Accelerating the Creation of Customized, Language-Specific IDEs in Eclipse

Philippe Charles, Robert M. Fuhrer,
Stanley M. Sutton Jr., Evelyn Duesterwald
IBM T. J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598
pcharles, rfuhrer, suttons, duester@us.ibm.com

Jurgen Vinju
CWI
Amsterdam, Netherlands
jurgen.vinju@cwi.nl

Abstract

Full-featured integrated development environments have become critical to the adoption of new programming languages. Key to the success of these IDEs is the provision of services tailored to the languages. However, modern IDEs are large and complex, and the cost of constructing one from scratch can be prohibitive. Generators that work from language specifications reduce costs but produce environments that do not fully reflect distinctive language characteristics.

We believe that there is a practical middle ground between these extremes that can be effectively addressed by an open, semi-automated strategy to IDE development. This strategy is to reduce the burden of IDE development as much as possible, especially for internal IDE details, while opening opportunities for significant customizations to IDE services. To reduce the effort needed for customization we provide a combination of frameworks, templates, and generators. We demonstrate an extensible IDE architecture that embodies this strategy, and we show that this architecture can be used to produce customized IDEs, with a moderate amount of effort, for a variety of interesting languages.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Software—integrated development environments

General Terms Languages

Keywords IDE, Eclipse, generation, meta-tooling, IDE workbench.

1. Introduction

After decades of activity, programming languages remain a vital area of active research, and new languages are introduced with surprising frequency. Motivations include basic research into new computing and system architectures, new application domains, new programming paradigms, and pedagogical purposes. Some topics of recent special interest include parallel languages (e.g., X10 (Charles et al. 2005)), domain-specific languages, scripting languages (e.g., Ruby (ruby-lang.org) or Python (python.org)), and aspect-oriented languages (e.g., AspectJ (eclipse.org/aspectj)). In fact, this trend may even deepen: "language-oriented programming" (Fowler) aims to make the development of languages cost-effective even for use in a single application.

Alongside this proliferation of languages, the last 25 years have seen Integrated Development Environments (IDEs) rise from novelty status to fundamental need. IDEs provide critical tooling, such as editors, viewers, dependency management, and build support, that enable programming languages to be used most effectively. The existence of a full-featured IDE has become critical to a language's widespread adoption.

Such a rich feature set comes at a cost, however: modern IDEs are large and complex software systems, incorporating large amounts of highly customized language-specific functionality. As a result, constructing a modern IDE from scratch requires a level of effort and cost that is often prohibitive.

An alternative approach to IDE development relies on automatic generation. Examples include the Synthesizer Generator (Reps and Teitelbaum 1984), the ASF+SDF Meta-Environment (van den Brand et al. 2001), and SmartTools (Attali et al. 2001). In this approach, meta-tools consume a specification of the syntax and (possibly) semantics of the language, and generate tools such as parsers, compilers, interpreters, and debuggers. Some of these systems are capable of generating IDE tooling, such as editors and viewers. Although this can greatly relieve the burden of producing the IDE, it comes with significant limitations. Specifically, while generation-based IDEs provide basic functionality for the given language, they do so at the expense of the language-specific customizations in appearance and behavior that give

OOPSLA 2009 October 25–29, 2009, Orlando, Florida, USA.

the greatest productivity boost. Often, their feature sets provide only the "lowest-common denominator" across all languages, resulting in IDEs that do not offer the leverage that makes IDEs popular. Additionally, the meta-tools for generation often dictate the use of specific parsing or compiler technology. This can be highly impractical in cases where a compile front-end already exists that would have to be rewritten to match the required parsing technology. For these reasons, generation-based approaches, while successful in limited applications such as parser generators, have not had corresponding success in the domain of IDEs.

In fact, these two development approaches represent the end-points of a spectrum. Building a high-powered IDE from scratch is feasible when a sizable group of people with the appropriate expertise make a substantial and sustained resource commitment. Generating an IDE from a language specification can be performed by a small group (or an individual) when the main interest and expertise is in language development and a basic IDE is adequate to their purposes. We see a practical middle ground between these two extremes of IDE development, one where a high degree of reuse is balanced with fulfillment of language-specific requirements, where incremental efforts are rewarded by incremental value, and where developers can pick the degree of functionality and the level of investment.

We hypothesize that this middle ground can be effectively addressed by an open, semi-automated strategy for IDE development. First, we relieve the IDE developer of as much of the burden of IDE development as possible, particularly for those parts of the IDE that are of least interest, such as the internal details of user interface componentry. Second, to afford maximum flexibility for customization, we define open interfaces for the implementation of IDE services and make it possible to provide a service in entirely original ways. To reduce the effort that is needed, we facilitate the implementation and customization of IDE services by a combination of frameworks, templates, generators, and domain-specific languages. Finally, we provide other avenues to IDE customization, including the ability to select which services are incorporated into the IDE and the ability to introduce services *ad hoc*. This strategy effectively combines the power of tool generation with the flexibility of manual construction.

Our contribution in this paper is two-fold. First, we demonstrate an extensible IDE architecture that separates and encapsulates the language-independent framework, exposes significant points for language-specific extensions and customizations, and generally enables these extensions and customizations to be implemented with minimal reference to the underlying framework. Second, we show that this architecture is practical and can be used to produce useful IDEs for a variety of interesting languages with a moderate amount of effort. Additionally, we have implemented our approach as the IDE Metatooling Platform (IMP), which is available as an Eclipse Project on www.eclipse.org/imp. This paper updates and elaborates work previously published in (Charles et al. 2007), describing new features, giving additional details about the development process and architecture, and reporting new information on experience and evaluation.

The rest of this paper is organized as follows. Section 2 states our vision of an IMP IDE and contrasts this briefly with some other approaches. Section 3 states our goals and approach more specifically. As a means to document the level of effort involved in developing an IMP IDE, the IMP IDE development process is explained in Section 4. Section 5 presents the IMP architecture, highlighting aspects that make the IMP approach to IDE development possible. Section 6 describes experience with IMP (mostly our own) and Section 7 gives an evaluation based mainly on that experience. Section 8 discusses related work. Section 9 presents some open issues, and we conclude in Section 10.

2. An IMP IDE

As stated earlier, our primary goal is to facilitate the development of language-specific IDEs in Eclipse. The premier example for such an IDE is the Java Development Toolkit (JDT) (eclipse.org/jdt). The JDT exemplifies both a sensible approach to IDE architecture and a useful set of tools, features, and functions.

On the architectural level, the JDT consists of a collection of Java-oriented tools which operate on an abstract model of Java workspaces, projects, and programs. Some of the JDTs tools are defined as extensions of more general user-interface elements provided by the Eclipse framework. Additional services that support software development in general, such as resource management, version control, preferences support, and support for plug-in development, are also provided by Eclipse. Moreover, the Eclipse framework's extensibility permits the creation of additional Java-specific or Java-compatible tools and services.

Similarly, an IMP IDE is a collection of mainly language-specific tools organized around an abstract program model that are situated in the larger context of an Eclipse workspace. IMP also supports the development of language-independent IDE features. Many IMP-based tools extend elements in the Eclipse framework. Users of an IMP IDE are able to draw on other components available through Eclipse to address services that IMP does not support. Like Eclipse, IMP is itself extensible and offers support for an expanding variety of tools and functions.

Although IMP provides support specifically for building IDEs in Eclipse, many aspects of its approach generalize readily to other IDE frameworks. Also, IMP does not directly address IDE development through infrastructure such as notification mechanisms, communication buses, or object repositories (e.g., (Reiss 1990; Purtilo 1994; Kadia 1992)). At the same time, IMP does nothing to prevent the use of such mechanisms; e.g., Eclipse provides an extensive frame-

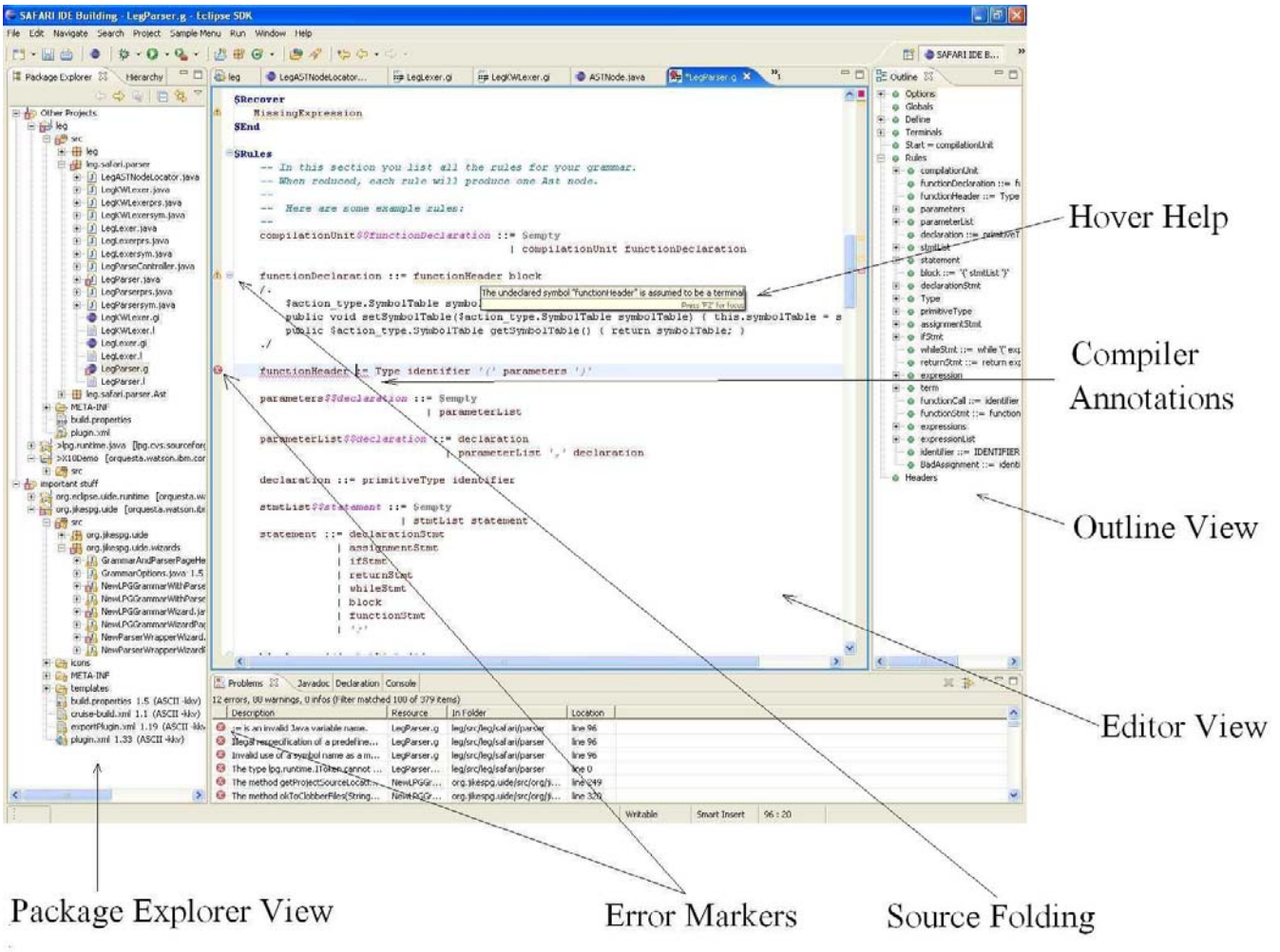

Figure 1. An IMP-based IDE for the LPG grammar specification language

work for tool notification and resource management that interoperates with IMP.

Similar to the JDT, an IMP IDE offers what many developers have come to expect from a modern IDE: a language-sensitive editor with features such as syntax highlighting, hover help, hyper-linking (e.g., from references to declarations), and content assist; structural and navigational views (e.g. outline views); and program building services. Figure 1 shows a screenshot of an IMP-based IDE with various elements called out. While the look and feel of the depicted IDE is similar to the JDT, the language for which it is intended is the grammar specification language for the LPG parser generator (lpg.sourceforge.net).

Figure 2 shows a listing of IDE services that are available (or under development) in IMP, as an image of the menu by which service implementations are created. Many of these services will be familiar to users of modern IDEs; several are discussed in detail later in the paper. A few menu entries are not targeted at specific services but instead represent generic hooks for the introduction of arbitrary functionality in different contexts. These include the Editor Actions Contributor, which adds commands to the editor's context menu, the Introduced Editor Service, which specifies functions to be invoked automatically along with other editor services, and the Refactorings Contributor, which augments the set of language-specific program refactorings.

The "LPG" category of services in Figure 2 addresses syntax definition and parser generation via the LPG parser generator (lpg.sourceforge.net). However, IMP does not mandate any particular technology for syntax definition, parsing, or program representation – the IDE developer can choose any desired technology (see Sections 4 and 5). Nonetheless, the IMP distribution includes, as an option, the LPG parser generator and an accompanying IDE in order to offer a more complete package for out-of-the-box IDE development.

As of this writing, IMP does not provide any framework components or meta-tooling for developing or interacting with language runtimes and debuggers. Although IMP's agnostic position with respect to language technology makes

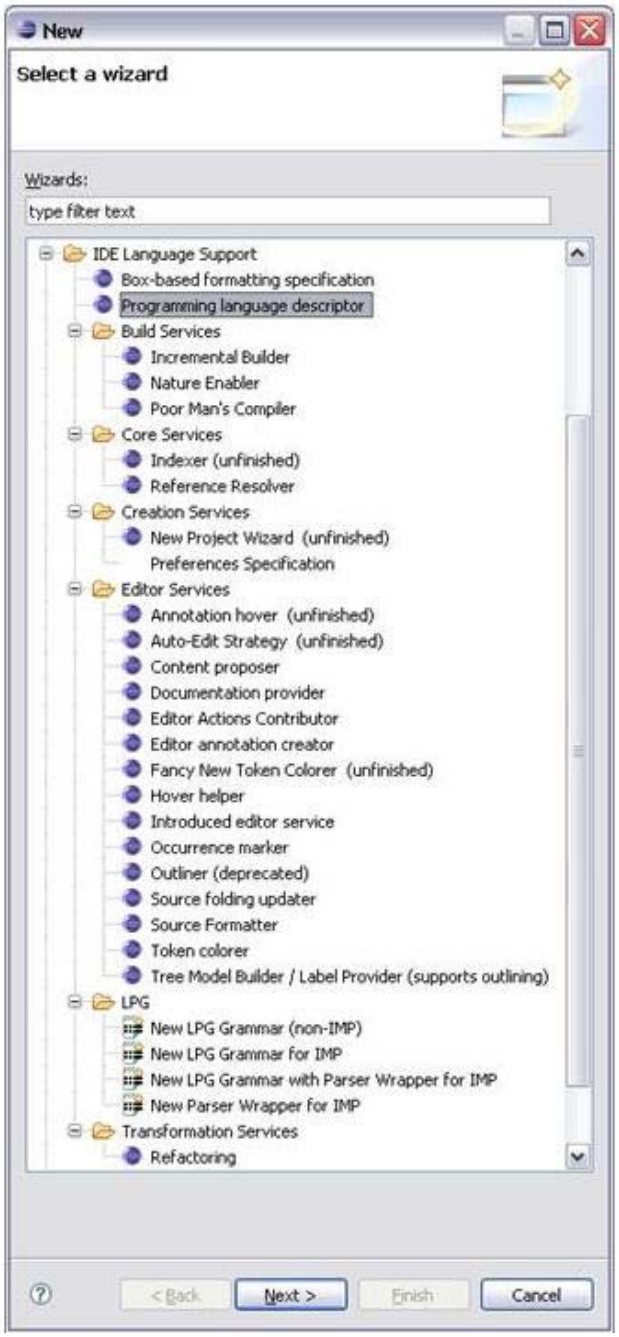

Figure 2. IMP support for IDE services.

defining runtime and debug-time tooling more difficult (requiring a bridging layer of semantic descriptions), we believe it is still eminently possible. This is an open area for research and further development.

3. Goals and Approaches

Through a hybrid approach to IDE development, combining the flexibility of manual construction with those of automated generation, we hope to achieve the following goals:

To support the development of IDEs by people whose primary competency is not IDE development or user-interface frameworks. This includes, for example, language developers.

To support the development of "effective" IDEs, that is, IDEs that have a substantial subset of the features and functions that are expected in a modern IDE, and that have acceptable performance and size characteristics. We consider the Eclipse Java Development Toolkit (eclipse.org/jdt) as the "gold standard" of IDE effectiveness.

To significantly reduce the time and effort required to develop a useful IDE. Where it might now take several IDE specialists person-months of effort to develop a modest IDE for a moderately sized language, our goal is to reduce that to a few person-weeks of effort by a non-specialist. Ultimately, we intend to shift the economics of IDE development such that IDEs can be developed "on demand" as a normal part of language definition, application development, research projects, and academic courses.

To enable significant customization of both the set of IDE features, and the behavior of those features. Customization for specific languages, users, and purposes is a key to an IDE's success.

To accommodate alternative tooling for key IDE components such as parsers, AST representations, and editors. This allows IDE developers to use whatever software assets they already have to lower development costs, and still gain the benefits of an IDE framework.

To allow IDEs to be developed selectively, iteratively and incrementally. This allows IDE developers to focus first on priority features, and allowing them to add additional features if and when the need arises.

To realize these goals, we further base our approach to IDE meta-tooling on the following design principles:

- Build IDE components and services upon models of the program under development (rather than linking services directly to one another). This enables incremental development of IDE services and increases the substitutability of IDE components.
- Enable as much of the IDE as possible to function independently of any particular service. This facilitates selective and incremental development.
- Provide templates for IDE service implementations that help developers focus on language-specific customizations and other important aspects of IDE behavior and appearance.
- Provide language independent base classes for service implementations. Where possible, provide concrete classes that provide a useful level of functionality without modification.
- Separate the creation, management, and invocation of editor services from the editors and other views, so that the editor is replaceable.

- Support an open-ended set of services with public extension points. This supports the usability, extensibility, and customizability of developed IDEs.

The following sections address our implementation of these approaches.

4. IMP Development Process

This section gives an overview of the process of developing an IMP-based IDE. We illustrate that IMP affords the following benefit automatically: (i) that in-depth knowledge of the framework (especially that of the underlying Eclipse framework) is not required to implement substantial, customized IDE services; (ii) that the effort required is proportional to the benefit received (i.e., that there is low development overhead relative to the functionality of interest); and (iii) that the expertise required for IDE development centers around knowledge of the language.

Before beginning development of the various services, the developer must supply basic information about the language. This is done through a "New Programming Language" wizard, which collects a list of filename extensions along with a unique identifier for the language (used to match content to the appropriate service implementations).

For the most part, the IDE developer can select the desired services and implement them in any order. The most obvious exception to this is the parsing service, which generally comes first, since most services rely on the token stream and ASTs. There are a few additional (though fairly obvious) dependencies that imply an order (e.g., content assist typically depends on reference resolution).

Each step in the process is initiated by means of a wizard through which the user identifies the target project, language, and service implementation classes. Certain wizards permit limited customization through additional fields, rather than by modifying the service implementation code.

Finishing the wizard typically has two effects. First, an extension is created in the plug-in meta-data that registers the service implementation with the IMP runtime. Second, a skeletal service implementation is generated, which may provide limited functionality without further implementation (e.g., the skeletal token-colorer highlights keywords). However, most services naturally require additional work to implement the desired functionality. IMP alleviates the burden of this additional work by factoring the language-independent infrastructure concerns into framework classes, enabling the developer to focus on language-specific concerns. IMP also provides domain-specific languages for certain services, to minimize syntactic and semantic overhead.

The generated skeletons contain an example implementation for a simple, block-structured, imperative language, called "LEG" ("Little Expression Grammar"). The generated files are automatically opened in the appropriate editor and positioned where customizations are to be performed.

4.1 IDE Services

IDE services can be divided into user visible services (e.g. token coloring) and internal services (such as parsing and reference resolution). We discuss below the implementation of a representative selection of services of both kinds.

The first service typically implemented is the parser service, which must implement the "parse controller" interface. IMP is designed to work with any parser, whether hand-coded or generated, so this interface is appropriately neutral. For example, it includes methods to parse a string (returning the resulting AST), to get the keywords of the language, or to return an iterator over tokens within a given range in the source text. Additionally, IMP interfaces typically represent abstract syntax trees (ASTs) and other language-specific entities as plain objects.

To facilitate the creation of a parser, the IMP meta-tooling includes a complete IMP-based IDE for the LPG parser generator (lpg.sourceforge.net). (Similar support can easily be added to IMP for other parser generators.) Like many modern parser generators, LPG features include automatic generation of AST classes from the grammar, as well as AST visitor classes to assist in AST traversal.

For developers using other parser generators, or existing parsers, or who want to write their parser manually, IMP also provides a "parser-wrapper" wizard that creates a skeleton for the parse controller class that delegates all operations to the existing parser class.

Once this step is complete, the nascent IDE is already (minimally) usable: the source editor provides "live parsing," presenting syntax errors as source annotations.

Perhaps the simplest user-visible service to implement is **token coloring** (syntax highlighting). The principal method in the requisite interface for this service is

```
getColoring(IParseController, Object)
```

where the `Object` represents the text to be "colored" and the parse controller provides access to the AST, among other things. The `Object` given to `getColoring` is typically a lexical token. In any case, the entity's kind is used to determine the text attributes to apply to the corresponding source. Information about the context of the token in the surrounding AST can also be consulted. To determine how much text to re-color for any given textual change, the token colorer interface defines the method

```
calculateDamageExtent(IRegion)
```

which takes a "damaged" (modified) text region and returns a possibly larger one whose coloring needs to be updated. By default, this method simply returns the region it was given.

The **label provider** and **documentation provider** services are implemented similarly to the token colorer in that both take a program entity (often representing an AST node) and return a value based on the type of the entity. In the former case, the value returned is the text or image to be used to represent the given entity in various UI views. In the latter,

the value is the relevant documentation for the given entity (e.g., its JavaDoc).

Some IMP services, such as the **source-text folder** and **tree-model builder**[1], are easily implemented using AST visitors. The service implementor provides a `visit(...)` method for each AST-node type for which source folding or an outline item is desired. For a source folder, the trivial `visit(...)` method for any foldable AST-node type consists of a call to the method `FolderBase.makeFoldable()`. For a tree-model builder, the visitor implementation is only slightly less trivial, as it typically must ensure that the tree model mirrors the AST's structure. Both the source folder and tree-model builder permit more complex logic, e.g., to fold regions of text that don't correspond exactly to AST nodes or to rearrange the tree model (e.g. sorting a node's children based on their type or labels). As shown in Section 7, though, the typical implementation of a visitor method is just a couple of lines. As a result, the size of these service implementations depends mainly on the number of nodes addressed.

Reference resolution is a core internal service in IMP that is used by several other services. The principal method to implement is `getLinkTarget(...)`. If a compiler front end is available to provide binding information, this method can simply return the precomputed binding. If not, it must produce this information by other means.

The **hover-help** service is one for which IMP provides a default implementation but which also facilitates considerable customization. This implementation takes advantage of a reference resolver and a documentation provider, if available. First, if a reference resolver exists, it is used to find the declaration corresponding to the AST node over which the cursor hovers, if that node is a reference node; otherwise, the "hovered node" is used. Next, if a documentation provider exists, the hover helper returns whatever the provider produces for that node. If no documentation provider exists, the hover helper simply returns the source text associated with that AST node. Finally, if no reference resolver exists, the default hover helper simply returns the source text associated with the given AST node, if different from the "hovered node." Of course, if the above logic is insufficient, the developer can always create a custom hover implementation to completely control the information that is presented. In all cases, however, note that the IDE developer is responsible only for defining *what* to display, rather than *how* or *when* to display it.

4.2 Arbitrary Services

IMP provides extension points and supporting mechanisms explicitly designed for a number of anticipated IDE services such as token coloring and text folding (see Section 7 for a more complete list). The following additional extension points allow for the introduction of arbitrary services in a number of specific contexts:

- The "model listener" extension point permits arbitrary clients to be notified whenever the source model (AST) changes in response to text edits.
- The "refactoring contributor" extension point allows IDE plugins to contribute one or more language-specific refactorings. Refactorings are generally triggered on behalf of a selected program entity (e.g., in the source editor or outline), via the context menu.
- The "editor actions contributor" extension point allows IDE plugins to programmatically contribute actions to the Eclipse menu bar, tool bar, and status bar.

IMP provides wizards for each of these extension kinds that generate very basic implementation skeletons.In the case of the "New Refactoring" wizard, however, the code skeletons themselves encapsulate nontrivial knowledge about the relationship among several key Eclipse APIs for refactoring, structured text rewriting, undo support, and the like.

4.3 Help for the IDE Developer

The IMP development process is well documented, with a User's Guide and Eclipse cheat sheets. Cheat sheets are integrated into the IMP IDE, and provide a semi-interactive, step-by-step guide through the development process. Also included is documentation for LPG, the PrefSpecs language for specifying preferences and preference pages, and the source formatting language and IDE. The IMP IDE developer can actually run through all of the IMP wizards and create an operational LEG IDE without editing any code; this IDE can then be used for experimenting with alternative language formulations or service implementations. The IMP release contains the source for several functioning IMP IDEs that are part of IMP itself, notably for the LPG, PrefSpecs, and source formatting languages, as well as others that are under development. The IMP-based IDE for X10 is available as open-source on SourceForge.

5. IMP Architecture

5.1 Meta-Tooling and Runtime

IMP consists of two kinds of components: meta-tooling components, which are used when developing an IDE, and runtime components, which are used during the execution of that IDE. The relationship between these two sets of components is depicted in Figure 3. The meta-tooling components includes wizards, templates, and generators that are used (as described in Section 4) to declare a language and develop specific IDE services. The meta-tooling also includes several IMP-based IDEs for domain- specific languages to aid in implementing certain services. Among these languages are LPG grammar and preference page specifications (see Sections 2 and 4).

[1] The tree model is available for any client's use; the standard IMP outline view uses it to present the source text's structure.

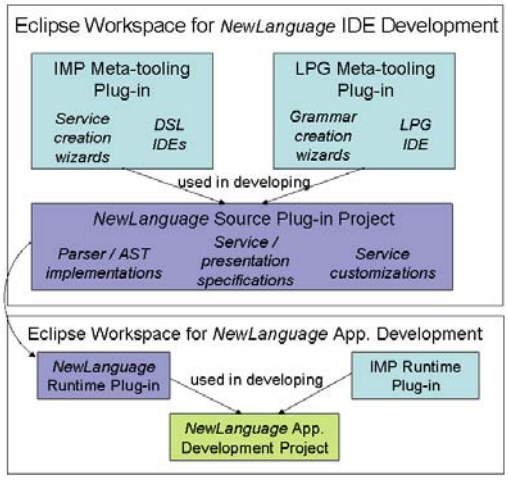

Figure 3. IDE Development and IDE Runtime

The IMP runtime framework builds on the "Eclipse Rich Client Platform", a set of generic, mostly programming-oblivious components, comprising text editors, tree views, and so forth. IMP components extend these to provide user-visible IDE services that are relevant to most programming languages such as source editors, parser problem annotations and markers and structural views. The IMP runtime framework also provides internal functions critical to an IDE's execution, such as the identification, instantiation, and dispatching of language-specific services (discussed below).

The IMP meta-tooling and runtime frameworks are language independent, but they naturally rely on language-specific code to provide the language-specific behavior for the various IDE services. To plug language-specific services into the language-independent framework, IMP uses the Eclipse extension-point mechanism (Bolour). Specifically, most IDE services managed by the IMP runtime framework correspond directly to IMP- or Eclipse-defined extension points. Thus, as mentioned in Section 4, IMP service creation wizards typically create one or more implementation classes and register them as extensions of the corresponding extension point. IMP runtime components (such as the Universal Editor) then query the extension registry to find the implementations for the language service in question.

As shown in Figure 4, IMP locates language-specific service implementations using an IMP-maintained mapping from filename extensions to known languages. This mapping is defined by extensions of the IMP "language descriptor" extension point.[2] Likewise, the extension metadata for each service implementation identifies the language for which it is intended.

[2] This is done in part so that service implementations can be distributed across multiple plug-ins.

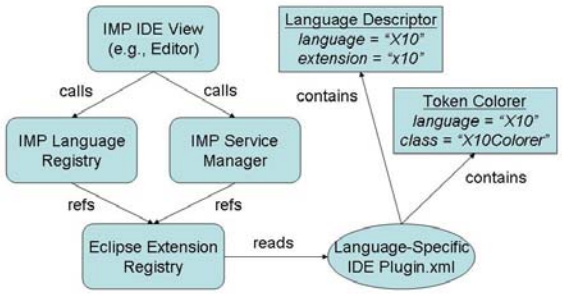

Figure 4. Locating service implementations

For example, IMP's source editor makes use of various language-specific service implementations. One of these is token coloring, which identifies the text attributes to be used in displaying each source text entity. For this service, IMP defines the `tokenColorer` extension point. Extensions of this extension point *must* implement the IMP `ITokenColorer` interface. The "New Token Colorer" wizard generates a skeleton implementation of this interface, and automatically registers it as an extension of the `tokenColorer` extension point. (The implementation will typically be customized by the IDE developer.) At runtime, when the IMP editor opens a file belonging to a given language, it consults the IMP Language Registry and Service Manager (as shown in Figure 4) to locate the extension of the `tokenColorer` extension point for that language. If one exists, the implementation class is used to color the source.

5.2 Runtime Operation

The execution flow of the token colorer is typical of most language services: it is invoked by the IMP framework, performs some language-specific analysis on its arguments, and returns the results to the framework. Interactions between the service and the rest of the IDE infrastructure are managed entirely by IMP. This arrangement enables the service implementations to focus on language-specific structure and semantics rather than on user interface APIs or other infrastructure components.

Figure 5 depicts the flow of events that lead from user editing actions to the resulting analyses and view updates. To first order, source document changes result in the updating of one or more models, such as the token stream, AST, and search indices. Changes to these models are propagated to listeners that form the basis for nearly all IDE services.

In more detail, editing actions, such as inserting text, result in a region of "damaged" source text, managed by the underlying Eclipse text components. The damaged region is propagated in the foreground (i.e., in the same thread) to an IMP "service controller" class, which mediates between the IMP framework components (e.g., the editor) and the language-specific services. In particular, this controller in-

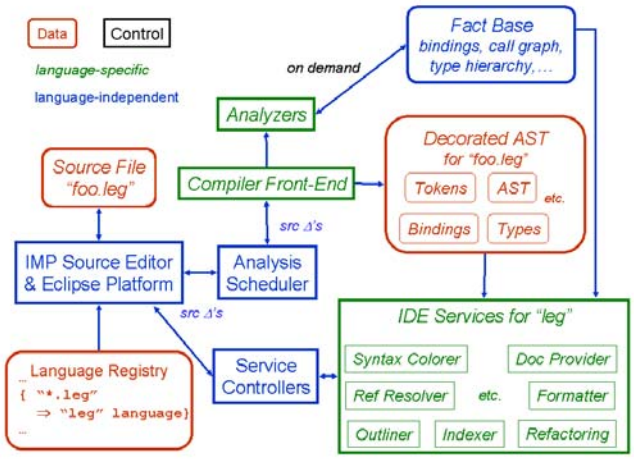

Figure 5. Service scheduling

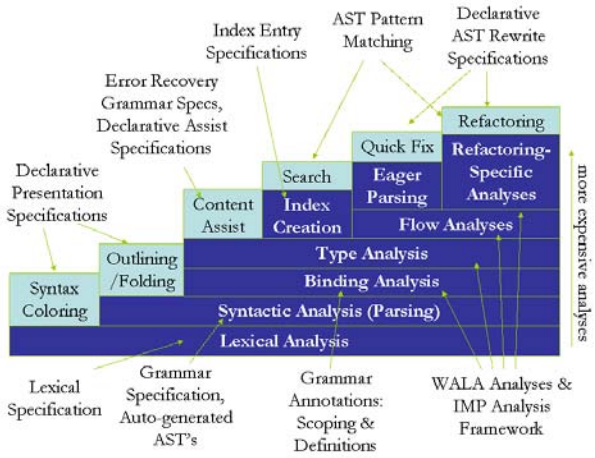

Figure 6. Service dependence on analyses

vokes the parsing service to produce the new token stream and AST from the source text. Certain light-weight listeners are notified synchronously (e.g., for token coloring). Other listeners are notified of model changes pending sufficient idle time. If more edits arrive in the interim, pending notifications are abandoned, and another model update cycle begins.[3] For heavier-weight analyses, such as search index creation, background jobs listen for changes to resources within the workspace, and perform the appropriate processing to update their results. Certain services, e.g., refactoring, may require additional analyses that are too expensive to perform eagerly; hence they are only performed on demand. Dependences of various services on various kinds of analysis are shown in Figure 6.

[3] In fact, parsing and other analyses can be interrupted, if the parser supports the appropriate API.

At present, model changes are represented as entirely new models; in the future, some analyses may process changes incrementally.

5.3 Architectural Substitution

The same architectural features that accommodate pluggable IDE services also supports substitution of major IDE components, such as the editor. Editor substitution is useful for languages with unusual syntactic requirements that are otherwise amenable to typical IDE services. For example, legacy languages such as FORTRAN or COBOL are column-sensitive and require specialized editing functionality that the Eclipse text editor does not support. By separating the Language Registry and Service Manager from the IDE components, as shown in Figure 4, and by encapsulating the initialization and configuration of the IMP runtime framework, editor substitution is straightforward. A foreign editor can be integrated by extending or wrapping it and using the Language Registry and Service Manager to access IMP metadata and services. It is similarly straightforward to do likewise with other services and views.

6. Experience

We have substantial experience in using IMP and IMP-based IDEs. Several IMP-based IDEs are part of the IMP release:

- The LEG language is a simple, procedural language that is provided for pedagogical purposes.
- The Box (van den Brand and Visser 1996) language is a simple DSL for text-formatting rules. This IDE was originally developed to support the debugging of Box tools and is now an integral part of IMP's source formatting specification editor.
- The PrefSpecs language is a DSL for the specification of preference pages, fields and values. We have used this IDE for the preferences of IMP itself, as well as for other IDEs listed here.
- The LPG grammar specification language is used by the LPG parser generator. We have used the LPG IDE in specifying the grammars for a wide variety of languages (e.g., Box, X10, PrefSpecs, COBOL, and the LPG grammar specification language itself). The LPG IDE can be used separately from IMP.
- The PSP language is a DSL for declarative aspects of IDE presentation.

We and others have also used IMP to develop IDEs that are not part of the IMP release:

- The X10DT, an IDE for the X10 language (Charles et al. 2005), an extension of Java for highly concurrent applications such as scientific and engineering applications (this IDE is available as part of the X10 release (X10.sourceforge.net))

- An extension to an existing COBOL IDE to provide support for COBOL development on Eclipse (released as part of IBM's Rational Developer for zSeries (RD/z))
- An IDE for SDF and ASF+SDF, which is a port of The Meta-Environment (van den Brand et al. 2001) to Eclipse using IMP.
- An IDE for Rascal, a DSL for source code analysis and manipulation based on a combination of algebraic specification and relational calculus primitives (Vinju et al. 2008). Its intended application domain is implementation of static analyses and refactorings for other DSLs intended for use in IDEs.
- An IDE for ToolBus Script, a coordination language based on algebra of communicating processes (ACP). It connects tools written in arbitrary languages to a coordination bus which is scripted in a formal language that can be easily analyzed (Fokkink et al. 2008).
- 'Spoofax/IMP is an IDE tooling platform integrating SDF and Stratego/XT into Eclipse.' It uses SDF(Heering et al. 1989) for defining grammar rules and Stratego (Visser 2004) for semantic rules and realizes the IDE using IMP (Kats and Kalleberg 2009).

IMP is also used to develop IDEs as part of other projects in IBM Research, such as an IDE for the SPADE stream-processing specification language (Gedik et al. 2008). Most of the IDEs listed above are publicly available.

To give just a few observations about our experience, the above languages range in grammar size from fewer than two dozen rules (for Box), to over 1000 rules (for COBOL). The size of the IDEs also varies widely, in terms of both code size and the number of features realized. Some representative numbers on code size are given in Section 7.3. Regarding the number of features, the set of typical "core" features for the IDEs that have been developed as part of IMP include

- Parsing with error annotations and problem markers
- Syntax highlighting (token coloring)
- Source-text folding
- Tree model builder (used, e.g., in outlining)
- Label provider (used, e.g., in outlining)
- Reference resolution (used, e.g., in hover help and occurrence marking)
- Hover help
- Occurrence marking
- Content assist
- Documentation provider
- Project building

These are all found in the LEG, LPG, and PrefSpecs IDEs and in X10DT. Additionally, the PrefSpecs IDE has a second documentation provider and the LPG IDE provides several refactoring and context-menu actions. The LPG and X10 IDEs have an IMP-based preference page (as does IMP itself).

Some IMP-based IDEs implement only a subset of these features. The COBOL IDE offers only six of the above-listed services, but these are used in conjunction with other services and tools developed earlier outside of Eclipse. The Box IDE has only parsing, syntax highlighting, and a builder, which were sufficient for its intended purpose of supporting debugging of Box tooling. The Rascal IDE has parsing, syntax highlighting, outlining, and a custom console. This clearly demonstrates that IMP reduces the cost of building IDEs to the point that it can be sensible to do so even for a relatively modest payoff.

The kinds of languages that have been supported through IMP include both specification languages and programming languages; the former have been mainly domain specific, while the latter have been more general purpose.

In developing IMP we have seen a number of opportunities where a domain-specific language could facilitate IDE development. Taking advantage of such opportunities was considerably easier because IMP made it practical to develop IDEs for these languages. Other IMP users have also found it opportune to couple IMP with DSLs to support aspects of IDE development (e.g., as for The Meta-Environment, Rascal (Vinju et al. 2008), and Spoofax/IMP (Kats and Kalleberg 2009)).

IMP has also provided the leverage to move some existing IDEs (e.g., for COBOL, The Meta-Environment) onto a new platform (i.e., Eclipse).

7. Evaluation

This section evaluates IMP against the goals set in Section 3. The basis for this evaluation is our experience in using IMP to build the IDEs described in the previous section. As discussed there, these languages and their IDEs vary widely is character and size. Many are in regular use, and most are publicly available. Table 1 lists several of the languages and gives measurements of the size of their grammars.

Language	# non-terminals	# rules
Box	9	21
LEG	23	37
PrefSpecs	66	71
LPG	66	100
ToolBus Script	–	118
X10[4]	69	204
Rascal[5]	200	582
Cobol	535	1124

Table 1. Languages with IMP IDEs and their grammar sizes

7.1 Ease of IDE Development

How hard is it for someone who is not an "IDE expert" to build an IMP IDE? In principle this could be measured experimentally, but such experiments are beyond the means of our project. Thus, we have not had the opportunity to conduct trials that would allow a truly robust quantitative evaluation. So, while the evidence collected to date is primarily anecdotal and qualitative, it nevertheless suggests that we have made substantial progress toward our goal.

We have observed a number of novice IMP users that were able to construct nontrivial IDEs with no particular background in IMP or the Eclipse infrastructure and without help much beyond the available documentation. We continue to find that IMP attracts both novice and experienced IDE developers, both within IBM and externally (see news://eclipse.org/imp/).

Furthermore, we know from our own experience that the amount of time needed to develop a nontrivial IDE or a significant IDE service is relatively small (hours, days, or weeks, depending on the scale of the effort). Also, the amount of code that needs to be written for many services is small in an absolute sense or relative to the complexity of the target language (e.g., the number of AST node types). Further details on these points are provided in Section 7.3.

Finally, we have made several decisions in the design of IMP in large part (if not entirely) to ease the burden of IDE development. Examples include:

- Use of simple specification languages and generators for some services, such as preference pages and IDE presentation
- The provision of base classes and implementation skeletons for many services that such that typical implementations can be programmed concisely and without significant reference to the Eclipse platform
- The handling of IDE related extensions and extension points (e.g., for service registrations and lookup) by the IMP meta-tooling and runtime, thus freeing the IDE developer from involvement with these details

Additionally, IMP's ability to support iterative and incremental development makes it possible for novice developers to "go slowly" as they are gaining experience.

We regard this as an area for continuous improvement in IMP. For example, we continue to refine base classes and interfaces as users help us discover how to improve their usability. At the same time, we take care to preserve opportunities for more sophisticated implementation approaches for those users with more demanding requirements.

[5] X10 data exclude elements of the Java, which serves as a base language.

[5] Rascal data include elements adopted from the Java statement and expression languages.

7.2 Effectiveness of IDEs

We consider the ability of IMP to support the development of "effective" IDEs from two angles: the set of services supported and IDE size and performance.

7.2.1 Supported Services

The set of services that an IDE may support is effectively open-ended. IMP provides direct support for a significant set of specific services, and the opportunity to add further, unanticipated services. Most of the services for which IMP provides development support are shown in Figure 2. This compares relatively favorably with the range of services provided by industry-leading Java IDEs, such as the Eclipse JDT (eclipse.org/jdt). As noted previously, IMP's support for execution, runtimes, debugging, and refactoring is presently very limited.[6] Support for indexing and searching in IMP is still under development. Also, IMP is lacking in the area of project-level navigation and support for entity-creation wizards (e.g., "New Project" wizards).

Most of the services indicated in Figure 2 are common in modern IDEs. The JDT in particular is a highly advanced IDE that has been in commercial development for almost a decade and continues to advance. In contrast, basic IMP-based IDEs can be constructed within days or weeks, depending on the complexity of the language, features of interest, and desired level of functionality.

7.2.2 Size and Performance

We examine IDE size and performance metrics for a set of IMP IDEs to which we have access. Specifically, we measured code size, memory footprint and interactive responsiveness.

Table 2 shows the total size of the JAR files that comprise the distribution of the IDEs under consideration. The size varies widely depending on the size of the language and features in the IDE, among other things. For example, the Cobol IDE provides fewer services than some of the others but for a much larger language, while the LPG IDE includes refactoring, search, and view support that is not found in most of the other IDEs. Also, unlike the other IDEs, the release for X10 does not include the AST representation (as the X10 compiler is packaged separately from the X10 IDE). Generally, though, these values are well within the range of typical IDE-scale software systems, which may be megabytes larger (a recent JAR file for JDT core source alone was about 3.5 MB).

To give an indication of the runtime size of an IMP IDE, we measured the heap memory footprint using a synthetic workload that consisted of running the IDE and opening six editors. The resulting memory footprint ranged from 23 MB for our smallest IMP-based IDE, the LEG IDE, up to 41 MB for our most complex IDE, the X10 IDE. For comparisons

[6] IMP provides minimal support for execution and debugging for languages that translate to Java.

IDE	JAR Size (KB)
LEG	164
PrefSpecs	298
LPG	1,299
X10	1,425
Cobol	2,815

Table 2. Size of exported JAR files for selected IMP IDEs. (Includes source, binaries, meta-data, and miscellaneous files.)

we ran the same experiment with the JDT, which resulted in a memory footprint of 35 MB. This demonstrates that the size of an IMP-based IDE can vary, depending on features of the IDE, but is generally within an acceptable range for an Eclipse IDE.

To assess IDE responsiveness we instrumented several IMP IDEs to measure IDE start-up time. Cold start-up time is measured as the time it takes to initially invoke the IDE with a representative input file until the IDE has initialized all views and is responsive to user input, excluding the Eclipse start-up time. Warm start-up time is the time it takes for consecutive editors to open. We measured cold and warm start-up times as the average over 4 experiments.

Cold start-up times ranged from 2.3 seconds for the LEG IDE (opening a file with 15 LOC) up to 4.7 seconds for the X10 IDE (opening a file with 400 LOC). Consecutive warm start-up times ranged from 0.04 seconds for the LEG IDE to 1.0 second for the X10 IDE. These numbers are more or less comparable to our experience with the Eclipse JDT and demonstrate that IMP-based IDEs, even complex ones, are sufficiently responsive for interactive use.

7.3 Reduction of Time and Effort

Our IMP IDEs have not been developed under controlled conditions, but our experience indicates that useful IMP IDEs can be developed in days or weeks. For example, development of the initial PrefSpecs IDE took about one week by one person, beginning with the specification of the grammar and generation of the parser using LPG, and including implementations for token coloring, outlining, text folding, reference resolution, content assist, two alternative documentation providers, and a builder.

For the Cobol IDE, the time required to implement various services has ranged from one hour or less (e.g., for the token colorer and source-text folder) to several hours (e.g., for the occurrence marker). The very simple Box IDE took less than one day. The Rascal IDE (with parser, token colorer, outlining, and a custom console) took an experienced IMP developer less than two days. The ToolBus Script IDE was developed by two people in two days. Despite the emphasis that we have placed on feature richness, sometimes a simple language-sensitive editor will meet a developer's needs.

In our experience, the most difficult part of building an IMP IDE may be defining the grammar. Grammer definition requires careful consideration for any non-trivial language. IMP shields the language developer from the complexities of IDE development, but we do not attempt to obviate the intellectual challenges of language design. For a complex language, the definition of the grammar may take more time than the construction of a basic IDE using IMP.

In order to provide an objective approximation of IDE development effort using IMP we measured the amount of custom code the IDE developer has to write in order to complete an IDE service implementation. Table 3 shows, for a representative set of services, the numbers of custom lines of code (LOC), the number of AST node types to which the service applies, and the corresponding LOC per AST node type. The top part of Table 3 shows four user-visible services and the bottom part shows four internal IDE services.

Most services shown in Table 3 require very little custom code, especially when considering the amount of custom code per relevant AST node type. Builders are a service that tends to be either trivial (e.g., LEG and PrefSpecs builder) or highly customized in a fairly complex way (e.g., X10 builder). Services like the occurrence marker and the tree-model builder, which are typically based on the Visitor pattern, may be relatively large (if there are many AST node types) but still conceptually simple. The amount of custom code for most reference resolvers is relatively small. The one exception here is for PrefSpecs which, unlike the other IDEs, builds its own symbol table rather than relying on one provided by the parser. Note that the Cobol IDE does not yet have a builder and documentation provider.

Some IMP services may not need any customization. The IMP framework provides a default hover helper implementation that works automatically with an existing reference resolver and documentation provider. This has worked well for the LEG, LPG, and X10 IDEs; the PrefSpecs IDE used a non-default hover helper to obtain more control over the source of documentation.

Table 3 shows that the service implementations for IMP-based IDEs are relatively small, often quite small in proportion to the number of AST node types involved. Considering that most of the service implementations are also fairly stylized (e.g., implementing a visitor), this demonstrates substantial success in isolating the language-specific elements of these service implementations and in simplifying their customization by IDE developers.

7.4 Enabling of Customization

IMP affords many points of customization to the IDE developer, such as customization of the parsing technology, the AST representation, and the editor. Beyond that, an IMP-based IDE can be customized with respect to the services or features it includes. With very few restrictions, the available services can be combined in almost arbitrary ways, and the

Service	LEG	Pref-Specs	LPG	Cobol	X10
Occurence marker (Template = 255 LOC)					
LOC	176	272	70	153	59
AST nodes	14	30	3	10	10
LOC/AST node	12	9	23	15	6
Token Colorer (Template = 46 LOC)					
LOC	8	38	26	27	12
AST nodes	4	26	6	14	9
LOC/AST node	2	2	4	2	2
Source folder (Template = 26 LOC)					
LOC	4	24	82	68	65
AST nodes	1	6	17	17	21
LOC/AST node	4	4	5	4	3
Builder (Template = 79 LOC)					
LOC	1	1	450	N/A	977
Document provider (Template = 38 LOC)					
LOC	8	79	17	N/A	576
AST nodes	3	31	4	-	35
LOC/AST node	3	3	4	-	16
Label provider (Template = 83 LOC)					
LOC	20	56	113	145	178
AST nodes	6	19	50	35	18
LOC/AST node	3	3	2	4	9
Reference resolver (Template = 23 LOC)					
LOC	5	178	11	5	60
AST nodes	1	2	1	2	14
LOC/AST node	5	89	11	3	4
Tree-model builder (Template = 44 LOC)					
LOC	22	133	247	258	56
AST nodes	5	19	26	35	19
LOC/AST node	5	7	8	7	3

Table 3. Size of templates and additional custom lines of code (LOC) for representative service implementation in representative IDEs. Also shown, the number of AST node types referenced in the service implementation and the corresponding LOC per referenced node type (approximate whole number).

IMP framework provides extension points for the introduction of arbitrary services.

Individual IDE services can be customized through parameters specified at the time that initial implementations are generated or through the way in which service implementations are completed or extended. To facilitate service implementation, we provide base classes and default, skeleton implementations for many services.

Some open points in the default implementations for selected services are shown in Table 4. However, customizations are not restricted to predefined points; any service can be implemented using an arbitrary program.

Following are some examples of customizations that were made in the IDEs we evaluated:

Service	Example Customizations
Syntax colorer	Definition of text attributes Assignment of text attributes to tokens Use of non-local information (e.g., from AST) Determination of region to re-color
Outliner	Included elements Element order Indentation structure Text labels and icons Use of extra information
Builder	Tests for file types (src, incl, ...) Marker IDs Dependency computation Compile method Overrides of defaults for message management, dialogs, ...

Table 4. Example customizations for selected IDE services

- The use of Polyglot (Nystrom et al. 2003) rather than LPG as the basis for representing AST nodes in the X10 IDE.
- The use of a javacup-based parser in the ToolBus Script IDE
- The use of a preexisting column-oriented editor in the Cobol IDE rather than the standard IMP editor
- The use of a custom console in the Rascal IDE based on the the ScriptConsole class from the Dynamic Languages Toolkit (eclipse.org/dltk).
- The provision of "language-oriented" documentation, rather than the more common "program-oriented documentation", in hover help for the PrefSpecs IDE
- The use of different algorithms for reference resolution in the X10 (prescreening link source nodes), LPG (no prescreening link source nodes), and PrefSpecs (use of token stream instead of the AST to find references)
- The recognition of specially-formatted comment tags (e.g. "TODO") in the X10 compiler
- The computation of cross-compilation unit dependencies in the X10 compiler

These are in addition to the selection of different service sets for different IDEs and more routine choices about syntax coloring, and outline construction.

7.5 Iterative, Incremental, and Selective Development

IMP supports the ability to freely select which services to provide in an IDE. No two of the evaluated IMP IDEs offer exactly the same set of services. Furthermore, development of an IMP IDE is incremental since services are generated one at a time and are mostly individually completed. This

incrementality is beneficial in that it allows IMP IDEs to be verified at each stage of development.

Iterative development is possible both for adding new services to an IDE and for refining or replacing existing services. However, not all incremental changes to an IMP IDE can be localized impact-free. For instance, the biggest crosscutting concern in any IMP IDE is inevitably the AST representation and node types. If the underlying grammar changes, the AST types will generally also change, with more or less impact. If new AST node types are added, the existing IDE services should continue to work, albeit ignoring the new node types. The services can be updated individually as needed to address the new types. If existing node types are modified or deleted, then dependent IDE services will have to be repaired before they can be recompiled and used. Better support for IDE evolution is a topic for future work.

Finally, an important advantage of IMP's incremental development is the ability to construct basic limited-functionality IDEs with very little effort. In situation where simple IDEs with limited functionality are sufficient, IMP provides an ideal framework to arrive at a solution quickly.

7.6 Accommodating Language Changes

Several of the above languages (notably LPG, PrefSpecs, and X10) were evolving while their IDEs were under development. This raises another important question in assessing the effectiveness of our framework: how difficult is it to keep an IDE implementation in sync with language changes? This section offers some qualitative experience to address that issue.

Specifically, the LPG grammar was refactored and augmented to effect various syntactic enhancements. In this case, the bulk of the IDE implementation (coloring, folding, outlining, reference resolution, etc.) was affected little, requiring only a few isolated lines of alteration. The grammar refactoring implementations and grammar analysis, however, were naturally impacted more significantly by the changes to the AST hierarchy.

Likewise, the PrefSpecs language was enhanced to support additional preference data types, multiple hierarchical preference pages, and to make various specification items optional. Again, the bulk of the IDE required little or no work to accommodate these changes. On the other hand, the PrefSpecs compiler that generates Java implementation classes for various user interface componentry, and its data structures and code generator required significant adaptation, as one would expect.

The X10DT, on the other hand, faced significant changes of two different kinds: (a) in the initialization and invocation of the underlying Polyglot compiler framework and the organization of compiler passes (which the X10DT's builder extends), and (b) in the AST hierarchy and type system APIs, in partial support of generic types. In essence, the former changes were more difficult to accommodate, requiring a nontrivial reworking of the life cycle of the IParseController and related classes. It seems unlikely that the IDE framework could do much to insulate the IDE developer from such problematic API changes without a tight integration between the IDE framework and the compiler, which would violate one of our key design goals (technology agnosticism). The second set of changes, on the other hand, corresponds more directly to language changes, and was actually much easier to accommodate, particularly as they were systematic.

In short, for the most part, the bulk of the effort in accommodating language changes lies mainly in the code that is intrinsically complex, as it should be.

8. Related Work

There is a long tradition of work on the automatic generation of programming-related tools from language definitions and related specifications (Reps and Teitelbaum 1984; Borras et al. 1988; Henriques et al. 2005; van den Brand et al. 2001). These systems differ from one another, and from IMP, in the form of language definition used, the particular tools or services generated, and aspects of processing, architecture, and infrastructure. Here, though, we are more concerned with the general goals and approach. All of these systems, including IMP, share the goal of simplifying the creation of programming tools, thereby making it easier to develop and adopt new programming languages, and improving the quality of programs and the programming process. All also make use of some combination of language-independent library components and language-specific generated components.

The main difference between IMP and the earlier approaches is that IMP puts a heavier emphasis on customization of the resulting IDE. In the previous work, once the tools are generated, they are done. This certainly minimizes the subsequent work of the IDE developer, but at the cost of customizability. With IMP, once tools are generated, additional customization is possible, if not required. In effect, IMP shifts the focus of customization from the language definition to the tool implementation. This means that the programmer typically has work to do subsequent to tool generation, but it affords very broad opportunities for customization. We have attempted to minimize the amount of work that is required of programmers, though, by providing default implementations where feasible, introducing specification languages to enable some implementations to be customized declaratively, and by stripping away most of the IDE-related parts of tool implementations so that the developer can focus on the language-related parts.

MPS (Jetbrains.com) has goals that are very similar to IMP's. Its authors claim that it is an implementation of Language Oriented Programming that facilitates the definition of domain-specific languages with full IDE support (code completion, navigation, refactoring, and more) that also allows the user to add specialized support (such as special editors). Unfortunately, not enough technical detail on this work

is available in the literature to enable us to make a detailed comparison to IMP.

The work on Gen-Voca (Batory et al. 2002) describes the application of a sort of aspect-oriented composition framework to the creation of variant IDEs for variants of Java. Language variants are defined by extensions to Java; IDE variants are defined by extensions of tools in the Java IDE. In contrast, IMP starts with a language-independent IDE framework, and it incorporates language-specific extensions through extension point and object-oriented mechanisms. Whereas (Batory et al. 2002) focused mainly on the combination of existing modules in alternative ways, IMP is focused on supporting the initial development of the modules. IMP might benefit from compositional technologies like Gen-Voca, but research in that area is beyond our current scope.

Marama is an Eclipse-based set of meta-tools for generating visual programming languages that support diagramming applications (Grundy et al. 2008). As with IMP, a goal for Marama is to enable the rapid development of simple modeling tools while allowing more complex tooling to be developed over time. Unlike IMP, both the meta-modeling tools and the resulting modeling notations are highly visual (although constraints and behaviors can be specified in textual formulae and, as a last resort, in Java). Also unlike IMP, but like most generation-based IDEs for textual languages, a major part of the framework and tooling are dedicated to execution support. Visual languages are beyond the current scope of IMP, but the prospect of combining meta-tooling for textual and visual languages is an interesting challenge for the future.

The Dynamic Languages Toolkit (DLTK) (eclipse.org/dltk) is an Eclipse technology project aimed at facilitating the development of JDT-like IDEs for dynamic languages. In this it shares some context and high-level goals with IMP. However, the DLTK and IMP differ in important ways. Since the DLTK is targeted toward dynamic languages it can adopt a "common tooling" approach: common AST representation, common runtime, and common services. It provides execution support for these languages and aims for language interoperability. IMP, in contrast, does not restrict its focus to a particular family of languages. To maintain language-independence the IMP framework provides language-neutral APIs to language-specific services. For example, it does not rely on any particular internal representation; it supports custom ASTs and accommodates the use of alternative compiler front-ends. IMP does not provide any special support for program execution or language interoperability. However, one of our goals for future work is to support IDE and language extensibility.

9. Open Issues

There is a clear tension between our desire to provide a framework that is independent of any particular language technology and our desire to make things easy for IDE developers, who obviously must select a specific language technology. As a consequence, our APIs are necessarily agnostic, e.g., using `Object` to represent tokens or AST nodes. To help resolve this tension, our runtime framework provides base classes to ease the implementation of language processing for LPG-based parsers and scanners, and program analyses for Polyglot-based front-ends. Similarly specialized base classes can easily be provided for other parser/compiler technologies and program representations. Better support for execution and debugging is a topic for future work.

Another challenge which language-technology independence presents is the ability to introspect on the target language's syntax or semantics. For example, the presentation specification language IDE would benefit from being able to query the sets of non-terminals and terminals (e.g. for content assist), regardless of the kind of parser generator being used. It requires carefully crafted APIs that are both agnostic of particular parsing technologies but accommodating to them. The `IParseController` interface does this for parsing services, but the analogous interface exposing language properties has yet to be designed. Such an interface would enable promoting certain development-time meta-tooling services that are presently language-specific into the language-independent framework.

Another ongoing concern is how best to create new IDE service implementations. This is presently handled by a combination of wizards, domain specific languages, generators, and Java implementation. All have proven useful, but the most appropriate balance among them is not clear. IMP currently relies to a significant degree on wizards to initiate the development of an IDE service. In so doing, many details of setting up the service can be hidden from the user, simplifying the process. However, the user may not have a clear mental model of the wizard's effect on the system. In particular, the user may wish to change a decision made when interacting with a wizard, but not know whether it is safe to re-run the wizard.[7] Moreover, decisions made when interacting with a wizard often "disappear into the code" (or the meta-data), making it hard to discover later on what choices were already made. In contrast, most IDE developers have well-defined expectations regarding compilers and builders. Thus, it may be more user-friendly to move more of the setup and creation of IDE services out of wizards and into a DSL compiler. This might keep the specification of IDE services simple while providing greater transparency to developers.

This raises another issue, however, that of the tradeoff between implementation via DSLs versus Java. The issue is exacerbated by the relatively large number of services supported by IMP. On the specification side, it is generally considered that DSLs are easier to write than imperative programs,

[7] Sadly, there seems to be little consistency in this regard among wizards in general.

since specifications tend to be more abstract and yet more concise.One extreme is to provide a separate DSL for each service. This requires learning a new language for each service – though each language may be relatively simple. The other extreme is to use one DSL for many services. Although this requires learning just one language, the language will naturally be more complex. On the programming side, the IMP IDE developer almost certainly has good knowledge of Java, and many IMP services can be implemented with very little code. As a result, the IDE developer may be more comfortable implementing services in Java than specifying them in one or more DSLs. Fortunately, IMP's use of the Eclipse extension point mechanism permits implementation of any given service by any of the above means, so that an IDE developer need not make a single choice for all services.[8] We are still evaluating the various approaches.

10. Conclusions

We have presented an approach to the construction of language-specific IDEs in Eclipse that draws on both automated generation and manual implementation. We do this by a combination of meta-tooling and carefully engineered runtime interfaces that cleanly separate language-specific concerns from the details of IDE and UI infrastructure and interoperation. For the IDE developer, this approach is intended to strike a flexible balance between customizability and ease of development. For the IDE user, the benefit should be IDEs that offer a good range of features that are well adapted to details of language syntax and semantics. This combination of functionality and language sensitivity appears to be critical for the widespread adoption of an IDE.

Our design goals represent a unique blend of elements that distinguishes our work from prior work in the area. In this regard, important aspects of our approach are our emphasis on language-specific customization, great flexibility in the selection of IDE features, neutrality with respect to language/parsing technology and program representation, and the ability to swap out major components of the architecture, such as the editor. This combination makes the approach a practical solution for a wide range of IDE development scenarios.

We have realized our approach in the Eclipse IDE Metatooling Platform – IMP (http://www.eclipse.org/imp/). IMP has been used, by us and others, to develop IDEs for a wide variety of domain-specific and general-purpose languages. IMP has been used to develop entirely new IDEs (often for new languages) and to help migrate existing IDEs onto Eclipse. Some of the IDEs support meta-tooling in IDE development (e.g., in support of grammar specification and parser generation), while others represent tooling for application development (e.g., in COBOL). Many of these IDEs are in regular use and most are publicly available.

[8] In fact, the Spoofax IDE uses a single DSL interpreter as the implementation for a significant subset of the IDE services that IMP supports.

We have also presented an evaluation of IMP that addresses (a) ease of IDE development, (b) feature sets and performance of IMP-based IDEs, (c) reduction of IDE development effort, (d) enabling of IDE customization, and (e) support for incremental, iterative, and selective IDE development. Our measurements and experience to date suggest that IMP significantly simplifies IDE development, produces practical IDE implementations, and significantly reduces IDE development effort. Overall, IMP appears to offer an effective solution to many crosscutting problems presented by this complex development domain.

One present limitation of our framework is that the abstraction of language-processing technologies and program representations makes it somewhat more difficult to provide execution and debugging support. Similarly, we currently have limited capability to introspect on grammar structure, which complicates providing certain meta-tooling services in a language-independent way. We are investigating how to provide additional leverage in these areas.

Another area of ongoing research is how best to support IDE developers, especially novice developers, in the implementation of IDE services. We currently use a combination of wizards, DSLs, generators, and manual programming. All are useful, but we continue to look for further opportunities to simplify and clarify the IDE development process.

Acknowledgments

This work is based in part upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

References

Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Didier Parigot, and Claude Pasquier. Smarttools: A generator of interactive environments tools. In *CC*, pages 355–360, 2001.

Don Batory, Roberto E. Lopez-Herrejon, and Jean-Philippe Martin. Generating product-lines of product-families. In *IEEE Conf. on Automated Software Engn.*, page 81, 2002.

Azad Bolour. Notes on the eclipse plugin architecture. `http://www.eclipse.org/articles/Article-Plug--in-architecture/plugin-architecture.html`.

P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *ACM Symposium on Practical Software Development Environments*, pages 14–24, 1988.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005. ISBN 1-59593-031-0.

Philippe Charles, Robert M. Fuhrer, and Stanley M. Sutton Jr. IMP: a meta-tooling platform for creating language-specific IDEs in eclipse. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 485–488, 2007.

eclipse.org/aspectj. AspectJ project. http://www.eclipse.org/aspectj/.

eclipse.org/dltk. Dynamic Languages Toolkit. http://www.eclipse.org/dltk/.

eclipse.org/jdt. Eclipse Java Development Tools. http://www.eclipse.org/jdt/.

Wan Fokkink, Paul Klint, Bert Lisser, and Yaroslav S. Usenko. Towards formal verification of toolbus scripts. In *AMAST 2008: Proceedings of the 12th international conference on Algebraic Methodology and Software Technology*, pages 160–166, Berlin, Heidelberg, 2008. Springer-Verlag.

M. Fowler. Language workbenches: The killer-app for domain specific languages? http://www.martinfowler.com/articles/language-Workbench.html.

Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1123–1134, New York, NY, USA, 2008. ACM.

John Grundy, John Hosking, Jun Huh, and Karen Na-Liu Li. Marama: an eclipse meta-toolset for generating multi-view environments. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 819–822, New York, NY, USA, 2008. ACM.

J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf—reference manual—. *SIGPLAN Not.*, 24(11):43–75, 1989. ISSN 0362-1340.

Pedro Rangel Henriques, Maria Joao Varanda Pereira, Marjan Mernik, Mitja Lenic, Jeff Gray, and Hui Wu. Automatic generation of language-based tools using LISA. *IEE Proceedings - Software*, 152(2):54–69, April 2005.

Jetbrains.com. JetBrains Meta Programming System. http://www.jetbrains.com/mps/.

R. Kadia. Issues encountered in building a flexible software development environment: lessons from the arcadia project. *SIGSOFT Softw. Eng. Notes*, 17(5):169–180, 1992. ISSN 0163-5948.

Lennar Kats and Karl Trygve Kalleberg. StrategoXT–Spoofax-IMP. http://strategoxt.org/Stratego/Spoofax-IMP, 2009.

lpg.sourceforge.net. LPG. http://www.sourceforge.net/projects/lpg/.

N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for Java. In *CC*, pages 138–152, 2003.

James M. Purtilo. The polylith software bus. *ACM Trans. Program. Lang. Syst.*, 16(1):151–174, 1994. ISSN 0164-0925.

python.org. Python. http://www.python.org/.

Steven P. Reiss. Connecting tools using message passing in the field environment. *IEEE Softw.*, 7(4):57–66, 1990. ISSN 0740-7459.

T. Reps and T. Teitelbaum. The synthesizer generator. In *ACM Symposium on Practical Software Development Environments*, pages 42–48, April 1984.

ruby-lang.org. Ruby. http://www.ruby-lang.org/.

Mark van den Brand and Eelco Visser. Generation of formatters for context-free languages. *ACM Trans. Softw. Eng. Methodol.*, 5(1):1–41, 1996. ISSN 1049-331X.

Mark van den Brand et al. The ASF+SDF meta-environment: A component-based language development environment. In *Computational Complexity*, pages 365–370, 2001.

Jurgen Vinju, T. van der Storm, Paul Klint, Bas Basten, and Arnold Lankamp. Rascal: A domain specific language for software analysis and transformation. Poster presentation, Scientific ICT-Research Event Netherlands (SIREN), Sep. 29, 2008, 2008.

Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

X10.sourceforge.net. X10. http://www.sourceforge.net/projects/x10/.