Proceedings of the
13th International Conference on Applied Informatics
Eger, Hungary, February 19–21, 2026
DOI: 10.17048/icai.2026.227
pp. 227–243

Formal Verification of Questionnaire Logic Using SMT Solvers

Péter Sághelyi, Péter Bereczky

Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary
psaghelyi@hotmail.com, berpeti@inf.elte.hu

Abstract. Modern computer aided surveys (CAPI/CATI) can contain hundreds of questions with complex conditional logic. In order to guarantee correctness through all possible traversal paths is a computationally demanding challenge. We propose to replace the traditional imperative, command-based questionnaire design with a constraint-based paradigm where correctness is verified statically rather than through path discovery. This work presents a formal, declarative framework for questionnaire specification and SMT-based verification that enables this separation. We formalize questionnaires as tuples of items with preconditions and postconditions inspired by Hoare Logic, classify item reachability and postcondition feasibility through satisfiability checks, and establish a four-level validation hierarchy with five theorems characterizing the relationships between per-item, global, and path-based analysis. The method is evaluated based on real-world questionnaires.

Keywords: questionnaire verification, SMT solvers, formal methods, Z3, preconditions, postconditions

2020 *Mathematics Subject Classification:* 68V15, 68Q60, 68N30

1. Introduction

The formal study of questionnaire structure spans six decades, from Picard's foundational *Théorie des questionnaires* [18] through Fellegi and Holt's systematic approach to edit rules [9], to Willenborg's proof that checking questionnaire consistency is NP-complete [26]. Graph-based methods [1, 6, 7, 24] have provided visualization and documentation tools, while path enumeration approaches [8] have enabled systematic testing. However, none of these approaches provide formal guarantees about questionnaire correctness—a fundamental limitation of testing

that Dijkstra famously observed applies to all software systems [5].

The proliferation of AI-powered survey generation [14, 17] has amplified this need. AI systems can generate questionnaires with hundreds of interconnected questions whose conditional logic exceeds what human reviewers or even the generating AI can verify.

Our approach. We adopt a declarative, constraint-based paradigm inspired by Hoare Logic [10]. Just as Hoare triplets $\{P\}C\{Q\}$ specify program correctness through preconditions and postconditions, our framework specifies questionnaire correctness through logical constraints. Preconditions become Boolean formulas determining item visibility; postconditions become constraints on valid responses. This declarative specification maps naturally to SMT constraint satisfaction, enabling the Z3 theorem prover [15] to analyze questionnaire properties with mathematical precision.

Contributions. This paper makes four contributions:

1. A formal framework and domain-specific language (called QML [21]) for representing questionnaire logic using preconditions and postconditions.
2. A four-level SMT-based validation hierarchy (per-item, global, cycle detection, path-based) characterized by five theorems with formal proofs.
3. A compiler transforming the questionnaire logic expressed in QML with embedded Python code blocks into Z3 constraints.
4. The evaluation of our framework with real-world questionnaires.

The rest of this paper is structured as follows. Section 2 introduces the formal model for questionnaires followed by Section 3 discussing our approach for validating questionnaires based on the model. Section 4 introduces a concrete implementation, which connects the model to the Z3 SMT solver [15], and evaluates it with real-world questionnaires. Section 5 summarises related work, and Section 6 concludes and points out future directions.

2. Formal Questionnaire Model

We formalize questionnaires by defining their components axiomatically and then composing them into a complete structure.

Definition 2.1 (Item)**.** An **item** I_i is a questionnaire element that presents content and optionally collects responses. Items may be informational (no outcome), scalar questions (single integer outcome), question groups (vector outcome), or matrix questions (matrix outcome).

Definition 2.2 (Outcome Variable)**.** For each item I_i collecting responses, the **outcome variable** S_i represents the respondent's answer.

In this paper, we only discuss scalar outcome variables ($S_i \in \mathbb{Z}$) for clarity; however, real-world questionnaires require more complex answer structures. Our framework is capable of handling *vector* outcome variables $S_i \in \mathbb{Z}^k$ for question groups, or $S_i \in \mathbb{Z}^{m \times n}$ for matrix questions too.

We denote the complete vector of outcome variables as $\mathbf{S} = (S_1, \ldots, S_n)$. An *assignment* is a vector $\mathbf{a} = (a_1, \ldots, a_n)$ over the corresponding domains.

Definition 2.3 (Domain Constraint)**.** Each outcome variable S_i has a **domain constraint** $D_i(S_i)$ restricting valid values, specified either as a range $\ell_i \leq S_i \leq u_i$ or an enumeration $S_i \in \{v_1, \ldots, v_k\}$.

The conjunction of all domain constraints forms the **base constraint**:

$$B := \bigwedge_{i=1}^{n} D_i(S_i)$$

Let the set of domain-respecting assignments be $\text{Dom} := \{\mathbf{a} \mid \mathbf{a} \models B\}$. Here, we use $\models$ to denote the SMT satisfaction relation[1] used by Z3 [15]: $\mathbf{a} \models \phi$ means assignment $\mathbf{a}$ satisfies formula ϕ.

Definition 2.4 (Precondition)**.** Each item I_i has a **precondition** P_i, a Boolean formula over $\mathbf{S}$ determining whether the item is presented. If $\mathbf{a} \models P_i$ for assignment $\mathbf{a}$, item I_i is displayed; otherwise it is hidden.

Preconditions encode the conditional logic that creates branching paths through the questionnaire based on previous answers.

Definition 2.5 (Postcondition)**.** Each item I_i has a **postcondition** Q_i, a Boolean formula over $\mathbf{S}$ constraining valid responses. Postconditions are only evaluated when the item is accessible, requiring: $P_i \Rightarrow Q_i$.

Postconditions ensure logical consistency between answers and can dynamically restrict acceptable values based on prior responses.

Definition 2.6 (Questionnaire)**.** A **questionnaire** $\mathcal{G}$ is a tuple $(\mathcal{I}, \mathbf{S}, \mathcal{D}, \mathcal{P}, \mathcal{Q}, I_{\text{start}})$ where:

- $\mathcal{I} = \{I_1, \ldots, I_n\}$ is a finite set of items
- $\mathbf{S} = (S_1, \ldots, S_n)$ is a vector of outcome variables
- $\mathcal{D} = (D_1, \ldots, D_n)$ specifies domain constraints
- $\mathcal{P} = (P_1, \ldots, P_n)$ specifies preconditions
- $\mathcal{Q} = (Q_1, \ldots, Q_n)$ specifies postconditions
- $I_{\text{start}} \in \mathcal{I}$ is the designated starting item

Throughout, we write $\text{SAT}(\phi)$ when formula ϕ has a satisfying assignment, $\text{UNSAT}(\phi)$ when none exists.

[1] We essentially rely on a subset of propositional logic with integer and array theory.

2.1. Precondition Reachability Classification

For each item I_i, we classify its precondition P_i relative to B into three categories:

$$\begin{aligned} \textbf{ALWAYS}: &\quad \mathrm{UNSAT}(B \wedge \neg P_i) \\ \textbf{NEVER}: &\quad \mathrm{UNSAT}(B \wedge P_i) \\ \textbf{CONDITIONAL}: &\quad \mathrm{SAT}(B) \end{aligned}$$

An **ALWAYS** item is presented to every respondent; **NEVER** means no valid assignment can satisfy the precondition, indicating a design error (the item is dead code); **CONDITIONAL** is the normal case where the item depends on prior answers.

2.2. Postcondition Classification

We classify postconditions relative to their preconditions into four categories:

$$\begin{aligned} \textbf{VACUOUS}: &\quad \mathrm{UNSAT}(B \wedge P_i) \\ \textbf{TAUTOLOGICAL}: &\quad \mathrm{UNSAT}(B \wedge P_i \wedge \neg Q_i) \\ \textbf{CONSTRAINING}: &\quad \mathrm{SAT}(B \wedge P_i \wedge Q_i) \text{ and } \mathrm{SAT}(B \wedge P_i \wedge \neg Q_i) \\ \textbf{INFEASIBLE}: &\quad \mathrm{UNSAT}(B \wedge P_i \wedge Q_i) \end{aligned}$$

VACUOUS correlates with NEVER reachability—the postcondition is irrelevant. **TAUTOLOGICAL** means the postcondition adds no constraint beyond what domains and the precondition already enforce, suggesting a redundant rule. **CONSTRAINING** is the normal case: the postcondition meaningfully restricts valid responses. **INFEASIBLE** is a design error—no valid response exists when the item is reached.

3. Comprehensive Validation

This section establishes the precise relationship between global and per-item validation, proving what each level can and cannot detect. These results guide the practical choice of validation strategy.

Definition 3.1 (Global Satisfiability Formula)**.** The **global satisfiability formula** for a questionnaire is:

$$\mathcal{F} \;:=\; B \;\wedge\; \bigwedge_{i=1}^{n} (P_i \Rightarrow Q_i)$$

The questionnaire passes global validation if $\mathrm{SAT}(\mathcal{F})$.

Intuitively, the global formula asks whether there exists *at least one* complete assignment of answers that respects all domain constraints and satisfies all postconditions for every reached item.

Definition 3.2 (Witness Formula)**.** For item I_i, the **witness formula** is $W_i := B \wedge P_i \wedge \neg Q_i$. If $\mathrm{SAT}(W_i)$, there exists an assignment where item I_i is reached but its postcondition is violated. Such an assignment *witnesses* a validation failure.

The diagnostic question is whether W_i is satisfiable: $\mathrm{UNSAT}(W_i)$ means item I_i is *locally valid* (whenever it is reached, its postcondition must hold), while $\mathrm{SAT}(W_i)$ indicates a potential violation. The witness assignment provides a concrete counterexample showing exactly which answer combination causes the failure.

3.1. Key Properties of Global Satisfiability

The following five theorems capture the exact relationships between per-item, global, and path-based validation.

Theorem 3.3 (Soundness of Per-Item Validity)**.** *If* $\mathrm{SAT}(B)$ *and for every* $i \in \{1, \ldots, n\}$ *we have* $\mathrm{UNSAT}(W_i)$*, then* $\mathrm{SAT}(\mathcal{F})$*.*

Proof. Assume $\mathrm{SAT}(B)$ and $\mathrm{UNSAT}(W_i)$ for all i, where $W_i = B \wedge P_i \wedge \neg Q_i$.

Step 1: Transform unsatisfiability to entailment. For each i, $\mathrm{UNSAT}(B \wedge P_i \wedge \neg Q_i)$ means:

$$\nexists\, \mathbf{a} : \ \mathbf{a} \models (B \wedge P_i \wedge \neg Q_i)$$

Equivalently, $\forall\, \mathbf{a} : \ \mathbf{a} \not\models B$ or $\mathbf{a} \not\models P_i$ or $\mathbf{a} \models Q_i$. Using $(\neg A \vee \neg B \vee C) \equiv ((A \wedge B) \Rightarrow C)$: if $\mathbf{a} \models B \wedge P_i$ then $\mathbf{a} \models Q_i$. By the deduction theorem: $B \models (P_i \Rightarrow Q_i)$.

Step 2: Combine entailments. Since $B \models (P_i \Rightarrow Q_i)$ for each i, by monotonicity of entailment:

$$B \models \bigwedge_{i=1}^{n} (P_i \Rightarrow Q_i)$$

Step 3: Construct a model for $\mathcal{F}$. By hypothesis, $\mathrm{SAT}(B)$, so there exists $\mathbf{a}$ with $\mathbf{a} \models B$. By Step 2, every model of B is also a model of $\bigwedge_{i=1}^{n}(P_i \Rightarrow Q_i)$. Therefore:

$$\mathbf{a} \models B \wedge \bigwedge_{i=1}^{n} (P_i \Rightarrow Q_i) = \mathcal{F}$$

Hence $\mathrm{SAT}(\mathcal{F})$. □

Theorem 3.4 (Local Invalidity Does Not Imply Global Failure)**.** *If* $\mathrm{SAT}(W_i)$ *for some* i*, it does not follow that* $\mathrm{UNSAT}(\mathcal{F})$*.*

The witness W_i only shows that there *exists* a violating assignment; the global check asks whether there *exists* a (possibly different) assignment satisfying all implications simultaneously.

Proof. By counterexample. Consider two items:

- I_1: age, $S_1 \in [0, 120]$, $P_1 = \mathit{true}$, $Q_1 = \mathit{true}$
- I_2: driving experience, $S_2 \in [0, 100]$, $P_2 = (S_1 \geq 16)$, $Q_2 = (S_2 \leq S_1 - 16)$

The postcondition Q_2 ensures driving experience cannot exceed years since turning 16.

The witness $W_2 = B \wedge (S_1 \geq 16) \wedge (S_2 > S_1 - 16)$ is satisfiable: $S_1 = 20$, $S_2 = 10$ gives 10 years of experience at age 20, violating Q_2 since $10 > 20 - 16 = 4$. Thus SAT(W_2)—item I_2 is locally invalid.

However, $\mathcal{F}$ is satisfiable: $S_1 = 30$, $S_2 = 5$ satisfies B and $(P_2 \Rightarrow Q_2)$ since $5 \leq 30 - 16 = 14$. Thus SAT($\mathcal{F}$) despite SAT(W_2). The local invalidity shows that *some* respondents could violate the postcondition, but valid completions still exist. □

Theorem 3.5 (Accumulated Constraints Can Cause Global Unsatisfiability)**.** *There exist questionnaires where no item is* INFEASIBLE *individually (each* SAT($B \wedge P_i \wedge Q_i$)*), yet* UNSAT($\mathcal{F}$)*.*

Per-item checks examine each postcondition independently against B, missing conflicts between postconditions that must be satisfied simultaneously during questionnaire execution.

Proof. By counterexample. Consider a satisfaction survey where I_1 asks "Rate your overall experience" with $S_1 \in [1, 100]$, $P_1 = \mathit{true}$, and postcondition $Q_1 = (S_1 > 50)$ requiring a positive rating. Item I_2 asks "Would you like a follow-up?" with $S_2 \in \{0, 1\}$, $P_2 = \mathit{true}$, and postcondition $Q_2 = (S_1 < 30)$ intended to trigger follow-up only for dissatisfied respondents. Each item is individually feasible:

- SAT($B \wedge P_1 \wedge Q_1$) with $S_1 = 75$: a positive rating satisfies Q_1
- SAT($B \wedge P_2 \wedge Q_2$) with $S_1 = 20$: a low rating satisfies Q_2

Neither is INFEASIBLE. However, since both preconditions are *true*:

$$\mathcal{F} = B \wedge (\mathit{true} \Rightarrow S_1 > 50) \wedge (\mathit{true} \Rightarrow S_1 < 30) = B \wedge (S_1 > 50) \wedge (S_1 < 30)$$

This is unsatisfiable—no S_1 can simultaneously satisfy both constraints. The per-item checks miss this conflict because they evaluate each postcondition against all possible values of S_1, not accounting for the fact that Q_1 constrains the same variable that Q_2 references. □

3.2. Dependency Graph and Cycle Detection

To ensure questionnaires can be evaluated in a consistent order, we analyze dependencies between items. Construction proceeds in three stages:

Stage 1: Data flow graph. Through static analysis of preconditions, and postconditions (and code blocks after introducing them in Section 4), we construct a directed graph with item vertices and variable vertices. Edges capture data flow: answering item I_j assigns outcome S_j (edge $I_j \to S_j$).

Stage 2: Item dependency graph. From the data flow graph, we derive $G = (V, E)$ where $V = \{I_1, \ldots, I_n\}$ and $(I_j \to I_i) \in E$ iff there exists a directed path from I_j to I_i in the data flow graph.

Stage 3: Topological sort and cycle detection. We compute a topological ordering of G using Kahn's algorithm [11]: initialize with zero in-degree items, iteratively remove items and their outgoing edges. If edges remain after termination, cycles exist and the questionnaire cannot be evaluated—each item on the cycle depends on others in the cycle.

3.3. Path-Based Validation

Path-based validation addresses a limitation of the global formula and per-item checks: items that appear reachable in isolation but become unreachable when predecessor constraints accumulate.

Definition 3.6 (Execution Path)**.** An **execution path** $\pi = (I_{k_1}, \ldots, I_{k_n})$ is a permutation of items respecting the dependency order where each item's precondition is satisfied given the accumulated constraints from preceding items.

Definition 3.7 (Path Constraint Formula)**.** For path $\pi = (I_{k_1}, \ldots, I_{k_n})$, the **path constraint formula** is:

$$\Phi_\pi \;:=\; B \;\wedge\; \bigwedge_{j=1}^{n} (P_{k_j} \wedge Q_{k_j})$$

The path is **feasible** if $\mathrm{SAT}(\Phi_\pi)$.

Note the key difference from the global formula: path constraints use conjunctions $(P_i \wedge Q_i)$ for items on the path, not implications $(P_i \Rightarrow Q_i)$. This ensures that preconditions are actually satisfiable, not just vacuously handled.

Theorem 3.8 (Global Validity is Necessary for Path Validity)**.** *If* $\mathrm{UNSAT}(\mathcal{F})$*, then for every execution path* π*,* $\mathrm{UNSAT}(\Phi_\pi)$*.*

Proof. By contraposition. Suppose $\mathrm{SAT}(\Phi_\pi)$ for some path $\pi = (I_{k_1}, \ldots, I_{k_m})$ with satisfying assignment $\mathbf{a}$.

For items on the path: $\mathbf{a} \models P_{k_j}$ and $\mathbf{a} \models Q_{k_j}$, so $\mathbf{a} \models (P_{k_j} \Rightarrow Q_{k_j})$.

For items not on the path: the execution path includes all items whose preconditions hold under $\mathbf{a}$ (by definition, a maximal set respecting the dependency order). Therefore, for items $I_i \notin \pi$, we have $\mathbf{a} \not\models P_i$, making $(P_i \Rightarrow Q_i)$ vacuously true under $\mathbf{a}$.

Combining: $\mathbf{a} \models B \wedge \bigwedge_{i=1}^{n} (P_i \Rightarrow Q_i) = \mathcal{F}$, giving $\mathrm{SAT}(\mathcal{F})$.

By contraposition: $\mathrm{UNSAT}(\mathcal{F})$ implies $\mathrm{UNSAT}(\Phi_\pi)$ for all π. □

Theorem 3.9 (Global Validity Does Not Guarantee All Items Reachable)**.** *There exist questionnaires where* $\mathrm{SAT}(\mathcal{F})$ *but some conditionally-reachable item is unreachable on every feasible path.*

Proof. We provide an example. Let $I_1 = (S_1 \in [1, 100]$, $P_1 = true$, $Q_1 = (S_1 > 80))$ and $I_2 = (S_2 \in \{0, 1\}$, $P_2 = (S_1 < 50)$, $Q_2 = true)$. The dependency graph gives $I_1 \to I_2$ since P_2 references S_1.

Per-item analysis: $P_2 = (S_1 < 50)$ is CONDITIONAL: SAT$(B \wedge S_1 < 50)$ with $S_1 = 25$. Neither item is INFEASIBLE.

Global analysis:

$$\mathcal{F} = B \wedge (true \Rightarrow S_1 > 80) \wedge ((S_1 < 50) \Rightarrow true) \equiv B \wedge (S_1 > 80)$$

This is satisfiable with $S_1 = 90$, so SAT$(\mathcal{F})$.

Path analysis: Any path including I_2 requires:

$$\Phi_\pi = B \wedge (S_1 > 80) \wedge (S_1 < 50)$$

which is unsatisfiable. Item I_2 is classified CONDITIONAL by per-item analysis, passes the global check, yet is unreachable on every valid path. This is **dead code**: the global formula masks I_2's unreachability because $(P_2 \Rightarrow Q_2) = ((S_1 < 50) \Rightarrow true)$ is vacuously true when Q_1 forces $S_1 > 80$. □

3.4. Accumulated Reachability

To detect dead code systematically without enumerating all paths—which would face the same exponential blow-up as testing—we define accumulated reachability, which accounts for constraints imposed by predecessor items in the dependency order.

Definition 3.10 (Accumulated Reachability)**.** For item I_i, let $\text{Pred}(i) = \{j : I_j \prec I_i\}$ be predecessor items in the topological order. The **accumulated constraint formula** is:

$$\mathcal{A}_i \;:=\; B \;\wedge \bigwedge_{j \in \text{Pred}(i)} (P_j \Rightarrow Q_j)$$

Item I_i is **accumulated-reachable** if SAT$(\mathcal{A}_i \wedge P_i)$.

If UNSAT$(\mathcal{A}_i \wedge P_i)$ for a conditionally-reachable item, it is dead code—its precondition can never be satisfied given accumulated predecessor constraints.

Example 3.11. Consider three items:

$$I_1 = (\textit{annual income}, S_1 \in [0, 1000000], P_1 = true, Q_1 = true)$$
$$I_2 = (\textit{tax bracket}, S_2 \in \{1, 2, 3\}, P_2 = true, Q_2 = (S_1 \geq S_2 \cdot 10000))$$
$$I_3 = (\textit{low-income assistance}, S_3 \in \{0, 1\}, P_3 = (S_1 < 10000), Q_3 = true)$$

There are two possible topological orders $I_1 \to I_2 \to I_3$ and $I_1 \to I_3 \to I_2$ (since both I_2 and I_3 refer to S_1). If we consider the first order, per-item analysis classifies P_3 as CONDITIONAL: SAT$(B \wedge S_1 < 10000)$ holds with $S_1 = 5000$. However, with predecessor set $\text{Pred}(3) = \{1, 2\}$:

$$\mathcal{A}_3 \wedge P_3 = B \wedge \underbrace{(S_1 \geq S_2 \cdot 10000)}_{Q_2} \wedge \underbrace{(S_1 < 10000)}_{P_3}$$

This is unsatisfiable: since $S_2 \geq 1$, Q_2 forces $S_1 \geq 10000$, contradicting P_3. Item I_3 (low-income assistance) is dead code—the postcondition Q_2 sneaks a lower bound on S_1 through I_2, making I_3 unreachable even though per-item analysis sees no problem. This order-independent fact is captured exactly by the path constraint formula Φ_π (Definition 3.7), which conjoins $P_i \wedge Q_i$ over all path items and is unsatisfiable here for both orders; in practice we detect such cases by evaluating accumulated reachability under a fixed canonical topological order.

3.5. Validation Hierarchy

The presented validation approaches can be combined into a hierarchy. Furthermore, we can utilise SMT-solvers (in our case, Z3 [15]) for instance-based verification: the presented validation formulas can be automatically checked for concrete questionnaires. Questionnaire validation operates at four levels of increasing thoroughness and computational cost:

1. **Per-item validation** (Sections 2.1 and 2.2) classifies each item independently by testing its precondition reachability and postcondition feasibility. This is the fastest level, as item analyses are parallelizable and each requires only 2–3 SMT queries.
2. **Global validation** (Definition 3.1) performs a single satisfiability check on the global formula $\mathcal{F}$, determining whether at least one valid questionnaire completion exists.
3. **Cycle detection** (Section 3.2) analyzes the dependency graph for circular dependencies, which indicate that the item ordering cannot be resolved.
4. **Path-based validation** (Definition 3.10) verifies accumulated reachability for each conditional item under predecessor postconditions. This is the most thorough level, capable of detecting dead code that earlier levels miss, at the cost of one SMT query per item with accumulated constraints.

Validation outcomes. The hierarchy yields six diagnostic cases: (1) all W_i UNSAT implies all levels pass (Theorem 3.3); (2) some SAT(W_i) with UNSAT($\mathcal{F}$) means conflicting postconditions—no valid path exists (Theorems 3.5 and 3.8); (3) SAT($\mathcal{F}$) with all items accumulated-reachable is the ideal outcome; (4) SAT($\mathcal{F}$) with some items not accumulated-reachable indicates dead code (Theorem 3.9); (5) any INFEASIBLE item is always a design error; (6) any NEVER-reachable item signals an unsatisfiable precondition.

4. Implementation of the Formal Model

We introduce the Questionnaire Markup Language (QML, specified in [21]), a YAML-based specification language where preconditions and postconditions are expressed as Python boolean expressions. Items may contain imperative code blocks

for intermediate computations. These code blocks must be transformed into Z3 constraints (influencing pre- and postconditions), a non-trivial challenge since Z3 variables are immutable while Python allows mutation, conditional branching, and iteration. We present Example 3.11 in Section A with QML syntax, and refer to [20] for further, more involved examples.

4.1. Compilation Strategy

The compiler [22] transforms Python code blocks into Z3 formulas through three key techniques:

Single Static Assignment (SSA). SSA [4] ensures each variable is assigned exactly once by creating versioned copies at each reassignment point (e.g., $\text{total}_0 = 0$, $\text{total}_1 = \text{total}_0 + x$, $\text{total}_2 = \text{total}_1 + y$). Each SSA variable maps directly to an immutable Z3 expression, eliminating mutation while preserving the semantics.

Phi functions for conditional branching. When control flow paths merge after conditional statements, *phi functions* [4] select the appropriate variable version: $x_{\text{new}} = \phi(\text{cond}, x_{\text{true}}, x_{\text{false}})$, mapping to Z3's `If`$(\text{cond}, x_{\text{true}}, x_{\text{false}})$. Nested conditionals produce nested `If` expressions, preserving full semantic equivalence with the original Python code.

Bounded loop unrolling. Since Z3 cannot represent arbitrary loops, we employ bounded loop unrolling [3, 13] for `for` loops with statically determinable bounds (range calls with compile-time constants, container literals). Each iteration is expanded into sequential SSA-transformed constraints. The compiler unrolls up to 20 iterations, issuing a warning (not an error) for larger loops and using partial results for validation.

4.2. Unified Analysis

A distinguishing feature of this approach is the unified analysis of both declarative conditions and imperative code blocks within a single constraint system. Variables introduced in code blocks flow through the dependency graph alongside outcome variables; therefore, we extend the data flow graph construction step to include these variables too (i.e. data flow edges can contain these new variables too), but item dependencies and the topological sort is determined exactly the same way as we discussed in Section 3.2.

Only code portions that ultimately influence a precondition or postcondition are compiled; auxiliary computations are ignored. This selective compilation reduces constraint complexity without affecting verification completeness.

The compiler supports a Python subset covering common questionnaire patterns: arithmetic, comparisons, Boolean operators, if/elif/else, bounded for loops, variable assignment, list literals, and attribute access [20]. Vector and matrix operations map to Z3's array theory.

4.3. Evaluation

We converted 12 real-world questionnaires (26–303 pages) from their original imperative format into QML and ran the full validation hierarchy on each [19]. In these traditional instruments, branching between questions is expressed imperatively through skip logic or explicit *GOTO* jump instructions that route the respondent to a target question. The corpus spans major national surveys from Canada, the U.S., and Europe, totalling over 4,200 items in 12 QML files. A companion benchmark harness measures how validation time and memory scale with questionnaire size and complexity [23]. We provide a summary extracted from this benchmark in Section B.

Conversion methodology. Each questionnaire was converted through a three-phase LLM-assisted pipeline. First, the source PDF was OCR-extracted and parsed into a structured question inventory cataloguing every item, its response options, and its imperative routing instructions (GOTO targets, interviewer checks, filter conditions). Second, the inventory was transformed into declarative QML by systematically converting GOTO-based skip patterns into preconditions and interviewer consistency checks into postconditions, following eight identified conversion patterns (e.g., forward GOTOs become disjunctive preconditions over skipped items; range-check interviewer instructions become postconditions on the preceding question). Third, the resulting QML files were validated using the full four-level hierarchy described in Section 3.

Structural equivalence. All GOTO-based routing was successfully expressed as preconditions, confirming that traditional imperative questionnaires can be faithfully translated into the declarative model without loss of semantics or structure. All 12 questionnaires passed the full validation hierarchy. This is expected—these are mature, production instruments that have been refined over many survey cycles.

Postconditions and data consistency. Despite their structural soundness, the questionnaires reveal a systematic gap in data quality assurance. Half of the 12 questionnaires contain zero postconditions—no range checks, no cross-field edits, no logical constraints of any kind. Across the entire corpus, only 63 constraining postconditions exist (1.5% of items), concentrated almost entirely in a single questionnaire. Traditional practice handles such inconsistencies through post-collection data cleaning rather than preventing them at the point of collection. QML postconditions enable a shift from correction to prevention: at runtime the survey engine enforces them, rejecting inconsistent responses before they enter the dataset.

5. Related Work

Questionnaire logic formalization. The formal study of questionnaires began with Picard [18], who established the mathematical foundations of question-

naire theory as a branch of combinatorics. Fellegi and Holt [9] introduced edit rules—consistency constraints on survey responses—and systematic procedures for detecting and correcting violations. Willenborg [25] formalized routing structures (skip patterns) and edit structures as separate concepts, later proving [26] that checking their combined consistency is NP-complete. We extend this line from post-collection cleaning to pre-deployment verification via SMT solving.

Graph-based approaches. Fagan and Greenberg [7] pioneered graph representations for skip patterns in census questionnaires. Bethlehem and Hundepool [1] developed TADEQ, a comprehensive tool for documentation and analysis of electronic questionnaires, using directed graphs to visualize routing logic. Elliott [6] derived basis path sets from questionnaire graphs for systematic testing. Schiopu-Kratina et al. [24] formalized the relationship between questionnaires and labeled directed graphs, emphasizing that edges must carry conditions. Our dependency graph construction extends this tradition with SMT-based formal verification.

SMT solvers and bounded model checking. SMT solvers [2, 15, 16] combine Boolean satisfiability with decision procedures for background theories (integers, arrays, bitvectors). The DPLL(T) architecture [16] enables efficient handling of constraints mixing Boolean logic with integer arithmetic—exactly the combination questionnaire validation requires. Bounded model checking [3, 13] applies similar techniques to verify software properties by unrolling loops to a bounded depth, analogous to our loop unrolling strategy. Our compilation approach also parallels symbolic execution [12].

6. Conclusion

We have presented a formal framework for questionnaire validation based on pre- and postconditions inspired by Hoare Logic. The framework provides a four-level validation hierarchy (per-item, global, cycle detection, and path-based) with theorems characterizing what each level can and cannot detect. A compiler bridges practical questionnaire authoring in the QML language [22] with formal verification via Z3 SMT constraints. Evaluation on 12 real-world questionnaires (over 4,200 items) confirmed that imperative designs translate faithfully into the declarative model, that mature instruments pass structural validation as expected, and that postconditions address a systematic gap by shifting data consistency enforcement from post-collection cleaning to the point of collection.

Future work. We plan to extend language support (functions, exception handling), develop incremental validation for efficient re-checking after modifications, and apply the framework to domains beyond surveys—compliance audits, medical triage protocols, and safety-critical system checklists—where formally verified conditional logic is essential.

References

[1] J. G. Bethlehem, A. Hundepool: *TADEQ: A Tool for Documentation and Analysis of Electronic Questionnaires*, Journal of Official Statistics 20.2 (2004), pp. 233–263.

[2] A. Biere, M. Heule, H. van Maaren, T. Walsh: *Handbook of Satisfiability*, vol. 185, Frontiers in Artificial Intelligence and Applications, IOS Press, 2009.

[3] E. Clarke, A. Biere, R. Raimi, Y. Zhu: *Bounded Model Checking Using Satisfiability Solving*, Formal Methods in System Design 19.1 (2001), pp. 7–34, doi: 10.1023/A:1011276507260.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck: *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, ACM Transactions on Programming Languages and Systems 13.4 (1991), pp. 451–490, doi: 10.1145/115372.115320.

[5] E. W. Dijkstra: *Notes on Structured Programming*, in: Structured Programming, ed. by O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, London: Academic Press, 1972, pp. 1–82.

[6] D. Elliott: *The Application of Graph Theory to the Development and Testing of Survey Instruments*, Survey Methodology 38.2 (2012), pp. 195–206.

[7] J. T. Fagan, B. V. Greenberg: *Using Graph Theory to Analyze Skip Patterns in Questionnaires*, in: Proceedings of the Section on Survey Research Methods, American Statistical Association, 1988, pp. 193–198.

[8] G. Feeney, S. Feeney: *On the Logical Structure of Census and Survey Questionnaires*, Genus 75.1 (2019), pp. 1–23, doi: 10.1186/s41118-019-0065-y.

[9] I. P. Fellegi, D. Holt: *A Systematic Approach to Automatic Edit and Imputation*, Journal of the American Statistical Association 71.353 (1976), pp. 17–35, doi: 10.1080/01621459.1976.10481472.

[10] C. A. R. Hoare: *An Axiomatic Basis for Computer Programming*, Communications of the ACM 12.10 (1969), pp. 576–580, doi: 10.1145/363235.363259.

[11] A. B. Kahn: *Topological Sorting of Large Networks*, Communications of the ACM 5.11 (1962), pp. 558–562, doi: 10.1145/368996.369025.

[12] J. C. King: *Symbolic Execution and Program Testing*, Communications of the ACM 19.7 (1976), pp. 385–394, doi: 10.1145/360248.360252.

[13] D. Kroening, M. Tautschnig: *CBMC—C Bounded Model Checker*, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Lecture Notes in Computer Science, Springer, 2014, pp. 389–391, doi: 10.1007/978-3-642-54862-8_26.

[14] J. Lerner: *The Promise and Pitfalls of AI-Augmented Survey Research*, tech. rep., NORC at the University of Chicago, 2024.

[15] L. de Moura, N. Bjørner: *Z3: An Efficient SMT Solver*, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), vol. 4963, Lecture Notes in Computer Science, Springer, 2008, pp. 337–340, doi: 10.1007/978-3-540-78800-3_24.

[16] R. Nieuwenhuis, A. Oliveras, C. Tinelli: *Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)*, Journal of the ACM 53.6 (2006), pp. 937–977, doi: 10.1145/1217856.1217859.

[17] F. Olivos, M. Liu: *ChatGPTest: Opportunities and Cautionary Tales of Utilizing AI for Questionnaire Pretesting*, Field Methods (2024), doi: 10.1177/1525822X241280574.

[18] C. Picard: *Théorie des questionnaires*, Paris: Gauthier-Villars, 1965.

[19] P. Sághelyi: *Questionnaire Markup Language (QML): Evaluation Corpus of Real-World Questionnaires*, Accessed: 2026-06-18, 2026, url: https://github.com/askalot-io/QML/tree/main/evaluation.

[20] P. Sághelyi: *Questionnaire Markup Language (QML): Example Specifications*, Accessed: 2026-06-18, 2026, URL: https://github.com/askalot-io/QML/tree/main/tests/fixtures.

[21] P. Sághelyi: *Questionnaire Markup Language (QML): Language Specification (JSON Schema and EBNF Grammar)*, Accessed: 2026-06-18, 2026, URL: https://github.com/askalot-io/QML/tree/main/askalot_qml/schema.

[22] P. Sághelyi: *Questionnaire Markup Language (QML): Reference Implementation and Compiler*, Accessed: 2026-03-30, 2026, URL: https://github.com/askalot-io/QML.

[23] P. Sághelyi: *Questionnaire Markup Language (QML): Validator Scaling Benchmark Harness*, Accessed: 2026-06-18, 2026, URL: https://github.com/askalot-io/QML/tree/main/benchmark.

[24] I. Schiopu-Kratina, C. M. Zamfirescu, K. Trépanier, L. Marques: *Survey Questionnaires and Graphs*, Electronic Journal of Statistics 9.2 (2015), pp. 2005–2035, DOI: 10.1214/15-EJS1067.

[25] L. Willenborg: *Computational Aspects of Survey Data Processing*, PhD thesis, University of Amsterdam, 1988.

[26] L. Willenborg: *Testing and Checking Questionnaires*, in: Proceedings of the ASA Section on Survey Research Methods, American Statistical Association, 1995, pp. 1032–1037.

A. QML Source for the Running Example

Listing 1. QML specification for the income survey of Example 3.11.

```yaml
qmlVersion: "2.0"

questionnaire:
  title: "Income Survey with Dead Code"

  blocks:
    - id: block1
      kind: Group
      title: "Financial Information"
      items:
        - id: q_income
          kind: Question
          title: "What is your annual income?"
          input:
            control: Editbox
            min: 0
            max: 1000000

        - id: q_tax_bracket
          kind: Question
          title: "Select your tax bracket"
          input:
            control: Radio
            labels:
              1: "Low"
              2: "Medium"
              3: "High"
          postcondition:
            - predicate: q_income.outcome >= q_tax_bracket.outcome * 10000
              hint: "Income must cover the tax bracket's lower bound"

        - id: q_low_income_assist
          kind: Question
          title: "Would you like information about low-income assistance programs?"
          input:
            control: Radio
            labels:
              0: "No"
              1: "Yes"
          precondition:
            - predicate: q_income.outcome < 10000
```

B. QML Benchmark Data

To characterise how validation cost scales, a companion harness [23] generates synthetic QML questionnaires that vary one structural axis at a time – item count, number of preconditions, number of postconditions, and dependency depth – and times the full per-item validation pipeline (Section 3.5). Each questionnaire is validated in a fresh subprocess so that peak memory reflects Z3's native allocations;

reported values are medians of five repetitions after one warm-up run. The measurements were collected with Z3 4.15.3 and the QML compiler [22] on a single thread of an x86-64 Linux host.

Wall-clock time is split into three phases: YAML *parse* and AST construction, constraint construction, and the *Z3 solve*; we plot parse, Z3 solve, and their sum (*total*; constraint construction is the small remainder). Figure 1 shows that the Z3 solve dominates at scale and is by far most sensitive to the number of preconditions, whereas dependency depth has negligible effect. Figure 2 shows peak memory remains within a narrow ∼49–61 MiB band throughout, so validation is compute-bound rather than memory-bound at these sizes.

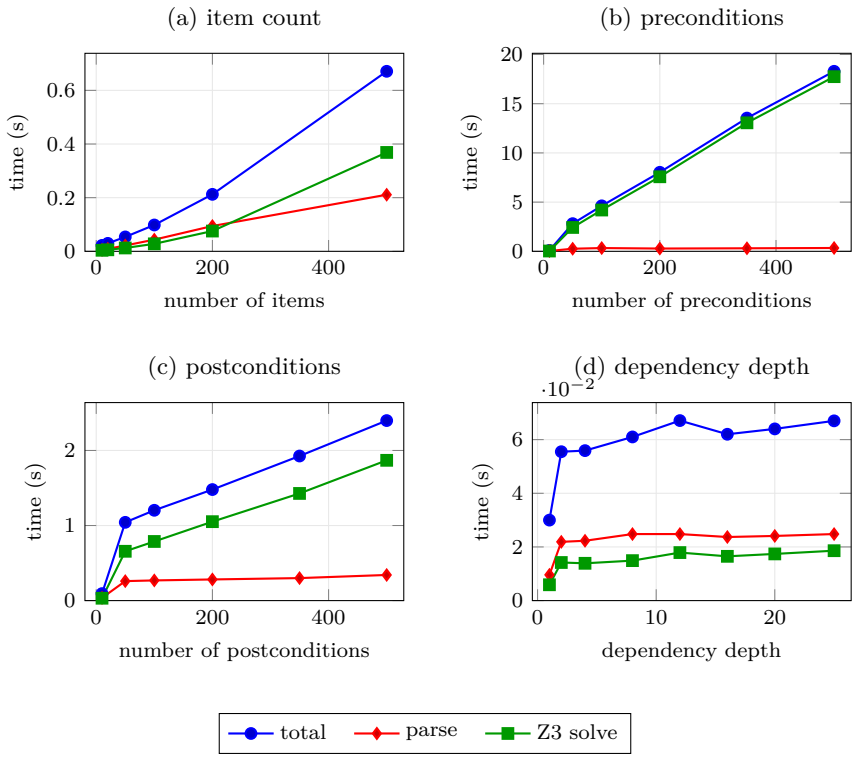

Figure 1. QML validation wall-clock time vs. questionnaire size and complexity, decomposed into end-to-end *total*, YAML *parse*/AST build, and the isolated *Z3 solve*. Each marker is the median of five repetitions. The Z3 solve dominates and grows steeply with the number of preconditions (panel b); item count grows roughly linearly (a); postconditions grow mildly (c); dependency depth is essentially free (d, note the 10^{-2} scale).

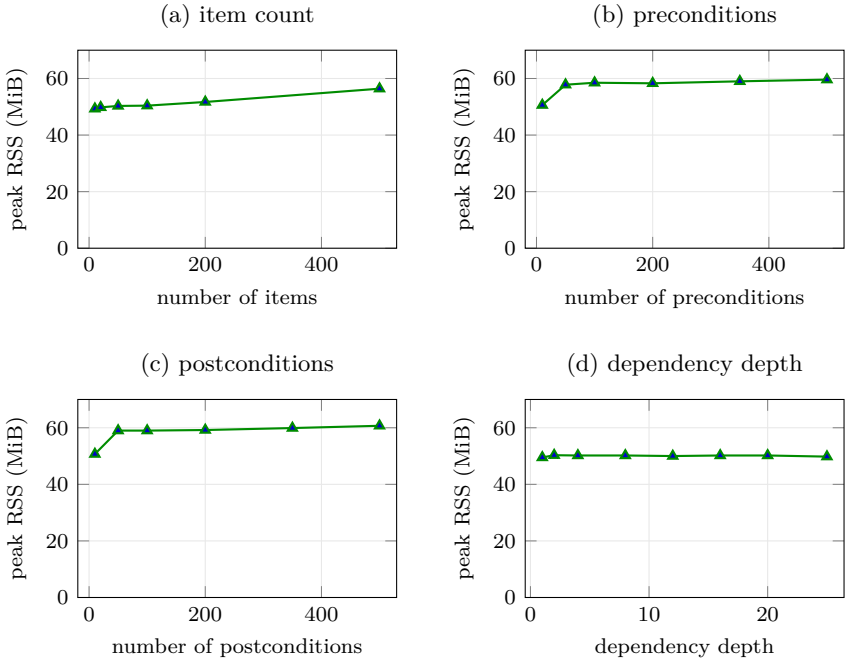

Figure 2. Peak resident set size (whole-process RSS, including Z3's native allocations) for the same sweeps. Memory stays within a narrow ∼49–61 MiB band across every configuration (axes start at 0 to make the flatness explicit), confirming that validation is compute-bound, not memory-bound, at these scales.