SpecEdit: Projectional Editing for TLA+ Specifications

Riwan Cuinat
ENSTA Bretagne
Brest, France
riwan.cuinat@ensta-bretagne.org

Ciprian Teodorov
Lab-STICC, ENSTA Bretagne
Brest, France
ciprian.teodorov@ensta-bretagne.fr

Joel Champeau
Lab-STICC, ENSTA Bretagne
Brest, France
joel.champeau@ensta-bretagne.fr

***Abstract*—High quality requirements and specifications are the premises of efficient software system engineering. Formal approaches propose precise and unambiguous requirements amenable to automated reasoning. TLA+, for instance, is used by major companies, such as Microsoft and Amazon, to specify high-profile business critical systems. However, despite its undeniable strengths for the specification of complex distributed systems, TLA+ suffers from the duality of its syntax, which is likely to hamper its large-scale industrial adoption. A system engineer can easily read mathematical specifications in TLA+, produced through LaTeX. However, for writing TLA+ specifications, he must learn the discommoding ASCII syntax, which requires unnecessary effort and dedicated learning time.**

This paper introduces SpecEdit, an IDE for TLA+ with a projectional editor that solves this issue. SpecEdit exposes the mathematical syntax of TLA+ for both reading and writing specifications, without requiring external transformations. This approach minimizes the cognitive effort and streamlines the formal system specification process. We illustrate the benefits of our approach using the specification of the Elasticsearch cluster coordination module. We furthermore emphasize the complementarity with the existing TLA+ tools. Through SpecEdit, TLA+ gains the specification editor that was missing without compromising compatibility with the existing tools.

***Index Terms*—TLA+, projectional editor, formal specifications**

I. Introduction

To reach the goal of automatically analyzed requirements, a formalization is required and the resulting expressions must be closer to the expertise domain of the engineers. As noticed by formal methods users on industrial cases [1], formal specification based on simple discrete math with basic set theory and predicates notation is quite familiar to engineers. Numerous research efforts address this challenge through mathematical formalism equipped with powerful tools for reasoning and handling specifications [2].

Temporal Logic of Actions (TLA+) is a formal specification language created by Leslie Lamport [3]. TLA+ defines a temporal logic based on set theory that facilitates the specification of dynamic systems. TLA+ specifications are amendable to formal verification either through model-checking via TLC [4] or theorem proving using TLA+ Proof System (TLAPS) [5]. Particularly well adapted for the specification of distributed systems, TLA+ has been successfully used in both academia and industry. Amongst its industrial uses, we can cite its usage in production, for capturing the design requirements of todays most influential cloud infrastructures, S3 from Amazon [1], Azure from Microsoft [6] and Elasticsearch [7].

The transition from natural language requirements to formal specifications is not always as smooth as end-users might want it to be, even when provided with adequate analysis tools. As observed by Green [8], the alignment of the language with the domain greatly influences the ability to effectively express facts in that domain. In the case of TLA+, the language concepts and semantics are directly mapped to mathematics, offering the premises for both formalization and automation. However, at the syntactic level, TLA+ suffers from the duality of its syntax, which introduces a gap between the conceptual view of the specification and its encoding in ASCII. According to Green, this duality entails an arduous cognitive dimension [8] for the specification designer.

Moreover, TLA+ is targeted at non-programmers. Leslie Lamport answered in 2014 to some user's feedback [9]:

> "As for the "pretty-printed" version versus the ASCII, a TLA+ user at Intel wrote that one of the good things about TLA+ is that if he doesn't understand what a TLA+ construct means, he can look it up in a math book. **Math books don't write math in ASCII**, they use standard mathematical symbols. **I want TLA+ users to think in terms of math**, which means thinking in terms of its symbols. You will soon get to be bilingual, reading math and its TLA+ ASCII versions equally well."

He indicated at the time that it is inconceivable to give up the mathematical affiliation of TLA+, and that it is therefore inevitable to keep the mathematical notation in parallel with the code written in ASCII. This leads us to the following research question: **Is it possible to hide TLA+'s syntax duality in a viable bilingual Integrated Development Environment (IDE) to reduce the mental efforts of system engineers?** Such an IDE would expose only the mathematical syntax to the user, translating it to the ASCII version for ensuring compatibility with the existing tools.

At first glance this does not seem so trivial because the input device we use, the keyboard, does not allow for the direct input of special characters, like $\in$. Furthermore, the use of the Unicode ID (U+2208) or of complex key combinations needed for writing the corresponding Unicode characters are no better solution.

This paper introduces SpecEdit, an IDE with a projectional editor for TLA+ that solves this problem. SpecEdit lets the designer use standard mathematical symbols in the specifications. This approach is meant to minimize the mental effort and streamline the formal system specification process.

We illustrate the benefits of our approach using the openly available specification of the cluster coordination of Elasticsearch. We furthermore assess the complementarity of SpecEdit with respect to the existing TLA+ tools and emphasize some of the advantages of projectional editors for writing formal requirements and specifications.

Section II describes the issue addressed in this paper while presenting the related work. Section III presents SpecEdit's architecture discussing some of the difficulties encountered. Section IV illustrates our approach on a practical case-study and discusses its complementarity with the existing tools. Finally, Section V concludes this paper giving some future research directions.

II. Background And Related Work

The first part of this section introduces the TLA+ specification language, and the associated tools while underlying the *syntax duality*, which creates a gap between the conceptual specification and its TLA+ equivalent. In the second part, we overview the basics of domain-specific languages (DSL) design, emphasizing the relation to the problem addressed.

a) TLA+ Language and Ecosystem: TLA+ is a high-level specification language based on set theory and predicate logic, enriched with temporal logic primitives [3]. This combination of features renders the language well adapted for specifying and reasoning on complex dynamic systems, represented as sets of behaviors. The TLA+ syntax thus naturally finds its roots in mathematical notation and features more than 80 non-ASCII mathematical symbols. Concise but expressive, easy to write on paper and well understood by engineers, the mathematical syntax is however historically not well supported by text editors and IDEs. For TLA+, this led to the fallback of using mnemonic ASCII words, similar to the LaTeX notation. Consider, for instance, the subset operator '$\subseteq$' (U+2286 in Unicode), which is represented by the **\subseteq** mnemonic in the TLA+ syntax. Although listed in the language documentation [10], these mnemonics considerably steepen the learning curve, due to the need to lookup the notation corresponding to the wanted symbol. Moreover, once the mnemonic is inserted, the specification designer either has: *a)* to become accustomed with thinking in terms of ASCII notation or, *b)* to continuously generate and compile the LaTeX-based PDF representation of the specification.

The TLA+ Toolbox [11], is a free and open-source IDE for TLA+. The TLA+ Toolbox provides a set of tools for manipulating TLA+ specifications: *a)* specification editor; *b)* explorer, for specification navigation; *c)* model editor, for extracting closed specifications for model-checking; *d)* trace explorer, for trace-based diagnosis. The TLA+ Toolbox is coupled with the TLC model-checker [4] and TLAPS [5] to create a state-of-the-art formal specification ecosystem. However, the specification editor lacks not only the much needed mathematical symbols but also most of the features proposed by the current IDEs [12] and implemented in SpecEdit.

b) Domain-Specific Language Design: Domain-specific language design focuses on finding a common language capable of finely capturing a domain of interest. From this perspective, TLA+ can be seen as a DSL for specifying interacting dynamic systems (distributed systems). The two main ingredients of a domain-specific language design are *syntax* and *semantics*. The syntax defines the words in the language, either conceptually (abstract syntax) or concretely (concrete syntax). Semantics gives meaning to the words. To facilitate the formalization of semantics usually an *abstract syntax* is defined (a metamodel), which captures a conceptual view of the words of the language. In this case, the *concrete syntax* only establishes an interface between the DSL user and the *abstract syntax*. The *abstract syntax* is a pivot language on which we can define multiple views (concrete syntaxes). In this study, we are particularly interested in the interface (concrete syntax) between the specification designer and the TLA+ language.

Typically, to ease the implementation effort, the concrete syntax definition is based on compiler-compiler tools, such as ANTLR (ANother Tool for Language Recognition) [13] to transform a grammatical definition of the syntax to a parser, which instantiates the abstract syntax. The TLA+ syntax problem can be partially solved by defining a Unicode-based grammar, which uses mathematical symbols. However, to address the Unicode symbol input problem, the user will still have to rely on mnemonics interpreted by the text editor.

Intentional programming [14], offers another solution for implementing the concrete syntax of DSLs through the use of projectional editors. Projectional editors allow the direct use of the abstract syntax. To view the model, the abstract syntax is projected on a user interface (a view is created). To instantiate the needed model elements the user intention is projected to abstract syntax operations via User Interface actions. JetBrains' MetaProgramming System (MPS) [15] offers a DSL design workbench based on projectional editing. The produced editors allow the mixing of graphical, textual and mathematical notation, which is particularly adapted for the *syntax problem* of TLA+. Moreover, the modularity of projectional editing allows the creation of various model views enabling further specialization of the TLA+ editor.

III. A Projectional Editor for TLA+

This section overviews the architecture of SpecEdit discussing the creation of a basic projectional editor as well as the improvements needed for a better user experience and compatibility of the IDE with existing tools.

A. SpecEdit's Architecture

SpecEdit solves the syntax duality problem of TLA+ by providing an IDE with a projectional editor integrating modern features such as syntax highlighting and autocompletion.

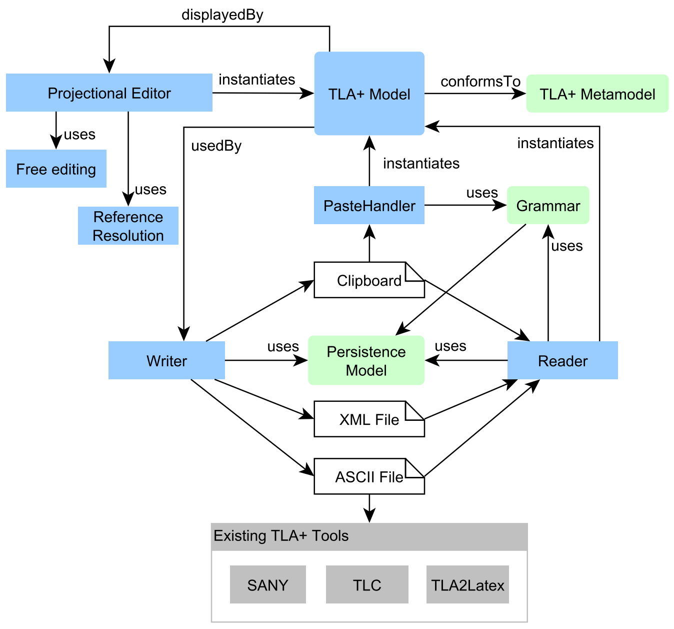
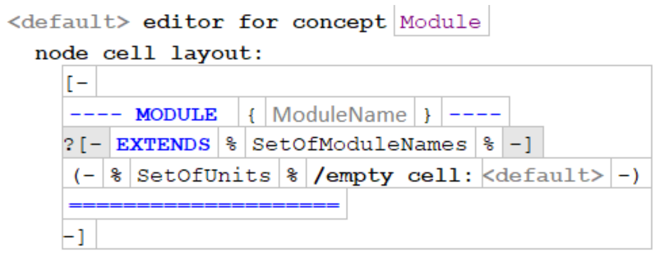


Fig. 1. Underlying architecture of SpecEdit



```
<default> editor for concept Module
  node cell layout:
    [-
    ---- MODULE { ModuleName } ----
    ?[- EXTENDS % SetOfModuleNames % -]
    (- % SetOfUnits % /empty cell: <default> -)
    =====================
    -]
```

Fig. 2. MPS Editor definition for TLA+ Module

Furthermore, SpecEdit offers a flexible backend that preserves the compatibility with the existing tools.

An overview of the high-level architecture of SpecEdit is shown in Figure 1. In the Figure, the core components of SpecEdit are emphasized by the green (representing metamodels) and blue boxes (representing the tools). The *Projectional Editor* displays the *TLA+ model* that is instantiated based on user input. As usual, the model *conformsTo* the metamodel. For SpecEdit, we created a syntax-driven *TLA+ metamodel*, to finely capture user intentions. SpecEdit supports *Free editing* and *Reference resolution*, besides other features inherited from the underlying language workbench, such as syntax highlighting and refactoring (not shown in the Figure). The backend is composed of the *PasteHandler*, the *Writer*, and the *Reader*. The *PasteHandler* enables on-the-fly conversion from ASCII TLA+ specifications (stored in the clipboard) to instantiated TLA+ model nodes. The *Writer* represents the serialization modules, which, based on the *Persistence model*, outputs different file-based representations of the model. The *Reader* represents the file-input modules, which can instantiate the *TLA+ Model* according to the *Persistence model* (and after parsing the files according to the respective *Grammar* rules). SpecEdit supports copy-paste, both internally by exchanging AST nodes through the *Writer-Clipboard-Reader* path, and externally by processing plaintext coming from the system clipboard via the PasteHandler. SpecEdit backend supports TLA+ ASCII files (which is the standard supported by the existing TLA+ tools) and an XML-based serialization format (easy to parse, internal format).

SpecEdit is implemented with the MPS language workbench, a mature, commercially supported technology for DSL design, which facilitates the creation of IDEs with a projectional editor. The first step towards the creation of an IDE with a language workbench, such as MPS, consists in the definition of the language (abstract syntax, concrete syntax, pretty-printing) with the tools of the workbench. Section III-B describes the process of defining TLA+ as a Language in MPS.

Projectional editors rely extensively on the abstract syntax, and allow direct editing of the underlying model. Users do not write code, but they instantiate metamodel elements (AST nodes) via numerous specialized editors. MPS gives the possibility to use the completion menu to select a new node to be instantiated. MPS will thus show, in the menu, only nodes that can be instantiated at the cursor position. However, this process is slow and very frustrating when not combined with free editing. To overcome this limitations, SpecEdit implements free editing support on top of the MPS specialized editors. *Free editing*, combined with *Reference Resolution* (Section III-C), optimizes the user experience.

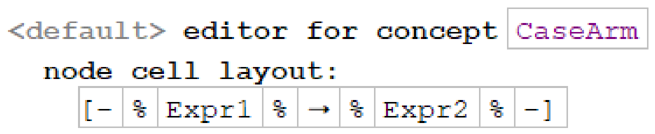
By default, MPS saves models as XML files, which are easy to parse and manipulate programmatically. However, to ensure the compatibility with the existing TLA+ tools, two custom modules were created, namely a *PasteHandler* and a custom ASCII persistence model (section III-D).

B. Basic Projectional Editor for TLA+

Since MPS frees programmers from defining a grammar for their languages, the MPS Structure Language is provided as an alternative. Concepts in MPS, defined via the Structure Language, represent the abstract syntax (types and hierarchy of AST nodes) and they reference children nodes, parent nodes, properties (of primary data types), etc.

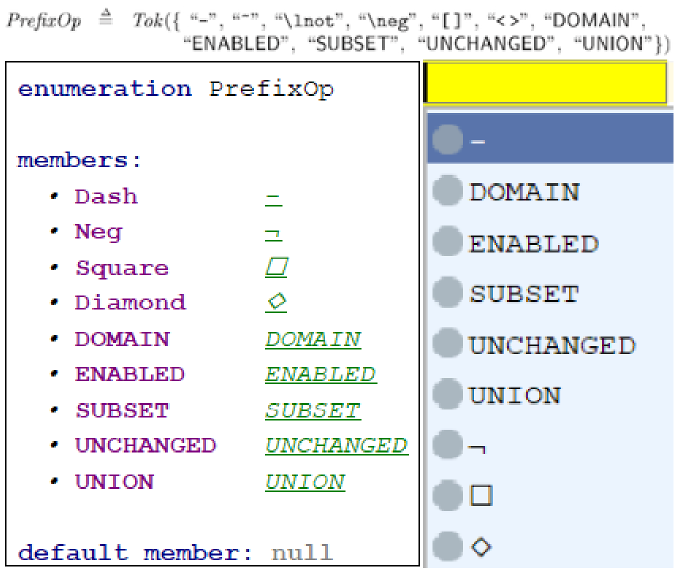
To define the underlying language metamodel in MPS, the TLA+ grammar (defined explicitly in [10]) was converted into a set of MPS Concepts. The extraction of the language Concepts from the concrete syntax resulted in a metamodel with 110 interconnected Concepts.

For the definition of the concrete syntax, each Concept was associated with a specific MPS Concept Editor. From the model-view-controller design-pattern perspective, these Concept Editors play the role of “views” and “controllers” for the associated language Concepts (“models”). An Editor is described by the cells it is composed of, like a template. Figure 2 illustrates an Editor for a TLA+ Module. In blue are represented read-only fields. ModuleName designates a mandatory string that must be entered by the user, SetOfMod-



```
<default> editor for concept CaseArm
  node cell layout:
    [- % Expr1 % → % Expr2 % -]
```

Fig. 3. MPS Editor definition for TLA+ CaseArm



$PrefixOp \triangleq Tok(\{$ "-", "⁻", "\lnot", "\neg", "[]", "<>", "DOMAIN", "ENABLED", "SUBSET", "UNCHANGED", "UNION"$\})$

```
enumeration PrefixOp

members:
  • Dash        -
  • Neg         ¬
  • Square      □
  • Diamond     ◇
  • DOMAIN      DOMAIN
  • ENABLED     ENABLED
  • SUBSET      SUBSET
  • UNCHANGED   UNCHANGED
  • UNION       UNION

default member: null
```

Fig. 4. MPS Enum for TLA+ PrefixOp

uleNames optional strings and SetOfUnits a set of expressions and statements that may constitute the TLA+ Module.

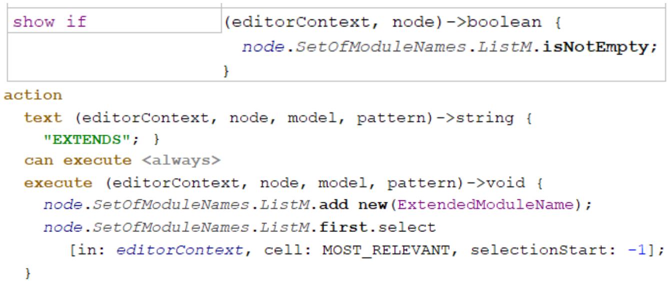
Since Unicode characters are supported by MPS, they were used to customize the Editors and thus to obtain a rendering that integrates the mathematical notation. In Figure 3, the case branch is automatically instantiated, by the customized Editor, with an $\rightarrow$ (instead of the ASCII $->$) as a read-only field.

To allow users to insert symbols, in fields that accept inputs (direct input or completion), MPS Enums were used. Enumerations in MPS allow to define properties with values from pre-defined sets. The possible values are shown, in a context-menu, if the given values can be inserted at the current cursor position according to the metamodel. Figure 4 shows the mapping between the TLA+ grammar rule ”PrefixOp” (top) and the corresponding MPS Enum node (left) as well as the rendering in the completion menu when prompted to users (right). Note for instance the suggestion of the $\diamond$ symbol, representing an eventuality in temporal logic, instead of the $<>$ string.

C. Customization of user experience

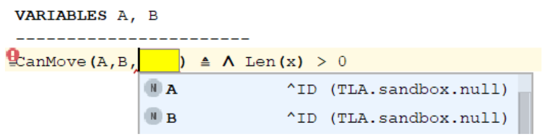
This section discusses the solutions implemented in SpecEdit for free editing and reference resolution support.

1) Free editing: In MPS, by default, free editing is prohibited. This means, that, unless the user is given a predefined box, users cannot use the keyboard to write code. To allow free editing, programmers have to make fields editable in MPS Editors. Once editable, users will be able to write



```
show if        (editorContext, node)->boolean {
                 node.SetOfModuleNames.ListM.isNotEmpty;
               }
action
  text (editorContext, node, model, pattern)->string {
    "EXTENDS"; }
  can execute <always>
  execute (editorContext, node, model, pattern)->void {
    node.SetOfModuleNames.ListM.add new(ExtendedModuleName);
    node.SetOfModuleNames.ListM.first.select
      [in: editorContext, cell: MOST_RELEVANT, selectionStart: -1];
  }
```

Fig. 5. Example of optional field management with TLA+ Module import



```
VARIABLES A, B
------------------------
CanMove(A,B,    ) ≜ ∧ Len(x) > 0
             A          ^ID (TLA.sandbox.null)
             B          ^ID (TLA.sandbox.null)
```

Fig. 6. Completion menu with referenced identifiers

freely with their keyboard but MPS will point to syntax errors since writing with the keyboard does not instantiate new Concepts. To solve this issue, predefined strings were mapped to the Concepts as aliases (via MPS Aliases and the Transformation Menu Language which allows to trigger actions when given patterns are identified). Not only do aliases appear in completion menus and context assistants but MPS also instantiates the corresponding Concepts in the current model the user is editing when the alias or a part of it (if no other possible match) is typed in.

Another editing issue is hiding optional model elements, when they are not needed. For instance, in TLA+, a module can optionally extend other modules. In Figure 2 the optional block is shown as a set of cells preceded by a question mark. In the case of a standalone TLA+ module, the ”EXTENDS” clause should be hidden. However, if it is hidden, there is no mechanism defined to allow users to make it appear and thus to import modules. In MPS this user intention is captured through ”display conditions” and ”side tranform actions”, which allow to implement specific mechanisms in the editor.

Figure 5 illustrates this principle. A Transformation Menu can be invoked by a user typing “EXTENDS” right to the cell to which it has been associated. It leads to the instantiation of a new element in the list of extended module names. The condition in the “show if” property is then verified, and the hidden field is unhidden.

2) Reference Resolution: Syntax predictions are an asset to be used in parallel with free editing. Predictions are not only meant to suggest AST nodes to instantiate but also strings to fill in editable fields. Reference resolution is one of the functionalities meant to enhance the predictions provided by the IDE.

In MPS, a reference creates a link between two nodes of the AST outside the tree containment hierarchy. For instance, a “VariableDeclaration” node is contained in a ”Module”, but can be referred to, using a “VariableReference” node, from any following definition. The principle is thus to use pointers in Editors targeting the variable declaration identifier. The main goal is to improve user experience by automatically providing (in the context menu) the identifiers (or names) defined in the specification scope. Figure 6 shows that when editing the "CanMove" definition, the user is presented with the previously defined variables A and B. Furthermore, connecting the concepts through references enables refactoring transformations, such as renaming, which applies seamlessly to all occurrences of the reference.

D. *Plaintext support with ANTLR*

This section discusses pasting and file-loading and saving in SpecEdit; three simple actions which need special attention in projectional editors.

1) Paste handler: When a user tries to paste an element into the IDE, a background routine is in charge of retrieving the clipboard content and inserting it in the code editor. This happens transparently when working with text. In a projectional editor, however, the text from the clipboard should be first parsed and interpreted to instantiate the corresponding nodes in the model.

SpecEdit uses the ANTLR parser-generator framework and a TLA+ grammar definition to process plaintext. ANTLR builds a parse tree and generates a skeleton visitor class containing methods for traversing parse trees. In the case of SpecEdit, the corresponding visitor class is subclassed and each visit method overridden to instantiate MPS Concepts. The TLA+ Concepts defined are programmatically accessible within MPS. As opposed to regular transpiling, in the case of SpecEdit, the input and output language are the same. Their respective definitions are however different (ASCII grammar versus MPS Language).

To integrate the modules generated with ANTLR into MPS and consequently into SpecEdit, a Java archive containing the compiled code and the various dependencies (like the parser, lexer and visitor) was created and imported under a new solution in MPS (as a stub model). Based on this process, we created an MPS plugin inserting a new entry in the context menu. Thanks to this plugin, when the user clicks on the new entry in the menu, a method retrieving the content of the clipboard is called and checks that the retrieved text verifies a given pattern. The text retrieved from the clipboard, which is tokenized, parsed, visited and mapped with MPS Concepts allows the instantiation of an AST.

2) Custom persistence model: By default, models in MPS are saved in a proprietary XML-based format. The idea we had was to create a custom persistence model allowing to remove any formatting specific to the IDE in order to save TLA+ source code files directly in plaintext and ensure the compatibility of SpecEdit with the existing TLA+ tools. This implied modifying the reading (opening files) and writing (saving files) procedures of the IDE.

The approach followed is very similar to the one used for the realization of the paste handler. It involved using the modules generated using ANTLR to ensure the transpiling. As in the case of the paste handler, a new dedicated MPS plugin build solution was created that imports a custom persistence model. Classes were created to override the persistence logic, encapsulate the parsing and visiting procedures and implement the different interfaces that are essential for dealing with the internal working mechanisms of MPS. [16]

Apart from the import source, which is a file and not the clipboard, the processing principle for loading ASCII TLA+ files is exactly the same as for the paste handler (lexing, parsing, visiting, mapping). Note however that we created an explicit plugin descriptor for MPS to be aware that this plugin provides a model factory. The writing procedure is delegated to text generators implemented via MPS TextGens.

The two import approaches selected, namely pasting and customizing the persistence model, were chosen because of their respective merits. While importing unstructured models in MPS (i.e. models written in plaintext) is essential from a user's point of view, being able to insert pieces of TLA+ code from external editors into ongoing MPS projects is also useful.

Finally, the basic projectional editor, introduced in Section III-B, once tuned-up, became SpecEdit, a TLA+ IDE which solves the syntax duality problem of TLA+ without compromising either the user experience or the compatibility with the existing tools.

IV. SpecEdit in Practice

SpecEdit benefits extensively from the architecture provided by MPS. The resulting multiplatform standalone IDE provides:

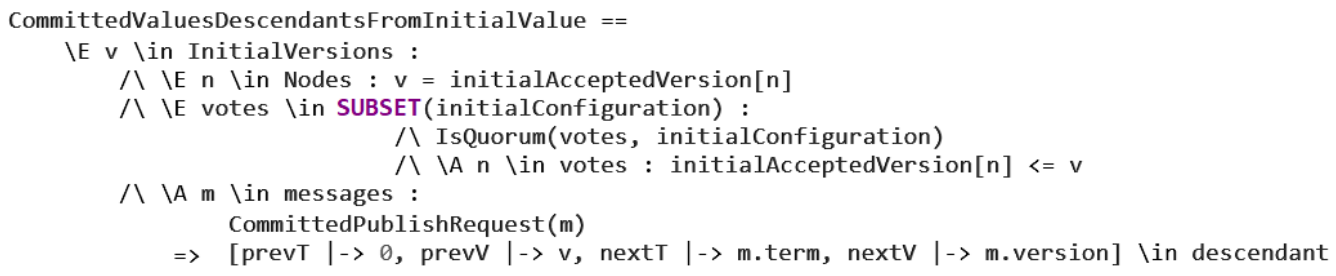
- Basic features of an IDE (syntax highlighting, tree view, autocompletion, predictions, syntax verification, reference resolution, free editing support, etc.);
- Mathematical notation support, freeing TLA+ users from the ASCII syntax;
- Plain text support, based on ANTLR modules, MPS TextGens and a custom persistence model, which preserve the compatibility with other tools.

The GitHub repository of SpecEdit is available at github.com/RiwanC/SpecEdit together with a demo video which illustrates the features offered by the IDE as well as its usability.

A. *Concrete example*

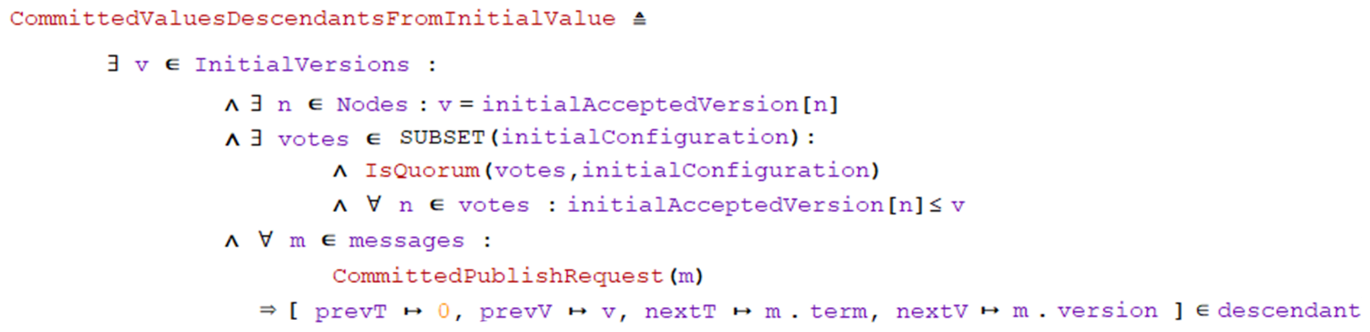
Elasticsearch is a distributed search engine developed in Java. Distributed coordination is a problem the creators of the engine faced and that is known to be difficult to solve [17]. The Elasticsearch designers relied on TLA+ for the specification and the validation of their cluster coordination module. The full TLA+ specification (ASCII syntax) can be found in the Elastic GitHub repository [18]. Its rendering inside SpecEdit is available in the GitHub repository of SpecEdit.

The meaning of this specification, discussed in [7], is out of the scope of this article. However, for illustration purposes we have selected one definition from this specification, which is presented in Figure 7. For comparison, Figure 8 illustrates the same definition in SpecEdit. Compared to the ASCII specification, note the use of the standard mathematical symbols for: *a)* the definitions, which are introduced by the "equal by definition" operator $\triangleq$ instead of the $==$ (line 1 in the



```
CommittedValuesDescendantsFromInitialValue ==
    \E v \in InitialVersions :
        /\ \E n \in Nodes : v = initialAcceptedVersion[n]
        /\ \E votes \in SUBSET(initialConfiguration) :
                            /\ IsQuorum(votes, initialConfiguration)
                            /\ \A n \in votes : initialAcceptedVersion[n] <= v
        /\ \A m \in messages :
                CommittedPublishRequest(m)
            =>  [prevT |-> 0, prevV |-> v, nextT |-> m.term, nextV |-> m.version] \in descendant
```

Fig. 7. Illustration of an excerpt from a TLA+ Elasticsearch algorithm rendering in the editor of the TLA+ Toolbox



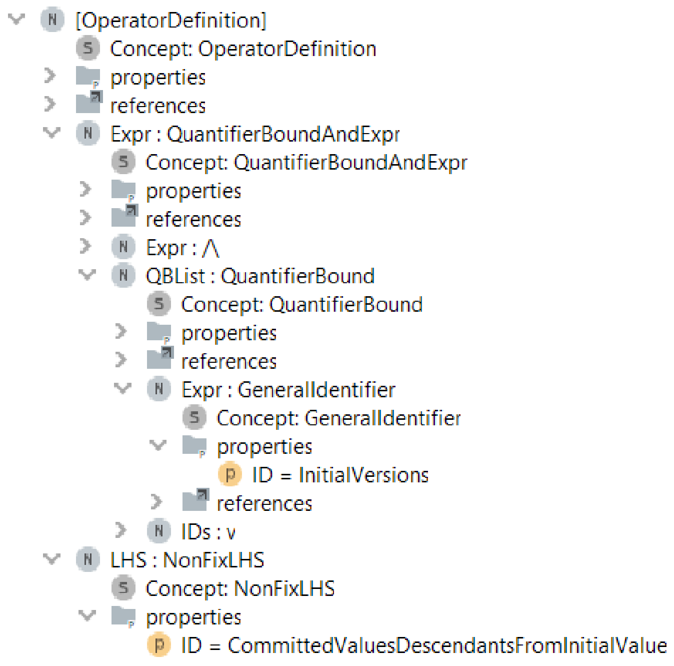
```
CommittedValuesDescendantsFromInitialValue ≜

        ∃ v ∈ InitialVersions :

                ∧ ∃ n ∈ Nodes : v = initialAcceptedVersion[n]
                ∧ ∃ votes ∈ SUBSET(initialConfiguration):
                        ∧ IsQuorum(votes,initialConfiguration)
                        ∧ ∀ n ∈ votes : initialAcceptedVersion[n] ≤ v
                ∧ ∀ m ∈ messages :
                        CommittedPublishRequest(m)
                  ⇒ [ prevT ↦ 0, prevV ↦ v, nextT ↦ m . term, nextV ↦ m . version ] ∈ descendant
```

Fig. 8. Illustration of an excerpt from a TLA+ Elasticsearch algorithm rendering in SpecEdit

example); *b)* the standard logic operators, $\wedge$ instead of /\ or \land for conjunction (for readability, TLA+ provides a unary conjunction operator, to be used in long conjunction chains, line 3 in the example), $\Rightarrow$ instead of $=>$ for implication (last line in our example); *c)* comparison operators, $\leq$ instead of $<=$ for less-or-equal (note the ambiguity with the right-to-left logical implication in the ASCII version); *d)* the existential and universal quantifiers, $\exists$ and $\forall$ instead of \E and \A respectively; *e)* the set inclusion operator, $\in$ instead of \in (present in most lines); *f)* the mapping operator for tuples, $\mapsto$ instead of $|->$ (last line in Figure 8). Not having to deviate from mathematical writing (in both writing and reading) is what streamlines the engineering process and reduces the mental effort. Without providing a full textual description of this definition, note that the first line in SpecEdit reads like standard mathematical formulas: "*Exists v in InitialVersions such that ...*". The colon operator ":" reads as "such that" in TLA+. In this case, similarly to the TLA+ export to LaTeX, we decided to preserve it, although using another symbol can easily be achieved within SpecEdit.

SpecEdit provides bidirectional compatibility with the existing TLA+ tools. Since SpecEdit provides an ASCII export option, users can edit the specifications in SpecEdit before importing them in the TLA+ Toolbox [11] for further manipulation, like model-checking with TLC [4]. Moreover, the ASCII support provided by the ANTLR modules enables importing existing specifications in SpecEdit.

Through projectional editing, SpecEdit will guide the user into writing syntactically correct specifications. Since the user input is projected directly on the abstract syntax, it is impossible to have syntax errors. SpecEdit is meant to delegate the semantic verification to the existing TLA+ backend, named SANY, which performs all necessary verifications. The virtue of this approach is that, regardless of the editor, all specifications are validated by a unique semantic verification engine, prohibiting semantic divergence between the editors and the analytic tools.



```
[OperatorDefinition]
    Concept: OperatorDefinition
    properties
    references
    Expr : QuantifierBoundAndExpr
        Concept: QuantifierBoundAndExpr
        properties
        references
        Expr : ∧
        QBList : QuantifierBound
            Concept: QuantifierBound
            properties
            references
            Expr : GeneralIdentifier
                Concept: GeneralIdentifier
                properties
                    ID = InitialVersions
                references
            IDs : v
    LHS : NonFixLHS
        Concept: NonFixLHS
        properties
            ID = CommittedValuesDescendantsFromInitialValue
```

Fig. 9. Tree view of an excerpt from a TLA+ Elasticsearch algorithm in SpecEdit

SpecEdit is open and extensible, the state-of-the-art MPS infrastructure is mature and maintained, supporting many extensions from language generators to Application Programming Interfaces (APIs) which would allow to expose novel functionalities to the user. Consider, for instance the tree-like representation of an excerpt from the Elasticsearch specification, in Figure 9. This view, produced by MPS, closely follows the internal syntactic representation of the specification. Like most of the SpecEdit features inherited from MPS, this view can be customized to create a specification explorer, which would hierarchically show the content of specifications.

B. Lessons Learned

The integration of plaintext support in SpecEdit ensures that the benefits of projectional editing are maximized while guaranteeing the compatibility with legacy tools. The major advantages of the projectional approach and MPS are to be able to dissociate the model from the view and to have a high composability of the language definition modules.

Merging the projectional approach with parsing nevertheless bears its limits. The concern is that each time the underlying structure of the language is modified both the projectional editor and the ANTLR modules are impacted. TLA+ however, is a mature language with a stable syntax, thus, the underlying abstract syntax is considered stable. Since the ANTLR grammar maps the parse-tree to the TLA+ Concepts, the Editors can be modified without creating conflicts with ANTLR modules.

The projectional approach is not the most widespread approach in code editors nowadays. It however has the potential to play an important role in the future to map DSL definition to any syntax. This approach efficiently solves the difficulties of translating requirements into formal specifications. In the case of SpecEdit, the use of MPS as a backbone for the creation of a tool dedicated to TLA+ has proved particularly fruitful. Not only does MPS offer a complete customizable architecture, but also provides access to non-trivial mechanisms for advanced users. SpecEdit transcends the current duality of the syntax of TLA+ and promises improvements that will facilitate the daily work of systems engineers to bridge the gap between conceptual view and syntax. The transition from one tool to another is not an easy matter in daily professional life. The projectional editor, though self-sufficient, combined with a traditional parsing approach, addresses this concern in the specific context of TLA+ by providing a bridge between SpecEdit and the existing TLA+ tools.

V. Conclusion and Perspectives

SpecEdit provides a solution that unifies the two existing syntaxes of TLA+ and eases the work of engineers. To do so, not only was it necessary to formalize a new model of the language but also was it crucial to work on the input modes provided to the users. Not yet a full-fledged IDE, SpecEdit nonetheless is a viable TLA+ specification editor, as illustrated on the Elasticsearch case-study. It is meant to epitomize what can be achieved through projectional technology for improving the experience of systems engineers using specification languages such as, but not limited to, TLA+.

Some future research directions include the use of model-federations [19] to ensure the traceability of textual requirements translated into TLA+ specifications via SpecEdit. We are also considering implementing various projections for TLA+ specifications, based on tabular/graphical Editors.

References

[1] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How amazon web services uses formal methods," *Communications of the ACM*, vol. 58, no. 4, pp. 66–73, 2015.

[2] J.-M. Bruel, S. Ebersold, F. Galinier, A. Naumchev, M. Mazzara, and B. Meyer, "The role of formalism in system requirements (full version)," 2019. [Online]. Available: https://arxiv.org/abs/1911.02564v6

[3] L. Lamport, "The temporal logic of actions," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, p. 872–923, May 1994.

[4] Y. Yu, P. Manolios, and L. Lamport, "Model Checking TLA+ Specifications," in *Correct Hardware Design and Verification Methods*, L. Pierre and T. Kropf, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 54–66.

[5] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz, "A TLA+ Proof System," in *Knowledge Exchange: Automated Provers and Proof Assistants (KEAPPA)*, Doha, Qatar, 2008. [Online]. Available: https://hal.inria.fr/inria-00338299

[6] D. Langworthy, "TLA+ at Microsoft: 16 Years in Production," in *TLA+ Conference (keynote), St. Louis, MO, USA*, 2019.

[7] Y. Welsch, "Using TLA+ for fun and profit in the development of Elasticsearch," in *TLA+ Conference (keynote), St. Louis, MO, USA*, 2019.

[8] T. R. G. Green, "Cognitive dimensions of notations," in *Proceedings of the Fifth Conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and Computers V*. USA: Cambridge University Press, 1990, p. 443–460.

[9] Google Groups. (2020) Some user feedback. [Online]. Available: https://groups.google.com/forum/#!msg/tlaplus/WdDT5sKY7rI/t_jswui_GkcJ

[10] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[11] M. A. Kuppe, L. Lamport, and D. Ricketts, "The TLA+ Toolbox," *Electronic Proceedings in Theoretical Computer Science*, vol. 310, p. 50–62, Dec 2019.

[12] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning, "The state of the art in language workbenches," in *Software Language Engineering*, M. Erwig, R. F. Paige, and E. Van Wyk, Eds. Cham: Springer International Publishing, 2013, pp. 197–217.

[13] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, 2013.

[14] C. Simonyi, "The death of computer languages, the birth of intentional programming," Tech. Rep. MSR-TR-95-52, September 1995. [Online]. Available: https://www.microsoft.com/en-us/research/publication/the-death-of-computer-languages-the-birth-of-intentional-programming/

[15] M. Voelter and V. Pech, "Language modularity with the mps language workbench," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 1449–1450.

[16] JetBrains. (2020) Custom persistence cookbook. [Online]. Available: https://www.jetbrains.com/help/mps/custom-persistence-cookbook.html

[17] S. Ossowski and R. Menezes, "On coordination and its significance to distributed and multi-agent systems," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 4, pp. 359–370, 2006.

[18] Y. Welsch and D. Turner. (2019) Zenwithterms.tla. [Online]. Available: https://github.com/elastic/elasticsearch-formal-models/blob/master/ZenWithTerms/tla/ZenWithTerms.tla

[19] F. R. Golra, F. Dagnat, J. Souquières, I. Sayar, and S. Guerin, "Bridging the gap between informal requirements and formal specifications using model federation," in *Software Engineering and Formal Methods*, E. B. Johnsen and I. Schaefer, Eds. Cham: Springer International Publishing, 2018, pp. 54–69.