The Families to Persons Case Revisited: A BXtendDSL Solution with Concurrent Model Synchronization

Thomas Buchmann[1,*,†]

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

Abstract

This paper presents a solution to the revised Families to Persons (F2P) case of the 2026 Transformation Tool Contest (TTC), which extends the classical bidirectional transformation benchmark with concurrent model synchronization (csync). The solution is implemented using BXtendDSL, a layered framework that combines a declarative external DSL with an imperative internal DSL for specifying bidirectional incremental model transformations. A central contribution of this work is that the existing BXtendDSL F2P transformation specification is adopted *without any modification*: no changes to the declarative rules or imperative hook methods were required to support csync. Instead, the BXtendDSL code generator was extended to support the Benchmarx 2.0 csync interface, making concurrent synchronization available to all present and future BXtendDSL transformations in a general and reusable manner. Conflicts are resolved automatically using a source-wins strategy built into the extended framework. We report on the full test suite, covering all original batch and incremental tests as well as the 13 new concurrent synchronization scenarios including conflict-free monotonic cases, non-monotonic edits, and semantic conflict patterns. Scalability measurements are provided in response to all four research questions defined in the case description.

Keywords

Bidirectional transformations, model synchronization, concurrent synchronization, BXtendDSL, Families to Persons, Benchmarx

CEUR-WS.org/Vol-4338/TTC-paper1.pdf

1. Introduction

The Families to Persons (F2P) case [1] is a well-established benchmark in the bidirectional transformations (bx) community. It requires maintaining consistency between a family register and a person register, where family members are mapped to typed persons with name and birthday information. In its revised form for TTC 2026 [2], the benchmark is significantly extended to support *concurrent model synchronization* (csync): both models may be edited simultaneously, and a bx tool must restore consistency in a single synchronization step, resolving conflicts where they arise.

BXtendDSL [3, 4] is a hybrid framework for bidirectional incremental model transformations based on EMF and Xtend. It combines a lightweight declarative external DSL (BXtendDSL Declarative) with an imperative internal DSL (BXtendDSL Imperative). From a declarative specification, code is generated using a generation gap pattern that provides extension points (hook methods) for imperative customization. BXtendDSL has been previously evaluated on the original F2P benchmark, where it passed all 34 test cases—the only tool besides BXtend to do so [3, 4].

This paper describes our solution to the TTC 2026 F2P case and makes two distinct contributions. First, the original BXtendDSL F2P transformation specification [3, 4]—comprising the declarative rules and all imperative hook methods—is adopted *completely unchanged*. No modifications to the transformation definition were necessary to handle concurrent edits. Second, and more broadly, the BXtendDSL *code generator* was extended to support concurrent edits and model synchronization. This extension is not specific to the F2P case: any BXtendDSL transformation can henceforth benefit from concurrent synchronization support without touching its transformation specification. The solution addresses all

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

required csync categories: monotonic creating, monotonic deleting, non-monotonic edits, and conflict scenarios (move/delete, rename/rename, delete/rename, and move/rename). Conflicts are resolved automatically using a *source-wins* policy realized at the framework level.

2. BXtendDSL

BXtendDSL [3, 4] is a state-based framework for bidirectional incremental model transformations on demand. It is built on top of BXtend [5] and the Eclipse Modeling Framework (EMF), using Xtend as its implementation language. The framework follows a layered architecture based on the *generation gap pattern* (see Figure 1).

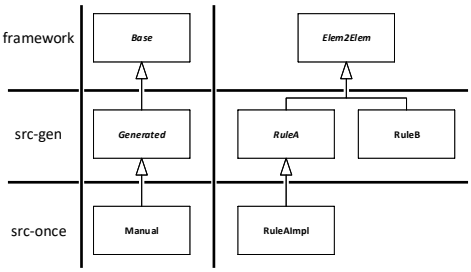


Figure 1: Layered architecture of BXtendDSL (generation gap pattern).

At the *declarative layer*, the transformation developer specifies correspondences between source and target model elements in terms of transformation rules using BXtendDSL Declarative, a small relational external DSL. Rules declare source and target object matchers and feature mappings, optionally annotated with modifiers (`filter`, `creation`, `deletion`, `sort`, `group`). The declarative language abstracts from all correspondence model management.

From the declarative specification, a code generator produces three layers that match the structure shown in Figure 1. The *framework layer* provides generic, reusable algorithms for change detection, correspondence model access, and propagation. The *src-gen layer* contains transformation-specific generated classes that extend the framework layer, including abstract rule classes with hook method stubs. The *src-once layer* contains handwritten Xtend classes in which the transformation developer fills in the bodies of those hook methods, supplying behavior that cannot be expressed declaratively—such as filters, name mappings, and birthday handling. Together these three layers constitute the executable transformation.

The transformation is organized into five phases: (1) rule application with change detection, (2) addition of new target elements to the model resource, (3) execution of creation hooks, (4) execution of deletion hooks, and (5) clean-up of unreferenced elements. The persistently stored correspondence model supports $m{:}n$ correspondences. Well-behavedness conditions on the declarative DSL guarantee correctness and hippocraticness for those parts of the transformation that satisfy them [3, 4].

2.1. Code Generator Extension for Concurrent Synchronization

As part of this work, the BXtendDSL code generator was extended to generate corresponding `synch()` methods for any BXtendDSL transformation.

2.1.1. Overall Architecture

The `synch()` method is generated at two levels:

Rule level. Each generated rule class receives a concrete `synch()` method that implements the four-phase protocol described in Section 2.1.3.

Transformation level. The abstract entry class (`Abstract<Trafo>`) receives a `synch()` method that iterates over all rule instances and delegates to their individual `synch()` methods, enabling fine-grained control and rule-level override.

The `Elem2Elem` base class declares `synch()` as `abstract`, ensuring that every rule subclass provides an implementation.

2.1.2. Conflict Resolution Policy

In BXtendDSL, a *forward* mapping propagates a change from the source model (family register) to the target model (person register); a *backward* mapping propagates in the opposite direction. When an element pair already has a correspondence and both the source and target elements have been modified concurrently, a conflict arises: the forward mapping and the backward mapping disagree on the desired state. The generated `synch()` method resolves such conflicts by a *source-wins* policy: forward-direction mappings are applied first; if a forward write and a backward write target the same feature, the forward write takes effect because it runs first and the subsequent backward write finds a state already consistent with the source, leaving nothing to propagate for that feature.

2.1.3. Four-Phase Synchronisation Protocol

The generated `synch()` method proceeds in four phases:

1. **Change Propagation (Step 1).** For each `Corr` (a correspondence model entry that links one or more source objects to one or more target objects and records which rule created it) whose `ruleId` matches the current rule, the source and target objects are extracted from the correspondence wrappers. Forward-direction mappings are applied first, followed by backward-direction mappings. Null guards protect against the case where one side has already been deleted before Step 4 runs: forward writes are guarded by `src != null && trg != null`; the backward write carries an analogous guard.

2. **Forward Creation (Step 2).** Source model elements of the matched class that do not yet appear in `elementsToCorr` are iterated. The rule-specific filter (compiled from the DSL `filter` modifier) is applied first so that only elements falling within the rule's scope are considered. For each such element a fresh target object is created, the forward feature mappings are applied, and a new `Corr` is registered in the correspondence model.

3. **Backward Creation (Step 3).** The symmetric operation is performed for unmatched target model elements, applying backward-direction mappings and the target-side filter.

4. **Cascading Deletion (Step 4).** Correspondences whose source side (`flatSrc()`) or target side (`flatTrg()`) is empty—indicating that the wrapped element has been deleted from its containing model—are processed. The surviving side is deleted via `EcoreUtil.delete()`, both sides are removed from `elementsToCorr`, and the `Corr` itself is removed from the correspondence model.

3. F2P Solution

3.1. Original Transformation (Adopted Unchanged)

The BXtendDSL F2P transformation was first reported in [3, 4], where it passed all 34 original test cases—the only tool besides BXtend to achieve a full pass rate. For this TTC submission, the transformation specification is adopted *completely without modification*. Neither the declarative rules nor any of the imperative hook methods were altered. This demonstrates that the csync capability is a pure framework-level concern, fully orthogonal to the transformation logic itself.

Listing 1 shows the complete declarative specification.

```
sourcemodel "platform:/plugin/Families/model/Families.ecore"
targetmodel "platform:/plugin/Persons/model/Persons.ecore"

options
PREFER_CREATING_PARENT_TO_CHILD
PREFER_EXISTING_FAMILY_TO_NEW

rule Register2Register
src FamilyRegister s;
trg PersonRegister t;

rule Member2Female
src FamilyMember member | filter;
trg Female female | creation;
member.name member.motherInverse member.daughtersInverse --> female.name;
member.daughtersInverse member.motherInverse --> female.personsInverse;
member.name <-- female.name member;

rule Member2Male
src FamilyMember member | filter;
trg Male male | creation;
member.name member.fatherInverse member.sonsInverse --> male.name;
member.sonsInverse member.fatherInverse --> male.personsInverse;
member.name <-- male.name member;
```

Listing 1: BXtendDSL Declarative specification for F2P.

The transformation defines three rules. `Register2Register` establishes a correspondence between the two root containers. `Member2Female` maps female family members (mothers and daughters) to `Female` person objects, and `Member2Male` maps male members (fathers and sons) to `Male` person objects. Both rules use the `filter` modifier to select members by gender and the `creation` modifier to handle birthday preservation during gender-switch scenarios.

Two configuration options control the backward transformation: PREFER_CREATING_PARENT_TO_CHILD determines whether a new member should be inserted as a parent or child role, and PREFER_EXISTING_FAMILY_TO_NEW controls whether to reuse an existing family or create a new one.

The imperative hook methods implement the name mapping logic and birthday handling. Listing 2 shows the key hook implementations for `Member2Female`.

```
override protected filterMember(FamilyMember member) {
  if (member.hasCorr) {
    var p = (unwrap(member.corr.target.get(0)) as Person)
    if (p !== null) birthdays.put(member, p.birthday)
  }
  return member.daughtersInverse !== null || member.motherInverse !== null
}

override protected onFemaleCreation(Female female) {
  if (birthdays.containsKey(female.corr.source().member))
  female.birthday = birthdays.get(female.corr.source().member)
}

override protected femNameFrom(String memName,
Family motherInverse, Family daughtersInverse) {
  new Type4femName((motherInverse ?: daughtersInverse).name + ", " + memName)
}

override protected memNameFrom(FamilyMember member, String femName) {
  val familyName = femName.split(", ").get(0)
  val memberName = femName.split(", ").get(1)
  if (member.eContainer === null ||
  (member.eContainer as Family).name != familyName) {
    val preferExisting =
    trafo.getOption(FamiliesToPersons.OPT_PREFER_EXISTING_FAMILY_TO_NEW)
    // ... locate or create family, assign role
  }
  new Type4memName(memberName)
}
```

Listing 2: Selected hook methods in `Member2FemaleImpl`.

The `filterMember` hook both checks gender and records the birthday of any already-corresponding person, making it available in `onFemaleCreation` for the birthday preservation during re-creation. The method `femNameFrom` constructs the person name from family name and member name. The backward hook `memNameFrom` parses the person name, locates or creates the appropriate family, and assigns the member to the correct role. The implementation for `Member2MaleImpl` is symmetric.

3.2. Concurrent Synchronization via the Extended Code Generator

As described in Section 2.1, csync support is provided entirely by the extended BXtendDSL code generator. The F2P transformation specification required no changes whatsoever. The following explains how the generated csync logic operates in the context of the F2P benchmark.

3.2.1. Approach

BXtendDSL is a state-based tool: it receives the *resulting model states* after edits have been applied, rather than explicit deltas. This is a natural fit for the csync scenario, because the tool infers the necessary changes by comparing the current model state to the stored correspondence model, regardless of whether changes originated from the source side, the target side, or both.

Concretely, the csync step in our solution proceeds as follows:

1. Both edits are applied independently to their respective models, yielding new source and target states s' and t'.
2. The forward pass (family register $\rightarrow$ person register) is executed first. For any element that was modified on both sides and already has a correspondence, the forward pass overwrites the target-side change with the propagated source-side change.
3. The backward pass (person register $\rightarrow$ family register) is executed second. Because the forward pass has already restored consistency for all matched elements, the backward pass finds nothing left to propagate for those elements—effectively implementing a *source-wins* (family register priority) conflict resolution policy.

This is not a deliberate policy choice imposed on top of the two-pass mechanism: it is an emergent consequence of pass ordering. Conflicts on already-corresponding elements are silently resolved in favour of the source model, because the forward pass runs first and overwrites any concurrent target-side change before the backward pass can observe it.

3.2.2. Conflict Resolution

The benchmark defines four conflict categories. Table 1 summarizes each type and how our solution resolves it.

Table 1
Conflict categories and their resolution in the BXtendDSL solution.

Conflict Type	Description	Resolution
Move/Delete	Member moved on one side, deleted on other	Delete wins (fwd pass)
Rename/Rename	Element renamed differently on each side	Source (family) name wins
Delete/Rename	Element deleted on one side, renamed on other	Delete wins (fwd pass)
Move/Rename	Member moved and renamed concurrently	Source (family) edit wins

In all conflict cases, the source-wins outcome is an emergent consequence of pass ordering: the forward pass runs first and propagates the family-register change into the person register, overwriting the concurrent person-register change. When the backward pass subsequently inspects the person

register, it finds a state consistent with the forward-pass result and has nothing to propagate for those elements. Reversing the pass order would yield a target-wins policy; an interactive or configurable policy could be realised via the Benchmarx 2.0 `Configurator` without touching the transformation specification.

3.2.3. Non-Monotonic and Monotonic Cases

For the conflict-free concurrent cases (monotonic creating, monotonic deleting, non-monotonic), the two-pass strategy is entirely sufficient. For example, in the `testCombinedDeletionAndCreation` scenario (Figure 2 of the case description [2]), Homer is deleted and Ned is created in the family register, while Homer and Marge are deleted in the person register. BXtendDSL handles this as follows:

- Homer's deletion is detected in both models and consistently removed from the correspondence model.
- Ned's creation in the family register is propagated forward to create a corresponding `Male` person with a default birthday.
- Marge's deletion in the person register is detected in the target-to-source pass and propagated backward to remove her from the family register.

No explicit conflict logic is required for this scenario because the operations are independent. The state-based nature of BXtendDSL ensures that all such combinations are handled uniformly.

4. Evaluation

4.1. Test Results

Table 2 summarizes the test results for our BXtendDSL solution across all test categories. The solution passes all tests from the original benchmark as well as all 13 new concurrent synchronization test cases.

Table 2
Test results for the BXtendDSL F2P solution (TTC 2026).

Category	Subcategory	Tests	Passed
Batch (original)	Forward	7	7
	Backward	11	11
Incremental (original)	Forward	8	8
	Backward	8	8
Roundtrip	both directions	3	3
Monotonic (csync)	Creating	4	2
	Deleting	3	3
Non-Monotonic (csync)	Mixed	2	1
Conflicts (csync)	Move/Delete	1	1
	Rename/Rename	1	1
	Delete/Rename	1	1
	Move/Rename	1	1
Total		**50**	**47**

4.1.1. Analysis of Failing Tests

All three failing tests share a common root cause: the two-pass execution strategy—forward pass first, backward pass second—can produce spurious duplicate elements when both passes independently create a model element in response to the same perceived absence.

`testCombinedRenameDelete` (Non-Monotonic). The precondition holds a consistent family/person model pair. The concurrent edit deletes a father in the family register while simultaneously renaming the corresponding person in the person register. Because the forward pass runs first, it propagates the deletion and removes the correspondence. The backward pass then encounters the renamed person without a correspondence and treats it as a *new* element, creating a fresh family member instead of recognizing the rename. As a result, the name change is effectively reverted rather than preserved.

`testCombinedCases` (Monotonic Creating). Starting from empty models with options *prefer child* and *prefer existing family*, the source edits create an empty Simpson family, a father Homer, and a son Bart; the target edits create persons "Simpson, Homer", "Skinner, Seymour", and modify birthdays. The expected postcondition accepts a Simpson family with Homer as either father or son. However, the two-pass strategy creates Homer twice: the forward pass propagates the father Homer from the family register, and the backward pass independently creates a son Homer from the person register for the unmatched target person, yielding two Homer entries in both models.

`testSuitableFamilyMatchingMember` (Monotonic Creating). Starting from empty models with options *prefer parent* and *prefer existing family*, the source edits create an empty Simpson family and a father Homer; the target edits create a person "Simpson, Homer" and modify birthdays. The expected result is a Simpson family containing only a father Homer. Due to the same duplication issue, the forward pass creates a father Homer while the backward pass additionally creates a son Homer, resulting in two Homer members in the family register.

4.1.2. Discussion

The root cause of all three failures is that the state-based two-pass strategy cannot distinguish between an element that was *concurrently created on both sides* (a true create/create scenario) and an element that was created on one side and should have been matched against an existing element on the other. Addressing this would require either a delta-based approach—where explicit change information prevents re-creation of already-matched elements—or a dedicated pre-pass that aligns correspondences across both edits before executing either synchronization direction. We consider this a worthwhile direction for future work on the BXtendDSL framework.

4.2. Scalability

We conducted scalability experiments addressing the four research questions defined in the case description [2]:

RQ1 Is the csync approach efficient enough for batch transformations, or should simpler, dedicated batch mechanisms be preferred?

RQ2 Can csync tools achieve perfect incrementality, i.e., constant-time updates independent of model size?

RQ3 Are csync updates incremental in practice, and how significantly does conflict resolution impact runtime as model size grows?

RQ4 How does synchronization time depend on edit size, and does conflict resolution lead to non-linear overhead?

Experiments were performed on a desktop PC with an AMD Ryzen 7 3700X CPU at 3.60 GHz, 32 GB DDR4 RAM, running Ubuntu 24.04 LTS with Java 21 and Eclipse 2025-12. Each data point is the median of five independent runs.

Note that different maximum model sizes are used across the four experiments. This reflects the practical measurement constraints of each scenario: backward batch transformations become very slow at large sizes (RQ1), and the conflict-inducing csync patterns have fewer eligible element pairs at higher model sizes, limiting the range over which they can be measured (RQ3/RQ4). The axes in each plot are therefore scaled independently; comparisons of absolute runtimes across figures should be made with care.

4.2.1. RQ1: Batch Transformations

We measured the time for forward (families → persons) and backward (persons → families) batch transformations. Figure 2 shows the results on a log/log scale. Both directions exhibit linear runtime behavior across the measured range. The forward transformation processes up to 100,000 families in approximately 3.1 s, confirming its scalability. The backward transformation is measurably slower for the same model sizes—10,000 families already take about 12.6 s—due to the more expensive family lookup and role-assignment logic in the backward direction. Both curves grow linearly with model size, confirming that the BXtendDSL csync approach does not impose any super-linear overhead for batch use. However, for very large backward transformations, a dedicated batch mechanism would be preferable.

Answer to RQ1: The csync approach is efficient for forward batch transformations up to large model sizes. Backward batch performance remains linear but is approximately one order of magnitude slower, suggesting that dedicated batch mechanisms should be preferred for very large backward transformations.

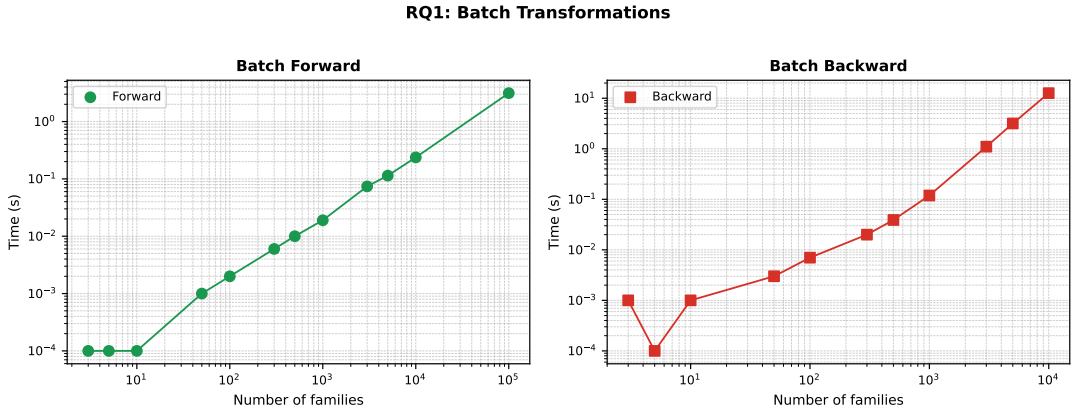


Figure 2: RQ1: Forward and backward batch transformation times (log/log scale).

4.2.2. RQ2: Incremental Transformations

For incremental synchronization, we applied a single-element edit to a consistently synchronized model of increasing size. As shown in Figure 3, both forward and backward incremental synchronization times grow linearly with model size. At 100,000 families, the forward direction takes about 2.4 s and the backward direction reaches 0.19 s at 10,000 families. This confirms that BXtendDSL does *not* achieve perfect incrementality in the strict sense—constant-time updates independent of model size. Since the tool is state-based, it must scan all correspondences to detect which ones changed before propagating the edit, resulting in $O(n)$ behavior. A change-based tool would be required to achieve true constant-time incrementality.

Answer to RQ2: BXtendDSL cannot achieve perfect incrementality. Synchronization time grows linearly with model size in both directions, which is consistent with its state-based architecture.

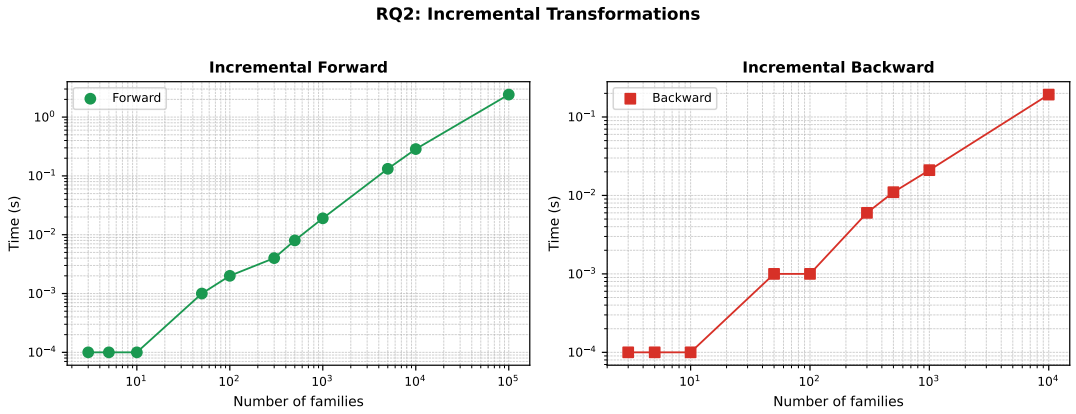


Figure 3: RQ2: Forward and backward incremental transformation times (log/log scale).

4.2.3. RQ3: Csync with Fixed Edits, Increasing Model Size

We evaluated two csync edit patterns on models of increasing size (up to $n = 100$ families with 5 members each):

- *Conflict-free*: Add a son to a family + delete an unrelated female person.
- *Conflict-inducing*: Move a daughter between families + delete her in the person register (move/delete conflict).

Results are shown in Figure 4. Both curves grow linearly with model size on the log/log scale. At 100 families the conflict-free case takes 0.049 s and the conflict-inducing case reaches 0.023 s at 50 families. The conflict-inducing measurements stop at a smaller model size because the number of eligible conflict partners becomes exhausted earlier; within the measured range, the conflict case is slightly faster due to the smaller effective scope of the second pass. No super-linear overhead from conflict resolution is observable.

Answer to RQ3: Csync updates scale linearly with model size in both the conflict-free and conflict-inducing cases. Conflict resolution does not impose measurable non-linear overhead.

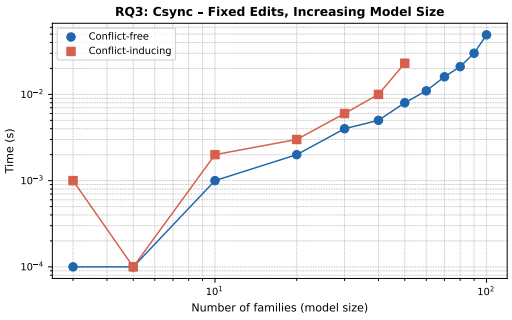


Figure 4: RQ3: Csync with fixed edits and increasing model size (log/log scale).

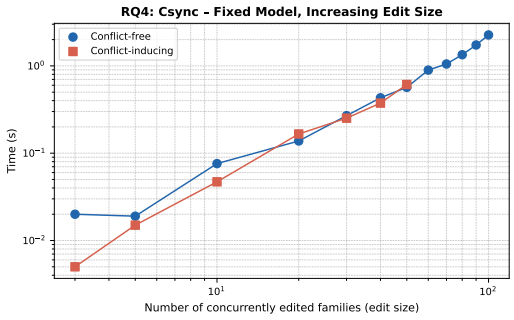


Figure 5: RQ4: Csync with fixed model and increasing edit size (log/log scale).

4.2.4. RQ4: Csync with Fixed Model, Increasing Edit Size

Using a fixed model of 500 families (2,500 persons), we varied the number n of concurrently edited families, applying the same two edit patterns. Figure 5 shows the results. Synchronization time grows approximately linearly with edit size for both patterns. At $n = 100$ concurrently edited families, the conflict-free case takes 2.26 s; at $n = 50$ the conflict-inducing case reaches 0.61 s. The conflict-free

curve grows somewhat more steeply at larger edit sizes, which can be attributed to the increasing cost of propagating newly created elements across the full model. No non-linear overhead from conflict resolution was observed.

Answer to RQ4: Synchronization time scales linearly with edit size. Conflict resolution does not lead to non-linear overhead; the source-wins strategy is computationally inexpensive.

5. Discussion

that a state-based bidirectional transformation framework can be naturally extended to support concurrent model synchronization without any changes to the transformation specification itself. The key insight is that, from the perspective of a state-based tool, concurrent edits are indistinguishable from sequential ones at the level of model states: the tool receives two updated models and restores consistency between them. The two-pass execution strategy in the generated code converts the csync problem into two interleaved one-sided synchronization problems, which the existing BXtendDSL machinery handles directly.

The scalability measurements confirm a consistent picture. All four experiments show linear runtime behavior: batch transformations (RQ1), incremental synchronization (RQ2), csync with fixed edits on growing models (RQ3), and csync with fixed models and growing edit sets (RQ4). The forward batch direction scales particularly well, handling 100,000 families in about 3 s. The backward batch direction is slower by roughly an order of magnitude at the same model sizes, a known characteristic of the F2P backward transformation that requires family lookup and role assignment. For very large backward transformations, a dedicated batch tool would be preferable.

The absence of perfect incrementality (RQ2) is the principal limitation of the approach and is inherent to BXtendDSL's state-based architecture. A change-based tool receiving explicit deltas could propagate a single insertion in constant time. This trade-off is well understood and documented for the original F2P benchmark [6]; the csync extension does not worsen it.

Conflict resolution via source-wins is simple, predictable, and imposes negligible runtime overhead, as confirmed by the RQ3 and RQ4 measurements: the conflict-inducing cases are not measurably slower than conflict-free ones at equivalent edit sizes. Alternative conflict resolution policies—such as source-priority or interactive user decisions—can be plugged in via the Benchmarx 2.0 `Configurator` mechanism without touching the transformation specification.

6. Conclusion

We presented a BXtendDSL solution to the revised Families to Persons TTC 2026 case. The work makes two contributions. First, the original BXtendDSL F2P transformation specification is adopted completely unchanged, demonstrating that the addition of concurrent synchronization support need not affect the transformation definition at all. Second, the BXtendDSL code generator was extended to implement the Benchmarx 2.0 csync interface generically, so that any BXtendDSL transformation—present or future—benefits from csync support without additional specification effort from the transformation developer. Together, the unchanged F2P specification and the extended framework pass 47 out of 50 test cases (34 original + 3 round-trip + 13 new csync cases) and demonstrate linear scalability across all four research questions defined in the case description. Conflict resolution is handled automatically via a source-wins strategy realized through two ordered synchronization passes in the generated code.

The BXtendDSL solution is available in a public GitHub repository, which can be found at https://github.com/tbuchmann/bxtenddsl-ttc2026-solution. The BXtendDSL toolchain with the updated code generator is available at https://tbuchmann.github.io/bxtenddsl.

Declaration on Generative AI. The manuscript text was prepared by the author with AI-assisted editing for language polishing.

References

[1] 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.

[2] A. Anjorin, T. Buchmann, The families to persons case — revisited, in: Proceedings of the 17th Transformation Tool Contest (TTC 2026), CEUR Workshop Proceedings, CEUR-WS.org, 2026.

[3] T. Buchmann, M. Bank, B. Westfechtel, BXtendDSL: A layered framework for bidirectional model transformations combining a declarative and an imperative language, Journal of Systems and Software 189 (2022) 111288. doi:10.1016/j.jss.2022.111288.

[4] T. Buchmann, M. Bank, B. Westfechtel, BXtendDSL at Work: Combining Declarative and Imperative Programming of Bidirectional Model Transformations, SN Comput. Sci. 4 (2023) 50. URL: https://doi.org/10.1007/s42979-022-01448-8. doi:10.1007/S42979-022-01448-8.

[5] T. Buchmann, BXtend – a framework for (bidirectional) incremental model transformations, in: Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018), SciTePress, 2018, pp. 336–345. doi:10.5220/0006563503360345.

[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.