Monto: A Disintegrated Development Environment

Anthony M. Sloane, Matthew Roberts, Scott Buckley, and Shaun Muscat

Department of Computing, Macquarie University

Abstract. Integrated development environments play a central role in the life of many software developers. Integrating new functionality into these environments is non-trivial and forms a significant barrier to entry. We describe our Monto architecture which aims to address this problem. Monto components communicate via text messages across an off-the-shelf messaging layer. The architecture imposes limited constraints which enables easy combination of components to form an environment. A prototype implementation shows that this approach is practical and shows promise for full-featured development environments.

1 Introduction

Integrated development environments (IDEs) are an important part of the toolkit of many software developers. They provide facilities for editing, interrogating, transforming, running and debugging source code. Their integrated nature means that developers can perform all of these tasks without leaving the IDE.

In recent years, impressive progress has made it easier for software language engineers to extend IDEs. The IDEs themselves provide extension frameworks that allow new plugins to be combined with existing facilities. Language engineers have integrated their tools into these frameworks to achieve high-level specification of IDE components.

Despite this progress in bringing language engineering tooling closer to language designers and developers, tying that tooling to a particular IDE framework is a serious limitation. For example, the considerable effort used to develop an Eclipse plug-in for a new language probably doesn't provide any support for other environments. This tie-in also makes it harder for researchers to make new research results from language engineering accessible to practitioners. For example, having great tooling in Eclipse is of no help to developers who write their code in IntelliJ IDEA, Netbeans or a text editor. Requiring developers to move to a particular IDE platform is often not practical. Even if researchers can settle on an IDE they then have to make their tool infrastructure work with that IDE which may require language changes or other compromises.

An alternative to a highly coupled framework for IDE extension is one that aims to limit coupling to a bare minimum while still allowing feature integration. We call this sort of framework a *disintegrated development environment (DDE)* to indicate that it comprises parts that are as separate as possible but maintains

the overall goals of IDEs. This paper describes our prototype *Monto DDE*, its architecture and our preliminary experiences using it to build IDE-like facilities. Our goal with the Monto project is to explore a minimalist approach, its design and practicality; this paper reports our first steps.

We motivate Monto by discussing problems met by tool builders and developers that arise from a highly integrated approach (§2). We also discuss related work that environment builders have proposed to solve similar problems and upon which we build. These considerations led us to a view that a broadcast architecture should be used to reduce coupling between components. Communication should be as simple as possible to minimise overhead and enable components to be written quickly in any language.

The Monto architecture distinguishes between *sources* that publish notifications when changes to user-edited text occur, *servers* that provide functionality, and *sinks* that consume products from servers (§3). A *broker* mediates between sources, servers and sinks. All communication between Monto components is text encoded in JSON messages (§4). The ZeroMQ library [1] is used for fast communication between components. Using off-the-shelf technology for communication means that Monto components can be written in a wide variety of languages.

We discuss our experience with a prototype implementation of the Monto architecture (§5). We have implemented sources and sinks as plug-ins for the Sublime Text editor [2]. Our experiments show that components can be integrated with little effort using the Monto approach. Use of simple messages and a fast messaging layer means that interactive performance is good enough for live update even though many messages and processes are involved.

Our contribution is to show that this approach to building environments is practical and suffices to implement basic features of more integrated approaches. Future experiments are needed to explore more advanced functionality.

2 Motivation and Related Work

Integrated development environments such as Eclipse, IntelliJ IDEA and NetBeans provide powerful facilities for program development. However, it is widely agreed that developing plug-ins for non-trivial new languages in these environments is not for the faint-hearted. Success stories such as the Java Development Tools in Eclipse are the product of many years of development by many developers. Effort on this scale is beyond all but the most well-resourced organisations.

Many researchers have attempted to address the difficulty of adding support for new languages to IDEs or similar systems. Most notably, the IDE Metatooling Platform (IMP) for Eclipse [3, 4] abstracts the Eclipse framework to make it easier to build language-specific services. The Spoofax/IMP project integrates the Stratego term rewriting language and related domain-specific languages into Eclipse [5, 6]. Language designers can easily use Spoofax to develop custom support for new languages, including syntax highlighting, code folding and name definition-use navigation. All of these facilities integrate well with the rest of Eclipse. As the name suggests, Spoofax/IMP is based on an evolution of

IMP rather than on the core Eclipse frameworks. Spoofax is a form of language workbench which is a category of environment specifically designed to make it easy to build new language support [7]. There are many other workbenches that implement different approaches to language specification. For example, the Meta Programming System (MPS) provides a general editing framework in which new languages can be specified by defining abstract syntax, projections from that syntax to text, analyses, code generation, and so on [8].

Text editors are the other main kind of front-end used for software development. Similar to IDEs, editors often provide plug-in architectures, but usually operate at a lower level. For example, most editor extension mechanisms rely on text-based processing such as regular expression matching to perform syntax highlighting, in contrast with IDE plug-ins that usually integrate full parsers. The tendency in editor plug-in frameworks is to make it easy to add extensions, usually at the cost of having to operate in a reasonably primitive environment. Some editors support sophisticated extensions that reuse existing infrastructure. For example, the ENhanced Scala Interaction Mode for Emacs (ENSIME) project reuses the Scala compiler to provide support for IDE-like features in a Scala programming mode for Emacs [9].

Much of this work on providing language-support in IDEs, workbenches and text editors is impressive, but it is based on a fundamental assumption. Developers of new language tooling are expected to use a specific platform, such as Eclipse plus Stratego, MPS or Emacs. This assumption means that it is non-trivial to use this tooling in other settings. For example, there is no easy way for a developer who prefers the IntelliJ IDEA environment to use Spoofax.

The difficulty of using tooling in different integrated contexts leads to general component architectures for software tooling. The idea is to develop a framework in which a variety of tools can cooperate, yet remain somewhat separate. Communication between tools allows them to exchange information. A primary motivation for our work is the ToolBus coordination architecture which is based on message passing [10]. ToolBus has been used to develop coordinated tooling in the language engineering space [11]. Another related approach is embodied in the Linda coordination language which bases communication between parallel processes around a shared store of general data tuples [12].

Architectural approaches such as ToolBus and Linda are a step in the right direction since they allow individual separated components to provide functionality while the framework handles the communication between components. However, they still impose non-trivial integration requirements. For example, ToolBus uses a process algebra-based scripting language to describe how tools interact. While such a description undoubtedly provides benefits, it does impose a barrier to entry. Linda requires custom support to access the tuple store.

These considerations led us to wonder whether we could reduce coupling between components even further while still employing a largely decoupled approach in the style of ToolBus and Linda. The novel aspect of our solution is to disintegrate as much as possible and remove the need for a coordination language by simplifying the interaction between components. In the Monto architecture

components play simple defined roles and are unaware of the existence of other components. No overall coordination specification is required. In architectures like ToolBus many different kinds of messages are sent between components. We reduce the number of message types to two. Moreover, we follow the lead of Unix and Web technology by only sending text messages with a simple structure to keep coupling low.

3 Monto Architecture

Monto contains *sources*, *servers* and *sinks* (Figure 1). The components run independently either as separate processes or as threads in one or more processes, all of which may be running on a single machine or on multiple machines. Most likely a single process will interact with the user by operating as both a source and multiple sinks, while many servers run as separate local or remote processes. A source reacts when text is changed by a user (step 1). The source publishes a complete version of the changed text (step 2) which is passed to servers by a broker (step 3).

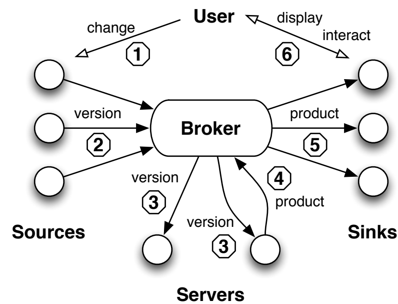

Fig. 1. Monto architecture overview.

In this paper we assume that versions are sent in a fine-grained manner so that each change results in a separate message. A typical source might be a plug-in for a text editor that is triggered each time the user makes a modification to a file buffer. To keep things simple in our prototype, the broker passes on every version to every server. A registration scheme could easily be added to reduce message traffic but we haven't found it to be necessary.

Servers react to versions by sending responses that contain products which are derived from the version text (step 4). A single server may respond to every version or just to certain ones. For example, a server that knows how to perform semantic analysis checks for a particular language will only respond versions written in that language, whereas one that provides information about version control status will respond to every version that involves a tracked file.

The broker passes the products to the sinks (step 5). Usually a sink will display some part of the product to the user, possibly inducing some further user interaction (step 6). As for servers, sinks are often designed to only react to certain kinds of product. A typical sink might react to a product containing an outline by showing the outline in a text editor buffer or IDE view. A sink that knows how to handle text completion might display options from a completion product so that the user can select one.

The Monto architecture is specifically designed to minimise coupling between the components. The broker exists so that sources do not need to be aware of the identity or location of the servers and sinks. Similarly, servers can work without

having to be aware of the sinks that consume their products. Sinks do not need to know anything about the servers that produce the products they receive.

4 Communication

The choice of communication technology directly affects which languages can be used to implement Monto components and hence indirectly influences which other technologies can participate. For example, choosing a Java-specific communication mechanism would mean that JVM-based languages could easily be used but others would be ruled out. Basing things on Java would mean that Eclipse and other Java-based IDEs would be able to participate as sources or sinks, but text editors that are implemented in C could not be easily incorporated.

We use the ZeroMQ messaging technology [1] to implement communication in Monto. ZeroMQ is a convenient layer on top of basic socket-level communication, but otherwise does not impose any constraints on the information that can be communicated. ZeroMQ-compatible libraries exist for most mainstream languages, so it is easy for components to interoperate without sharing an implementation language. ZeroMQ is also very fast since it imposes minimal overhead above the basic communication layer. Speed is important since messages from sources to servers to sinks are being used to provide interactive functionality.

Monto sends messages over ZeroMQ sockets as text and the ZeroMQ layer takes care of issues such breaking large messages into smaller pieces and reassembling them at the other end. ZeroMQ also takes care of queueing messages. Sending a message using ZeroMQ is typically a couple of lines of code. Servers and sinks use a blocking operation to wait for an incoming message to arrive.

Message formats. The choice of message format strongly affects the simplicity and power of the framework. Using message formats that make few assumptions about the information that is being communicated means that the framework will not impose too much on the way that it can be used. For example, if changes were notified by sending an abstract syntax tree of the version according to some grammar, we would make it inconvenient to write servers that wish to process the version as lines of text, perhaps to perform a spell check.

The simplest message format we can use is uninterpreted plain text. However, it is useful to have slightly more structure so that servers and sinks have something by which to discriminate between messages. We use the JSON structured text format. As for ZeroMQ, an advantage of JSON is that encoders and decoders are available for many languages.

Version messages. Messages that describe a version contain:

- source: a unique string that identifies the source of the version,
- language: the name of the language in which the source is written,
- contents: the complete text of the version, and
- selections: objects that describe the current selected regions in the source.

The source string is usually the name of a file that is backing the content that is being edited. The contents of a version message do not necessarily correspond to the current contents of the file since the user may not have saved it.

The language field is used so that servers can react only to text that is written in a language that they understand. The language "text" is used if no other language is suitable. Using a string to encode the language is the simplest method, but introduces some imperfections. For example, who determines which language names are acceptable in a version message? We could specify the legal names up-front using some form of enumeration type, but we stick with a string so that the framework is as flexible as possible. Coordination of language names must be done by convention outside the framework.

The contents field contains the complete text of the version. One obvious possibility for modification of this design is to send just the nature of the change itself. Our view is that this kind of extension would complicate the messaging too much and would tie the framework too closely to particular kinds of changes. The price we pay is that servers may be recalculating information that could have otherwise been determined in a more incremental fashion, or they must become somewhat stateful. So far we have not found this to be a limitation.

Most sources have some notion of the *current selection* which describes the editing position in the text and what, if any, of the text has been highlighted by the user. The selection field of a version message supports servers that take the user's current focus of attention into account. For example, a server that determines completion possibilities needs to know where the cursor is.

Product messages. Messages that communicate a product contain:

- source: the unique identifier of the source to which this product relates,
- product: the type of the product,
- language: the language in which the product text is written, and
- contents: the content of the product as text.

The source field is used to associate the product with the source of the version that triggered it. Sinks can react to products that pertain to a source in which they are interested. For example, a sink that is waiting for a code completion product would react to products that apply to the initiating source.

The product field identifies its type and is used by sinks to react only to products that they can handle. For example, a sink that wants to display an outline view for any source would ignore the source field but check that the product field indicates an outline. Monto enforces no discipline on product names, so like language name they must be agreed by convention outside the framework.

The language and contents fields are used similarly to their role in version messages. Sometimes a server will produce text in a particular language. For example, if the server is formatting the version text then the product language will be the same as that of the corresponding source message. If the server is compiling Java code then the product language might be "JVM byte code".

Version and product messages can contain extra fields to communicate information above the basic level mandated by the framework. For example, a particular source might include information about the change that created a version, in case that information is of use to a server but would be hard for the server to calculate itself. Similarly, a server can provide extra information

in a product message for use by sinks. This sort of extra information would be provided and used by convention between developers of Monto components.

Running Monto. Monto consists of a loose collection of components that run autonomously. A script simplifies starting and stopping the broker and any servers that the user desires to use. The script is driven by a simple configuration file that specifies paths, command-line arguments, etc.

To avoid overwhelming the servers with many small changes to the same source in a small period of time, the broker collects only the most recent version message for each source and periodically sends it to the servers. The timing has been adjusted to balance between sending too many messages to the servers and not reacting quickly enough for good interactive use. The broker can be implemented in any language that can communicate using JSON messages over ZeroMQ. In our prototype it is implemented by about 40 lines of Python but can easily be implemented in a compiled language if speed becomes a problem.

Other than the broker and servers, the user must also run sources and sinks. Normally these components will be implemented by plug-ins in an editing environment of the user's choice so they will be automatically started up when that environment starts or as the result of user commands.

5 Experience

We have been experimenting with the Monto prototype framework to build various sources, servers and sinks. Our aim so far has been to explore to see if our simple approach is sufficient to encompass typical IDE-like functionality. We particularly wanted to see whether an approach that requires sending messages between components performs well enough to make a usable environment.

Sublime Text. Our current experiments use the Sublime Text 3 editor [2]. We have built a Monto plug-in for Sublime Text in 250 lines of Python. The plug-in relies on 100 further lines that are independent of Sublime Text and can be used by any Python-based Monto component. When the plug-in is loaded, Sublime Text acts as a source for any buffer that the user is editing. A version is published each time a buffer is created, modified and when a selection is moved.

The plug-in provides a command by which the user can create new views that display Monto products, which we call *Monto views*. A Monto view can optionally display products that relate to all sources or just those for the source that held the focus when the command was run. Similarly, new arrivals of a product can be appended to the existing text in a Monto view or replace it.

Figure 2 shows a Sublime Text window editing a factorial program written in the Java-subset language MiniJava (top left). The user has three Monto views to display products: the abstract syntax tree of the program (a form of outline, top right), the abstract syntax tree pretty-printed as MiniJava code (bottom left), and a translation of into Java Virtual Machine bytecode (bottom right).

The Monto views in Figure 2 are updated continuously as the developer edits the program. Adding a new local variable declaration in the `ComputeFac` method will cause a new node representing that declaration to appear in the tree view, a

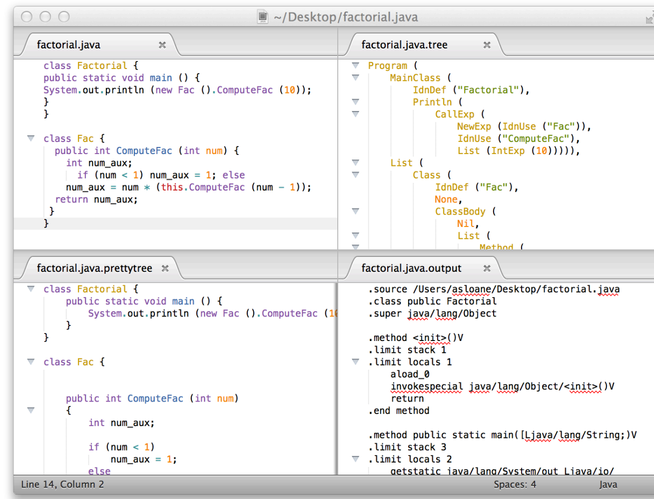

Fig. 2. Sublime Text: MiniJava factorial program and three Monto views.

pretty-printed version of that declaration to appear in the pretty-print view, and the bytecode to be updated to reflect that a new local variable slot is needed.

All of these updates happen nearly instantaneously so the overhead of interpreting messages and reacting to them appears to be low. This observation confirms that at least for basic functionality the performance of a Monto-based environment is sufficient for interactive use. We have not conducted a comprehensive benchmark against other alternatives and we make no claim about how more advanced features will perform since those features are part of future work.

The plug-in provides other ways in which Monto products can be used. For example, a product containing formatted source code might replace the current selection. A product can also be used by a code completion command as suggestions in a pop-up menu. In fact, since a product is just text, the only limitation on the way it can be used is the capability of the editor.

Any other extensible text editor or IDE could play the role that Sublime Text does in our experiments. All that is required is a way to detect when the user has made changes to the text that they are editing, a way to send a ZeroMQ message containing that version, and a way to react to products coming in from servers. If an editor can be extended in Python it can reuse the Monto library used by the Sublime Text plug-in. Otherwise, similar functionality would need to be implemented in that editor's extension language.

There is no requirement that a single program act as both the source and sink as Sublime Text does. For example, products that result from changes happening in one editor can be displayed in another one. A fan-out structure could be used to send products to more than one viewer, so that multiple developers can observe editing as it happens during a pair coding session. A server that simply reflects versions back out as products would enable live observation of editing, but observing developers could also create other views as needed. For example, a

server that automatically runs tests on changed code could report to developers who are running a test result display sink.

MiniJava compiler server. One typical use of a DDE is to interface with existing compiler code. Rather than duplicate the compiler code within the environment, we wish to reuse it. In fact, the products shown in Figure 2 were produced using a server that is a small extension of existing Scala code for a MiniJava compiler. The compiler is written using our Kiama language processing library [13, 14]. 90 lines of Scala wraps any Kiama compiler so that it acts as a Monto server; no modifications must be made to the compiler code. The wrapping code uses off-the-shelf Java libraries for JSON encoding and ZeroMQ.

If a syntax error is introduced in the MiniJava source then the products shown in Figure 2 will be empty since those products are not defined when the version text doesn't parse. The MiniJava compiler also has an error product that reports any syntax or semantic errors from the compilation process. Thus, if desired, a developer can augment the shown views with one that continuously updates with the current compiler error messages.

Wrapping text-based tools. Many command-line tools exist that would be of use in a development environment but were developed independently with their own user interface. For example, there are many lint tools that produce a textual report of code quality problems beyond those diagnosed by compilers. We have built a wrapper script to enable these sorts of tools to be used with other Monto components. The script runs as a server that executes a shell command each time a version is received, captures the output of the command, and sends it back as a product. It is easy to use this wrapper to incorporate the output of those tools in a Monto view so that, for example, lint reports can be viewed in the editor and are updated automatically as code changes.

6 Conclusion and Future Work

Our initial experiments have shown that a minimalist disintegrated development environment approach has some promise. With a relatively small amount of effort we were able to build a simple framework that provides an editing experience with quick feedback to source code changes. By factoring the framework into independent components that communicate via simple messages, we do not require component developers to buy into a complex framework. Having said that, we do not claim on this evidence that a Monto-based environment can rival well-established IDEs with complex plug-ins.

Current work is investigating more advanced facilities, how they fit into a disintegrated world and whether our simple framework is sufficient to support them with acceptable performance. Of particular interest is the ability of Monto to incorporate servers that reside across the network, perhaps to provide access to functionality that is impossible or hard to install on a local machine. Some other areas of current investigation are: source mapping to relate product text to version text; incorporation of a project view so that servers can work at the project level not just at the file level; products that are HTML or SVG and

sinks that are web browsers; sinks that display graphical output; build feedback; execution-based products for live coding, debugging and testing; wrapping version control tools; and read-eval-print-loop-based servers.

Acknowledgements. Štěpán Šindelář provided useful feedback on the paper. We also thank the anonymous reviewers for their helpful suggestions.

References

1. Hintjens, P.: ZeroMQ: Messaging for Many Applications. O'Reilly (2013)
2. Skinner, J.: Sublime Text 3. http://www.sublimetext.com/3
3. Charles, P., Fuhrer, R.M., Sutton, Jr., S.M.: IMP: A meta-tooling platform for creating language-specific IDEs in Eclipse. In: Proceedings of Conference on Automated Software Engineering, ACM (2007) 485–488
4. Charles, P., Fuhrer, R.M., Sutton, Jr., S.M., Duesterwald, E., Vinju, J.: Accelerating the creation of customized, language-specific IDEs in Eclipse. In: Proceedings of Conference on Object Oriented Programming Systems Languages and Applications, ACM (2009) 191–206
5. Kats, L.C.L., Kalleberg, K.T., Visser, E.: Domain-specific languages for composable editor plugins. In: Proceedings of the Workshop on Language Descriptions, Tools, and Applications. Volume 253 of Electronic Notes in Theoretical Computer Science., Elsevier (April 2009) 149–163
6. Kats, L.C., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: Proceedings of Conference on Object Oriented Programming Systems Languages and Applications, ACM (2010) 444–463
7. Erdweg, S., Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G.D., Molina, P., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., Vlist, K., Wachsmuth, G.H., Woning, J.: The state of the art in language workbenches. In: Software Language Engineering. Volume 8225 of Lecture Notes in Computer Science. Springer (2013) 197–217
8. Voelter, M.: Embedded software development with projectional language workbenches. In: Model Driven Engineering Languages and Systems. Volume 6395 of Lecture Notes in Computer Science. Springer (2010) 32–46
9. Cannon, A.: Enhanced Scala Interaction Mode for Emacs (ENSIME). https://github.com/ensime/ensime-src
10. Bergstra, J., Klint, P.: The discrete time ToolBus—a software coordination architecture. Science of Computer Programming **31**(23) (1998) 205–229
11. van den Brand, M.G.J., van Deursen, A., Heering, J., Jong, H., Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The ASF+SDF Meta-Environment: a component-based language development environment. In: Proceedings of Conference on Compiler Construction. Volume 2027 of Lecture Notes in Computer Science., Springer (2001) 365–370
12. Ahuja, S., Carrier, N., Gelernter, D.: Linda and friends. Computer **19**(8) (1986) 26–34
13. Sloane, A.M.: Lightweight language processing in Kiama. In: Generative and Transformational Techniques in Software Engineering III. Volume 6491 of Lecture Notes in Computer Science. Springer (2011) 408–425
14. Programming Languages Research Group, Macquarie University: The Kiama language processing library. http://kiama.googlecode.com