# Comparison of Feature Implementations across Languages, Technologies, and Styles

Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz, and Andrei Varanovich
Software Languages Team, University of Koblenz-Landau, Germany

***Abstract*—We describe and validate a method for comparing programming languages or technologies or programming styles in the context of implementing certain programming tasks. To this end, we analyze a number of 'little software systems' readily implementing a common feature set. We analyze source code, structured documentation, derived metadata, and other computed data. More specifically, we compare these systems on the grounds of the NCLOC metric while delegating more advanced metrics to future work. To reason about feature implementations in such a multi-language and multi-technological setup, we rely on an infrastructure which enriches traditional software artifacts (i.e., files in a repository) with additional metadata for implemented features as well as used languages and technologies. All resources are organized and exposed according to *Linked Data* principles so that they are conveniently explorable; both programmatic and interactive access is possible. The relevant formats and the underlying ontology are openly accessible and documented.**

***Index Terms*—Programming languages; Programming technologies; Programming style; Software chrestomathies; Software metrics; Feature location; Reverse engineering; *Linked Data***

## I. Introduction

Consider the following research question: '*Which programming language or technology or style is more suited for implementing a certain programming task?*'

For instance, let us pick the task of 'totaling salaries of employees in a company', which is a possible functional requirement for an information system. This task could be implemented in any given language in many different styles, making use of many different programming technologies. The task could be implemented in *Java* in such a manner that the company structure is represented in either plain objects or *DOM*-based objects for *XML*. The latter option is illustrated by using the *JDOM*[1] API:

```
// doc represents an input XML document
Iterator<?> iterator =
  doc.getDescendants(new ElementFilter("salary"));

// Iterate over all salary elements
while (iterator.hasNext()) {
  Element elem = (Element)iterator.next();
  Double salary = Double.valueOf(elem.getText());
  total += salary;
}
```

Consider another implementation of the task; this time in *Haskell* while assuming a designated data model for the company structure and the use of a generic programming style ('Scrap your boilerplate' (SYB) [1]) for processing the data, also subject to a designated library.[2] Thus:

```
-- Traverse "everything" to aggregate all floats (salaries)
total :: Company -> Float
total = everything (+) (extQ (const 0) id)
```

We want to compare many such different implementations. In principle, feature implementations can be discovered using either static text retrieval techniques [2], dynamic program analysis [3], or combinations [4]. For the shown samples, it may be straightforward to locate feature 'Total'. Consider another feature regarding the hierarchical organization of the company structure in terms of top-level departments breaking down hierarchically into sub-departments. We refer to this feature as 'Hierarchical company'. At the code level, this feature may be associated with a recursively defined type for departments. The *Haskell* implementation does indeed define such a data model (not listed here). The feature's implementation is *implicit* though in the *JDOM*-based *Java* implementation because of the lack of an explicit data model.

We describe and execute a methodology for the comparison of feature implementations across languages, technologies, and styles. We leverage the 101companies project[3] (or just '101') [5], as it provides a suitable chrestomathy [6], i.e., a collection of systems exercising different languages, technologies, and styles, while being comparable in terms of implemented features. The comparison relies on feature location and metrics calculation. 101's infrastructure is readily leveraged for all required analysis. All resources including available metadata and computed information are organized and exposed according to *Linked Data* principles [7], [8] so that they are conveniently explorable; both programmatic and interactive access is possible. The relevant formats and the underlying ontology are openly accessible and documented. The methodology is 'context-aware' in that different viewpoints and interactions are considered for feature location. (We adopt this meaning of context awareness from [9].) For instance, we distinguish 'as intended' features versus 'as implemented', i.e., features specified by the system documentation versus features located by the analysis of source code.

Eventually, we compare the same feature set in different systems (i.e., implementations or 'contributions' according

[1] http://www.jdom.org/

[2] http://hackage.haskell.org/package/base-4.6.0.1/docs/Data-Data.html

[3] http://101companies.org/

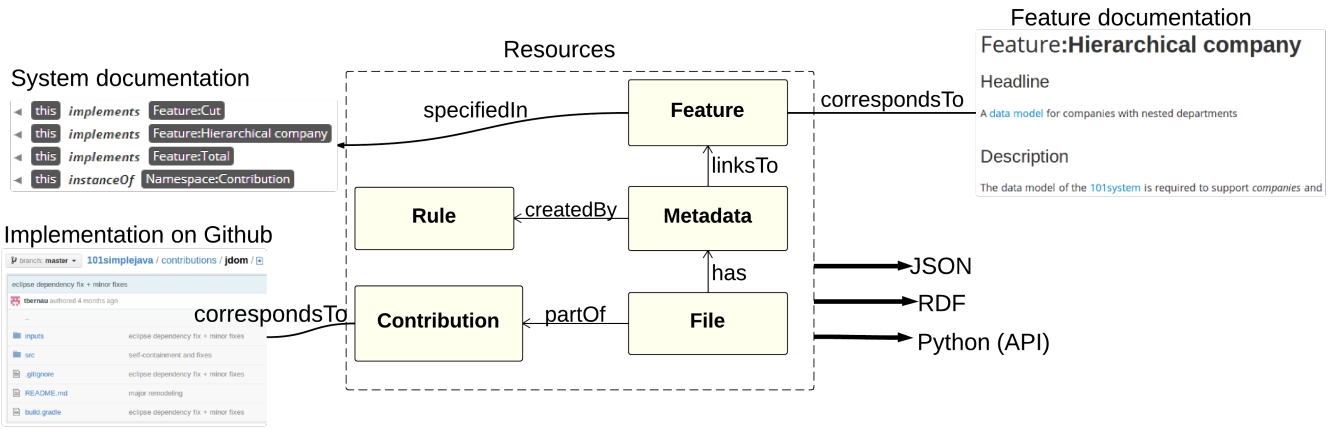


Fig. 1. Overview of the underlying infrastructure hinting also at 'links' in the sense of *Linked Data*

to 101) in terms of the NCLOC metric. We delegate more advanced metrics to future work. The methodology includes non-automated aspects for the sake of selecting targets for comparison, the validation of results, and their interpretation.

*Road-map:* The rest of this paper is structured as follows. §II briefly describes the underlying infrastructure including *Linked Data* aspects. §III describes the methodology for comparison. §IV executes the methodology in one particular way. §V describes the results, thereby returning to the research question stated at the beginning of this section. §VI discusses the research directions for future work. Related work is discussed in §VII and the paper is concluded in §VIII.

## II. The Underlying Infrastructure

The infrastructure for creating, analyzing and accessing resources is briefly summarized in Figure 1. We expose and access all involved resources according to *Linked Data* principles [7], [8]:

1) Use URIs as names for 'things'.
2) Use HTTP URIs so that people can look up names.
3) Provide useful information in the HTTP response.
4) Include links to other 'things' to make them discoverable.
5) Use standards for response formats and query languages (RDF, SPARQL, etc.).

System documentation is provided by a wiki platform—the 101wiki. A textual description is enriched with a metadata section for semantic properties; see the upper left corner of Figure 1. Authoring such properties is part of the structured documentation process that goes beyond plain source-code documentation. For instance, some properties specify the features thought to be implemented by the system. Such knowledge is exposed through 'links'—some of them are shown in Figure 1. Features are also documented on the 101wiki; see the upper right corner of Figure 1.

Source code is maintained in a (GitHub-based) repository—the 101repo; see the lower left corner of Figure 1. We use a computational infrastructure, the 101worker, to analyze source files for the sake of deriving resources and dumps, which are again published. In the present paper, we are specifically interested in computed metadata as follows: *a)* the features implemented by source files, subject to feature location; *b)* the languages and technologies used by the source files, subject to simple heuristics; *c)* data for source-code metrics, in fact, NCLOC.

To produce *a)* and *b)* we use simple, automated rules expressed in 101meta—a language for associating metadata units with files [10]. For instance, the Java code from the paper's introduction is associated with these units, rendered here in JSON notation:

```
[ { "language" : "Java" },
  { "technology" : "JDOM" },
  { "feature" : "Total" } ]
```

To compute derived resources such as metrics data in the sense of *c)* we plug designated modules into the 101worker. There is a metrics module which associates the source file $f$ (i.e., a 'primary' resource) with a 'derived' resource $f$.metrics.json with metrics data formatted again in JSON.

## III. Methodology

We use the following methodology for comparison of feature implementations across languages, technologies, and styles in terms of the diversity present in 101:

*Feature selection:* Determine a set of features that are often enough implemented to be promising in terms of a comparison. There are some features in 101, which are indeed very popular. The feature set needs to be small enough to make it relatively easy to validate the correctness of all results, as manual validation and interpretation is necessary; see below.

*Implementation selection:* Determine a set of implementations (contributions in 101's terminology) that are promising in terms of comparison. We can hardly assume that feature location and other algorithmic aspects of the methodology are completely robust across the diversity at hand. Thus, the set of selected implementations must be small enough to allow validation. Here we note that 101's contributions often modularize feature implementations with one source file per feature, thereby enabling straightforward consideration of implementations even with supersets of the selected features.

```
# Walk over all files of all contributions and build a mapping from features to files
featureIndex = {}
for folders, files in walk(Namespace('contributions')):
   for file in files:
      for feature in file.features:
         featureIndex.setdefault(feature, []).append(file)

# Validation step
validateAutomaticTagging(featureIndex)
for contribution in config.selectedImpls:
   for f in (set(contribution.implements) & config.selectedFeatures):
      if not any(file.member == contribution for file in featureIndex.get(f, [])) and
         not f in config.implicit.get(contribution.name, []):
            ... # Feature not found, error handling kicks in

# Count NCLOC for every file that belongs to a selected implementation and is concerned with a selected feature
contributionIndex = {}
for feature in config.selectedFeatures:
   for file in featureIndex.get(feature, []):
      member = file.member
      if member in config.selectedImpls and file.relevance == 'system':
         contributionIndex[member.name] =
            contributionIndex.get(member.name, 0) + file.metrics.ncloc
```

Fig. 2. Idealized *Python* code for the comparison. The code operates on *Linked Data*.

*Language, technology, and style detection:* Perform language and technology detection. In this paper, we only care about 'programming style' in so far as it is hinted at by the use of technologies. When interpreting comparison results, we may very well take knowledge of styles into account.

*Feature location:* Locate the selected features in the selected implementations. Some features may be missed because of their implicit implementation and thus require manual tagging. More generally, feature location must be validated. In this paper, for simplicity, we do not admit implementations where any source file mixes selected with additional features, as this would require a degree of feature location that is not available to us across many languages.

*Metric computation:* In this paper, we limit ourselves to NCLOC as metric, as there is hardly any other metric available at this point for many different languages, but see the discussion of §VII. The metric is computed for source files identified by feature location or manual tagging. The metric values are summed up for all the files of an implementation.

*Interpretation:* The different NCLOC sums are interpreted by an expert who consults the selected implementations.

## IV. Execution

We inform feature and implementation selection by building a mapping from all features to the files implementing a feature; see the first code block in Figure 2. To this end, we use the *Linked Data* access path to the 101repo holding (all source-code files of) all implementations of 101. Feature location has been applied to the 101repo upfront. We note in passing that 101repo is a confederated repository with many distributed, physical repositories, but the *Linked Data* access path shields the programmer from such complexity.

Based on inspection of the mapping, we should pick features that stand out as being popular and modularly implemented; see the parameter `config.selectedFeatures` in Figure 2. In this paper, we pick 101's features for totaling and cutting salaries on top of the data model for hierarchical companies. Eventually, we also pick a few implementations; see the parameter `config.selectedImpls` in Figure 2. One selection is discussed in §V.

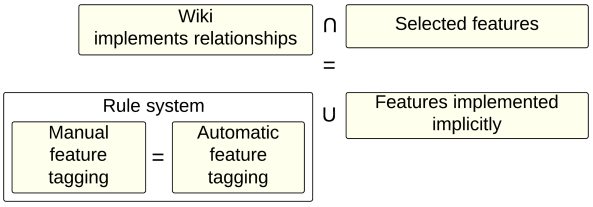


Fig. 3. Objective of the validation of feature location

Eventually, we compute a mapping from 101's contributions to NCLOC; see the last code block in Figure 2. To this end, we iterate over the selected features and, in turn, over all files concerned with each feature. If the file belongs to a selected implementation and is also tagged as being 'system' relevant (as opposed to generated code or included third-party code), then the file's NCLOC value is counted towards the general NCLOC value for the associated implementation ('contribution').

Feature location is validated semi-automatically; see the middle code block in Figure 2 and see Figure 3 for a summary of the validation objective. In particular, the results of rule-based feature location are compared with the documented ('specified') features, thereby revealing potential discrepancies between feature location and documentation. As an aside, for several contributions, feature location was also performed manually, thereby enabling the validation of the identified source files.

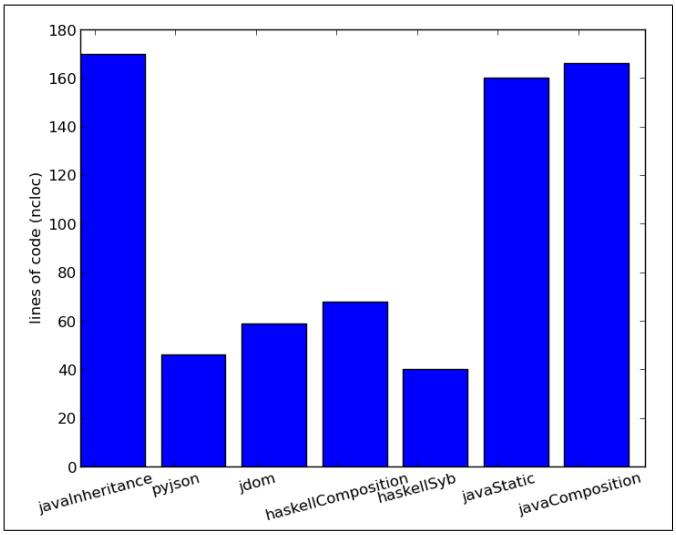


Fig. 4. An NCLOC-based comparison of implementations of features 'Total', 'Cut', and 'Hierarchical company'

Feature location may miss 'implicit' implementations, as discussed in the introduction. This status may be confirmed by code inspection. In those cases, the discrepancies between feature location and documentation are manually silenced by declaring the implicit status; see the parameter `config.implicit` in Figure 2. In future work, we plan to devise more sophisticated rules for feature location, which can also detect implicit implementations.

## V. Results

Within the scope of an early research achievement, we aim to address exemplarily the research question as to what language, technology, or style is more suited for implementing a certain programming task.

Figure 4 shows the chart for the execution of §IV for seven selected implementations with a coverage of three languages—*Java*, *Python*, and *Haskell*.

There are four Java-based implementations. Three of them (see the names java...) show very similar metrics, which is a consequence of the fact that they are implemented in different but basic styles of OO programming (class inheritance versus object composition versus families of static methods). The fourth Java-based implementation, *jdom*, is strikingly more concise. This is the case because an implicit data model and a query API for features such as 'Total' are used.

The *Haskell*-based implementation *haskellComposition* uses an explicit data model and it uses no special API. Nevertheless, it is nearly as small as the *Java*-based implementation *jdom*. The *Python*-based implementation *pyjson* is even more concise; it does not leverage an explicit data model. It turns out that *haskellSyb* is the most concise implementation, despite its explicit implementation of a data model. Code inspection and expertise suggests the following arguments. Haskell is generally very concise, when compared to Java. Python may be similarly concise and it may benefit from the omission of explicit data models. However, Haskell's SYB style [1] makes the implementation of query and transformation features such as 'Total' and 'Cut' highly concise; such a style is not established in Python or Java.

## VI. Research Directions

*Broad metrics:* Further investigation of a technology- and language-parametric set of metrics is needed. Such metrics should be context-aware, that is, take into account not only the pure source code, but also a 'profile' of the involved APIs. (See §VII for the discussion of metrics.) Such metrics should agree on the 'objective', paradigm-independent complexity of implementations of features under study.

*Feature location:* Benchmarking of feature location has been stated as a worthwhile objective [11]. Some limited datasets are available.[4] It may be that 101 can be completed into a powerful dataset with its 178 contributions in 40 languages and 96 technologies. We should strive to better use this data set in feature location research. The systems are relatively small, but not trivial. Thus, they are both meaningful and amenable to validation, as also shown in the present paper.

*Full* Linked Data *compliance:* The remaining challenge lies in the heterogeneous setup at hand. Source code repository (101repo), wiki-based documentation (101wiki), and derivatives (as computed by the 101worker modules) evolve independently and with different scopes of change. Thus, we need to surface 101 in a way that confederates all resources in a useful and sufficiently efficient manner. This disqualifies, for example, a naive approach of dumping all data into a triple store as its consistency would be hard to maintain.

*Artifact granularity:* Our current infrastructure is designed in a way that files from the repository serve as primary resources; they are associated with URIs for derived resources that contain (meta)data. We work on a generalization from files to fragments.

## VII. Related Work

The comparison aspect of the presented research is original. However, there is related work, along different dimensions, which could inform improvements of the methodology for comparison and suggest more interesting experiments.

*Feature location*: Feature (or concept) location [12] is a common maintenance activity [11] performed by developers. A systematic survey is provided in [11]. In the absence of external documentation, advanced information retrieval methods, such as latent semantic indexing (LSI), are applied [13]. So far, we use only a very basic, text-based location approach operating on source text including program identifiers and comments, while taking only advantage of knowledge of the lexical structure; see some of the rules in [10].

Novel context-aware approaches to feature location [9] suggest using both a requirement model (e.g., a feature model) and a program model as input. However, deeper investigation and adoption of 'just enough' requirement model is an open issue. An interactive exploration approach is proposed in [14] to

[4] http://www.cs.wm.edu/semeru/data/benchmarks/

support the human-oriented and information-intensive process of feature location. The results show an increase of developer productivity while using multi-faceted search.

*Program comprehension & API analysis*: Creating a conceptual model of the source code is used in various program comprehension activities such as multi-language cross-referencing [15] or business-rules extraction [16]. As we demonstrated, API usage might encapsulate most of the code of a more traditional feature implementation. We should bring models of APIs, e.g., classifiers for APIs, API domains, and API facets [17], into the scope of the rule-based system for metadata computation, thereby imposing more structure on comparison.

*Software metrics*: We use NCLOC as a starting point for comparing implementations. There exists empirical evidence about correlation of the size of a software system with its fault-proneness [18] and maintainability [19]. In the case of object-oriented systems, more advanced size metrics, such as NIM (Number of Instance Methods) or TNOS (Total Number Of Statements) [20] can be considered. However, the methodology clearly requires metrics that can be used across various languages and paradigms. Furthermore, any selected metrics should also agree with complexity as perceived by programmers [21]. More research is needed on metrics suitable for comparison.

*Linked data*: We are aware of another effort concerning the provision of facts about source-code repositories using *Linked Data*. That is, *SeCold* is an open and collaborative platform for sharing software datasets, as introduced in [22]. In its first release, the dataset contains about two billion facts such as source-code statements, software licenses, and code clones from 18 000 software projects exposed using *Linked Data* principles. Overall, our research community is just at the beginning of handling repositories and derived artifacts as *Linked Data*.

## VIII. Conclusion

We have described an approach to comparing feature implementations across languages, technologies, and styles. The contribution of this paper does not lie so much in the manifestation of a significant metric difference across languages and technologies; it rather lies in the methodology and the underying infrastructure that enable one to reveal those differences. We have leveraged 101 in this paper and we clarified the assumptions made. That is, one relies on a chrestomathy [6] that collects a sufficient number of diverse enough but usefully comparable implementations of well-defined features. 101's systems are small, but not trivial; they are designed to demonstrate programming styles, languages, technologies, concepts, and best practices.

Our specific experiment showed a high variability in NCLOC of the implementations for the same features across language and technology. For instance, functional programming implementations in *Haskell* were typically of smaller size, when compared to object-oriented programming implementations in *Java*. Technology usage, appropriate for a certain feature (e.g., the use of query members of an *XML* API) has a big impact. Our early observations support the hypothesis that the combination of programming language + technology + style is the multi-dimensional scope needed for comparing 'efficiency' of implementations for a given task. Clearly, more work is needed on the methodology of comparison and its execution in experiments.

## References


[1] R. Lämmel and S. P. Jones, "Scrap your boilerplate: a practical design pattern for generic programming," in *Proc. of TLDI 2003*. ACM, 2003, pp. 26–37.

[2] A. Marcus and J. I. Maletic, "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing," in *Proc. of ICSE 2003*. ACM, 2003, pp. 125–137.

[3] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *Software Engineering, IEEE Transactions on*, vol. 29, no. 3, pp. 210–224, 2003.

[4] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *Software Engineering, IEEE Transactions on*, vol. 33, no. 6, pp. 420–432, 2007.

[5] J.-M. Favre, R. Lämmel, T. Schmorleiz, and A. Varanovich, "*101companies*: a community project on software technologies and software languages," in *Proc. of TOOLS 2012*, ser. LNCS, vol. 7304. Springer, 2012, pp. 59–74.

[6] R. Lämmel, "Software chrestomathies," 2013, liber amicorum Paul Klint. Available online http://softlang.uni-koblenz.de/chrestomathy/.

[7] C. Bizer, R. Cyganiak, and T. Heath, "How to publish Linked Data on the web," 2007, online tutorial http://wifo5-03.informatik.uni-mannheim.de/bizer/pub/LinkedDataTutorial/.

[8] T. Heath and C. Bizer, *Linked Data: Evolving the Web into a Global Data Space*, ser. Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan & Claypool, 2011, 1st edition.

[9] X. Peng, Z. Xing, X. Tan, Y. Yu, and W. Zhao, "Iterative context-aware feature location (NIER track)," in *Proc. of ICSE 2011*. ACM, 2011, pp. 900–903.

[10] J.-M. Favre, R. Lämmel, M. Leinberger, T. Schmorleiz, and A. Varanovich, "Linking documentation and source code in a software chrestomathy," in *Proc. of WCRE 2012*. IEEE, 2012, pp. 335–344.

[11] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[12] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Proc. of IWPC 2002*. IEEE, 2002, pp. 271–280.

[13] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," in *Proc. of WCRE 2004*. IEEE, 2004, pp. 214–223.

[14] J. Wang, X. Peng, Z. Xing, and W. Zhao, "Improving feature location practice with multi-faceted interactive exploration," in *Proc. of ICSE 2013*. ACM, 2013, pp. 762–771.

[15] B. Kullbach, A. Winter, P. Dahm, and J. Ebert, "Program Comprehension in Multi-Language Systems," in *Proc. of WCRE 1998*. IEEE, 1998, pp. 135–143.

[16] V. Cosentino, J. Cabot, P. Albert, P. Bauquel, and J. Perronnet, "Extracting business rules from cobol: A model-based framework," in *Proc. WCRE 2013*. IEEE, 2013, pp. 409–416.

[17] C. De Roover, R. Lämmel, and E. Pek, "Multi-dimensional exploration of API usage," in *Proc. of ICPC 2013*. IEEE, 2013, pp. 152–161.

[18] K. E. Emam, S. Benlarbi, N. Goel, and S. N. Rai, "The confounding effect of class size on the validity of object-oriented metrics," *IEEE Trans. Software Eng.*, vol. 27, no. 7, pp. 630–650, 2001.

[19] M. Dagpinar and J. H. Jahnke, "Predicting Maintainability with Object-Oriented Metrics - An Empirical Comparison," in *Proc. of WCRE 2003*. IEEE, 2003, pp. 155–164.

[20] M. Lorenz and J. Kidd, *Object-oriented software metrics: a practical guide*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.

[21] B. Katzmarski and R. Koschke, "Program complexity metrics and programmer opinions," in *Proc. of ICPC 2012*. IEEE, 2012, pp. 17–26.

[22] I. Keivanloo, C. Forbes, A. Hmood, M. Erfani, C. Neal, G. Peristerakis, and J. Rilling, "A Linked Data platform for mining software repositories," in *Proc. of MSR 2012*. IEEE, 2012, pp. 32–35.