint plugins, and execute validation across the graph [5] [11]. You can think of the Engine as the orchestrator that knows all token values and relationships.
- **Constraint definitions (Policies):** Constraints are configured via JSON files (often located under a `themes/` directory, sometimes called policies or rules). These define the rules for each category – e.g., an "order" policy file listing token keys that should be monotonic, a "wcag" policy file listing color pairs and minimum contrast ratios, or a cross-axis rules file defining conditional requirements. This separation makes the validator flexible: the rules can be adjusted or expanded without changing code, and organizations can plug in their own policy files (for custom brand rules, etc.) on top of the default ones.
- **Plugins (Constraint Checkers):** Each plugin implements a specific constraint check. Internally, a plugin is typically a class with a `check(engine)` method that evaluates all its rules against the current token graph and returns any violations [11]. For example, the **MonotonicPlugin** takes a list of token key pairs and ensures the numeric value of the first is ≥ the second for each pair, reporting an error if not. The **WcagContrastPlugin** takes rules about color pairs and desired contrast ratio and computes the actual ratios to compare [12] [13]. This modular design means the core engine doesn't have hardcoded knowledge of every rule – instead, it invokes each plugin in turn, which makes it easier to maintain and extend. *(According to the roadmap, a future "plugin API" will even let users add custom constraint plugins without modifying DCV's core [14].)*
- **Violations:** A violation is a standardized object describing a rule breach (including severity level, the type of constraint, a message, and references to the relevant token IDs or graph edges). Violations are what the engine produces for reporting. They form the basis of DCV's output – e.g., a list of violations can be printed to the console, or an exit code can be set to fail a CI build if any violation is of error severity.

Overall, the architecture cleanly separates *data (tokens/graph)* from *rules (constraints plugins)* and *outputs (reports/visualizations)*, which helps in understanding and auditing the system in layers.

Reusability and Integration

DCV is designed to function both as a **self-contained tool** and as an **engine that can be embedded** in other systems:

- **Standalone CLI and Library:** The project is packaged on npm, providing a command-line interface (`dcv` command) for direct use in projects. Teams can run `npx dcv validate` on their token files to enforce constraints, and incorporate this into build scripts or CI pipelines. In fact, a typical use-case is to add DCV to CI checks – the tool can exit with a non-zero code when violations are found, blocking deployments that violate design rules [15]. DCV also exposes a programmatic Node.js API, so developers can `import { validate } from 'design-constraint-validator'` and run validations in custom scripts or tools [16] [17]. This dual CLI/API approach makes it easy to integrate DCV's functionality wherever needed (from local development, to automated tests, to other applications).
- **Core Engine for Other Tools:** The repository positions DCV explicitly as a **core validation engine** that can be reused by higher-level platforms. Notably, it's meant to underpin the upcoming

DecisionThemes Studio (a larger design system product) as its validation layer [18]. In the documentation, the author explains that *"design-constraint-validator is the core validation engine – it validates any design tokens against constraints,"* whereas *"DecisionThemes … is a complete design system framework that uses this validator under the hood"* [18]. In other words, DCV focuses on the generic mechanics of constraint-checking and doesn't impose a specific design methodology beyond that; a tool like DecisionThemes can build on it, adding a GUI, a specific 5-axis design model (Tone, Emphasis, Size, Density, Shape), mapping higher-level design decisions to tokens, etc., while relying on DCV to perform the heavy lifting of validation [19]. This separation of concerns shows that DCV is intended to be **reusable**: any external tool or pipeline that deals with design tokens can incorporate DCV to ensure the tokens meet desired constraints.

- **Configuration and Extensibility:** Users can tailor DCV to their needs. It looks for default file locations (`tokens/*.json`, `themes/*.json` for constraints) but allows customization via a config file [20]. The constraint rules themselves can be extended – you can supply your own policy JSONs (for example, company-specific accessibility requirements or additional monotonic sequences to enforce). Future enhancements (per the roadmap) include a plugin API for custom constraints and integration features like VS Code diagnostics [21], which will further enable DCV's engine to be embedded in different environments. All of this underscores that DCV is built as a **generic engine** with well-defined inputs/outputs, rather than a one-off script; it's meant to be a building block in the design token tooling ecosystem.

Mathematical Foundations and Models

DCV's approach is underpinned by formal concepts from mathematics and computer science (graphs, partial orders, and logical rules):

- **Poset (Partially Ordered Set) Modeling:** Many design constraints can be seen as partial order requirements – for instance, a set of font sizes must obey an order (largest to smallest), which forms a poset under the "$\geq$" relation. DCV explicitly leverages this idea: it represents tokens and their constraints as a graph that can be visualized as a **Hasse diagram** (the graphical representation of a poset) [3]. By treating the design system as a set of elements with defined order/relationship constraints, DCV can apply **graph algorithms** to validate consistency. The "Graph Intelligence" feature (Hasse/poset export) is essentially exposing this internal model, showing how, say, a color palette forms a lightness-ordered chain, or how typography levels relate. This mathematical framing ensures that if the poset's requirements are satisfied, the design system's logical rules are upheld.
- **Directed Acyclic Graph and Dependency Reasoning:** Internally, the token graph is a DAG, which guarantees no cycles (circular dependencies) and allows for clear dependency tracing. The **acyclic** property is important for performing validations in a topologically ordered way and for doing things like incremental validation (only re-checking tokens that changed or that depend on changed tokens). DCV can traverse this graph to answer questions like "which tokens depend on this token?" or "what base tokens does this token derive from?", which is valuable for understanding the propagation of changes. The `why` analysis command uses graph traversal to provide provenance – it finds all upstream dependencies of a token and any constraints affecting it [22] [23]. This kind of analysis is akin to querying a dependency graph to explain a particular node's state, a technique rooted in graph theory and logic programming.
- **Numerical and Colorimetric Computation:** DCV enforces numeric relationships and uses proper mathematical conversions for comparisons. For example, for color contrast, it doesn't just compare hex codes – it converts colors to a linear lightness scale to compute relative luminance and contrast

ratio (using the WCAG formula). All colors are converted to the **OKLCH color space** (a perceptually uniform color space) internally for accurate comparison of lightness and contrast [24] [25]. This ensures that calculations like contrast ratio or lightness ordering are based on sound color science (so a "lightness" constraint truly reflects human-perceived lightness). For numeric tokens (sizes, spacing, etc.), DCV parses units (px, rem) into absolute values so it can compare magnitudes mathematically [26]. The monotonic constraints then operate on these numeric values, treating the design tokens as sequences of numbers subject to ≥ or ≤ relationships. All of this shows a strong quantitative foundation – DCV is essentially encoding design guidelines as mathematical inequalities or equations (e.g. *value_X* ≥ *value_Y*, *contrast(fg, bg)* ≥ *4.5:1*, *size_px* ≥ *44*).
- **Rule Logic (Conditional Constraints):** The cross-axis constraint feature introduces a more advanced logical model: rules that have a premise and a conclusion (if-then structure). These rules are evaluated by injecting token values into predicate functions (often small JavaScript arrow functions in the JSON) that return booleans [27] [28]. This is conceptually similar to a logical implication or a rule in a logic engine (e.g., *if P(token) is true, then Q(token) must be true*). DCV thus incorporates a **rule-based system** element, where the rules are user-defined but executed by the engine across the token set. This is grounded in boolean logic and functional programming – effectively, the design system's heuristic knowledge (like "make buttons larger on mobile screens for accessibility") is captured as code-like constraints that DCV evaluates. The combination of these rules with the underlying graph means DCV is doing a form of constraint solving or constraint satisfaction checking each time it runs validation.

In summary, DCV's design is heavily informed by mathematical models: it treats design tokens as elements of sets with relations, uses **graph theory** for dependency management and explanation, and relies on **formal criteria** (numerical thresholds, ratios, conditional logic) to validate design principles. This foundation is what allows it to go beyond simple linting and provide guarantees of coherence in a design system [2]. The result is a self-consistent, explainable validation process for design tokens, which can be trusted in high-stakes scenarios like enforcing brand guidelines or accessibility standards in an automated fashion.

Sources:

- CseperkePapp/design-constraint-validator – *README* (overview, features, philosophy) [2] [10] [18]
- CseperkePapp/design-constraint-validator – *Documentation* (Architecture and Constraints details) [5] [4]
- CseperkePapp/design-constraint-validator – *Package Definition* (package.json with description and metadata) [1]
- *WCAG 2.1 Guidelines* (referenced for contrast and sizing rules in DCV's policies) [29] [25]

1 package.json
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/package.json

2 3 10 14 15 16 17 18 19 20 21 README.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/README.md

4 24 25 26 27 28 29 Constraints.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/Constraints.md

5 6 7 8 9 11 12 13 22 23 Architecture.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/Architecture.md