Let the Agent Sync: LLM-Driven Code Generation for the Families to Persons Case

Thomas Buchmann[1,*,†]

[1]Hof University of Applied Sciences, Hof, Germany

Abstract

We present *BXAgent*, a tool that automatically generates Java code for bidirectional model transformations between EMF/Ecore metamodels from a natural-language description. Given two `.ecore` files and a short textual specification, an LLM-backed pipeline produces a single Java class that supports (i) batch transformations in both directions, (ii) incremental synchronisation after single-sided edits, and (iii) concurrent synchronisation (*csync*) when both models are edited independently. Unlike conventional bx tools that require a formal specification in a dedicated transformation language (e.g. TGGs or lenses), BXAgent's only user-facing input is natural language. The generated code is deterministic and stable: because the transformation logic is produced by a fixed FreeMarker template, repeated runs yield identical output for the same mapping specification. We applied BXAgent to the revised Families-to-Persons (F2P) benchmark of TTC 2026, which extends the classic case with 13 concurrent test cases and 3 round-trip cases in addition to the original 34 batch and incremental tests. BXAgent passes **46 out of 50** tests, including all four genuine conflict cases. We describe the tool architecture, the generated synchronisation algorithm, our conflict resolution strategy, and discuss the four failing cases together with their root causes.

Keywords

bidirectional transformations, model synchronisation, concurrent synchronisation, LLM code generation, EMF, Benchmarx

1. Introduction

Bidirectional model transformations (bx) are notoriously difficult to implement correctly: the forward and backward directions must be mutually consistent, incremental updates must not overwrite unrelated state, and concurrent edits on both sides demand a coherent conflict-resolution strategy [1]. Dedicated bx languages and frameworks (e.g. eMoflon, BXtend, NMF Synchronizations) address these concerns, but they require the developer to write a formal specification in a dedicated notation — a triple-graph grammar, a lens expression, or a declarative synchronisation rule — before any code is generated. The expertise barrier is substantial.

BXAgent [2] takes a different approach: it uses a large language model (LLM) to *generate* the transformation code from a high-level, natural-language description of the desired correspondence. The developer supplies only the two Ecore metamodels and a short English paragraph; the LLM infers the structural mapping and emits a typed `TransformationSpec`, which a deterministic FreeMarker template then expands into a complete Java class. Because the template is fixed, the generated code is identical across runs for the same spec: stability is a property of the architecture, not of the LLM. In practice, for the F2P case no compilation-repair round was necessary — the LLM produced a valid mapping on the first attempt.

This design also addresses the limitations of our earlier, chat-mode exploration [3]: in that work, the transformation was built up through many sequential prompts, leading to non-deterministic output, regression failures when previously working fragments were refined, and an inability to reach beyond batch transformations. BXAgent avoids these problems by separating the LLM's role (mapping extraction only) from the algorithmic responsibility (code structure, incremental logic, and csync algorithm), which

Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC. Rennes, France, June 29-July 3, 2026

*Corresponding author.

thomas.buchmann@hof-university.de (T. Buchmann)

0000-0002-5675-6339 (T. Buchmann)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-26

CEUR-WS.org/Vol-4338/TTC-paper2.pdf

is handled entirely by the template.

One important caveat: the incremental synchronisation path scales as $O(N)$ in total model size, not $O(\Delta)$ in edit size, because change detection relies on scanning all correspondence entries. This limits applicability to large, frequently edited models; we discuss the issue and a targeted fix in Sections 4 and 5.

The revised Families-to-Persons benchmark of TTC 2026 [4] provides an ideal evaluation vehicle. It retains all original 34 forward, backward, and incremental test cases, adds 13 concurrent synchronisation scenarios ranging from conflict-free monotonic edits to genuine move/delete, rename/rename, and delete/rename conflicts, and adds 3 round-trip cases. BXAgent is applied to this extended suite without any manual post-editing of the generated code.

The remainder of this paper is structured as follows. Section 2 briefly summarises the F2P case and the new test categories. Section 3 describes the BXAgent pipeline and the architecture of the generated transformation class. Section 4 reports test results and scalability measurements. Section 5 discusses limitations and the four failing cases. Section 6 surveys related work, and Section 7 concludes.

2. The F2P Case (TTC 2026)

The Families-to-Persons (F2P) case transforms a *family registry* (containing `FamilyRegister`, `Family`, and `FamilyMember` objects) into a *person register* (containing `PersonRegister`, `Male`, and `Female` objects, both extending the abstract `Person`) [5].

The central consistency relation is a bijection between family members and persons: a member in the `father` or `sons` role maps to a `Male` person, a member in the `mother` or `daughters` role maps to a `Female` person, and the person's name is composed as `"familyName, memberFirstName"`.

TTC 2026 introduces three new categories on top of the original 34 tests:

Round-trip tests (3). A model is transformed forward and then backward (or vice versa), and the result must be semantically equivalent to the original. These tests verify invertibility of the transformation.

Concurrent non-conflicting tests (9). The source and target models are independently edited (monotonic creations, monotonic deletions, or mixed non-monotonic edits), and the tool must restore consistency by propagating all edits without loss.

Concurrent conflict tests (4). Edits on both sides are semantically incompatible: a family member is both moved to another family *and* deleted in the person register (move/delete); a member is renamed on both sides to different names (rename/rename); a member is deleted on one side and renamed on the other (delete/rename); and a member is moved on one side and renamed on the other (move/rename). The tool must detect the conflict and apply a documented resolution strategy.

3. BXAgent

3.1. Overview

BXAgent is an LLM-backed code-generation pipeline for bidirectional EMF transformations. The tool is implemented in Java 21 with Maven and supports three LLM backends (Anthropic Claude, OpenAI, Ollama) configured via an `agent.properties` file.

Figure 1 shows the tool pipeline.

3.2. Pipeline Stages

Stage 1 – EcoreParser. Both metamodels are loaded via EMF reflection. For each `EClass` the parser collects its attributes, references (including containment and `eOpposite` flags), and supertypes, and assembles a compact *MetamodelSummary* in a format suitable for inclusion in an LLM prompt. Listing 1 shows an excerpt of the summary produced for the F2P family metamodel.

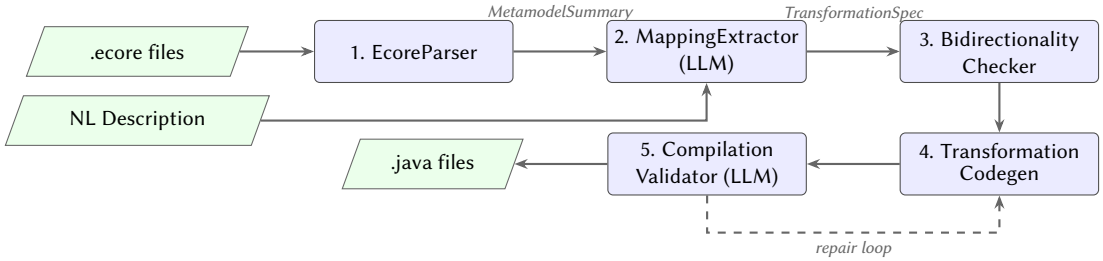


Figure 1: The BXAgent pipeline. Stages 2 and 5 involve LLM interaction; all other stages are deterministic.

Listing 1: Excerpt of the MetamodelSummary for the F2P family metamodel as passed to the LLM prompt.

```
EClass: FamilyRegister
  containment ref: families [0..*] -> Family

EClass: Family
  attribute: name : EString
  containment ref: father  [0..1] -> FamilyMember
  containment ref: mother  [0..1] -> FamilyMember
  containment ref: sons    [0..*] -> FamilyMember
  containment ref: daughters [0..*] -> FamilyMember

EClass: FamilyMember
  attribute: name : EString
```

Stage 2 – MappingExtractor. An LLM receives the two metamodel summaries and the user-provided natural-language description and is asked to respond with a single JSON object conforming to a `TransformationSpec` schema (Listing 2). The schema captures `TypeMappings`, `AttributeMappings`, `ReferenceMappings`, and the central `RoleBasedTypeMapping` construct for cases like F2P where one source type (`FamilyMember`) is distributed across several target types depending on its containment role. The LLM is given up to three retries on JSON parse errors.

Listing 2: Excerpt of TransformationSpec schema (simplified).

```
record TransformationSpec(
  List<TypeMapping>          typeMappings,
  List<AttributeMapping>     attributeMappings,
  List<ReferenceMapping>     referenceMappings,
  List<RoleBasedTypeMapping> roleBasedTypeMappings,
  List<TransformationOption> transformationOptions,
  List<BackwardConfig>       backwardConfigs
) {}

record RoleBasedTypeMapping(
  String sourceType,        // e.g. "FamilyMember"
  String intermediateType,  // e.g. "Family" (disappears)
  Map<String,String> roleToTargetType, // "father"->"Male",...
  String nameExpression,    // forward Java expression
  String backwardFamilyNameExpression,
  String backwardMemberNameExpression
) {}
```

Stage 3 – BidirectionalityChecker. Each `AttributeMapping` must supply both a `forwardExpression` and a `backwardExpression`. If the LLM left a backward expression empty (e.g. for a lossy mapping), the checker either prompts the user interactively or, when `BackwardConfig` parameters are present, proceeds automatically.

Stage 4 – TransformationCodegen. A FreeMarker template is instantiated with a data model derived from the `TransformationSpec`. The template generates a single, self-contained Java class with all required methods (see Section 3.3).

Stage 5 – CompilationValidator. The generated source is compiled in a temporary directory using the Java compiler API. On failure, compilation errors are fed back to the LLM for an automatic repair attempt (up to three rounds).

3.3. Generated Transformation Class

The generated class exposes the following public API:

Listing 3: Public API of the generated transformation class.

```
// Batch
void transform(Resource src, Resource tgt);
void transformBack(Resource tgt, Resource src);

// Incremental (single-sided)
void transform(Resource src, Resource tgt, Resource corrModel);
void transformBack(Resource tgt, Resource src, Resource corrModel);

// Concurrent synchronisation
SyncResult sync(Resource src, Resource tgt, Resource corrModel);
SyncResult sync(Resource src, Resource tgt, Resource corrModel, SyncConflictPolicy
    policy, DeletionPolicy delPolicy, Options opts);
```

3.3.1. Correspondence Model

All incremental and synchronisation methods rely on a *correspondence model* – a dynamically defined EMF EPackage stored as a `.corr.xmi` file in the same resource set as the source and target models. Each `CorrespondenceEntry` holds EReferences (not string IDs) to the matched source and target objects, together with stored fingerprints for both sides and the containment role. Crucially, when an EObject is deleted via `EcoreUtil.delete(obj, true)`, EMF automatically nulls all EReferences pointing to it – including those in correspondence entries. This design provides tombstone-free deletion detection without requiring explicit change notifications.

3.3.2. Fingerprint-Based Change Detection

Rather than relying on persistent XMI IDs or explicit change listeners, BXAgent computes a *content fingerprint* for each tracked object by concatenating the values of its designated key attributes (or all attributes as a fallback). For `RoleBasedTypeMapping`, a composite fingerprint is used: `family.getName() + "|" + memberFingerprint`, which detects both member renames and family-level renames in one step.

Change detection during incremental synchronisation then reduces to comparing the current fingerprint of each object with the stored value in its correspondence entry.

3.3.3. Concurrent Synchronisation Algorithm (`sync()`)

The `sync()` method implements a six-step algorithm over the four partitions of the correspondence model, determined by whether each entry's source and target references are null or non-null (Table 2).

Partition	src	tgt	Interpretation
1	$\neq$ null	$\neq$ null	Both sides exist; inspect fingerprints
2	$\neq$ null	null	Target deleted; recreate or propagate
3	null	$\neq$ null	Source deleted; recreate or propagate
4	null	null	Both deleted; garbage-collect entry

Figure 2: Four partitions of correspondence entries in `sync()`.

Step 1 – Partition 1: Fingerprint classification. Each entry with both sides non-null is classified by comparing current versus stored fingerprints on each side:

Δ_{src}	Δ_{tgt}	Action
yes	no	Case A: propagate forward
no	yes	Case B: propagate backward
yes	yes	Case C: conflict $\rightarrow$ apply `SyncConflictPolicy`
no	no	Case D: skip

For role-based types, structural changes (role switch, family reassignment) cannot be handled by attribute propagation alone. Instead, BXAgent uses *fingerprint neutralisation*: the stored fingerprint of the losing side is overwritten with its current value, signalling to the downstream role-based incremental step that no change has occurred on that side, so only the winning side's incremental method fires.

Step 2–4 – Partitions 2–4. Missing target objects (Partition 2) are recreated or propagated from the source; missing source objects (Partition 3) are handled symmetrically; double-null entries (Partition 4) are removed from the correspondence model.

Step 5a – Fuzzy matching. Before creating new objects for unmatched sources or targets, BXAgent checks whether they semantically correspond to an already existing object on the other side. This handles the scenario where both sides independently added the same real-world entity: the tool computes the *expected target fingerprint* of each unmatched source object and checks it against an index of unmatched target fingerprints. On a hit, a correspondence entry is created without duplicating the object.

Step 5b – Object creation. Remaining unmatched objects receive new counterparts.

Step 6 – Reference resolution. After all structural changes, the correspondence index is rebuilt and cross-references are re-resolved.

3.3.4. Conflict Resolution

The `SyncConflictPolicy` enum offers three strategies:

`SOURCE_WINS` (default) The source-side edit takes precedence; target attributes are overwritten.

`TARGET_WINS` The target-side edit takes precedence.

LOG_AND_SKIP No automatic resolution; conflicts are recorded in `SyncResult.conflicts()` and both sides retain their local state (divergent).

For the TTC benchmark, conflict test cases are evaluated with SOURCE_WINS unless the expected outcome requires otherwise.

3.4. Integration with Benchmarx 2.0

BXAgent does not directly implement the `BXTool<S,T,D>` interface of Benchmarx 2.0. As is common practice for all tools participating in the Benchmarx evaluation, a thin, manually written adapter bridges the framework's test infrastructure and the generated transformation class. This adapter translates Benchmarx's `performAndPropagateEdit(Supplier<IEdit<S>>, Supplier<IEdit<T>>)` calls into the corresponding `sync()`, `transform()`, and `transformBack()` invocations on the generated class, and delegates precondition and postcondition checks to the framework's utility functions. The adapter is small and generic; the transformation logic itself resides entirely in the generated code.

4. Evaluation

4.1. Test Results

All concurrent tests in this evaluation were run with the SOURCE_WINS conflict policy, meaning that whenever the source-side (family register) and target-side (person register) edits are in tension, the source-side change takes precedence.

Table 1 summarises BXAgent's performance on the complete TTC 2026 F2P test suite.

Table 1
BXAgent test results on the TTC 2026 F2P benchmark (conflict policy: SOURCE_WINS).

Category	Tests	Passed
Batch (forward / backward)	16	16
Incremental (forward / backward)	18	18
Round-trip	3	3
Concurrent – Monotonic Creating	4	4
Concurrent – Monotonic Deleting	3	1
Concurrent – Non-Monotonic	2	0
Concurrent – Conflicts	4	4
Total	**50**	**46**

BXAgent passes all batch, incremental, round-trip, conflict, and monotonic creation tests. All four failing tests are concentrated in the *MonotonicDeleting* and *NonMonotonic* concurrent categories. Notably, BXAgent handles all four genuine conflict scenarios correctly under the SOURCE_WINS policy. The root cause of the failures is a systematic ordering issue in the `sync()` algorithm, described in Section 5.

4.2. Scalability

All runtime measurements are median values over at least five independent runs on the same hardware. Model size is expressed as n families with 5 members each (forward direction) or $5n$ persons grouped by surname (backward direction). Times are given in seconds.

RQ1 – Batch transformations (Figure 3). Both forward (FWD) and backward (BWD) batch transformations scale linearly with model size. At $n = 1{,}000$ families (5 000 members) both directions complete in under 50 ms; at $n = 10{,}000$ forward takes 0.871 s and backward 0.847 s. The forward direction was also measured at $n = 100{,}000$, where it takes 43.5 s, confirming the linear trend but also showing that

the constant factor is non-trivial at very large scales. These numbers show that BXAgent's generated batch code is efficient enough for typical modelling workloads; for very large models a dedicated, non-reflective batch mechanism would be preferable (RQ1: partially yes).

RQ2 – Incremental transformations (Figure 4). The incremental paths (INCR_FWD, INCR_BWD) do *not* achieve constant-time updates. Both directions show a clear super-linear growth: INCR_FWD reaches 0.413 s at $n = 1{,}000$ and 57 s at $n = 10{,}000$; INCR_BWD reaches 0.849 s at $n = 1{,}000$ and 119 s at $n = 10{,}000$.

This is a fundamental architectural limitation: the incremental algorithm iterates over *all* correspondence entries to detect changes via fingerprint comparison, giving each synchronisation step $O(N)$ complexity in the total model size rather than $O(\Delta)$ in the size of the edit. Consequently, BXAgent's incremental mode is practical for small to medium-sized models but **not suitable as a drop-in replacement for purpose-built incremental bx tools** on large, frequently edited models. Perfect incrementality would require change notification (e.g. EMF adapters or explicit delta recording) rather than full-scan fingerprinting (RQ2: no — the current implementation is $O(N)$, not $O(\Delta)$).

RQ3 – Csync with fixed edits, increasing model size (Figure 5). Two edit patterns are compared on models of growing size: a conflict-free pattern (delete one son in the family model + add a new daughter in the person register; CDCsync) and a conflict-inducing pattern (move a daughter to another family + delete her in the person register; CDCFCsync). Both patterns show linear scaling. The conflict-free variant grows from 1 ms at $n = 3$ to 17 ms at $n = 50$; the conflict-inducing variant is slightly faster at small sizes but reaches 26 ms at $n = 100$, reflecting the additional fingerprint-neutralisation work for the conflict case. The overhead of conflict detection and resolution is modest and does not cause non-linear growth (RQ3: csync updates are near-linear; conflict resolution adds a small but manageable overhead).

RQ4 – Csync with fixed model, increasing edit size (Figure 6). Using a fixed model of 500 families (2 500 persons), the number n of concurrently edited families is increased. Again, two patterns are compared: conflict-free (CMCsync) and conflict-inducing (CMCFCsync). Both patterns grow roughly linearly with edit size; however, the conflict-inducing variant shows significantly higher absolute times — reaching 0.946 s at $n = 100$ edited families versus 0.244 s for the conflict-free pattern at $n = 50$. This suggests that the fingerprint-neutralisation and object-replacement steps triggered by conflict resolution carry a non-trivial per-conflict cost (RQ4: synchronisation time scales linearly with edit size; conflict resolution introduces a roughly $4\times$ overhead per conflicting family).

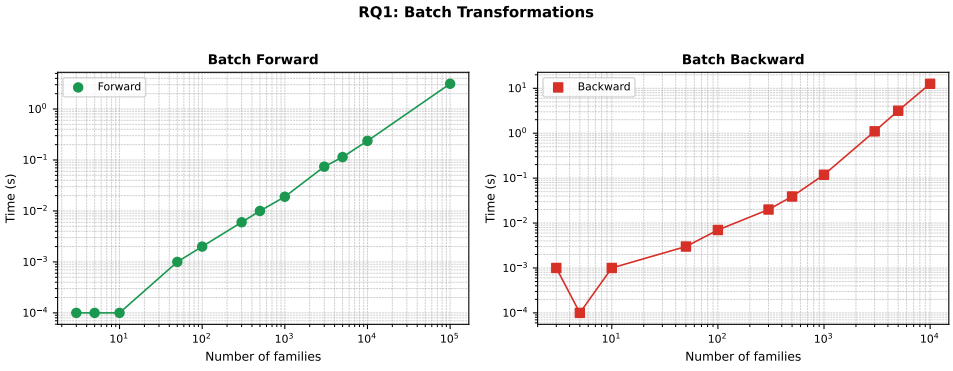


Figure 3: Batch transformation runtimes (forward and backward).

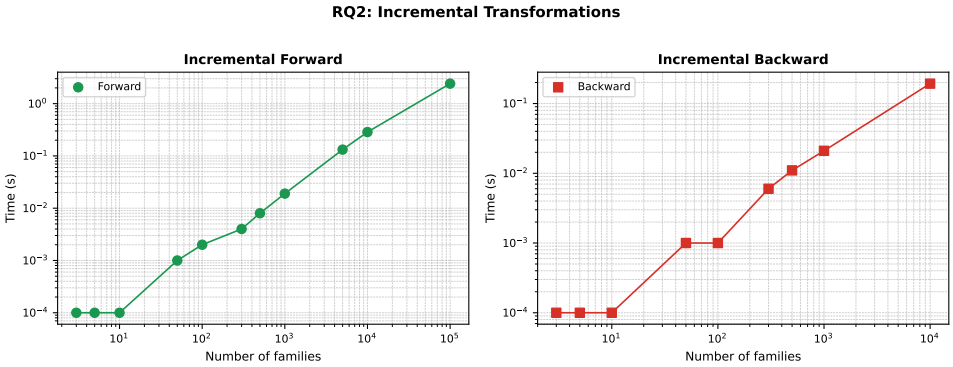


Figure 4: Incremental transformation runtimes (forward and backward).

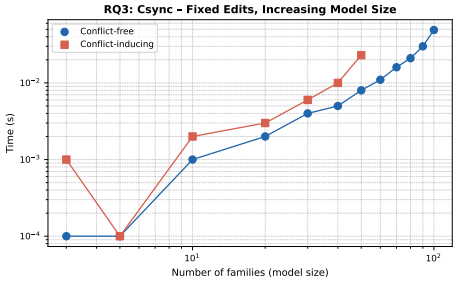


Figure 5: conflict-free vs. conflict-inducing.

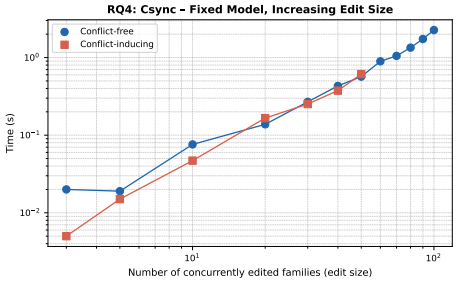


Figure 6: conflict-free vs. conflict-inducing.

5. Discussion

5.1. Failing Cases

All four failing tests share the same root cause: in BXAgent's `sync()` algorithm, attribute propagation (forward and backward) is performed *before* deletion is propagated. This ordering means that a deletion on one side can be undone by the attribute-propagation step, which still sees the to-be-deleted object in the correspondence model and faithfully recreates its counterpart on the other side.

`testCombinedDeletionAndCreation` (NonMonotonic). The precondition contains the Simpson family with Homer as father and Marge as mother. The concurrent edit deletes Homer in the family model and creates Ned Flanders as a new father; simultaneously, both Homer and Marge are deleted in the person register. Under SOURCE_WINS, Homer's deletion is consistent on both sides and requires no action. However, Marge's entry in the person register is deleted while the corresponding `FamilyMember` on the source side is still present. Because attribute propagation runs first and the correspondence entry for Marge still has a non-null source reference, the forward propagation step recreates Marge's `Female` person object before the deletion step has a chance to act. The postcondition therefore finds an unexpected `Female` object for Marge in the person register, and the test fails.

`testCombinedRenameDelete` (NonMonotonic). In this test, Homer is deleted in the family register while Lisa is renamed in the person register. Homer's deletion is straightforward in principle. Lisa's rename, however, is a backward change (target side) that is propagated back to the family model. Because backward attribute propagation fires before deletions are processed, the algorithm first overwrites Lisa's name in the family model from the (already changed) person side — so far correctly — but in the same propagation pass it also touches the correspondence entries for Homer. When the deletion of Homer is finally processed, the state-based detection no longer recognises the change consistently, and the test fails. This case illustrates that state-based (fingerprint) change detection without change recording cannot, in general, distinguish a deletion followed by an unrelated attribute update from a pure attribute update.

`testCombinedCases` (MonotonicDeleting). Rod Flanders is deleted in the family register, while Maggie Simpson is deleted in the person register. Before deletions are propagated, the backward attribute propagation step processes the still-present correspondence entry for Rod: it finds Rod existing on the target side (person register) and, applying `SOURCE_WINS` defaults, attempts to insert Rod into an existing family as a parent role. The subsequent deletion of Rod from the family model then conflicts with this already-applied reconstruction. Symmetrically, Maggie's deletion on the target side is undone by the forward propagation step, which recreates the `Female` person for Maggie before the deletion fires.

`testNonMatchingDeletion` (MonotonicDeleting). Homer is deleted in the family register; Maggie is deleted in the person register. The two deletions are on different objects and are therefore non-conflicting in intent. Nevertheless, the ordering problem manifests identically to the previous case: Homer's target-side object (a `Male` person) is restored by the forward propagation step, and Maggie's source-side object is restored by the backward propagation step, before either deletion is applied.

The common fix would be to perform *deletion detection first* — scanning for null references in the correspondence model — before propagating any attribute changes. This requires a structural reordering of the `sync()` steps and is a planned improvement for a future version of BXAgent.

5.2. Strengths

Low implementation effort. The developer provides two `.ecore` files and roughly one paragraph of English text. BXAgent produces a working bidirectional synchronisation class without any manual coding. This makes the approach accessible to modelling practitioners who are not bx experts.

Flexible conflict resolution. The three `SyncConflictPolicy` modes cover the main practical strategies (source priority, target priority, deferred manual resolution) without requiring the transformation author to implement them.

No change listeners required. Unlike approaches based on EMF adapters or change recording, BXAgent uses content fingerprints and EMF's null-propagation on deletion. The correspondence model can be loaded from disk between tool runs, making the approach well-suited for offline collaborative modelling workflows.

5.3. Limitations and Future Work

Deletion-before-propagation ordering. The most impactful current limitation, responsible for all four failing tests, is that the `sync()` algorithm performs attribute propagation *before* processing deletions. A deletion on one side is therefore temporarily invisible to the propagation step, which recreates the deleted object's counterpart on the other side before the deletion can be applied. Reordering the steps — scanning for null correspondence references first, removing affected entries, and only then propagating attribute changes for the surviving entries — would resolve all four failures.

Fingerprint collisions. If two objects share the same natural key (e.g. two family members named "Homer"), fingerprint matching may fail or produce incorrect correspondences. A more robust approach would use persistent XMI IDs as primary identifiers, making delete-recreate sequences unambiguously distinguishable from genuine new objects.

Backward role switch. When a `Male` person is changed to a `Female` on the person side, the corresponding family member should switch from a `father`/`sons` role to `mother`/`daughters`. The current backward incremental path reprocesses the person but does not perform an explicit object replacement on the family side. This is a known gap that does not affect the current test suite but would be relevant in practice.

LLM dependence. The quality of the generated mapping depends on the LLM's ability to interpret the natural-language description and the metamodels correctly. Complex metamodels or ambiguous descriptions can lead to incorrect Java expressions in `forwardExpression` / `backwardExpression` fields. The compilation validator catches syntactic errors, but semantic errors may not be caught before test execution.

Root elements. The generated transformation expects root `EObjects` to be pre-populated by the caller. This is consistent with the Benchmarx infrastructure but is a constraint for standalone use.
Maintainability of generated transformations. When a metamodel evolves — new attributes added, references renamed, or classes split — the generated Java class becomes stale. The intended maintenance workflow is to re-run the BXAgent pipeline with the updated `.ecore` files. Because the `TransformationSpec` is stored as a JSON artefact alongside the generated code, a developer can also edit the spec by hand and re-invoke the FreeMarker codegen step without triggering a new LLM call. If the natural-language description itself needs to change (e.g. a wrong attribute mapping was specified), the full pipeline must be re-run. In all cases the generated code is replaced wholesale rather than patched, which avoids the regression problem observed in our earlier chat-mode approach [3].

6. Related Work

The Benchmarx framework [6] has been used to compare a wide variety of bx tools on the original F2P case, including eMoflon (graph grammar-based), BXtend (propagation-based), NMF (declarative), and BiGUL (lens-based). Benchmarx 2.0 [7] extends the framework to concurrent synchronisation. All of these tools compile a developer-written *formal specification* — a TGG, a lens expression, or a declarative synchronisation rule — into executable code. BXAgent occupies a different point in the design space: its only user-facing input is natural language; there is no intermediate formal notation that the developer must master.

LLM-based code generation for model transformations is an emerging research direction. The closest antecedent from the same author [3] explored a chat-mode approach in which the F2P transformation was assembled through a sequence of LLM prompts. That approach proved fragile: the output was non-deterministic, previously working fragments regressed as new prompts refined the code, and only batch transformations could be reached; incremental and concurrent synchronisation were out of scope. BXAgent addresses these problems structurally by restricting the LLM to *mapping inference only* (Stage 2) and delegating all algorithmic logic to a fixed, deterministic FreeMarker template. The result is stable, reproducible, and covers the full synchronisation spectrum.

7. Conclusion

BXAgent demonstrates that LLM-driven code generation is a viable approach for bidirectional EMF model transformations, including the challenging case of concurrent synchronisation with conflict resolution. Applied to the TTC 2026 Families-to-Persons benchmark, the tool passes 46 of 50 test cases — including all four genuine conflict scenarios — with no manual post-editing of the generated transformation code. The four failing cases are all non-conflicting concurrent tests whose failures trace back to a single, well-understood algorithmic issue: attribute propagation is currently performed before deletion detection in the `sync()` method, causing deletions to be silently undone. Reordering these steps is a targeted fix that represents the most immediate direction for future improvement.

Declaration on Generative AI. The BXAgent tool itself uses LLMs as part of its pipeline to extract mapping specifications from natural-language descriptions. The manuscript text was prepared by the author with AI-assisted editing for language polishing.

Resources The BXAgent solution is available in a public GitHub Repository, which can be found at: https://github.com/tbuchmann/bxagent-solution-ttc2026

References

[1] J. Cheney, J. Gibbons, J. McKinna, P. Stevens, On principles of least change and least surprise for bidirectional transformations, J. Object Technol. 16 (2017) 3:1–31. URL: https://doi.org/10.5381/jot.2017.16.1.a3. doi:10.5381/JOT.2017.16.1.A3.

[2] T. Buchnmann, BXAgent: LLM-Driven Code Generation for Bidirectional and Incremental EMF Model Transformations, in: Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC, CEUR Workshop Proceedings, CEUR-WS.org, 2026.

[3] T. Buchmann, Prompting bidirectional model transformations - the good, the bad and the ugly, in: M. Wimmer, A. Egyed, B. Combemale, M. Chechik (Eds.), Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, MODELS Companion 2024, Linz, Austria, September 22-27, 2024, ACM, 2024, pp. 550–555. URL: https://doi.org/10.1145/3652620.3687802. doi:10.1145/3652620.3687802.

[4] A. Anjorin, T. Buchmann, The families to persons case — revisited, in: Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC, CEUR Workshop Proceedings, CEUR-WS.org, 2026.

[5] A. Anjorin, T. Buchmann, B. Westfechtel, The families to persons case, in: Proceedings of the 10th Transformation Tool Contest (TTC 2017), volume 2026 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2017, pp. 27–34.

[6] A. Anjorin, T. Buchmann, B. Westfechtel, Z. Diskin, H.-S. Ko, R. Eramo, G. Hinkel, L. Samimi-Dehkordi, A. Zündorf, Benchmarking bidirectional transformations: theory, implementation, application, and assessment, Software and Systems Modeling 19 (2020) 647–691. doi:10.1007/s10270-019-00752-x.

[7] A. Anjorin, T. Buchmann, L. Fritsche, Benchmarx 2.0: A benchmark for concurrent model synchronisation approaches, in: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, 2024, pp. 950–959.