Inconsistency Management using Synchronization Blocks

Georg Hinkel[1]

[1]RheinMain University of Applied Sciences and Arts, Unter den Eichen 5, 65195 Wiesbaden, Germany

Abstract

The development of complex systems involves multiple disciplines that each have matured models to represent a system in their discipline. Particularly in early stages of development, inconsistencies occur and only need to be resolved later in a project, as an explicit step. In this paper, we discuss how these inconsistencies can be managed using synchronization blocks, discuss the ICCM challenge and present an implementation of our approach inspired by the TTC 2017 Families to Persons case.

Keywords

Consistency Management, Synchronization Blocks, NMF

1. Introduction

The development of complex systems such as cyber-physical systems involves multiple disciplines that look at the system from different angles. In each discipline, tooling has evolved over many years, making changes to the underlying formats difficult and costly. In early stages of the development of a system, inconsistencies between these views may have to be tolerated as fixing them immediately required information not present, yet. This is the starting point of the ICCM challenge [1].

In this paper, we show how the inconsistency management can be solved by synchronization blocks [2] and their implementation in NMF Synchronizations. Because model synchronizations as such do not have a user interface and therefore do not directly allow users to interact with the models, we created an extension to AnyText [3] to synchronize textual model representations, using synchronization blocks at its core.

As the ICCM challenge [1] does not provide any concrete material such as metamodels for a demonstration, we use the *Families to Persons* benchmark introduced by Anjorin et al. [4] to demonstrate our approach. We provide an open-source implementation of a Visual Studio Code extension that synchronizes textual representations of these models and provides code lenses for the user to resolve inconsistencies.

2. Running Example: Families to Persons

As a running example in this paper, we use the *Families to Persons* benchmark introduced by Anjorin et al. [4]. The idea of this benchmark was that it encodes many recurring problems of bidirectional model synchronization in a relatively small example.

The metamodels in this case study are depicted in Figures 1a and 1b. They represent similar information in a different manner: Whereas the full name of a person is separated in its first name and surname in the *Families* model, this information is combined in the Person model in the format *"Surname, Firstname"*. Whereas the information on gender is explicitly modeled in the *Persons* metamodel, this information is implicitly encoded in the containment hierarchy in the *Families* metamodel. Lastly, the *Families* metamodel contains a family structure that is missing in the *Persons* model, meanwhile the birthday information is only present in the *Persons* metamodel.

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

georg.hinkel@hs-rm.de (G. Hinkel)

https://www.hs-rm.de/design-informatik-medien/ueber-uns/personen-am-fachbereich/person/georg-hinkel (G. Hinkel)

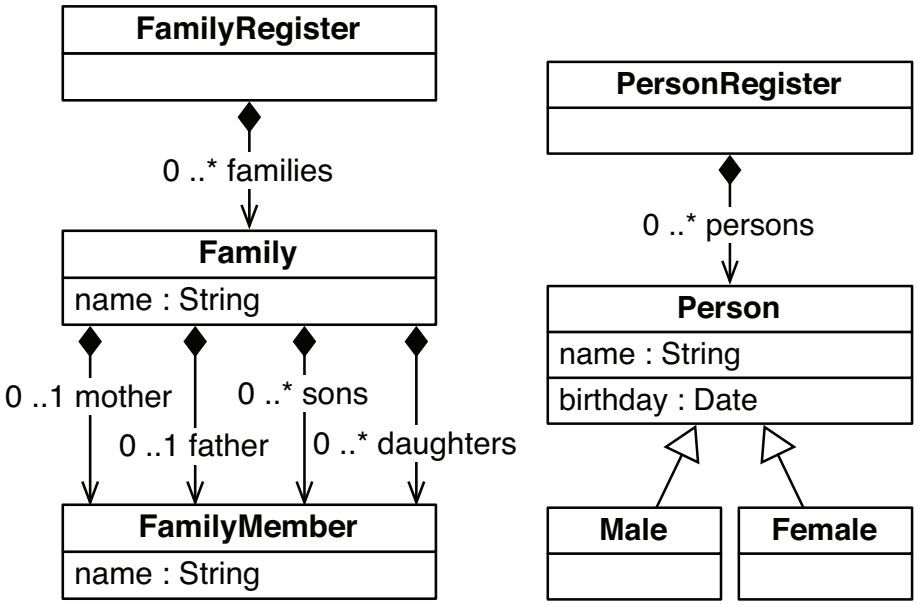
0000-0002-6462-5208 (G. Hinkel)

© 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/ICMM-paper2.pdf



(a) The Families metamodel (b) The Persons metamodel

Figure 1: Source and target metamodel in the FamiliesToPersons case study [4]

Since no model can be reconstructed entirely from the other, the task for solutions is to create a bidirectional transformation between both models. Meanwhile the direction from the *Families* metamodel to the *Persons* metamodel is straightforward, the backwards transformation is nondeterministic. Therefore, the behavior of the backward transformation has to be controlled by two flags PREFER_CREATING_PARENT_TO_CHILD and PREFER_EXISTING_FAMILY_TO_NEW. Although the naming conventions of these configurations suggest that they are constant, they actually may change during a synchronization.

3. NMF Expressions and NMF Synchronizations

NMF Expressions [5] is an incrementalization system integrated into the C# language. It takes expressions of functions and implicitly derives an incremental change propagation algorithm. This works by setting up a dynamic dependency graph that keeps track of the models' state and is adapted when necessary. The incrementalization system is extensible and supports large parts of the Standard Query Operators (SQO[1]). For some expressions, NMF Expressions is also able to lift the expression to a lens [6].

NMF Synchronizations is a model synchronization approach based on the algebraic theory of synchronization blocks. Synchronization blocks are a formal tool to run model transformations in an incremental (and bidirectional) way [2]. They combine a slightly modified notion of lenses [6] with incrementalization systems. Model properties and methods are considered morphisms between objects of a category that are set-theoretic products of a type (a set of instances) and a global state space Ω.

A (well-behaved) in-model lens $l : A \hookrightarrow B$ between types A and B consists of a side-effect free GET morphism $l \nearrow \in Mor(A, B)$ (that does not change the global state) and a morphism $l \searrow \in Mor(A \times B, A)$ called the PUT function that satisfy the following conditions for all $a \in A, b \in B$ and $\omega \in \Omega$:

$$l \searrow (a, l \nearrow (a)) = (a, \omega)$$

[1] http://msdn.microsoft.com/en-us/library/bb394939.aspx; SQO is a set of language-independent standard APIs for queries, specifically defined for the .NET platform.

$$l \nearrow (l \searrow (a, b, \omega)) = (b, \tilde{\omega}) \quad \text{for some } \tilde{\omega} \in \Omega.$$

The first condition is a direct translation of the original PutGet law. Meanwhile, the second line is a bit weaker than the original GetPut law because the global state may have changed. In particular, we allow the Put function to change the global state.

A (single-valued) synchronization block $\boldsymbol{S}$ is an 8-tuple $(A, B, C, D, \Phi_{A-C}, \Phi_{B-D}, f, g)$ that declares a synchronization action given a pair $(a, c) \in \Phi_{A-C} : A \cong C$ of corresponding elements in a base isomorphism Φ_{A-C}. For each such tuple in states (ω_L, ω_R), the synchronization block specifies that the elements $(f(a, \omega_L), g \nearrow (b, \omega_R)) \in B \times D$ gained by the lenses f and g are isomorphic with respect to Φ_{B-D}.

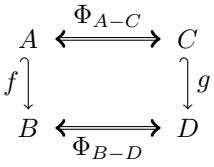


Figure 2: Schematic overview of unidirectional synchronization blocks

A schematic overview of a synchronization block is depicted in Figure 2. The usage of lenses allows these declarations to be enforced automatically and in both directions, if required. The engine computes the value that the right selector should have and enforces it using the Put operation. Similarly, a multi-valued synchronization block is a synchronization block where the lenses f and g are typed with collections of B and D, for example $f : A \hookrightarrow B*$ and $g : C \hookrightarrow D*$ where stars denote free monoids.

Synchronization blocks have been implemented in NMF Synchronizations, an internal DSL hosted by C# [7, 2] using NMF Expressions for the incrementalization. This DSL is able to lift the specification of a model transformation/synchronization in two orthogonal dimensions:

- **Direction:** A client may choose between transformation from left to right, right to left or in check-only mode
- **Change Propagation:** A client may choose whether changes to the input model should be propagated to the output model, also vice versa or not at all

This flexibility makes it possible to reuse the specification of a transformation for a broad range of different use cases. Furthermore, the fact that NMF Synchronizations is an internal language means that a wide range of advantages from mainstream languages, most notably modularity and tool support, can be inherited [8].

4. Inconsistency Management using Synchronization Blocks

The core idea of managing inconsistencies using synchronization blocks is to exploit the check-only mode implemented in NMF Synchronizations. In this mode, the synchronization engine does not automatically repair the inconsistency but reports it in a way that is actionable: the client can decide to execute the repair operation either on the right side or left side by executing appropriate abstract methods. What is important here is that the synchronization does not change any of the involved models until a client executes repairs through such methods. Information in a model that is not part of any synchronization block is simply ignored.

NMF Synchronizations further has the advantage that it is not limited to greenfield applications. This means, one can also synchronize existing models and attach a change propagation. This allows it to be used in contexts that evolve for a long time, in particular longer than an editor is open. However, NMF Synchronizations currently does not serialize the trace or the inconsistencies but inconsistencies and the trace are restored on the fly if the synchronization is started again. This has the consequence

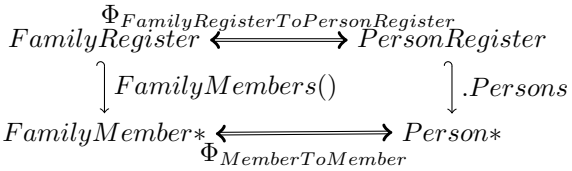


Figure 3: Synchronization block to synchronize family members with person elements

```
1  public class FamilyRegisterToPersonRegister : SynchronizationRule<FamilyRegister, PersonRegister> {
2    public override void DeclareSynchronization() {
3      SynchronizeMany(SyncRule<MemberToMember>(),
4                  fam => new FamilyMemberCollection(fam),
5                  persons => persons.Persons);
6    }
7  }
8  public class MemberToMember : SynchronizationRule<IFamilyMember, IPerson> {
9    public override void DeclareSynchronization() {
10     Synchronize(m => m.GetFullName(), p => p.Name);
11   }
12 }
```

Listing 1: Implementation of main synchronization blocks

that the nature of inconsistencies may change after the synchronization is restarted: If two elements are identified as corresponding but one is renamed, there is an inconsistency that the names do not match but if the synchronization is restarted, it may not see that these elements belong together and two inconsistencies are created to report that both elements have missing counterparts. NMF Synchronizations allows to override the decision when two model elements should correspond, but that means trace serialization would have to be implemented manually. However, considering the lifecycle of the supposed example model, we think that this kind of problem is unrealistic: in the mentioned case of the different views for brakes, it is unlikely that the requirements element for a brake renames the brake element when there is already a functional description for it.

In the running example of *families to persons*, we see two correspondences that constitute the synchronization [9]:

1. All family members contained in a family need to be synchronized with the people in the Persons model and
2. The full name of family members that consists of the name of the family and the name of the family member needs to be synchronized with the full name of the corresponding person.

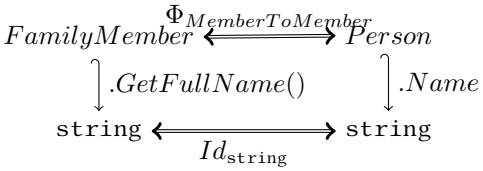


Figure 4: Synchronization block to synchronize names

Using synchronization blocks, these correspondences can be formulated in the diagrams of Figure 3 and Figure 4. The implementation in NMF Synchronizations is depicted in Listing 1.

While NMF is able to convert the simple member accesses for the persons into lenses, this does not hold for the helpers `FamilyMemberCollection` and `GetFullName` that we used in this implementation – these lenses are highly specific to the given scenario. Therefore, we have to explicitly provide an implementation of Put for these two lenses.

```
1  private static ObservingFunc<IFamilyMember, string> fullName =
2    new ObservingFunc<IFamilyMember, string>(m => m.Name == null ? null : ((IFamily)m.Parent).Name + ",
         "␣" + m.Name);
3
4  [LensPut(typeof(Helpers), "SetFullName")]
5  [ObservableProxy(typeof(Helpers), "GetFullNameInc")]
6  public static string GetFullName(this IFamilyMember member) {
7    return fullName.Evaluate(member);
8  }
9  public static INotifyValue<string> GetFullNameInc(this IFamilyMember member) {
10   return fullName.Observe(member);
11 }
12 public static void SetFullName(this IFamilyMember member, string newName) {
13   ...
14 }
```

Listing 2: Implementation of the `GetFullName` lens

```
1  private class FamilyMemberCollection : CustomCollection<IFamilyMember> {
2    public FamilyRegister Register { get; private set; }
3    public FamilyMemberCollection(FamilyRegister register)
4      : base(register.Families.SelectMany(fam => fam.Children.OfType<IFamilyMember>()))
5    { Register = register; }
6
7    public override void Add(IFamilyMember item) { ... }
8    public override bool Remove(IFamilyMember item) { ... }
9    public override void Clear() { ... }
10 }
```

Listing 3: Implementation of the `FamilyMemberCollection`

For the `GetFullName`-method, the PUT operation needs to be specified through an annotation. In addition, because NMF does not parse the contents of a method (only of lambda expressions), we need to specify an explicitly incrementalized version of the given helper method. To do this, we can reuse the implicit incrementalized lambda expression and also use that for the batch implementation to avoid code duplication. A sketched implementation is depicted in Listing 2.

In Listing 2, we create an `ObservingFunc` for the function to obtain a family members full name. This syntax allows NMF to get a model of the actual function and therefore, NMF Expressions is able to incrementalize it. In Lines 6–11, we use this object to represent the actual extension method and its incrementalization, which essentially forwards to the same `ObservingFunc` object. The incrementalization is connected with the original function through the annotation in line 5. Furthermore, we annotate the PUT method also using an annotation in line 4 where we reference the `SetFullName` method in line 12–14. The return type `void` of this method makes it clear that this is a persistent lens. This method may contain arbitrary C# code and is called when NMF Synchronizations needs to write a value, for instance as a consequence of an update in the *Persons* model where the last name of a person changed.

In the case of `FamilyMemberCollection` which as the name implies is a collection, we only have to provide the query how the results of this collection are obtained and implement the methods `Add`, `Remove` and `Clear`. NMF Expressions is able to automatically incrementalize the query when this is necessary and uses the provided model manipulation methods in case a model element has to be added to the collection. A schematic implementation is depicted in Listing 3.

Again, the model manipulation methods `Add`, `Remove` and `Clear` may contain arbitrary C# code. However, to add a family member to a family, the `Add` method has to know the family name of a person as well as its gender – information that is encoded using the containment hierarchy in the Families model and therefore unavailable before the element is added to a family. Therefore, we carry this information over from the corresponding element of the Person metamodel using a temporary stereotype: In NMF, all model elements are allowed to carry extensions. We use this to add an extension that specifies the last name and whether the given element is male. The stereotype is deleted as soon as

```
1  public class MemberToMale : SynchronizationRule<IFamilyMember, IMale> {
2   public override void DeclareSynchronization() {
3    MarkInstantiatingFor(SyncRule<MemberToMember>(),
4         leftPredicate: m => m.FatherInverse != null || m.SonsInverse != null);
5   }
6   protected override IFamilyMember CreateLeftOutput(IMale input, ...) {
7    var member = base.CreateLeftOutput(input, candidates, context, out existing);
8    member.Extensions.Add(new TemporaryStereotype(member) {
9     IsMale = true,
10    LastName = input.Name.Substring(0, input.Name.IndexOf(','))
11   });
12   return member;
13  }
14 }
```

Listing 4: The `MemberToMale`-rule

```
1  public override bool ShouldCorrespond(IFamilyMember left, IMale right, ISynchronizationContext
       context)
2  {
3     return left.GetFullName() == right.Name;
4  }
```

Listing 5: Specifying the correspondences initially

a family member is added to a family.

Furthermore, the fact that different genders are modeled through different classes in the Persons model, the synchronization rule `MemberToMember` needs to be refined to allow NMF Synchronizations to decide whether to create a `Male` or `Female` output element. This can be done in NMF Synchronizations through an instantiating rule.

The implementation of both of these concepts is depicted in Listing 4.

In particular, lines 3 and 4 mark the synchronization rule `MemberToMale` as instantiating for the rule `MemberToMember` on the condition that the family member is either a father or a son of a family. Further, we override the creation of an output model element for the left side by overriding the method `CreateLeftOutput`. This method calls the base implementation which simply uses the default constructor to create a new `FamilyMember` element. Then, it adds a case-specific extension called `TemporaryStereotype` that carries the information on the gender (through the `IsMale` attribute) and the last name.

Unlike the original TTC case, the synchronization also needs to be able to attach to existing models. Therefore, it is necessary to also implement the method `ShouldCorrespond` to decide whether two already existing model elements should match. This can be implemented by just matching the name of the person with the full name of the family member. The method can be implemented either in the `MemberToMember`-rule or in both `MemberToMale` and `MemberToFemale`, as the former by default delegates to the latter. Listing 5 depicts an implementation of the latter.

5. User Interface for Inconsistencies with AnyText

Users do not interact with the abstract model itself but with a concrete syntax, in many cases a domain-specific language. Hence, the way how the information about inconsistencies should be exposed to the user depends on the type of concrete syntax used. One approach could be to bring inconsistencies into the user's view as diagnostic items, for instance as a warning. However, considering the workflow of complex models that are developed stepwise, this may distract the user from actual problems with the model at hand: a problem caused by checks of static semantics might be lost if it drowned in warnings about inconsistencies that are accepted, especially at an early stage of the development.

When it comes to textual representations, we see another way that does not take away so much of the users attention: code lenses. In many editors for general-purpose languages, code lenses are a

```
grammar Families (families)
root FamilyRegister

FamilyRegister: families+=Family <nl>;
Family:
  'family' name=String 'with' <ind> <nl>
    ('father' father=FamilyMember | 'mother' mother=FamilyMember | 'son' sons+=FamilyMember | '
        daughter' daughters+=FamilyMember) * <unind><nl>;
FamilyMember: name=String <nl>;
terminal String: /\w+/;
```

Listing 6: A simple grammar for families

```
grammar Persons (persons)
root PersonRegister

PersonRegister: persons+=Person <nl>*;
Person: Male | Female;
Male:
  'male' name=String <nl>
  ('born' birthday=Date <nl>)?;
Female:
  'female' name=String <nl>
  ('born' birthday=Date <nl>)?;
terminal String: /\w+,\s\w+/;
terminal Date returns nmeta.DateTime: /\d{2}\.\d{2}\.\d{4}/;
```

Listing 7: A simple grammar for persons

```
await SynchronizationBootstrapper.RunLspServerOnStandardInStandardOutAsync(
   [ new FamiliesGrammar(), new PersonsGrammar() ],
   [ new ModelSynchronization<FamilyRegister, PersonRegister, FamiliesToPersonsSynchronization,
            FamiliesToPersonsSynchronization.FamilyRegisterToPersonRegister>
      {
         IsAutomatic = true,
         Direction = SynchronizationDirection.CheckOnly,
         LeftExtension = ".families",
         RightExtension = ".persons"
      } ]);
```

Listing 8: Startup logic for the editor extension

popular tool to notify the user of relevant but not crucially important information such as the number of references, the last edit or the number of failing tests using this method, if available. While such information is helpful for many use cases, it does not give the user an impression that there is something wrong with the current model[2].

To demonstrate this, we implemented two DSLs for Families and Persons as defined in [4] using AnyText [3], depicted in Listings 6 and 7.

Recently, AnyText does not only implement a parser but a fully bidirectional, incremental model-to-text synchronization such that grammar-based model-to-text synchronizations and a model-to-model synchronization can be glued together to form a bidirectional, incremental text synchronization [11]. We combined AnyText with NMF Synchronization such that the model synchronization and the grammars are all that a developer needs to implement in order to obtain an inconsistency management. AnyText comes with a code generator that can scaffold a Visual Studio Code extension including these grammars. However, this code generator does not assume that the languages supported by the extension are synchronized. Therefore, we need to modify the startup code as depicted in Listing 8.

This listing denotes the only glue code that is necessary to combine the model synchronization and the grammars. In particular, there is no manual code necessary to generate parts of either Families or

[2]Given that all of the model properties from Stachowiak [10] are fulfilled, we see code also as a model of the running software.

Persons model. Line 2 specify the grammars supported by the LSP server, while lines 3–10 specify the synchronization supported by the server. Here, the root types of the model, the model synchronization and the start rule to be used are given as types, as NMF Synchronizations often identifies instances with types. The configuration settings in lines 6–9 specify the file extensions the synchronization should react to, the direction of the synchronization and whether the synchronization should start automatically. If it does not start automatically, the framework provides a command to the editor to start synchronization explicitly. The direction `CheckOnly` means that the synchronization will be executed in check-only mode. If we set this to `LeftWins`, the synchronization would fix inconsistencies immediately, giving information in the family register model precedence to information in the persons model.

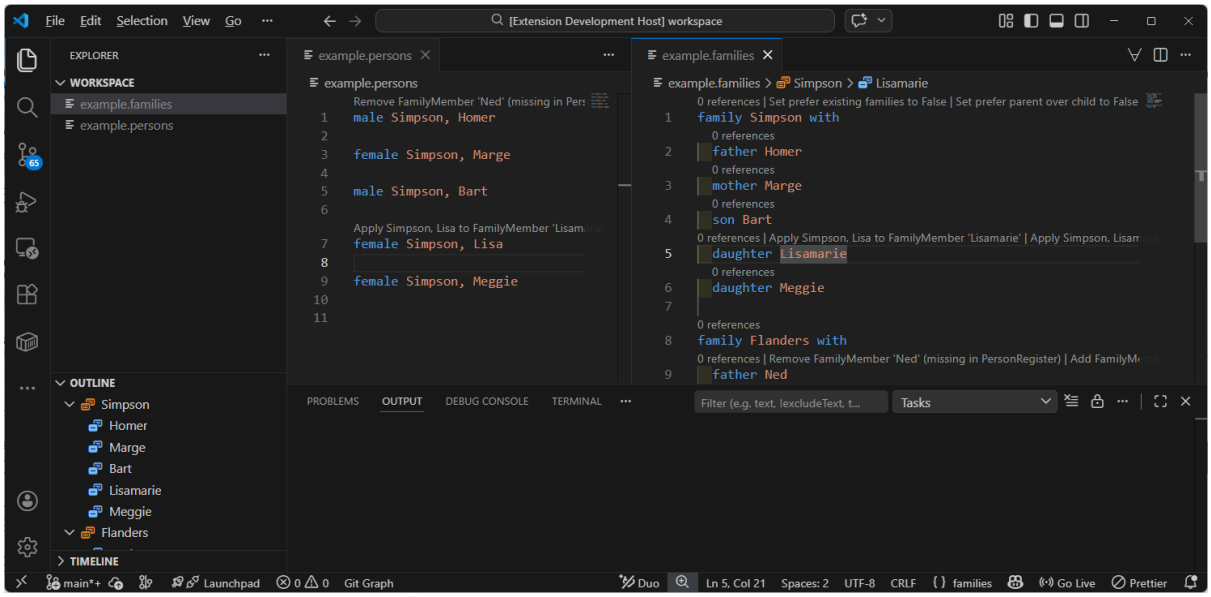


Figure 5: Visual Studio Code extension created from the running example, showing inconsistencies as code lenses

Figure 5 shows a screenshot of the resulting Visual Studio Code extension with example models open side by side. Although we extended NMF Synchronizations to allow developers to include custom inconsistency descriptions, we did not include them in the model synchronization and hence, the code lenses use the default descriptions built into NMF Synchronizations.

Note also that the incremental model synchronization may produce different inconsistencies than batch synchronization as it tracks model elements. In the example, we modified the first name of Lisa Simpson and the synchronization engine detects that just the name is no longer consistent, but correspondence between the model elements is kept.

The source code of the extension is publicly available at https://github.com/NMFCode/families-to-persons.

6. Related Work

There are several related works regarding the *Families-to-Persons* case collected by Anjorin et al. [4]. An approach that integrates synchronization results into the diagnostics view is the EVL+Strace [12] solution from Samimi-Dehkordi et al. [13]. While the TTC version concentrated on automating EVL, the tool usually foresees that inconsistencies are resolved manually from the diagnostics section.

A good overview on bidirectional model transformation (BX) languages, including a classification scheme, was created by Steven [14], Hidaka et al. [15] or Anjorin et al. [16]. The most prominent incremental and bidirectional transformation languages may be Triple Graph Grammars, originally introduced by Schürr [17]. Multiple tools implemented this paradigm [18] and some incremental

implementations exist [19, 20]. Buchmann created a BX approach, also using layered DSLs [21].

The bigER tool [22] also synchronizes text via the LSP but with a graphical representation of the model via Sprotty. Here, a separate transformation is responsible to synthesize text changes from model updates. Such an additional artifact is not necessary for AnyText as the functionality can be derived from the grammar and again, the focus is on automated synchronization.

7. Conclusion

The paper provides a concrete example, how inconsistencies between textual languages can be specified as synchronization blocks in terms of the abstract syntax, independent of the grammar. Synchronization blocks allow to specify consistency relationships in a way that allows the engine to fix them automatically or upon request. The synchronization capabilities of AnyText then automatically provides tooling that allows the user to fix inconsistencies through code lenses in the editor.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] T. Weber, A. Lange, H. Klare, J. Cederbladh, T. A. Völk, K. Bause, B. Jutz, N. Hagel, M. Kaushik, C. Atkinson, Icmm ((in-)consistency management in modeling) challenge 2026 - construction of a brake, 2026. URL: https://doi.org/10.5281/zenodo.18599405. doi:10.5281/zenodo.18599405.

[2] G. Hinkel, E. Burger, Change propagation and bidirectionality in internal transformation DSLs, Softw. Syst. Model. 18 (2019) 249–278. URL: https://doi.org/10.1007/s10270-017-0617-6. doi:10.1007/s10270-017-0617-6.

[3] G. Hinkel, A. Hert, N. Hettler, K. Weinert, Anytext: Incremental, left-recursive parsing and pretty-printing from a single grammar definition with first-class LSP support, in: Proceedings of the 18th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2025, Koblenz, Germany, June 12-13, 2025, ACM, 2025, pp. 98–111. URL: https://doi.org/10.1145/3732771.3742716. doi:10.1145/3732771.3742716.

[4] A. Anjorin, T. Buchmann, B. Westfechtel, Z. Diskin, H. Ko, R. Eramo, G. Hinkel, L. Samimi-Dehkordi, A. Zündorf, Benchmarking bidirectional transformations: theory, implementation, application, and assessment, Softw. Syst. Model. 19 (2020) 647–691. URL: https://doi.org/10.1007/s10270-019-00752-x. doi:10.1007/s10270-019-00752-x.

[5] G. Hinkel, R. Heinrich, R. Reussner, An extensible approach to implicit incremental model analyses, Software & Systems Modeling (2019). URL: https://doi.org/10.1007/s10270-019-00719-y. doi:10.1007/s10270-019-00719-y.

[6] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, A. Schmitt, Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem, ACM Transactions on Programming Languages and Systems (TOPLAS) 29 (2007). URL: http://doi.acm.org/10.1145/1232420.1232424. doi:10.1145/1232420.1232424.

[7] G. Hinkel, Change Propagation in an Internal Model Transformation Language, in: D. Kolovos, M. Wimmer (Eds.), Theory and Practice of Model Transformations: 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings, Springer International Publishing, Cham, 2015, pp. 3–17. URL: http://dx.doi.org/10.1007/978-3-319-21155-8_1. doi:10.1007/978-3-319-21155-8_1.

[8] G. Hinkel, T. Goldschmidt, E. Burger, R. Reussner, Using Internal Domain-Specific Languages to Inherit Tool Support and Modularity for Model Transformations, Software & Systems Modeling (2017) 1–27. URL: http://rdcu.be/oTED. doi:10.1007/s10270-017-0578-9.

[9] G. Hinkel, An NMF solution to the Families to Persons case at the TTC 2017, in: A. Garcia-Dominguez, G. Hinkel, F. Krikava (Eds.), Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences, CEUR Workshop Proceedings, CEUR-WS.org, 2017.

[10] H. Stachowiak, Allgemeine Modelltheorie, Springer Verlag, Wien, 1973.

[11] G. Hinkel, Incremental, bidirectional text-to-text synchronizations with AnyText, in: 19th ACM SIGPLAN International Conference on Software Language Engineering (SLE '26), July 02–03, 2026, Rennes, France, ACM, 2026. Accepted, to appear.

[12] L. Samimi-Dehkordi, B. Zamani, S. K. Rahimi, Evl+strace: a novel bidirectional model transformation approach, Inf. Softw. Technol. 100 (2018) 47–72. URL: https://doi.org/10.1016/j.infsof.2018.03.011. doi:10.1016/J.INFSOF.2018.03.011.

[13] L. Samimi-Dehkordi, B. Zamani, S. K. Rahimi, Solving the families to persons case using evl+strace, in: A. García-Domínguez, G. Hinkel, F. Krikava (Eds.), Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017, volume 2026 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2017, pp. 54–62. URL: https://ceur-ws.org/Vol-2026/paper14.pdf.

[14] P. Stevens, A landscape of bidirectional model transformations, in: R. Lämmel, J. Visser, J. a. Saraiva (Eds.), Generative and Transformational Techniques in Software Engineering II, volume 5235 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2008, pp. 408–424. URL: http://dx.doi.org/10.1007/978-3-540-88643-3_10. doi:10.1007/978-3-540-88643-3_10.

[15] S. Hidaka, M. Tisi, J. Cabot, Z. Hu, Feature-based classification of bidirectional transformation approaches, Software & Systems Modeling (2015) 1–22. URL: http://dx.doi.org/10.1007/s10270-014-0450-0. doi:10.1007/s10270-014-0450-0.

[16] A. Anjorin, T. Buchmann, B. Westfechtel, Z. Diskin, H. Ko, R. Eramo, G. Hinkel, L. Samimi-Dehkordi, A. Zündorf, Benchmarking bidirectional transformations: theory, implementation, application, and assessment, Softw. Syst. Model. 19 (2020) 647–691. URL: https://doi.org/10.1007/s10270-019-00752-x. doi:10.1007/s10270-019-00752-x.

[17] A. Schürr, Specification of graph translators with triple graph grammars, in: Graph-Theoretic Concepts in Computer Science, Springer, 1995, pp. 151–163.

[18] S. Hildebrandt, L. Lambers, H. Giese, J. Rieke, J. Greenyer, W. Schäfer, M. Lauder, A. Anjorin, A. Schürr, A survey of triple graph grammar tools, Electronic Communications of the EASST 57 (2013).

[19] H. Giese, R. Wagner, Incremental model synchronization with triple graph grammars, in: O. Nierstrasz, J. Whittle, D. Harel, G. Reggio (Eds.), Model Driven Engineering Languages and Systems, volume 4199 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2006, pp. 543–557.

[20] E. Leblebici, A. Anjorin, A. Schürr, Developing emoflon with emoflon, in: D. Di Ruscio, D. Varró (Eds.), Theory and Practice of Model Transformations, Springer International Publishing, Cham, 2014, pp. 138–145.

[21] 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:https://doi.org/10.1016/j.jss.2022.111288.

[22] P. Glaser, D. Bork, The biger tool - hybrid textual and graphical modeling of entity relationships in VS code, in: 25th International Enterprise Distributed Object Computing Workshop, EDOC Workshop 2021, Gold Coast, Australia, October 25-29, 2021, IEEE, 2021, pp. 337–340. URL: https://doi.org/10.1109/EDOCW52865.2021.00066. doi:10.1109/EDOCW52865.2021.00066.