The BCerT Solution for Bidirectional Model-Driven Transformation

Akram Idani

Univ. Grenoble Alpes, Grenoble INP, CNRS, VERIMAG.
F-38000 Grenoble France
Akram.Idani@univ-grenoble-alpes.fr

Abstract

The TTC'2026 Families to Persons case study focuses on bidirectional transformations with an emphasis on concurrent model synchronization, where simultaneous edits may occur on both source and target models. In this context, ensuring consistency while handling potentially conflicting updates is a key challenge. We present the BCerT solution providing a formal state-based approach to model transformation using the B method. Transformation rules are specified as invariants and preconditioned operations that are proved correct via theorem proving. Regarding execution, our approach follows a rule-based propagation strategy, where synchronization is achieved by leveraging the animation and constraint solving capabilities of the ProB model-checker. Our solution supports both forward and backward transformations as classical transformation cases, while extending to concurrent synchronization. Correctness is ensured by construction: since the transformation is proven, it obviously preserves both structural and semantic invariants. Testing is therefore not required for verification purposes, but is used in our approach for validation. To this end, we combine B with CSP, where CSP acts as an orchestrator of B operations, enabling systematic validation against the TTC benchmark test suite.

Artifacts and tutorials are available at:
`https://bcert-meeduse.github.io/ttc2026.html`

1 Introduction

In previous work [10], we participated in the TTC'2019 challenge [3] using Meeduse [5, 6, 7], a language workbench that we developed for defining and executing domain-specific languages (DSLs) based on the B method [1]. At that time, Meeduse was still a proof-of-concept framework and did not yet natively support model transformations. Since it was primarily designed to define execution semantics of DSLs rather than model-to-model transformations, the transformation problem was reformulated in terms of DSL execution semantics. Concretely, this required the development of dedicated drivers that encode transformations as consumption/production processes over a unified representation of the source and target domains. Our contribution received the Best Verification Award and the Third Audience Award, which was both encouraging and strongly motivated us to further develop Meeduse.

Since then, the framework has significantly evolved, gaining in maturity and being applied to several realistic applications from the safety-critical domain. More recently, we have developed BCerT [8], an extension of

Meeduse dedicated to model-to-model transformation and grammar-to-grammar transpilation. This extension provides a seamless integration between EMF and the ProB model-checker [11], allowing transformation rules specified in B to be executed directly on models without requiring additional implementation effort. This evolution has been further consolidated in a paper accepted at SLE 2026 [9], where the Families to Persons (F2P) transformation is used as a pedagogical example to illustrate how the B method can support the engineering of verified model transformations.

The TTC'2026 F2P case study provides an opportunity to evaluate our approach beyond the pedagogical setting presented in [9]. In addition to classical forward and backward transformations, the benchmark introduces concurrent synchronization scenarios, where edits may occur independently on both models and must be reconciled in a consistent way. The benchmark is structured as a set of test suites specifying expected behaviors through pre- and post-conditions. In our approach, transformation rules are defined in B and proved correct, ensuring the preservation of structural and semantic invariants. This guarantees that all reachable states satisfy the consistency constraints of the model. However, correctness with respect to invariants is not sufficient to meet the expectations of the benchmark, which are expressed in terms of observable behaviors. For this reason, validation is required to check that the execution of the transformation produces the expected results for each test case. To address this, we combine B with CSP[1] [4]. Transformation rules are expressed as B operations constrained by invariants, while CSP is used to orchestrate their execution. This allows us to systematically execute the benchmark test suites and verify pre- and post-conditions for each scenario.

The rest of the paper is organized as follows. Section 2 briefly introduces Meeduse and the BCerT extension. Section 3 presents the proposed solution to the TTC'2026 F2P case, including the involved meta-models, transformation rules, propagation strategies, and CSP-based test suites. Section 4 reports execution results and implementation metrics. Finally, Section 5 concludes the paper and discusses the scalability issue.

2 Brief introduction of Meeduse and BCerT

2.1 Models, meta-models, and semantics in Meeduse

The approach of Meeduse is a compiled approach; it translates meta-models and models built in the Eclipse Modeling Framework (EMF) into the technological space of the B method [1]. The overall approach of the tool is illustrated in Figure 1. From an ECore meta-model (or an Xtext Grammar), Meeduse generates a functional B machine (referred to as *Static Semantics*) that captures its structural aspects. The operational semantics of the language can then be defined in the same machine or as a separate one (referred to as *Dynamic Semantics*) that includes the functional one. This semantic layer allows reasoning about the correctness of the DSL using theorem proving tools such as Atelier B.

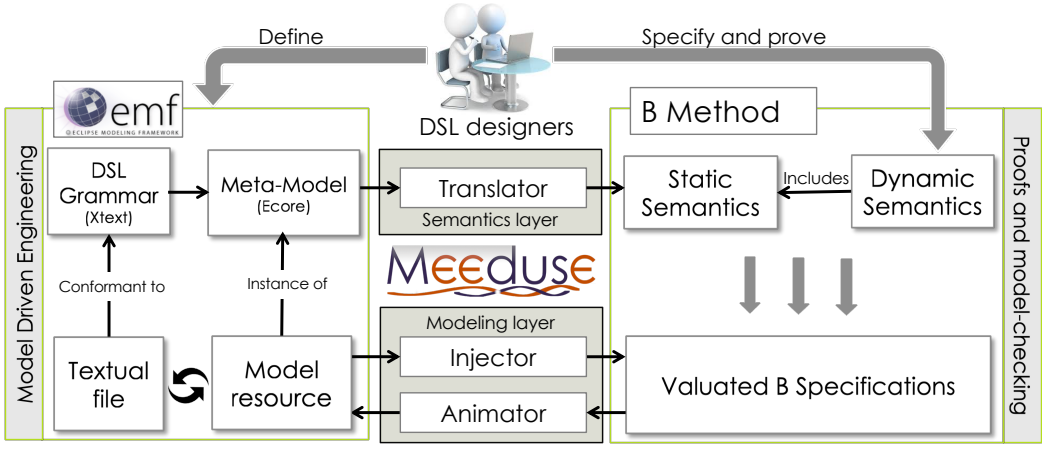

Figure 1: Meeduse architecture (taken from [5])

To support model execution, Meeduse embeds ProB [11], an animator and model checker of the B method; and uses it transparently from the user's perspective. Given a model, Meeduse produces a data refinement of the functional B machine. This results in a valued B specification that is semantically equivalent to the input model. Figure 2 illustrates this process: (1) the structural part of machine *MeeduseTuto* is generated from a

[1] Communicating Sequential Processes

meta-model containing a class `Counter` with an integer attribute `value`; and (2) the refinement *MeeduseTuto_r* is generated from a model containing one instance of this class, where `value` is equal to 0. The operations `inc` and `dec` define the dynamic semantics of the meta-model incrementing and decrementing attribute value, within the bounds specified by the invariant.

MACHINE *MeeduseTuto*
SETS *COUNTER*
CONSTANTS *Counter*
PROPERTIES $Counter \subseteq COUNTER$
VARIABLES *value*
INVARIANT
$value \in Counter \rightarrow \mathbf{INT} \wedge \mathbf{ran}(value) \subseteq -2 \,..\, 2$
INITIALISATION
$value := Counter \times \{0\}$
OPERATIONS
inc(cc) =
PRE $cc \in Counter \wedge value(cc) < 2$
THEN **value**(cc) := $value(cc) + 1$
END;
dec(cc) =
PRE $cc \in Counter \wedge value(cc) > -2$
THEN **value**(cc) := $value(cc) - 1$
END
END

REFINEMENT *MeeduseTuto_r*
REFINES *MeeduseTuto*
CONSTANTS *COUNTER1*
PROPERTIES
$COUNTER1 \in COUNTER$
$\wedge$ **card**($COUNTER$) = 1
$\wedge$ $Counter = \{COUNTER1\}$
INITIALISATION
$value := Counter \times \{0\}$

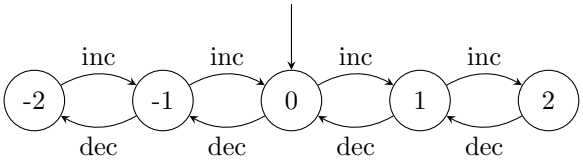

Figure 2: Models and meta-models in Meeduse

From the valued specification (right hand-side of Figure 2), ProB computes the enabled operations in the current state and produces an execution trace corresponding to all possible applications of these operations. At each step, Meeduse observes the state transition computed by ProB and propagates the corresponding updates back to the EMF model.

2.2 Model transformation: the BCerT extension

Building on the execution capabilities of Meeduse, the BCerT extension provides support for model-driven transformation and grammar transpilation. The key idea is that, once an EMF model can be executed through ProB, model transformation can be expressed within the same formal framework by taking into account multiple meta-models simultaneously. As shown in Figure 3, given a source meta-model $\mathrm{MM_{Source}}$ and a target meta-model $\mathrm{MM_{Target}}$, a pivot meta-model $\mathrm{MM_{pivot}}$ is introduced to reference both. The corresponding model $\mathrm{M_{pivot}}$ acts as an integration layer that links instances of the source model $\mathrm{M_{Source}}$ and the target model $\mathrm{M_{Target}}$.

The B specification derived from $\mathrm{MM_{pivot}}$ therefore includes all the structural elements required to represent both domains, as well as additional elements dedicated to managing correspondences and transformation traces. Transformation rules are specified as B operations that define how elements of the source model are consumed and how elements of the target model are produced or updated. On the B side, since all concepts are embedded within a single state space, the transformation is expressed as an execution process over a unified formal model.

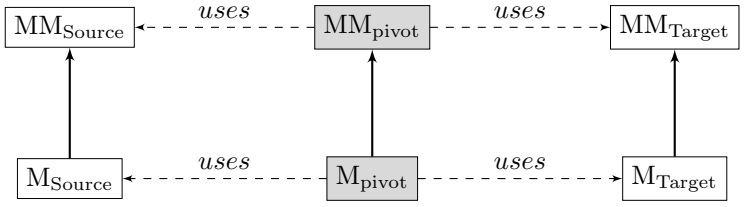

Figure 3: Architecture d'une transformation de modèles dans Meeduse

A well-known issue in rule-based transformations is how to control the order in which transformation rules are applied. In interactive mode, this choice is left to the user. For example, in state 0 of Figure 2, both operations

`inc` and `dec` are enabled, and the user can choose which one to execute. In automatic mode, this choice is made non-deterministically by the tool, typically at random, which is not suitable in a testing context where specific execution scenarios must be reproduced and validated.

This is precisely where the CSP‖B [2] approach becomes relevant. CSP [4] provides high-level operators such as sequencing, choice, and parallel composition to describe the control flow of systems. In the CSP‖B combination, the global behavior is defined as the parallel composition of a B machine and a CSP process. As a consequence, an operation can be executed only when it is both enabled by the B machine and permitted by the CSP controller. The process shown in Figure 4 enforces an initial choice between incrementing and decrementing, and then restricts the execution to the corresponding direction. In CSP, the operator [] denotes an external choice between alternative behaviors, while the prefix operator $a \rightarrow P$ specifies that event a must occur before the process continues as P. The resulting behavior is illustrated on the right-hand side of the figure, where the state space of the counter is reduced to a monotonic evolution depending on the initial choice.

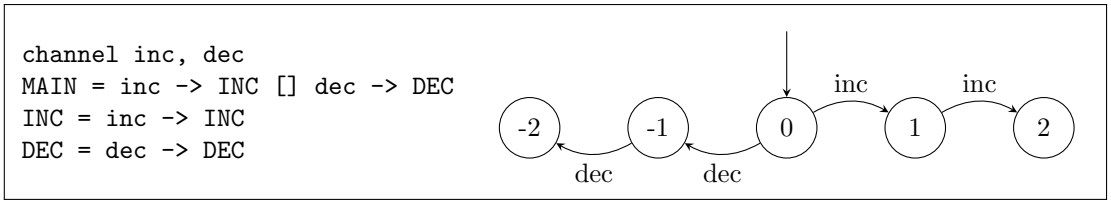

Figure 4: CSP control of the execution flow of the MeduseTuto machine

Recent versions of the ProB Java API provide support for the CSP‖B combination. This feature is exploited in the BCerT extension to allow the use of CSP for orchestrating the execution of transformation rules.

3 The BCerT solution to the TTC'2026 F2P case

Detailed B and CSP specifications are available in the online artefacts[2] and can be consulted for a complete view of the formal models (look inside folder `pivot/model/` files `*.mch` and `*.csp`). This paper does not aim at providing a full formalisation of the solution; instead, it focuses on giving an overview of how our approach addresses the main challenges of the benchmark.

3.1 Architecture

The TTC'2026 F2P case defines several transformation objectives, including batch forward, batch backward, incremental forward, incremental backward, concurrent synchronization, and roundtrip consistency. Only, the scalability aspect is not addressed in the current version of our solution and is left for future work. In our proposal, each objective is addressed following a common architectural pattern, structured into three layers, as illustrated in Figure 5:

- `MetaModel.mch` defines the core transformation rules together with basic model manipulation operations. It captures both the structural and behavioral aspects of the transformation, including invariants and operations that are formally proved correct.
- A family of machines `T_objective.mch` is introduced, each corresponding to a specific TTC objective. These machines define the propagation strategies that govern how transformation rules are applied in a given context.
- For each test suite of the benchmark, a CSP specification orchestrates the execution of the B operations defined in `T_objective.mch` by explicitly encoding the required execution scenarios and control flow.

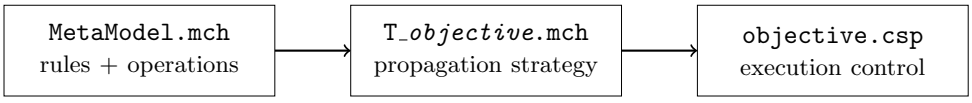

Figure 5: Generic architecture used for each TTC objective

This structure is uniformly applied across all objectives, ensuring a clear separation between transformation rules, their operational use, and the control of their execution.

[2] `https://bcert-meeduse.github.io/bcert-beta/2026_TTC_fp.zip`

3.2 Involved meta-models

The meta-models *Families* and *Persons* are provided by the contest. To bridge them as required by our infrastructure, we introduce a pivot meta-model (Figure 6). The central element is the class `Pivot`, which aggregates references to both `FamilyRegister` and `PersonRegister` through `familyModel` and `personModel` attributes. It also contains configuration parameters controlling the execution of the transformation, such as the direction (attribute `strategie`) and synchronization flags (`FAMILY_TO_NEW` and `PARENT_TO_CHILD`).

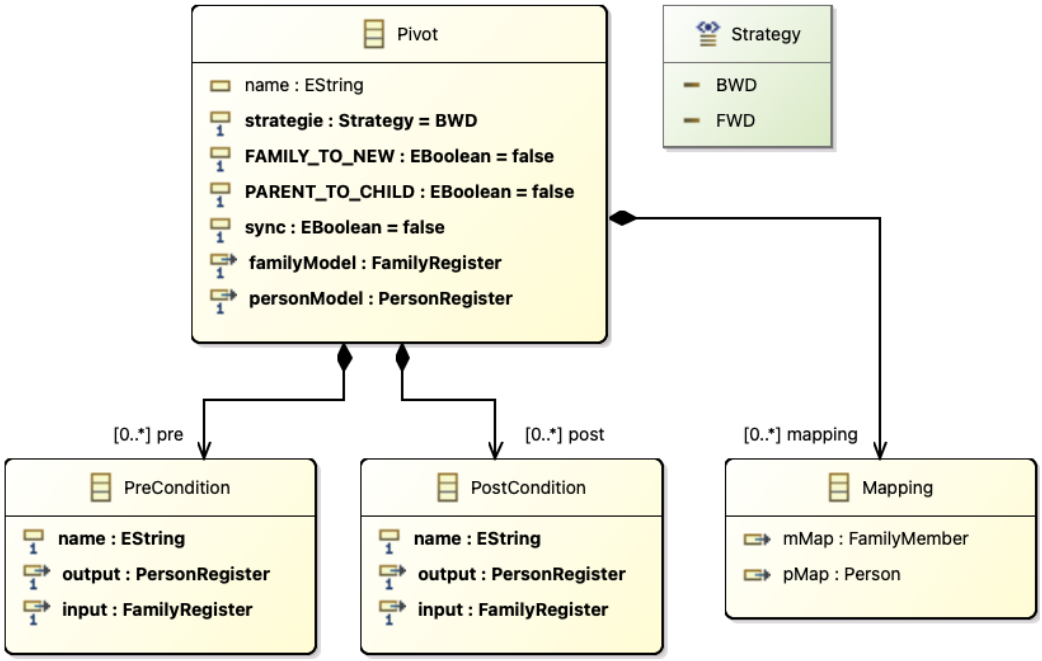

Figure 6: The pivot meta-model integrating Families and Persons

The pivot meta-model further introduces three key classes. First, `Mapping` elements explicitly represent correspondences between source and target elements, linking `FamilyMember` instances to `Person` instances. Second, `PreCondition` and `PostCondition` elements are used to describe the expected input and output configurations of test scenarios. These elements reference both the family and person registers and provide a structured way to express the conditions that must hold before and after the execution of transformation rules. Multiple `PreCondition` and `PostCondition` elements can be associated with a given `Pivot` instance. Each condition is identified by its `name` attribute.

3.3 Transformation rules in B

The B data structures (sets and relations) generated for the three meta-models are defined in the machine `MetaModel.mch`. Transformation rules are then specified as B operations over these structures, expressing how elements are created, updated, and related, while ensuring the preservation of structural and semantic invariants. These invariants capture both typing constraints (*e.g.*, well-formed relations between families, members, and persons), structural properties (*e.g.*, every `FamilyMember` belongs to exactly one `Family`, and every `Person` belongs to a `PersonRegister`), and consistency properties between the two domains (*e.g.*, mapped elements must respect gender and role constraints). As all transformation rules are proved to preserve these invariants, every reachable state of the system is guaranteed to be correct by construction. We distinguish three categories of rules: forward transformation rules, backward transformation rules, and rules dedicated to concurrent synchronization and conflict resolution.

1. Forward transformation rules propagate changes from the *Families* model to the *Persons* model:

   - `Member2Person` creates a new `Person` from an unmapped `FamilyMember`, initializes its attributes, and establishes the corresponding mapping.

- `ForwardCreateMissingPerson` completes an existing mapping by creating the missing `Person` when a `FamilyMember` is already mapped but no corresponding person exists.
- `ForwardRename` updates the name of a `Person` to reflect changes in the corresponding `FamilyMember`.
- `ForwardDelete` deletes a `Person` when the corresponding `FamilyMember` no longer exists, and removes the associated mapping.
- `ForwardRepairGender` repairs inconsistencies between the gender of a `Person` and the corresponding `FamilyMember`.

2. Backward transformation rules propagate changes from the *Persons* model to the *Families* model.

   - `Person2MemberNewFamily` creates a new `Family` and a corresponding `FamilyMember` from an unmapped `Person`, when no suitable family exists.
   - `Person2MemberExistingFamily` creates a new `FamilyMember` inside an existing family that matches the `Person`'s family name.
   - `BackwardRenameMemberName` updates the name of a `FamilyMember` to match the corresponding `Person`.
   - `BackwardDelete` deletes a `FamilyMember` when the corresponding `Person` no longer exists, and removes the associated mapping.
   - `BackwardMoveMaleToExistingFamily` and `BackwardMoveFemaleToExistingFamily` move a `FamilyMember` to an existing family when the associated `Person` refers to a different family.
   - `BackwardMoveMaleToNewFamily` and `BackwardMoveFemaleToNewFamily` create a new family and move the `FamilyMember` into it when the target family does not exist.

3. Concurrent synchronization and consistency rules handle situations where changes occur independently on both models and must be reconciled.

   - `ConcurrentMatchMemberPerson` creates a mapping between an existing `FamilyMember` and an existing `Person` when they match but are not yet linked.
   - `ConcurrentDeleteMatched` removes obsolete mappings when both the source and target elements have already been deleted.
   - `ConcurrentTargetRenameWins` resolves rename conflicts by prioritizing the target side and propagating the change back to the family model.
   - `ConcurrentDeleteWins` resolves conflicts between deletion and renaming by prioritizing deletion and removing the corresponding elements and mappings.

For illustration we show the example of rule `Member2Person` in Figure 7. This forward rule is triggered when a `FamilyMember` is not yet mapped to any `Person` (*i.e.*, `aMember` $\notin$ `ran(mMap)`), in forward mode (`strategie(pvtRoot) = FWD`) and outside synchronization (`sync(pvtRoot) = FALSE`). It creates a fresh `Person` (`aPerson` $\notin$ `Person`), assigns its gender according to the source element (`isFemale(aMember)`), and initializes its attributes, including a default birthdate (`birthday := birthday` $\lhd$ {`aPerson` $\mapsto$ `"DEFAULT"`}) and a name constructed from the family and member names. The new person is also inserted into the persons register (`persons := persons` $\lhd$ {`aPerson` $\mapsto$ `prsRoot`}). Finally, a new mapping is created (`aMapping` $\notin$ `Mapping`) and recorded consistently (`mMap := mMap` $\lhd$ {`aMapping` $\mapsto$ `aMember`}, `pMap := pMap` $\lhd$ {`aMapping` $\mapsto$ `aPerson`}), ensuring that the source element is now linked to its target counterpart.

3.4 Rule propagation

Depending on the considered objective, different propagation strategies are defined. For example, the basic propagation mechanism is defined by the machine `T0_MainTransformation.mch` presented in Figure 8). Unlike the machines (Figure 5) dedicated to benchmark test suites, this machine is not tied to a predefined scenario. It is intended for interactive use: the user edits an EMF model manually (a person model or a family model), launches the transformation in BCerT, and the tool computes via ProB the operations that are enabled in the current state. All enabled rules (those issued from clause `PROMOTES`) are then proposed to the user, who chooses which one to apply.

Member2Person =
  ANY *aMember*, *aPerson*, *aMapping* **WHERE**
    $aMember \in theMembers$
    $\wedge\ aMember \notin \mathbf{ran}(mMap)$
    $\wedge\ aPerson \in PERSON$
    $\wedge\ aPerson \notin Person$
    $\wedge\ aMapping \in MAPPING$
    $\wedge\ aMapping \notin Mapping$
    $\wedge\ strategie(pvtRoot) = FWD$
    $\wedge\ sync(pvtRoot) = \mathbf{FALSE}$
  THEN
    IF *isFemale*(*aMember*) **THEN**
      $Female := Female \cup \{aPerson\}$
    ELSE
      $Male := Male \cup \{aPerson\}$
    END
    $||\ Person := Person \cup \{aPerson\}$
    $||\ birthday := birthday \lhd \{aPerson \mapsto \text{"DEFAULT"}\}$
    $||$ **Persons_Person_name**$(aPerson) :=$
        $STRING\_CONC([Families\_Family\_name(familyOf(aMember)),$
          $\text{", "}, Families\_FamilyMember\_name(aMember)])$
    $||\ persons := persons \lhd \{aPerson \mapsto prsRoot\}$
    $||\ Mapping := Mapping \cup \{aMapping\}$
    $||\ mapping := mapping \lhd \{aMapping \mapsto pvtRoot\}$
    $||\ mMap := mMap \lhd \{aMapping \mapsto aMember\}$
    $||\ pMap := pMap \lhd \{aMapping \mapsto aPerson\}$
END ;

Figure 7: Example of a forward transformation rule specified in B

Obviously, some choices are non-deterministic. For instance, if a mapped pair exists but their names differ, both `ForwardRename` (propagating changes from the family model) and `BackwardRenameMemberName` (propagating changes from the person model) may be applicable. In the absence of additional control, the choice between these rules is left non-deterministic.

```
MACHINE     TO_MainTransformation
INCLUDES    MetaModel
PROMOTES
    Member2Person,
    ForwardDelete,
    ForwardRename,
    Person2MemberExistingFamily,
    Person2MemberNewFamily,
    BackwardDelete,
    BackwardRenameMemberName,
    BackwardMoveMaleToExistingFamily,
    BackwardMoveFemaleToExistingFamily,
    BackwardMoveMaleToNewFamily,
    BackwardMoveFemaleToNewFamily,
    ForwardCreateMissingPerson
    ConcurrentDeleteMatched,
    ConcurrentMatchMemberPerson,
    ConcurrentTargetRenameWins,
END
```

Figure 8: Main transformation machine

Regarding the test objectives of the TTC benchmark, the propagation of the underlying transformation rules is controlled by a dedicated operation `propagate` defined in each `T_objective.mch` machine.

- For the **batch forward** objective (`T1_BatchForward.mch`), the strategy consists in repeatedly applying the rule `Member2Person`. This reflects the fact that the transformation is performed in a single direction without considering deletions or updates.

- For the **batch backward** objective (`T2_BatchBackward.mch`), the strategy selects between two rules depending on the configuration. If the flag `FAMILY_TO_NEW` is enabled and a suitable family already exists (*i.e.*, a family with a matching name is found), the rule `Person2MemberExistingFamily` is applied. Otherwise, the rule `Person2MemberNewFamily` is used to create a new family. This introduces a conditional and data-dependent propagation strategy.

- For the **incremental objectives** (`T3_IncrementalForward.mch` and `T4_IncrementalBackward.mch`), the propagation is non-deterministic and defined using a `CHOICE` construct. At each step, one applicable rule is selected depending on the current state. In the forward case, this includes rules such as `Member2Person`, `ForwardDelete`, `ForwardRename`, and `ForwardRepairGender`, while in the backward case it includes creation, deletion, renaming, and movement rules (e.g., `Person2Member*`, `BackwardDelete`, `BackwardRenameMemberName`). No explicit priority is enforced between rules.

- The **concurrent** objective (`T5_Concurrent.mch`) introduces a priority-based propagation strategy. Rules are not applied arbitrarily but according to a fixed ordering. First, consistency-repair rules are considered: `ConcurrentDeleteMatched`, `ConcurrentMatchMemberPerson`, and `ConcurrentTargetRenameWins`. These rules resolve inconsistencies between the source and target models. Only when none of these rules is applicable does the system fall back to standard forward or backward transformations. In this second phase, an explicit direction is chosen (`SetFWD` or `SetBWD`), and rules are applied with internal priorities (e.g., deletion before renaming, and renaming before creation). This ensures that inconsistencies are resolved before structural updates are performed.

- Finally, the **roundtrip** objective combines both forward and backward rules in a fully non-deterministic manner. The `propagate` operation allows any applicable rule from both directions, as well as selected concurrent rules, to be applied. This reflects the absence of a fixed direction and enables exploring arbitrary sequences of transformations. However, in the test suite, the direction is explicitly controlled by invoking `SetFWD` or `SetBWD` before calling `propagate`.

Overall, propagation strategies range from deterministic (batch) to non-deterministic (incremental and roundtrip), and to priority-driven (concurrent), providing different levels of control over rule application depending on the test objective considered by the test suite.

3.5 Test suites

3.5.1 Model mutations

In the test suites provided by the TTC benchmark, the general approach consists in starting from an initial configuration where both the *Families* and the *Persons* models are empty. Depending on the test objective, these models are then progressively evolved through a sequence of mutations (e.g., creation, deletion, and renaming) before applying propagation or synchronization steps, and finally checking pre- and post-conditions. In the reference implementation, these mutations are encoded in Java using helper operations such as `srcEdit(helperFamily::createSimpsonFamily)`.

To reproduce these mutations within our formal framework, we introduce a set of basic model manipulation operations directly in B. For example, the operation `SonNEW` (Figure 9) creates a new `FamilyMember` and attaches it as a son to an existing family identified by its name. Based on these primitives, model mutations used in the reference implementation are reproduced by invoking our B operations with concrete parameter values in the objective-specific machines. For instance, the following operation from `T1_BatchForward.mch` creates a new family named “Flanders” and adds a son “Rod” (Figure 10).

SonNEW(*aFamilyName*, *aSonName*) =
PRE
aSonName ∈ *STRING* ∧
aFamilyName ∈ *STRING*
THEN
ANY *aFamily*, *aMember* **WHERE**
aFamily ∈ *Family* ∧ *Families_Family_name*(*aFamily*) = *aFamilyName*
∧ *families*(*aFamily*) = *fmlRoot*
∧ *aMember* ∈ *FAMILYMEMBER*
∧ *aMember* ∉ *FamilyMember*
THEN
FamilyMember := *FamilyMember* ∪ {*aMember*} ||
theSons := *theSons* ∪ {*aMember* ↦ *aFamily*} ||
Families_FamilyMember_name := *Families_FamilyMember_name* ⩤ {*aMember* ↦ *aSonName*}
END
END;

Figure 9: Example of a basic model mutation operation defined in B

fam ← **createFlandersFamilySonRod** =
BEGIN
fam ← *FamilyNEW*("Flanders") ;
SonNEW("Flanders", "Rod")
END;

Figure 10: Example of a call to SonNEW.

3.5.2 Test suites in CSP

Each `T_objective.mch` machine is associated with a CSP specification through the definition `CSP_GUIDE_FILE` in the header of the machine. This means that the execution of B operations is guided by a CSP controller. The role of CSP is to orchestrate three aspects: (*i*) the application of model mutations, (*ii*) the triggering of transformation rules through the `propagate` operation, and (*iii*) the verification of pre- and post-conditions. Instead of leaving rule application fully non-deterministic, CSP constrains the execution by defining admissible sequences of events.

A CSP test suite is structured as a set of scenarios, each corresponding to a TTC test case. A scenario consists of a sequence of mutation events followed by propagation phases and condition checks. The following excerpt illustrates this structure.

```
MAIN =
   testIncrementalInserts.true
     -> createInitialFamilies
     -> PROPAGATE_UNTIL_SYNC ;
     setBirthdayOfRod
     -> setBirthdayOfFatherBart
     -> createNewFamilySimpsonWithMembers -> unsync
     -> PROPAGATE_UNTIL_SYNC ;
     changeAllBirthdays
     -> createSonBart -> unsync
     -> PROPAGATE_UNTIL_STOP
```

Figure 11: Excerpt of a CSP test scenario (file `IncrementalForward.csp`)

In this example, the execution starts by selecting a specific test case (here `testIncrementalInserts`). A mutation is then applied to the source model (*e.g.*, `createInitialFamilies`). A propagation phase is then entered, during which transformation rules are repeatedly applied via the process `PROPAGATE_UNTIL_SYNC`, defined in Figure 12. This process triggers rule applications through events of the form `propagate!r`, where `r` denotes a

transformation rule. The execution continues until a synchronization event (`sync`) is reached. After this phase, additional mutations are applied, followed by another propagation phase. Finally, the execution ends with a last propagation phase, `PROPAGATE_UNTIL_STOP`, which is similar to `PROPAGATE_UNTIL_SYNC` but does not allow synchronization. As a result, rule applications continue until no further rule is enabled.

```
PROPAGATE_UNTIL_SYNC =
   [] r :  RULE @ propagate!r -> PROPAGATE_UNTIL_SYNC
   [] sync -> SKIP
```

Figure 12: CSP control of rule application

Pre- and post-conditions are encoded as explicit checks using events such as `assertPrecondition` and `assertPostcondition`. Depending on their evaluation, the execution either continues or stops, ensuring that each scenario satisfies the expected behavior defined by the benchmark.

4 Results

BCerT provides full support for the execution of CSP||B transformation scenarios. It ensures that model mutations, rule propagation, and condition checking are executed and reproducible. Figure 13 illustrates the execution of CSP||B model within BCerT of the *Concurrent* test suite.

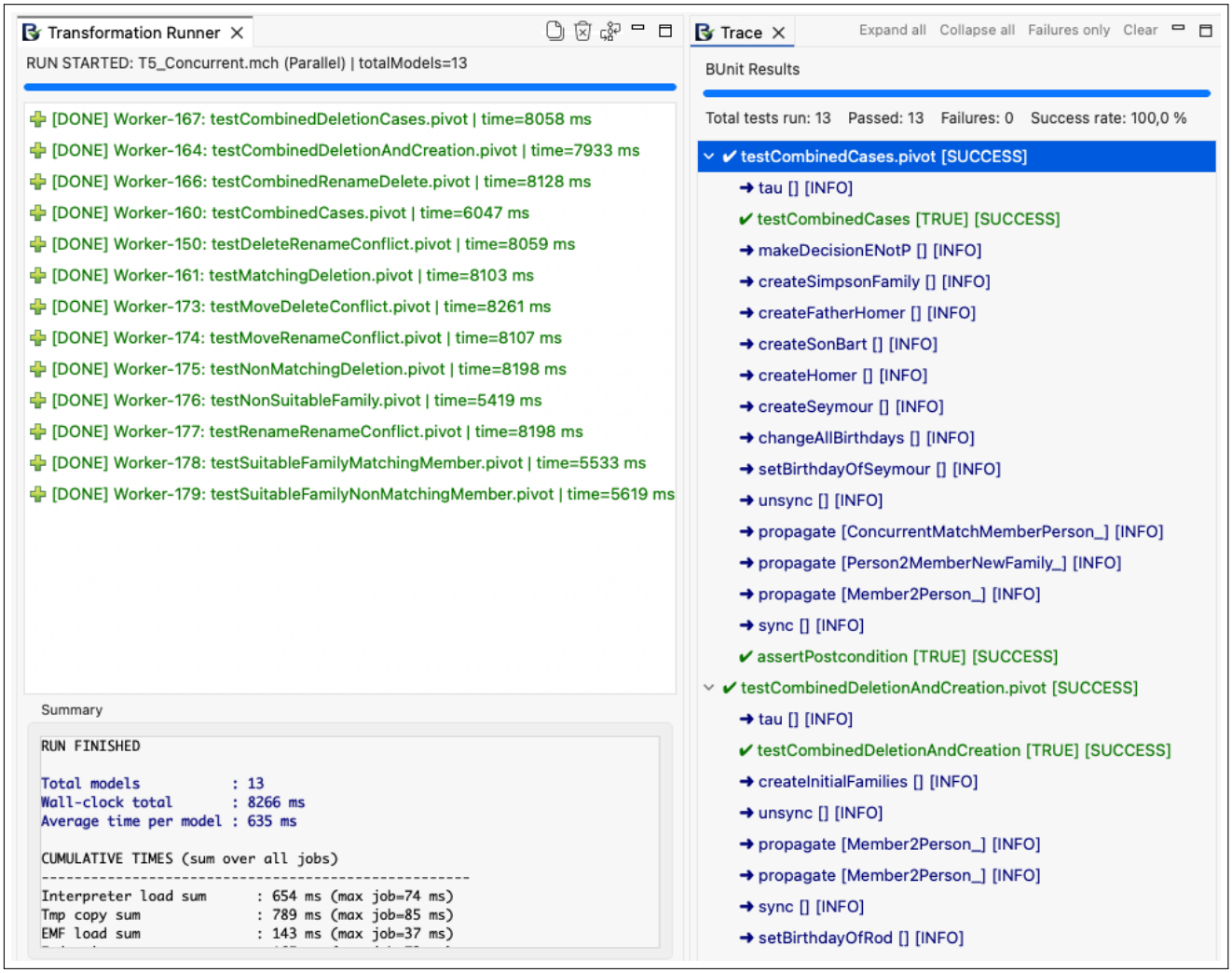

Figure 13: Execution of a CSP||B test suite in BCerT

The left-hand side shows the execution summary, where each test scenario is run as an independent job. In this example, the *Concurrent* test suite contains 13 scenarios, all of which are successfully executed in parallel. The total execution time is 8266 ms, with an average time of 635 ms per scenario. The right-hand side provides a detailed execution trace for each scenario. It shows the sequence of events triggered during execution, including model mutations, synchronization steps (`sync`/`unsync`), rule applications via `propagate`, and the evaluation of pre- and post-conditions. For instance, one can observe successive rule applications such as `ConcurrentMatchMemberPerson`, `Person2MemberNewFamily`, and `Member2Person`, followed by a synchronization point and a successful post-condition check. This execution trace demonstrates how CSP orchestrates the application of B-based transformation rules while ensuring that each scenario satisfies its expected properties. It also highlights the ability of BCerT to execute multiple scenarios concurrently and to provide detailed feedback on the correctness of each execution.

Table 1 summarizes the size of the main B and CSP specifications involved in our solution. The overall specification consists of 3275 lines of B code and 1006 lines of CSP code. The `MetaModel.mch` machine represents the core of the solution, as it defines the data structures, invariants, and transformation rules. Objective-specific machines (`T_objective.mch`) mainly define propagation strategies and remain relatively compact in comparison.

B Machines	**LOC**
BatchBackward.mch	205
MetaModel.mch	1278
RESET.mch	6
T0_MainTransformation.mch	24
T1_BatchForward.mch	109
T2_BatchBackward.mch	209
T3_IncrementalForward.mch	346
T4_IncrementalBackward.mch	367
T5_Concurrent.mch	502
T6_RoundTrip.mch	229
Total	**3275**

CSP Specifications	**LOC**	**Test scenarios**
BatchForward.csp	81	7
BatchBackward.csp	124	11
IncrementalForward.csp	170	8
IncrementalBackward.csp	250	8
Concurrent.csp	265	13
RoundTrip.csp	116	3
Total	**1006**	**50**

Table 1: Size of the B machines and CSP specifications, with number of test scenarios

5 Conclusion

This paper presented our BCerT solution to the TTC'2026 F2P case study. The solution is specified using B and CSP, and is fully executable within the BCerT environment. The core transformation logic is expressed as B operations, while invariants capture the structural and semantic consistency properties that must be preserved. Since the transformation rules are proved correct with respect to these invariants, the approach provides correctness by construction solution at the level of the formal model.

Beyond verification, the TTC benchmark requires validation against observable scenarios. To address this aspect, we used CSP as an orchestration layer for B operations. CSP controls model mutations, rule propagation, synchronization phases, and pre-/post-condition checking. This makes it possible to execute the TTC test suites in a reproducible way, while keeping the rule-based execution mechanism of BCerT and ProB. The solution covers batch forward, batch backward, incremental forward, incremental backward, roundtrip, and concurrent synchronization scenarios.

The current version of the solution does not address the scalability challenge. BCerT relies on ProB, which is both a model-checker and a constraint-based animator, and large-scale scenarios may require significant computational resources. Unfortunately, we did not have enough time to complete these experiments before submission. We plan to provide additional feedback on this point during the contest.

References

[1] Jean-Raymond Abrial. *The B-book: Assigning Programs to Meanings.* Cambridge University Press, New York, NY, USA, 1996.

[2] Michael Butler and Michael Leuschel. Combining CSP and B for specification and property verification. In *International Conference on Formal Methods*, volume 3582, page 221–236. Springer, 2005.

[3] Antonio Garcia-Dominguez and Georg Hinkel. Truth Tables to Binary Decision Diagrams. In Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, editors, *Proceedings of the 12th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2019) federation of conferences*, CEUR Workshop Proceedings. CEUR-WS.org, July 2019.

[4] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ, 1985.

[5] Akram Idani. Meeduse: A tool to build and run proved DSLs. In *16th International Conference on Integrated Formal Methods (IFM)*, volume 12546 of *LNCS*, pages 349–367. Springer, 2020.

[6] Akram Idani. Formal model-driven executable DSLs: Application to Petri-Nets. *Innovations in Systems and Software Engineering*, 18(1), 2022.

[7] Akram Idani. The B Method meets MDE: Survey, progress and future. In *16th International Conference on Research Challenges in Information Science (RCIS)*, volume 446, pages 495–512. Springer, 2022.

[8] Akram Idani. Transpilation Meets the B Method. In Egon Börger and Yamine Aït Ameur, editors, *Festschrift in Honor of Jean-Raymond Abrial*. Springer, 2026. Accepted book chapter.

[9] Akram Idani and German Vega. Engineering verified model transformations through a proof-based language workbench. In *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, 2026. Accepted long paper.

[10] Akram Idani, Germán Vega, and Michael Leuschel. Applying formal reasoning to model transformation: The meeduse solution. In *Proceedings of the 12th Transformation Tool Contest (TTC@STAF)*, volume 2550 of *CEUR Workshop Proceedings*, pages 33–44. CEUR-WS.org, 2019.

[11] Michael Leuschel and Michael Butler. ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, Mar 2008.