IDE as Code : reifying language protocols as first-class citizens

Pierre Jeanjean

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Pierre JEANJEAN

IDE as Code

Reifying Language Protocols as First-Class Citizens

Thèse présentée et soutenue à Rennes, le 29/04/2022
Unité de recherche : Inria, IRISA, Équipe DiverSE

Rapporteurs avant soutenance :

Mark VAN DEN BRAND	Professeur	Eindhoven University of Technology
Richard PAIGE	Professeur	McMaster University

Composition du Jury :

Attention, en cas d'absence d'un des membres du Jury le jour de la soutenance, la composition du jury doit être revue pour s'assurer qu'elle est conforme et devra être répercutée sur la couverture de thèse

Président :	François TAÏANI	Professeur	Université Rennes 1
Examinateurs :	Sebastian ERDWEG	Professeur	Johannes Gutenberg-Universität
	Richard PAIGE	Professeur	McMaster University
	Emma SÖDERBERG	Maître de Conférence	Lund University
	Mark VAN DEN BRAND	Professeur	Eindhoven University of Technology
Dir. de thèse :	Benoit COMBEMALE	Professeur	Université Rennes 1
Co-dir. de thèse :	Olivier BARAIS	Professeur	Université Rennes 1

ACKNOWLEDGEMENTS

While it is not mandatory at all to write an acknowledgements section, I still feel the need to write a few personal notes to finally accept that I'm done. I have always been told that "a good thesis is a finished thesis", it might sound silly but I cannot stress out enough the wisdom carried through this sentence. 3.5 years, that's how long it took to get there. With a global pandemic right in the middle... Yeah, that was fun. And now, here I am, with a complete manuscript, a succesful defense, a doctor's degree, and still SO many questions. "Could I have done something better?" "Should I have worked harder?" "Is there even a point to all of this?" "What now?" Impostor syndrome is a terrible condition, but I guess it's also the standard for a PhD student. But now, finally, the thesis is finished. I might have fallen off a horse a few times along the way, but I got back up, and finally I ate that horse. So, no matter what I may think now, it's time to accept that it's a good thesis. And if you, reader, are another PhD student stumbling upon these ramblings, I hope that you will be able to get there too.

Anyway, the first people I really need to thank are the ones that spent all this time with me: basically everyone from the DIVERSE reasearch team. I honestly cannot imagine reaching this point with any other team. Thank you so much to my two advisors, Prof. Benoit Combemale and Prof. Olivier Barais, for giving me this position and being such a driving force for the entire duration (despite overly busy schedules and multiple lockdowns). Thank you to Didier, Manuel and Dorian for mentoring me. Thank you to Gwendal for consistently interrupting my work, and also managing the defense. Now, I will not attempt to write an exhaustive list of all the members I met, but thank you all for the good mood, the support, and all the interesting ideas birthed by coffee breaks.

Next, I need to thank the members of the jury who accepted to review this work. In the end, you were the ones who made it possible to transform this time into a good thesis. I took your feedback into consideration, tried to apply as much as your suggestions as I could, and overall I definitely enjoyed exchanging with all of you.

On a more personal note, I have to thank my friends and family. Thank you Mom and Dad for bringing me here, and believing without any doubt I would get there at some point. Thank you Julie and Aurore for also believing and even assisting to the defense.

Thank you Mimi and Fidji for being here, even though you can't read I really need to include you. I also thank everyone from JAPAN7 for the all the silliness and for managing to constantly steal my attention. You are way to many though so I won't even try to name you. The guys from THE FRIENDS OF USUALLY (this is not a typo) too, we could have met more in this period but I'm glad we still did.

If you feel that you were not named here but should have been, well, sorry, but it's actually really hard to write an acknowledgments section. Every person that has helped me at some point probably deserves to be here but it's already long enough. I will just end this overly cheezy thing with a final thanks: music has always been a major part of my life, and consequently of this work too, and as such I want to thank Haru Nemuri, my favorite artist, for bringing me so many feelings when I needed them the most. She rules.

TABLE OF CONTENTS

INTRODUCTION EN FRANÇAIS

Contexte

L'ingénierie logicielle moderne consiste essentiellement à utiliser les langages de programmation les mieux adaptés à une tâche donnée, ce qui suppose de disposer du support d'outils permettant d'utiliser ces langages efficacement. Les environnements de développement intégrés (IDE) ont été initialement conçus comme un moyen de rassembler dans un seul logiciel les différents services d'un langage donné (par exemple, les services d'édition, de débogage, de vérification, de compilation, ...). Cependant, nous voyons de plus en plus de projets tirer parti des avantages de plusieurs langages de programmation en même temps, que ce soit pour des parties isolées (par exemple, dans le cadre de développements full-stacks, ou pour des architectures microservices) ou même dans la même base de code avec la tendance récente du *développement polyglotte* [37], [80]. Par conséquent, il est aujourd'hui nécessaire d'avoir accès aux services de plusieurs langages au sein d'un même environnement.

Pour éviter le développement de services de langages spécifiques pour chaque IDE existant, les protocoles de langage sont devenus ces dernières années un sujet d'intérêt dans la communauté de l'ingénierie des langages ([21], [65], [68]). En communiquant par le biais de protocoles bien définis, les principaux services de langage peuvent être réutilisés dans les différents IDE supportant ces protocoles. Une conséquence directe est que la responsabilité de fournir un support approprié pour un langage spécifique n'est plus du ressort du fabricant de l'IDE, mais incombe aux mainteneurs de langages qui développent les services indépendamment de toute plateforme particulière. Le premier protocole de langage, à savoir le Language Server Protocol (LSP) [1], a été proposé par Microsoft dans le cadre du développement de VS Code [2]. Le rôle de LSP est de prendre en charge les services d'édition communs de n'importe quel langage en fournissant une implémentation de serveur de langage conforme à des spécifications ouvertes et consultables librement. Il a été conçu autour d'un ensemble de services qui ont été

1. cf. https://microsoft.github.io/language-server-protocol/
2. cf. https://code.visualstudio.com/

extraits d'éditeurs de code spécialisés pour les langages généraux les plus couramment utilisés. LSP, le Debug Adapter Protocol (DAP) [3], et la plupart des autres protocoles de langage que nous voyons aujourd'hui spécifient la structure des données échangées entre un client unique (un composant de l'interface utilisateur d'un IDE) et un serveur unique (backend fournissant l'ensemble des services nécessaires au client), ainsi que les requêtes et les événements qui peuvent être envoyés de l'un à l'autre. La plupart des messages sont inclus dans ce que l'on appelle les « capabilities ». L'ensemble des capabilities est défini et fixé par les spécifications du protocole. L'idée est que les clients et les serveurs peuvent choisir d'implémenter un sous-ensemble des capabilities et d'en informer les autres, qui devraient être en mesure d'utiliser tous les messages correspondants, conformément à la spécification.

Énoncé du Problème

Avec le succès de LSP et de DAP, nous voyons apparaître de nouveaux protocoles de langage pour de nouveaux cas d'utilisation (par exemple, le Build Server Protocol [4] ou le Test Adapter Protocol [5]) et pour des fonctionnalités spécifiques mal prises en charge par les protocoles existants (par exemple, l'utilisation d'une syntaxe graphique dans LSP [68]). Le cas de la syntaxe graphique est particulièrement intéressant car il a conduit à la définition de deux protocoles différents pour le même objectif : le Graphical Language Server Protocol [6] et le Graphical Server Protocol [7]. Une autre façon d'étendre les fonctionnalités de LSP, qui a été utilisée dans des travaux tels que [60] et [50], consiste à ajouter arbitrairement le support de nouveaux messages à un serveur et à un client, et à considérer que toutes les implémentations existantes les ignoreront si elles ne les supportent pas. Bien que ces deux travaux apportent des contributions très utiles, une telle implémentation soulève des problèmes de maintenabilité et d'interopérabilité, ce qui motive davantage notre travail.

Sur le long terme, il serait contre-productif de continuer à créer des protocoles indépendants pour chaque cas d'utilisation, car cela irait à l'encontre de l'objectif de la définition de ces protocoles, à savoir : assurer un support approprié des différents

3. cf. https://microsoft.github.io/debug-adapter-protocol/
4. cf. https://build-server-protocol.github.io/
5. cf. https://github.com/microsoft/vscode-debugadapter-node/issues/154
6. cf. https://www.eclipse.org/glsp/
7. cf. https://obeonetwork.github.io/GraphicalServerProtocol/

services dans tous les environnements de développement. Dans une situation réelle, cela déplacerait le défi sur la composabilité et la compatibilité des protocoles existants, qui ne sont actuellement pas pris en compte. Par conséquent, au lieu d'avoir un *serveur* qui englobe tous les services pour un langage donné, nous explorons l'idée de le décomposer en services individuels interagissant les uns avec les autres. Pour assurer la communication entre ces services et le(s) *client(s)*, ainsi que leur chorégraphie, nous considérons les spécifications requises explicitement pour supporter toutes leurs interactions. Ainsi, au lieu de définir un protocole de langage à partir de zéro, lié à un cas d'utilisation spécifique, nous envisageons la définition d'un méta-protocole de langage à partir duquel nous pourrons dériver les instanciations les plus pertinentes (c'est-à-dire les configurations des services de langage). Grâce à cette approche, certaines parties des protocoles existants qui sont en fait génériques pourraient être séparées, et leurs impémentations mieux exploitées en étant réutilisées dans de multiples configurations de services. Cela apporte également plus de flexibilité à l'architecture globale : les services spécialisés peuvent être déplacés vers d'autres machines ou remplacés en fonction du cas d'utilisation, des fonctionnalités peuvent être ajoutées ou supprimées à tout moment, et il devient possible de contrôler finement le déploiement de chaque service indépendamment. Nous appelons cette vision *IDE as Code* (ou « Environnements de Développement Programmables »).

Comme l'illustre la Figure 1, nous attendons des concepteurs de langage qu'ils fournissent non seulement les spécifications de leur langage, mais aussi des informations sur les interactions protocolaires requises par les différents services. À partir de là, nous pouvons obtenir des « paquets de services de langage » qui sont des services de langage, indépendants et minimaux, qui interagissent les uns avec les autres. En utilisant ces paquets, les utilisateurs devraient pouvoir spécifier la configuration et le déploiement de leur IDE (ou utiliser des IDE prédéfinis) pour mettre en place un environnement de développement adapté à leurs besoins, en fonction de leur environnement de travail et de leur cas d'utilisation.

Les développeurs ont besoin d'un accès optimal aux services de langage, quels que soient leur lieu de travail et leur équipement. Avec la pandémie de COVID-19, il est devenu encore plus évident qu'il n'est pas toujours possible d'assurer l'accès à un environnement de travail stable, et de plus en plus d'efforts sont consacrés au développement d'IDEs infonuagiques footnotecf. https://ecdtools.eclipse.org/events/idesummit/2021/. Bien qu'ils permettent à leurs utilisateurs d'accéder à un environnement de développement persistant depuis n'importe où, ils présentent également des défauts qui doivent

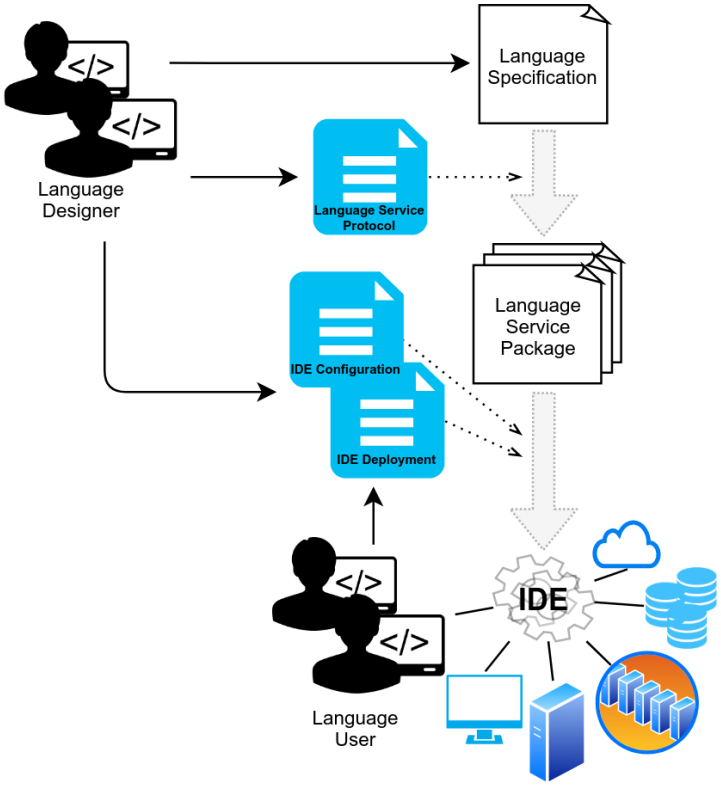

FIGURE 1 – Vision d'Ensemble de *IDE as Code*

encore être abordés : la nécessité d'un accès Internet stable et rapide, qui peut fortement fluctuer dans des situations telles que des voyages, une personnalisation limitée, et une mise à l'échelle insuffisante pour offrir à tous leurs utilisateurs une expérience transparente en même temps. Des travaux tels que [25] ont déjà pris en compte ces limitations et ont exploré l'idée de séparer les services de langage en microservices, avec des mécanismes de redéploiement dynamique, soit localement soit sur différents serveurs, comme solution à ces problèmes.

En ce qui concerne la prise en charge de différents cas d'utilisation, nous pouvons par exemple considérer les experts de domaine habitués à travailler avec un langage dédié via une syntaxe graphique, qui n'ont pas besoin d'une représentation textuelle ni des services pour la manipuler. Plusieurs études telles que [69] soutiennent que les professionnels peuvent souvent se sentir dépassés par leurs environnements lorsque ceux-ci offrent trop d'outils et de services. Cela peut être contre-productif car il est difficile de mettre en valeur les fonctionnalités les plus utiles lorsque les utilisateurs sont noyés sous les fonctionnalités inutiles. Par conséquent, l'un des objectifs de cette approche est de permettre aux utilisateurs de disposer d'un environnement propre et minimaliste où ils n'ont accès qu'aux outils qu'ils utilisent réellement. D'autre part, la connaissance que peuvent apporter les concepteurs de langages, en ce qui concerne l'ensemble des fonctionnalités les mieux adaptées à un cas d'utilisation particulier, est également précieuse, de sorte qu'ils devraient pouvoir spécifier des configurations d'environnements par défaut.

Les composants que l'on peut trouver dans un environnement de développement sont nombreux. Ils peuvent aller d'un analyseur syntaxique pour la syntaxe textuelle d'un langage à un débogueur pour l'exécution d'un programme. Dans cette thèse, nous mettons l'accent sur les composants d'exécution, c'est-à-dire les composants qui permettent et dirigent l'exécution des programmes (par exemple, les interpréteurs ou les débogueurs), dans le contexte des environnements interactifs. Nous abordons trois défis dans ce contexte :

— Le premier concerne directement la manière dont ces différents composants vont communiquer entre eux. Une simple approche client/serveur avec des protocoles fixes n'est pas suffisante pour supporter la customisation d'un tel environnement. Par conséquent, il est nécessaire de manipuler directement les protocoles de langage comme des objets de première classe, tout comme les composants qui les implémentent.

— La seconde consiste à établir une hiérarchie entre les différents composants d'exécution, et d'en déduire le plus basique qui peut être utilisé pour diriger les autres. L'objectif principal est d'améliorer la réutilisation des implémentations de services, tout en permettant de les standardiser autour d'APIs minimales et bien définies. Cela signifie également que nous devons spécifier les dépendances entre ces composants et définir les protocoles dont ils ont besoin pour interagir les uns avec les autres.

— Et la troisième consiste à produire réellement les composants du langage qui répondent aux spécifications des protocoles, en utilisant les informations qui existent déjà dans les spécifications des langages et en appliquant un processus possiblement génératif.

Contributions

La contribution principale de cette thèse est le concept d'*IDE as Code*. Nous discutons d'une implémentation possible d'outils de langage et de fonctionnalités fournis sous forme de microservices indépendants, qui peuvent être déployés à la demande par les utilisateurs pour adapter leur environnement de développement à leurs besoins. La configuration de l'environnement étant externe à toute plateforme spécifique, elle peut être utilisée pour mettre en place n'importe quelle infrastructure, qu'il s'agisse d'un IDE entièrement local ou d'un environnement dans le web, lié à un dépôt de code dans une forge telle que GitLab[8]. En manipulant les protocoles de langage comme des objets de première classe, nous pouvons tirer parti de la réutilisabilité des services qui les implémentent, et même définir une hiérarchie avec des protocoles spécifiques étendant ou utilisant une composition d'autres protocoles. Ceci est particulièrement intéressant lorsque nous prenons en compte les nombreux outils d'exécution qui peuvent être nécessaires pour un DSL donné, tels que : exécution complète d'un programme, interpréteur REPL et débogage. Ainsi, une deuxième contribution de cette thèse est la définition du composant le plus élémentaire qui peut diriger d'autres outils d'exécution de langage, et son protocole. Au-dessus de ce composant, nous pouvons fournir des outils d'exécution génériques pour chaque mode d'exécution identifié. Comme troisième contribution, nous décrivons une approche générative pour créer des interpréteurs REPL à partir de spécifications de langages dédiés, en utilisant le protocole mentionné ci-dessus.

8. cf. https://gitlab.com/gitlab-org/gitlab

Cela permet d'utiliser les langages dédiés dans des environnements de développement interactifs et modernes, tels que Jupyter Notebook, avec un minimum d'efforts, tout en assurant la cohérence de la sémantique avec les autres interpréteurs existants.

Implémentations et Évaluation

Les implémentations abordées dans ce manuscrit visent spécifiquement les langages dédiés (DSLs). À partir des spécifications de DSL existants, nous proposons une approche générative pour dériver un interpréteur REPL, un outil d'exécution de langage capable d'exécuter de manière incrémentale des programmes partiels. Un tel interpréteur peut être utilisé dans des environnements comme des notebooks (par exemple, Jupyter Notebook) que nous désignons comme « environnements de programmation interactifs ». Ces environnements reposent sur un processus de développement incrémental dans lequel le développeur peut écrire et documenter de petits bouts de code indépendants et obtenir des retours, ce qui mélange les approches de la programmation exploratoire [45] et de la programmation lettrée [49].

Les DSLs que nous avons utilisés pour évaluer cette implémentation offraient également des fonctionnalités de débogage avancées. En essayant de les intégrer dans de tels environnements, nous avons toutefois remarqué certains conflits entre les approches utilisées pour implémenter ces débogueurs et nos interpréteurs REPL. Dans certains cas, ils fournissaient touts les deux les mêmes fonctionnalités, et dans d'autres, ils se gênaient mutuellement. Cela a conduit à une définition de la hiérarchie des outils d'exécution, avec les interpréteurs REPL et les débogueurs s'appuyant sur une seule implémentation d'un composant exposant une sémantique opérationnelle formalisée. Par la suite, nous avons dérivé une autre approche pour obtenir ledit composant, soit implémenté manuellement par un ingénieur du langage, soit dérivé génératívement d'une sémantique opérationnelle existante. Nous présentons ensuite les autres outils d'exécution comme des implémentations génériques, interagissant avec ce composant via un protocole de langage spécifique.

Pour motiver notre vision et évaluer nos implémentations, nous considérons dans cette thèse un environnement de programmation interactif qui fournit des possibilités de débogage omniscient à partir d'une trace d'exécution, similaire à l'approche présentée dans [18]. Afin de fournir le support d'un débogueur omniscient à plusieurs plateformes différentes, nous pourrions en pratique utiliser DAP puisqu'il spécifie une requête *step-*

Back. Cependant, le mécanisme de retour en arrière tel qu'introduit dans [18] donne plus de contrôle que le simple retour en arrière : par exemple, on peut décider de revenir soit à l'intérieur soit à l'extérieur de fonctions/méthodes. À l'heure actuelle, aucun protocole de langage ne prend en charge un mécanisme de retour en arrière à ce niveau de granularité.

En outre, la génération et le stockage de la trace d'exécution est un processus très coûteux qui ne devrait pas être activé en permanence et qui gagnerait à être exécuté sur un hôte dédié. Idéalement, son activation devrait être contrôlable tout au long d'une session de débogage, et les protocoles de langage actuels n'offre aucune requête pour gérer cela.

CHAPTER 1

INTRODUCTION

1.1 Context

Modern software engineering consists heavily in using the programming languages most suitable for the task, which relies on having the tool support to use these languages efficiently. Integrated Development Environments (IDEs) have been initially designed as a way to gather in a single software the various services of a given language (*e.g.* facilities for editing, debugging, checking, compiling). However, we tend to see more and more projects leveraging on the benefits of multiple programming languages at the same time, either for isolated parts (*e.g.* full stack development, microservices architectures) or even in the same codebase with the recent trend of *polyglot development* [37], [80]. As a consequence, there is a need to have access to the services of several languages in the same environment.

To prevent the development of specific language services for each existing IDE, language protocols have become in the recent years a topic of interest in the language engineering community ([21], [65], [68]). By communicating through well-defined protocols, the main language services can be reused across the various IDEs supporting said protocols. A direct consequence is that the responsibility to provide a proper support for a specific language is no longer a concern of the IDE manufacturer, but befalls on the language maintainers developing the services independently. The first language protocol, namely the Language Server Protocol (LSP) [1], was proposed by Microsoft in the context of the development of VS Code [2]. The role of LSP is to support common editing services of any language providing a language server implementation that conforms to given open source specifications. It was designed around a set of services that was extracted from specialized code editors for the most commonly used general purposes languages. As a consequence, nowadays, any code editor can provide the same level of language support for GPLs that have a LSP implementation, *e.g.,* both VS Code and NeoVIM [3] offer

1. cf. https://microsoft.github.io/language-server-protocol/
2. cf. https://code.visualstudio.com/
3. cf. https://neovim.io

the same syntax highlighting, symbols documentation, type checking and refactoring capabilities for Python programs through Pyright [4]. LSP, the Debug Adapter Protocol (DAP) [5], and most of the other language protocols we see nowadays (for *e.g.,* graphical syntax, compilation, testing) specify the structure of the data exchanged between a single client (a UI component of an IDE) and a single server (backend providing the set of services needed by the client), and the requests and events that can be sent from one to the other. Most messages are included in what is referred to as "capabilities". The set of capabilities is set and fixed by the specifications of the protocol. The idea is that both clients and servers can choose to implement a subset of the capabilities and notify the other, which should be able to use all the corresponding messages according to the specification.

1.2 Problem Statement

Fixing the set of services at the level of the protocol means that it might not fit every use-case. For example, when considering domain specific languages (DSLs), we often find some unusual features that can be tied to either the meta-language approach used to design the language (*e.g.,* generic services [17]), the language itself (*e.g.,* paradigm, syntax), or even the program being developed (*e.g.,* current state representation). While we could argue that it would be pertinent to add some of these features as new capabilities to the existing protocols, it would be inconceivable to cover all specific use cases inside a single generic protocol. Still, the adoption of DSLs would greatly benefit from support by multiple major IDEs, as they would integrate better in the workflow of the users. Protocols are making this situation possible, although some features would be lost in the process [21].

With the success of LSP and DAP, we see new language protocols emerging for both new use cases (*e.g.,* Build Server Protocol [6], Test Adapter Protocol [7]) and specific features not properly supported by the existing ones (*e.g.* using a graphical syntax in LSP [68]). The case of the graphical syntax is particularly interesting, because it led to the definition of two different protocols for the same purpose: the Graphical Language Server Protocol [8]

4. cf. https://github.com/microsoft/pyright
5. cf. https://microsoft.github.io/debug-adapter-protocol/
6. cf. https://build-server-protocol.github.io/
7. cf. https://github.com/microsoft/vscode-debugadapter-node/issues/154
8. cf. https://www.eclipse.org/glsp/

and the Graphical Server Protocol[9]. Another way to extend the features of LSP that was used in works such as [60] and [50] consists in arbitrarily adding support for new messages to a server and a client, and consider that all existing implementations will ignore them if they don't support them. While these two works provide very useful contributions, such an implementation raises concerns of maintainability and interoperability, which further motivates our work.

In the long run, it would be counter-productive to keep making independent protocols for every use-case, as this would defeat the purpose of defining these protocols, *i.e.* to ensure proper support in all development environments. In a real-world situation, this would shift the challenge on the composability and compatibility of existing protocols, which is currently not addressed. Hence, instead of having a *server* that encompasses all services for a given language, we explore the idea of breaking it down into individual services interacting with each other. To ensure the communication between these services and the *client(s)*, as well as their choreography, we consider the specifications required explicitly to support all their interactions. Thus, instead of defining a language protocol from the ground-up, tied to a specific use-case, we envision the definition of a meta language protocol from which we can derive the most pertinent instantiations (*i.e.,* language services configurations). With this approach, some parts of existing protocols that are actually generic could be separated, and their implementations leveraged better by being reused in multiple services configurations. This also provides more flexibility to the overall architecture: specialized services could be moved to other machines or replaced depending on the use-case, features could be added or removed at any moment, and it would become possible to finely control the deployment of each service independently. We refer to this vision as *IDE as Code*.

As illustrated by Figure 1.1, we expect language designers to not only provide their language specifications but also information about the protocol interactions required for the different services. From there, we can obtain "language service packages" which are minimal independent language services interacting with each other. Using these packages, users should be able to specify the configuration and the deployment of their IDE (or use predefined ones) to set up a development environment in the way they need according to both their work environment and use case.

Developers require optimal access to language services no matter their workplace and equipment. With the COVID-19 pandemic, it became even more apparent that relying

9. cf. https://obeonetwork.github.io/GraphicalServerProtocol/

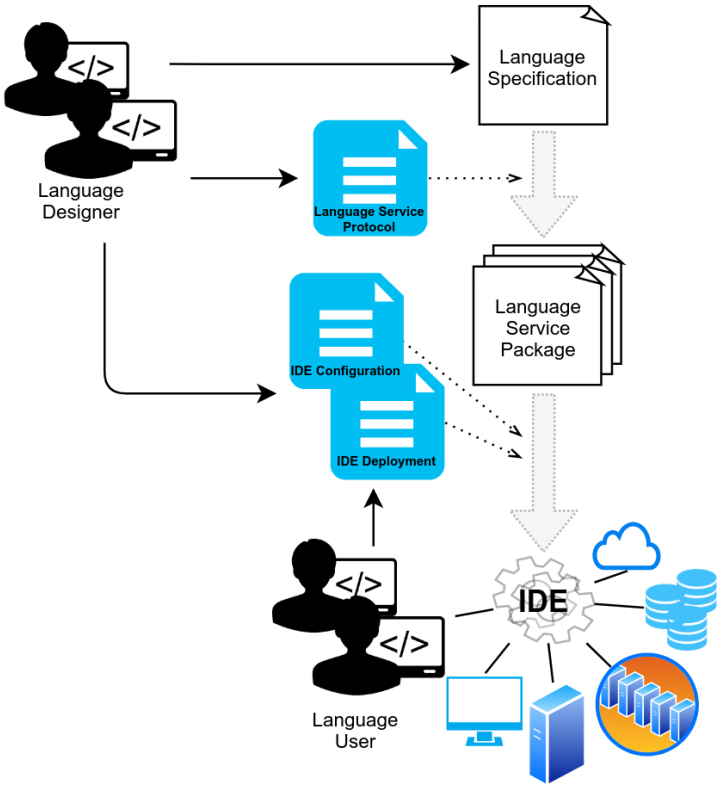

Figure 1.1 – Vision Overview for *IDE as Code*

on a stable work environment is not always sustainable, and more and more efforts are put towards the development of Cloud-based IDEs [10]. While they allow their users to access a persistent development environment from anywhere, they also possess flaws that still require to be addressed: the need for a stable and fast Internet access, which can greatly fluctuate in situations such as travels, limited customization, and insufficient scaling to offer all their users a seamless experience at the same time. Works such as [25] already took these limitations into account and explored the idea of separating language services into microservices, with mechanisms for dynamic re-deployment either locally or on different servers, as a solution to these issues.

On the topic of supporting different use cases, we can for example consider domain experts used to working with a DSL through a graphical syntax, that do not require a textual representation nor the services to manipulate it. Several studies such as [69] argue that professionals can often feel overwhelmed by their environments when they offer too many tools and services. This can be counterproductive as it is hard to properly showcase the most useful features when the users are drowned in useless ones. Consequently, a goal of this approach is to let users have a clean and minimalist environment where they have access to only the tools they actually use. On the other hand, the input of language designers in regard to the set of features best suited to a particular use-case is also valuable, so they should be able to specify default environment configurations. Thus, we consider the following research question: **How to provide a fully customizable development environment that empowers both DSL designers and users?**

The components that can be found in a development environment are numerous. They can range from a parser for the textual syntax of a language to a debugger for the execution of a program. In this thesis, we put the focus on runtime components, meaning the components that permit and drive the execution of programs (*e.g.*, interpreters, debuggers), in the context of interactive environments. We address three challenges in this context:

— The first one relates directly to the way these different components will communicate with each other. A simple client/server approach with fixed protocols is not enough to support the customization of such an environment. As a consequence, it is necessary to manipulate directly the language protocols as first class citizens, much like the components implementing them.

— The second one consists in establishing a hierarchy between the different runtime

10. cf. https://ecdtools.eclipse.org/events/idesummit/2021/

components, and deriving the most basic one that can be used to drive the others. The main goal is to improve the reuse of implementations of services, while also permitting to standardize them around minimal and well-defined APIs. This also means that we need to specify the dependencies between these components and define the protocols they require in order to interact with one another.

— And the third one is to actually produce the language components that meet the protocol specifications, by making use of the information that already exists in the specifications of the languages and applying a possibly generative process.

1.3 Contributions

The main contribution of this thesis is the concept of *IDE as Code*. We discuss a possible implementation of language tools and features provided as independent microservices, that can be deployed on demand by users to tailor their development environment to their needs. The configuration of the environment being external to any specific platform, it can be used to set up any infrastructure, either a fully local IDE or a basic web environment tied to a code repository in a forge such as GitLab [11]. By manipulating language protocols as first class citizens, we can leverage on the reusability of the services implementing them, and even define a hierarchy with specific protocols extending or using a composition of others. This is particularly interesting when we consider the multiple execution tools that can be required for a given DSL, *e.g.* complete program execution, REPL interpreter and debugging. As such, a second contribution of this thesis is the definition of the most basic component that can drive multiple language execution tools, and its protocol. On top of this component, we can provide generic execution tools for each identified execution mode. As a third contribution, we describe a generative approach to create REPL interpreters from DSL specifications, making use of the aforementioned protocol. This makes it possible to use DSLs in modern interactive development environments such as Jupyter Notebook with minimal efforts, while assuring the consistency of the semantics with other existing interpreters.

11. cf. https://gitlab.com/gitlab-org/gitlab

1.4 Implementations & Evaluation

The implementations addressed in this manuscript are targeting specifically domain-specific languages. From existing DSL specifications, we provide a generative approach to derive a REPL interpreter, a language execution tool able to incrementally run partial programs. Such an interpreter can be used in environments like notebooks (*e.g.,* Jupyter Notebook) that we identify as "interactive programming environment". These environments rely on an incremental development process in which the developer can write and document small independent code snippets and get feedback, which mixes the approaches of both exploratory programming [45] and literate programming [49].

The DSLs we used to evaluate this implementation also offered advanced debugging features. By trying to integrate them in such environments however, we noticed some conflicts between the approaches used to implement these debuggers and our REPL interpreters. In some instances, they would both provide the same features, and in others they would clash with one another. This lead to a proper definition of the hierarchy of execution tools, with both REPL interpreters and debuggers leveraging on a single implementation of a component exposing formalized operational semantics. As such, we derived another approach to obtain said component, either implemented manually by a language engineer or generatively derived from an existing operational semantics. We then introduce the other execution tools as generic implementations, interacting with this component through a specific language protocol.

To motivate our vision and evaluate our implementations, we consider in this thesis an interactive programming environment that provides omniscient debugging capabilities from an execution trace, similar to the one presented in [18]. In their approach, Bousse et al. rely on a debugger and a trace manager that are highly coupled. However, generating and storing the execution trace is a very expensive process, and as a result their debugger has trouble scaling for very large models. Considering a distributed approach based on microservices would make a lot of sense in this scenario, as we could imagine splitting the management of the full trace between several services, possibly hosted on separate hosts, and for which the activation could be finely controlled whenever needed. In order to provide support for an omniscient debugger to multiple platforms, in practice we could use DAP since it specifies a *stepBack* request. However, the backtracking mechanism as introduced in [18] gives more control than simply stepping back: for example, one can step inside and outside functions/methods. There is currently no language protocol

supporting such fine-grained backtracking mechanism.

1.5 Outline

The remainder of this thesis is structured in six chapters. Chapter 2 provides the background required to follow the rest of the manuscript. It discusses the evolution of IDEs and the challenges, both past and current, they face, while covering the current state of the art on the topics of IDE modularity and DSL integration. Chapter 3 further details the vision of *IDE as Code* introduced in this thesis and discusses possible implementations for configuring and deploying a development environment. Since we put the focus on program execution and interactive development environments, one of the main components that we need to support in our infrastructure is a REPL interpreter. We introduce the concept of REPL interpreters in Chapter 4 where we discuss the knowledge gained from studying existing implementations, and derive a principled approach to define them. Chapter 5 deals with the integration of services of executable DSLs in interactive environments, and presents a generative approach to derive REPL interpreters based on the principled approach. Chapter 6 defines a hierarchy between execution components, including REPL interpreters and debuggers, and presents our proposal for a language protocol dedicated to language runtimes that leverages on a unique definition of operational semantics. And finally, Chapter 7 concludes this manuscript with a research agenda following up on the work covered in this thesis and on the vision of *IDE as Code* as a whole.

1.6 Publications List

In this section we list all the publications made as part of this thesis, either accepted or still ongoing.

1.6.1 In-Press Publications

This section focuses on the accepted publications that are strictly related to this manuscript:

IDE as Code: Reifying Language Protocols as First-Class Citizens (Conference Paper)

Pierre Jeanjean, Benoit Combemale, Olivier Barais. In 14th Innovations in Software Engineering Conference, 2021 [40].

A Principled Approach to REPL Interpreters (Conference Paper) L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, Olivier Barais. In Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward!, 2020 [15].

From DSL Specification to Interactive Computer Programming Environment (Conference Paper) Pierre Jeanjean, Benoit Combemale, Olivier Barais. In Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE), 2019 [39].

1.6.2 Work in Progress

In this section, we list ongoing work initiated as part of this thesis but not published yet:

A Protocol for Decoupling Execution Services from Language Runtimes Pierre Jeanjean, L. Thomas van Binsbergen, Mauricio Verano Merino, Tijs van der Storm, Gurvan Le Guernic, Olivier Barais, Benoit Combemale.

A Language Independent Protocol for Exploratory Programming Mauricio Verano Merino, L. Thomas van Binsbergen, Pierre Jeanjean, Damian Frolich, Joey Lai, Tijs van der Storm, Benoit Combemale, Olivier Barais.

1.6.3 Others

This section includes accepted publications that explored other related topics to this PhD:

From Monolithic to Microservice Architecture: The Case of Extensible and Domain- Specific IDEs (Workshop Paper) Romain Belafia, Pierre Jeanjean, Olivier Barais, Gurvan Le Guernic, Benoit Combemale. In DevOps@MODELS 2021: ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems, 2021 [7].

Opportunities in intelligent modeling assistance (Journal Paper) Gunter Mussbacher, Benoit Combemale, Jörg Kienzle, Silvia Abrahão, Hyacinth Ali, Nelly Bencomo, Márton Búr, Loli Burgueño, Gregor Engels, Pierre Jeanjean, Jean-Marc Jézéquel, Thomas Kühn, Sébastien Mosser, Houari Sahraoui, Eugene Syriani, Dániel Varró, Martin Weyssow. Software and Systems Modeling 19.5, 2020 [58].

Runtime Monitoring for Executable DSLs (Journal Paper) Dorian Leroy, Pierre Jeanjean, Erwan Bousse, Manuel Wimmer, Benoit Combemale. In J. Object Technol. 19.2, 2020 [52].

CHAPTER 2

BACKGROUND & STATE OF THE ART

In this chapter, we introduce the background necessary to follow the rest of the manuscript, and explore the related state of the art. We first present a brief history of the evolution of IDEs (Section 2.1). Next, we cover domain-specific languages (Section 2.2) that will be one of the main study subjects in this thesis, followed by an overview of the requirements to execute programs in different development environments (Section 2.3). Then, we consider more globally language services and the issues to integrate them in multiple IDEs (Section 2.4). We also mention REPL interpreters, which are another core concept of our contributions, and we provide some background and examples on them (Section 2.5). Finally, we present past research works on protocols engineering, that shaped our thoughts on what a protocol for an IDE should look like and how it should be defined (Section 2.6).

2.1 Integrated Development Environments

The first mention to an Integrated Development Environment can be traced back to the work of P.S. Newman published in 1982 [59]. Such an environment should provide to a developer all the necessary documentation to understand the system either currently in development or being operated. In this paper, the author argues that the lack of system documentation at the time was partly due to development tool fragmentation: the lack of coordination between tools coupled with the absence of a unified environment giving access to them made it hard for developers to know what features they could use, and even if they did, there would be almost no interoperability. The higher cost of maintaining an unintegrated set of tools is also a strong argument in favor of integrated environments.

In those times, developers would mostly write source code through simple text editors such as *vi* or *Emacs*. Due to their efficiency at text editing, such tools are still popular today, but they have definitely evolved significantly throughout the years. For example, *vi* has been replaced almost exclusively by an "improved" version known as *vim*, which is itself slowly being supplanted by *neovim*. The integration of support for external plug-ins in these editors makes them still relevant for development purposes and not just basic

text editing. This in turns make them similar in nature to IDEs, as noted in [73].

As both programming languages and programs became more and more complex, environments following the IDE ideology did the same by integrating more and more complex and numerous language tools. This leads to two consequences:

— IDEs such as Eclipse [1] and IntelliJ IDEA [2] have so many features that their users can feel overwhelmed, to the point of not knowing about some of the most basic such as *References* ([69]);

— Developers who are satisfied with their IDEs are opposed to switching to others, even when they offer some useful specific features. So new tools need to be implemented for multiple environments in order to be used globally: development tool fragmentation has been replaced by development environment fragmentation.

The trend of IDEs is now switching towards cleaner and more user-friendly interfaces. At the same time, new concerns are emerging as today's developers need an environment offering support for multiple languages, sometimes even in the same project with polyglot programming ([37]), and accessible from anywhere. Modern IDEs such as Microsoft's *VS Code* [3], *Eclipse Theia* [4] or *JupyterLab* [5] are consequently implemented using web technologies, and support multiple languages through the implementation of newly defined language protocols such as LSP [6] and DAP [7].

2.2 Domain Specific Languages

Domain Specific Languages (DSLs) are software languages specifically designed to handle a specific application domain. As opposed to General-Purpose Languages (GPLs) such as *C* or *Java*, that are Turing complete languages aiming to be usable in any context, the role of DSLs is to capture the semantics of a given domain and only expose to their users the concepts they require to solve a specific problem. While DSLs may end up being Turing complete in some instances, this is usually not a goal when designing them. Examples of DSLs include *e.g.* markup languages such as *LaTeX* [8], a language used to

1. cf. https://www.eclipse.org/
2. cf. https://www.jetbrains.com/idea/
3. cf. https://code.visualstudio.com/
4. cf. https://theia-ide.org/
5. cf. https://jupyter.org/
6. cf. https://microsoft.github.io/language-server-protocol/
7. cf. https://microsoft.github.io/debug-adapter-protocol/
8. cf. https://www.latex-project.org/

annotate text documents with the necessary information to generate a consistent and standardized output (this thesis' manuscript was written entirely in *LaTeX*), modeling languages such as *SysML* [9], a language for specifying and analyzing complex systems, and programming languages such as *SQL* to interact with relational databases or *alda* [10] for music composition. The examples given previously are called "External" DSLs, because they use their own independent interpreter and/or compiler, and their editors rely on a parser dedicated to their specific syntax. Other kinds of DSLs are called "Internal" DSLs, and also consist in defining a grammar but on top of a host language to add domain-specific elements to it. These leverage on the existing tool support for the host language, both for edition and execution. Examples of Internal DSLs include Fluent APIs, APIs that are designed to be used mainly through methods chaining which makes their particular syntax explicit and easily readable, and language extensions that add new programming concepts to the host language such as multi-stage programming frameworks ([76]).

Throughout this thesis, we will focus on external DSLs that provide a specification designed to drive the development of a comprehensive development environment. A language specification defines all the concepts available to the language users through an abstract syntax, a metamodel whose instances are the programs written by the users. In order to actually manipulate the language, developers have access to one or several concrete syntaxes which map the concepts of the abstract syntax to concrete representations that they can reason with. Semantics then map the abstract concepts to actual behaviors within the context of the domain, making the language either executable or translatable to another executable language. The specifications considered here usually involve a concrete syntax definition in the form of a BNF-like description, an operational semantics in the form of an interpreter (or any variant such as a visitor pattern), and static semantics, that define all the context conditions ensuring statically correct conforming programs. In such a definition, the language usually encompasses a single execution entry point, a finely tuned execution context and an interpreter which defines a particular traversal of a given syntax tree to manipulate and update the execution context over the execution.

Such a language definition is now well-supported by advanced language workbenches that help language engineers to develop language tools such as structured editors, debuggers and simulators. For instance, tools like Xtext [11] support the generation of an

9. cf. https://www.omgsysml.org/
10. cf. https://alda.io/
11. cf. https://www.eclipse.org/Xtext/

advanced editor with a parser, syntactic validation, and all the features of modern code editors (*e.g.,* syntax coloring, auto-completion, etc.). Others, such as the GEMOC Studio [12], help to complement the language tooling with advanced execution engines and debugging facilities. A study of the main features provided by some language workbenches (*e.g.,* MPS, Rascal, Spoofax) can be found in [28]. Most of these workbenches are now mature enough to be included in industrial settings, and allow language engineers to automate the development of the tools for the main scenarios of the expected language users.

Let us consider for example the language *Logo* [13], illustrated in Figure 2.1. *Logo* is an educational language whose main focus is the animation of turtle graphics. As such, most of the statements are accompanied by feedback which is an action from the turtle, and this is part of what makes it interesting for teaching purposes. An abstract syntax of Logo specify that Program is made out of several instructions, that may also contain multiple kinds of expressions. Figure 2.2 shows the metamodel associated to a very minimal subset of this language. The associated concrete syntax is textual, and maps the instructions to the keywords FORWARD, RIGHT and LEFT, while also ensuring parsing of sums of integers to binary expressions and constants. Operational semantics define how the execution of each instruction will affect the movements of the turtle. From a comprehensive definition of the language, it is possible to automatically generate a dedicated and structured editor that supports the definition of complete logo programs, including a functional architecture and an explicit execution flow across the architecture.

2.3 Executability in Software Languages

Modern development environments support programming paradigms that differ from the traditional edit-compile-run loop process. As an example, notebooks, that merge REPL interpreters with the practice of literate programming, have recently gained popularity in many fields (*e.g.,* education [83], collaboration [81], domain exploration [71]) through the Jupyter initiative [48]. At its core, a Jupyter notebook requires a language kernel built on top of a REPL interpreter. Most of the popular languages nowadays provide REPL interpreters implementations [15], such as Java with the JShell tool officially maintained by Oracle. This makes these languages usable inside notebook environments. However,

12. cf. http://gemoc.org/studio.html
13. cf. https://en.wikipedia.org/wiki/Logo_(programming_language)

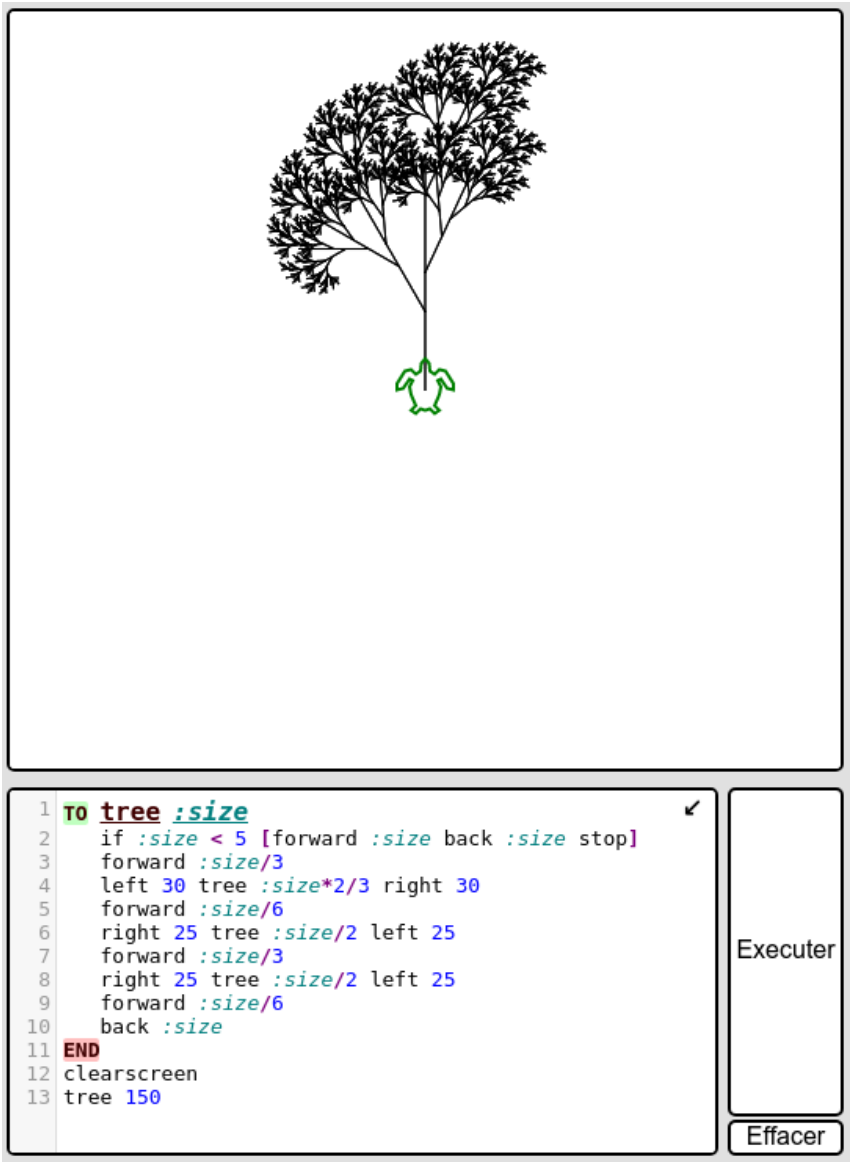

Figure 2.1 – Example of a free Logo Interpreter (https://www.calormen.com/jslogo)

with a few exceptions such as Python, these tools are usually implemented and managed independently of the other "more traditional" compilers or interpreters. They might reuse parts of the existing semantics implementations, but cannot guarantee behaviors fully consistent with the others. Tool support is also lacking: Oracle considers debugging as a non-goal for JShell [14], which also means that notebooks based on this tool will not be able to provide debugging facilities.

14. https://openjdk.java.net/jeps/222

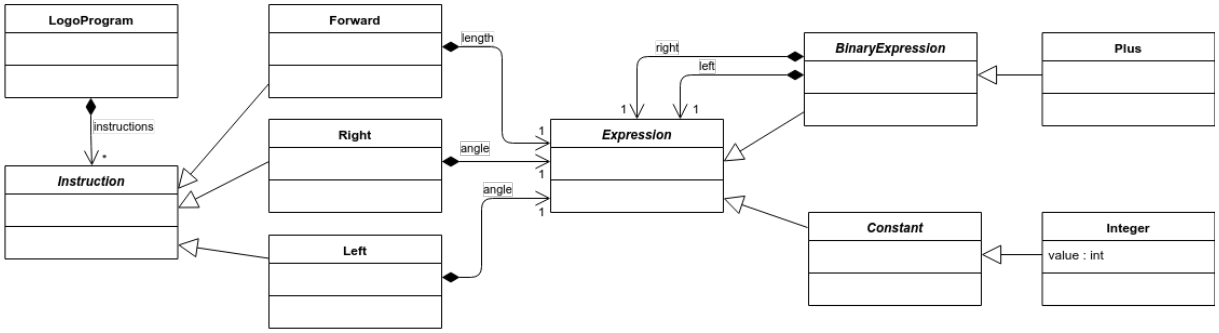

Figure 2.2 – Abstract syntax of a subset of the Logo language

When considering Domain-Specific Languages, these issues are even worse because it is quite often unrealistic in terms of resources to develop and properly maintain multiple implementations of the semantics for one language. Targeting one is already challenging enough, and would depend greatly on the habits of the users and the tools they require at a given time.

Besnard et al. studied such inconsistencies in [9] in the context of designing and deploying cyber-physical systems. At the time the current practice was to derive multiple semantics implementations from a first semantics definitions, each compatible with a specific diagnosis tool. However, all the generated semantics would end up being different from the one actually deployed on the target platform, which might invalidate the results of the different diagnosis obtained pre-deployment. To mitigate this, they contributed an approach based on a modular and reusable semantics implementation, and a generic API akin to a protocol in order to adapt it for the different analysis tools.

In recent years, language protocols have become a topic of interest in the language engineering community. The Language Server Protocol (LSP) provides an open source unified specification of language features for textual concrete syntax. Through its adoption in various development environments (*e.g.,* VS Code, Eclipse, Jupyter) language designers only require a single implementation of a *language server* to provide the same textual language features (*e.g.,* completion, formatting, validation) for users of different platforms. The Debug Adapter Protocol (DAP) similarly permits using a single debugger implementation in different environments.

A Foundational Subset for Executable UML Models (fUML [35]) was defined in order to leverage on unified semantics and execution tools for modeling concepts shared by

multiple domains (*e.g.,* activity diagrams, state machines). The specifications of fUML are very detailed and have been a great asset in the modeling community to support semantics and tool reuse for many years. However, one of their main limitations is that they do not, as yet, specify execution processes at a granularity low enough to handle debugging scenarios, as already noted by works such as [36] or [51].

In [17], Bousse et al. introduced the concept of *Execution Engine* to leverage on a single implementation of an operational semantics that follows a given interface. This approach allows the definition of generic *Engine Addons* to support different runtime scenarios. One of the addons contributed is a generic omniscient debugger (allowing both forward stepping and back in time exploration of the executed steps) that can be used by any sequential language interpreter that fits the interface. While sound in design, we argue that the proposed interface is too restrictive for emerging execution tools and frameworks: an *Execution Engine* expects a complete program as input, which is not compatible with the process of incremental building and exploration offered by notebooks.

2.4 Language Services

The term *Language Service* describes all the tools that can help users make an efficient use of a language. Program formatters, refactoring utilities and advanced debuggers can all be labeled as language services. Nowadays, the adoption of a programming language is directly tied to the tool support available to the users, *i.e.,* the language services.

2.4.1 Language Workbenches

The term *Language Workbench* was first introduced by Martin Fowler in 2005 ([32]) to describe the tools supporting the practice of *Language Oriented Programming*, namely software development that defines and makes use of multiple domain specific languages.

The point of these workbenches is to ease the design and implementation of DSLs, and to automatically derive the language services necessary for users to be, at least, as efficient with the DSLs that they would be with GPLs offering professional tool support. In practice, this translates into designing and creating a fully fledged IDE, tuned specifically for the target DSL, and making it available to users.

While language workbenches have existed and been continuously improving for years,

proper integration of the resulting DSLs into developers' workflow has only recently become a research focus. Providing a completely set up and ready to use IDE to users might seem sound, but it also means that the users need to learn how to use a completely new environment. Possibly, they might not even be able to import the settings that they've been using for years prior. In a blog post published in 2017, Meinte Boersma ([16]) addresses the different reasons holding back a more widespread adoption of language workbenches. He argues that the main issue is a "lack of integration with existing software development practices", and next that "even with a suitable language workbench, creating/finding and implementing a good DSL is not easy".

2.4.2 IDE Portability

A first improvement for integration is to provide support for the DSLs to the environment currently used by the targeted developers. Most language workbenches generate features and tools that can only be deployed on specific environments. This phenomenon is most generally observed for any IDE plug-in, and is known as the *IDE portability problem*. Keidel et al. addressed this issue in [44] and proposed a solution based on an intermediate representation of plug-in implementations. Environments require a Monto Plug-In made specifically for them. This Plug-In interacts with a message broker by sending a source message, containing the source code, at any change, and receiving product messages, that contain data formatted as Monto IR that translate to visual representations. The broker can interact with several environment independent language services, such as parsers or linters, by sending forwarding them the source messages and/or product messages obtained from other services.

The *Asf+Sdf* meta-environment [20] was probably the first language-parametric IDE system that used decoupled architecture, both for static aspects of a language definition [46], and for dynamic aspects such as debugging [19]. The approach uses an architecture based on *ToolBus*, a programmable bus made to connect different software, to coordinate heterogeneous components such as a parser, a text editor, or an interpreter.

A more recent approach to solve the *IDE portability problem* is the Language Server Protocol (LSP) proposed by Microsoft, that we already introduced in Section 2.3. A SWOT analysis of LSP with regard to DSLs was performed in [21]. LSP brings an opportunity to quickly deploy domain specific languages in multiple environments, either IDEs favored by the targeted developers or more simple and lightweight environments for user that are

not programming experts. However, it was designed around general purposes languages and is missing important features in order to really drive the deployment of DSLs, such as projectional editing. It is in theory possible to add these features through custom capabilities, but it would require implementing custom clients for each environment. The results of the SWOT analysis highlight potential limitations for the scalability of such an approach. They consider that each client requires the instantiation of complete independent server for each language used, and the lack of modularity and reusability of language features.

Similarly, the Debug Adapter Protocol (DAP) provides debugging support to multiple development environments, but targets general purposes languages and does not offer specific extensibility mechanisms for domain-specific concerns. The concept of moldable debuggers ([24]), *i.e.,* generic debuggers that provide a protocol to support domain-specific extensions, could provide an elegant solution to these limitations. But it would still require to first extend DAP with moldable capabilities.

2.5 REPL Interpreters

The acronym REPL stands for "Read-Eval-Print-Loop". A REPL interpreter is a language interpreter based on this principle of continuously reading user inputs as partial program, evaluating them in the same execution context, and printing feedback after each interpretation. These interpreters have a long history, and documentation on this history is scattered across sources. The Flexowriter system of Lisp I from 1960 is perhaps the oldest REPL implementation [54]. An early description of REPL behavior can be found in Peter Deutsche's memo on PDP-1 LISP [27]:

> *Each S-expression typed in will be evaluated, and its value printed out.*

The PILOT system [79] is one of the earliest and most advanced interactive REPL systems, also based on a LISP, in that it supports fully incremental and interactive evolution of programs. Teitelman writes that REPL-style interaction with Interlisp happened with the introduction of time-sharing at MIT in 1964 [78]. It is very well possible, however, that earlier Lisps and pre-1968 FORTH implementations [66] had REPL interfaces as well. The earliest programming language REPL that is not a Lisp we could find documentation of is the JOHNNIAC Open-Shop System (JOSS) [72]. Figure 2.3 shows an example of interaction with JOSS.

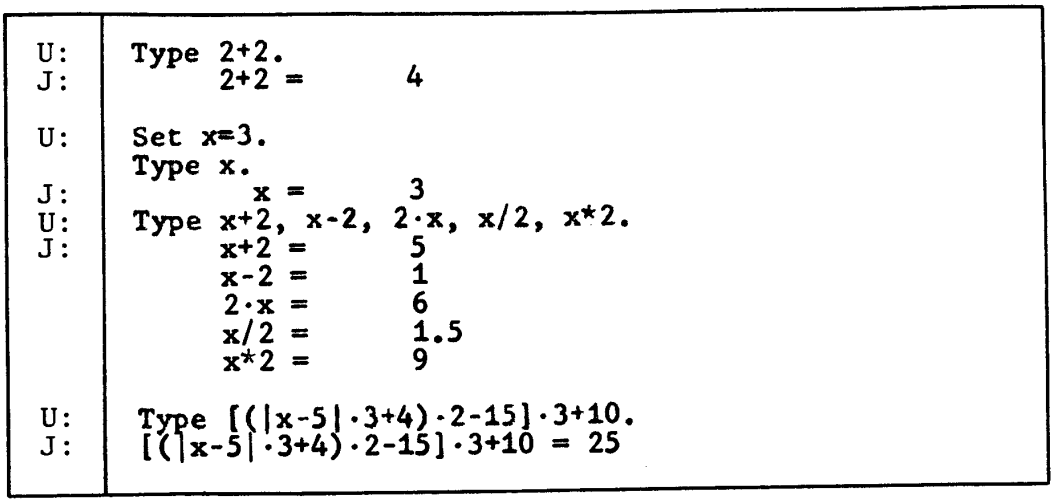

Figure 2.3 – Early user interaction using JOSS [72]

On their own, REPLs are useful tools to explore the possibilities offered by a language, since they enable trying small code snippets and getting instantly the results of the evaluation (including parsing and runtime errors), obtaining feedback on their effects (*e.g.,* by explicitly detailing the values changed in the execution context), and even navigating the execution context interactively after the initial evaluation. But they can also be used to drive more complex interactive environments such as computational notebooks, which were pioneered in the Mathematica system [82]. More recently, this style has been adopted in the context of other programming languages. IPython [62] and Jupyter [48] provide a means for computational story telling, where cells containing code are interleaved with output and prose cells. The language workbench framework Bacatá allows a language engineer to provide a notebook feature by reusing existing language specifications [55].

2.6 Protocols Engineering

Providing a language for specifying protocols and interactions and providing a compliance relationship to that protocol has been done for a while. Indeed, in the field of components based software engineering, Plasil *et al.* [63] provides within their architecture description language a way to specify this behavioral contract of each component. Software designers can define component's behavior. The paper defines a protocol con-

formance relation. Using these concepts, the designer can check the adherence of a component's implementation to its specification at runtime, while the correctness of refining the specification can be verified at design time. In the multi-agents community, several agent-oriented programming languages such as JADEL [8] provide abstractions to define agents interaction protocols. In the networking domain, Burgy *et al.* [22] proposes a new language-based approach for developing protocol-handling layers, to improve their robustness without compromising their performance. The approach is based on the use of a domain-specific language to specify the protocol-handling layer of network applications that use textual HTTP-like application protocols.

An earlier definition of a meta-protocol can be found in [1]. The approach aimed at providing a higher level of flexibility in protocol-based communications. Multiple protocol specifications are available inside a repository, and two parties can decide on which one to use during a negotiation phase. From the agreed upon specification, the actual implementation of the protocol is generated on the fly for the programming language used to define the component.

CHAPTER 3

THE VISION OF IDE AS CODE

In this chapter, we present an overview of the "IDE as Code" vision. We start by making clear the context and motivation behind the vision (Section 3.1). Then, we consider language protocols and how to specify them in order to obtain independent language services components (Section 3.2). And finally, we discuss a preliminary architecture implementation based around the concept of microservices (Section 3.3). This chapter is based on our ISEC'2021 publication ([40]).

3.1 Motivation

Our vision aims at providing more customizable and adaptable IDEs for the end users through language services that are independent of any client. It is driven both by the need to support domain-specific language services in multiple environments and to accommodate language users with a programming experience better tailored to their needs and technical background.

When we consider programming languages in the large, including both general-purpose and domain-specific languages, we need to account for an ever-growing number of language services. In contrast, at a given time, an IDE can only manage a subset of these. Since protocols such as LSP and DAP were designed around the features available in Visual Studio Code, it made perfect sense for Microsoft to also fix the set of the capabilities offered by these protocols to ensure that all the features would be fully supported in their IDE. However, the absence of a proper extension mechanism prevents the integration of additional services, such as the ones we can expect to find in new (domain-specific) languages.

Instead of fixing a priori the set of services that a user can use in the IDE, we envision a more open approach where all services are made available at any time, and the user can customize the IDE, possibly at runtime, based on a functional subset of these services. The customization could also be managed externally to provide to the users a development environment fully configured, and ready to use in the context of a specific use case.

To this end, in addition to their language specifications and the associated language

services, we expect language designers to also provide information about the protocol interactions and dependencies required to deploy and use the different services. This chapter focuses mainly on this part, starting with Section 3.2. From there, "language service packages", which are minimal independent language services interacting with each other, can be obtained and composed to instantiate the environment both required and wanted by a given language user. The deployment of these packages could also be finely controlled, for example to run some language services on suitable platforms, *e.g.*, with vast amounts of memory for execution logging or with powerful CPUs for compilation.

3.2 Specifying Language Protocols

In order to obtain usable and composable "language service packages", we first need to unify the specifications of the language protocols they will expose to communicate, both between them and with the environment accessible to the users. Current language protocols are basically provided as bidirectional APIs. While the data-flow between servers and clients is precisely specified, the control-flow is not explicit and can only be deduced from processing the documentation available in natural language. As a consequence, even though maintainers for servers and clients should be free of any specific implementation, in practice they end up relying on Visual Studio Code as a reference [1]. Here, we propose that the control-flow between language services should be formalized and become a part of the protocol specification. Whatever the implementations of "language service packages" end up being, they need to be agnostic to any technical concerns related to data transport.

As illustrated in Figure 3.1, individual "language service packages" should have their interactions properly specified in a language protocol. The left side of the figure represents the current practices in language engineering, where language services and their implementations can be automatically derived from a language specification, and the right side shows a conceptual metamodel of the specifications required to obtain "language service packages". Language services are provided as "language capabilities", and the corresponding UI components as "UI services". In addition, packages that are mandatory for development environments are reified as first-order concepts, like the notion of workspace for example. We make the assumption that language services can

1. https://www.reddit.com/r/vim/comments/b3yzq4

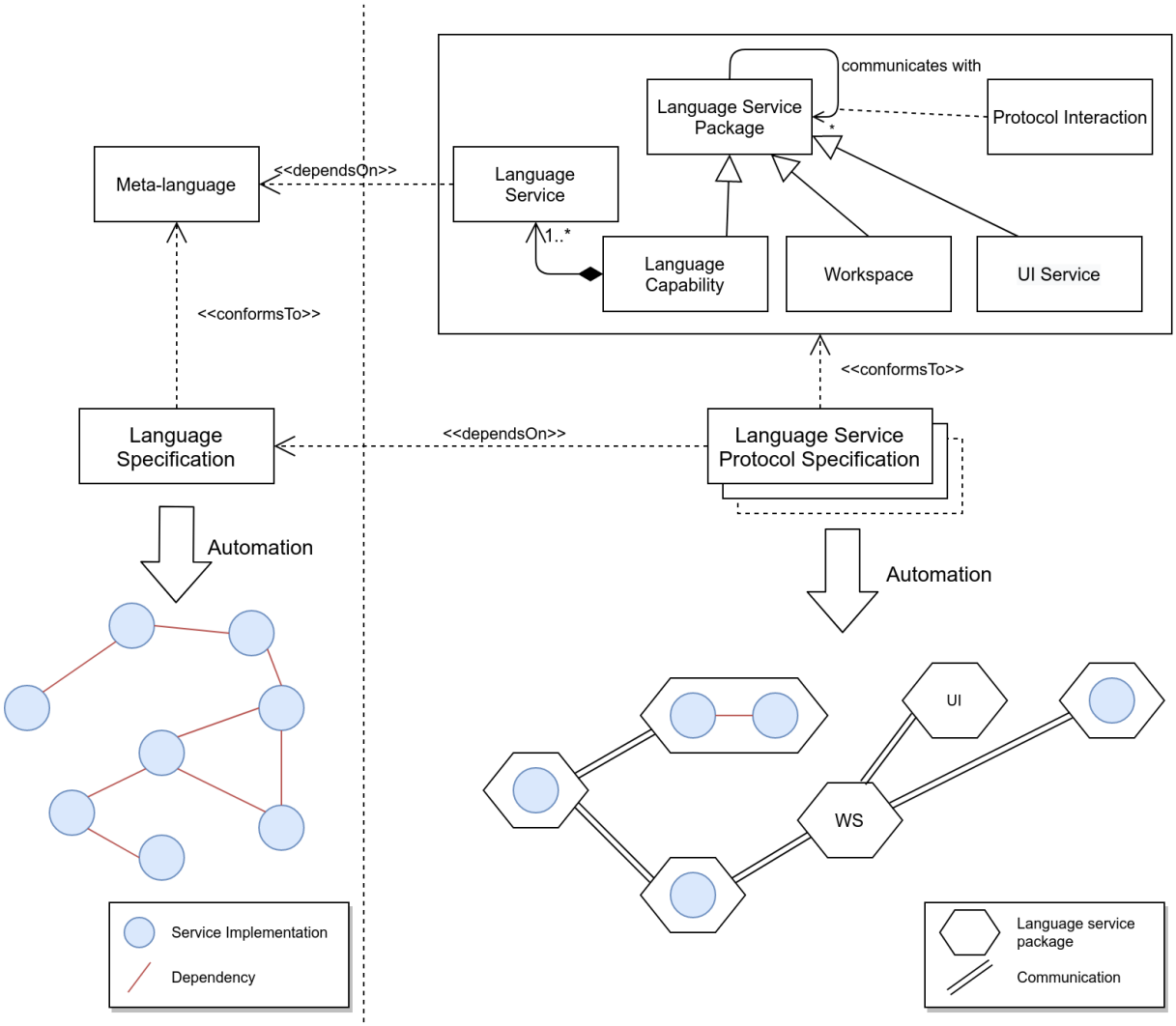

Figure 3.1 – Protocol Specification for *IDE as Code*

derive their implementations from a dependency to the language specification (*e.g.,* similarly to [25]), as their actual implementation is not the focus of this chapter. From a given language protocol specification, a generative approach supports the automatic creation and deployment of the language packages.

For customization purposes, "language service packages" can be replaced by more specialized implementations depending on the use-case. Services can be dynamically deployed, as long as the proposed deployment is coherent in regard to the dependencies and the protocol specifications. They might also specify particular hardware requirements, such as a need for RAM or disk storage, which would later drive the deployment on specific machines on a network infrastructure. Such an architecture is flexible enough to also support reusing existing language server components, such as existing LSP implementations, but since these implementations are usually monolithic they might not benefit from this approach as much as components specifically designed for it. For example, the overall scalability might suffer from using monolithic services ([25]).

3.3 Implementations

In this section, we discuss some implementation choices that we envisioned to illustrate the vision. We present a specific IDE architecture, using the scenario of integrating an environment that relies on an omniscient debugger.

Considering that they target to answer very specific problems, the added value of DSLs in comparison to general purpose languages resides primarily in the specific features they offer. As such, domain-specific language designers need the ability to provide specific language capabilities. An example of such is omniscient debugging, as introduced by [18]. The debug adapter protocol (DAP), a protocol that should address every debugging concern, is lacking in features in regard to this specific approach of omniscient debugging: it requires support for execution traces management, and a backtracking interface that makes the distinction between stepping into and stepping over statements.

However, DAP is still a very valuable and well needed contribution to unify the interfaces of different debuggers. For most of the capabilities it offers, it is in fact adapted for the different services needed to debug most textual DSLs. The main issue is the absence of a proper extension mechanism, that could make up for the missing features. As things stand, if one wanted to extend this protocol, they would have to define yet another protocol to wrap it that would end up being very specific to their use-case.

This raises concerns for both integration in environments, and compatibility with other services. Ideally, it should be possible to "import" an existing debug adapter as a language service, and specify its interactions with other services to deploy them as "language services packages". We could argue that this debug adapter could also, and should, be refined into smaller packages, but the re-usability of existing language servers is a major concern at this stage.

Thus, we propose an extensible architecture that can support existing language server implementations, as illustrated by Figure 3.2. In order to support resource scaling for web IDEs that need to serve multiple clients at the same time, the concept of "language services packages" is implemented as microservices. Microservices are minimal services providing a single feature to a global architecture, either through choreography or orchestration, and designed to be started and stopped on demand depending on the available resources and the required quality of service. An event broker lets both language microservices and UI microservices interact as part of a choreography. In the figure, UI microservices are all included in the same environment, but technically they could also be distributed and deployed independently: the service displaying the execution trace could very well be embedded inside a completely separate web page, or as part of a notebook that could also expose the necessary APIs to run analysis on the running context. Other implementation approaches such as OSGi plugins (targeting the Eclipse ecosystem) or Visual Studio Code extensions could also be considered, but comparing them is out of the scope of this work. Here, we also made the decision to use JSON serialization in order to stay close to LSP implementations, as well as websockets for the transport layer, but we have not yet evaluated sufficiently this setup and cannot comment on how pertinent this decision actually is.

The debug adapter microservice serves as a bridge between an existing debug adapter and the rest of the microservices. Its main goal is to turn the requests of DAP into events exchanges, in order to interact with the debug UI microservice. This particular microservice showcases that this architecture is flexible enough to include existing protocols and language servers.

The trace manager microservice receives events that consist of the different logical steps reached during the execution, and the data changes corresponding to each of them. Its role is then to build the execution trace and expose it to the rest of the environment. The events received by this service are sent by an execution microservice that has direct access to the program interpreter, and thus the execution context. The omniscient debug

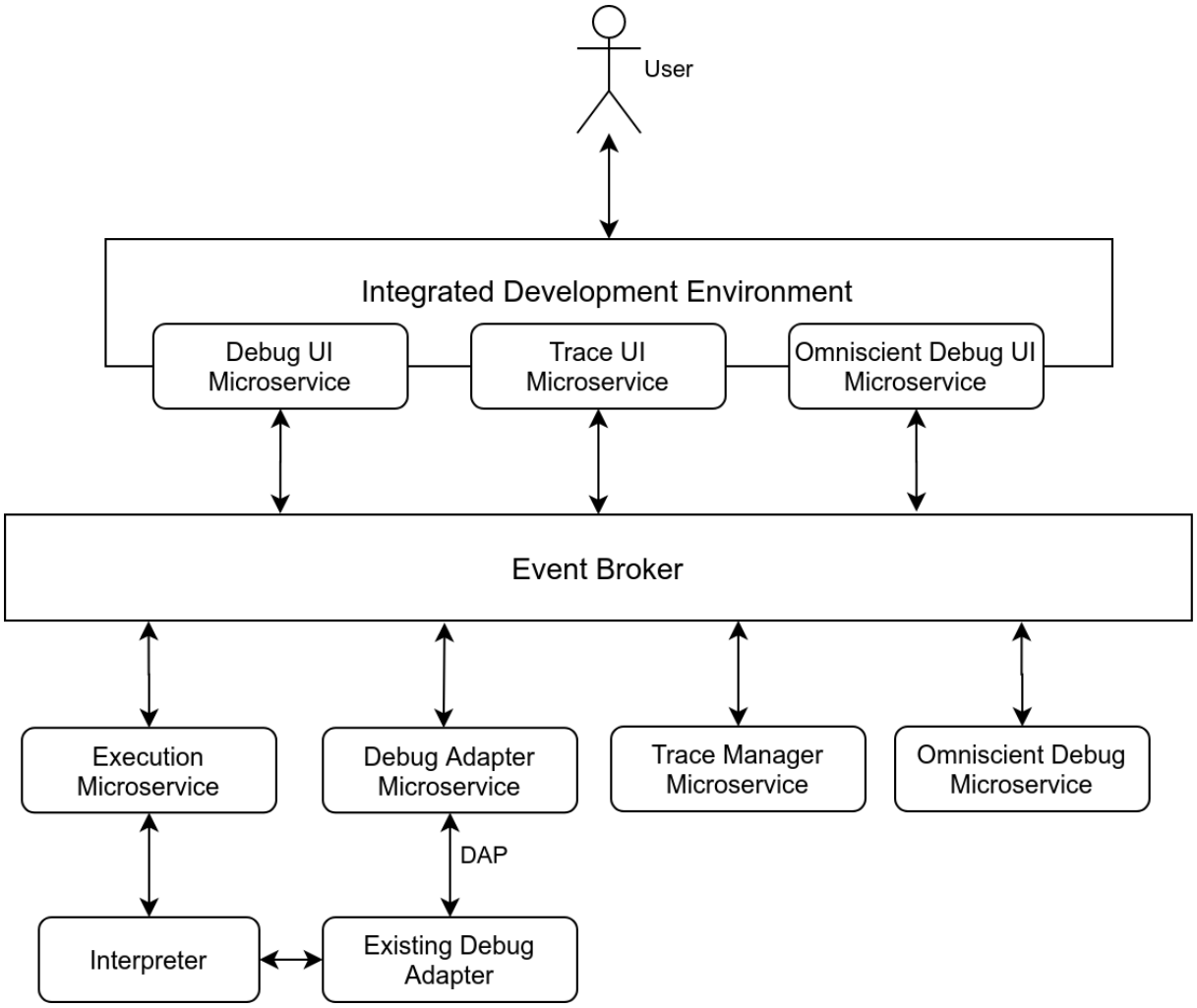

Figure 3.2 – Example of a Microservice Choreography for Omniscient Debugging (double arrows represent interactions)

microservice can make use of this trace to reset the execution context to the state it should have whenever the user needs to step back. The trace UI microservice provides feedback to the user about the execution trace and the current state of the program.

While the omniscient debug microservice relies on the trace manager, the configuration of microservices presented here is not fixed. As a choreography, every microservice is aware of the addition or removal of the others, and is able to react if there is an impact on their workflow. So it is possible to only provide the “normal” debugging services from the debug adapter depending on the resources available at a given time. It is also possible to generate the trace for only parts of the execution, and thus enable the omniscient debugger on the condition that a trace is indeed available.

A domain-specific language would help in defining and maintaining the specifications of these microservices. We are aware that a language to specify a service-oriented architecture is nothing new, but one dedicated to manage language services is yet to exist and would assist tremendously in the adoption of such an architecture. As such, we are considering a metalanguage designed to specify language services and their protocols, that provides constructs specific to IDEs, such as workspaces, development resources (files), and run configurations for example. It also drives the choreography by letting the microservices define complete workflows in their protocol specification.

Figure 3.3 shows an example of using such a language. Data structures can be specified, and used as arguments for the different events. By separating events into several blocks, multiple communication channels can be managed inside the event broker that will deliver them. The different packages can explicitly declare that they require other packages in order to be relevant, through a mechanism of dependencies. Then, their workflow when receiving events is explicitly defined, and the tasks can consist in waiting for other events, sending events, or calling methods from imported language services.

```
data {
    TraceState {
        backInto: Step
        backOver: Step
        backOut: Step
    }
    Step { ... }
}

events {
    trace {
        state
        stateResult(TraceState)
    }
    stepBackIn
    stepBackInDone
    setState(Step)
    setStateDone
}

packages {
    tracemanager depends on execution listens trace {
        recv step -> call updateState(Step)
        recv trace/state -> call getStateResult(result)
            -> send trace/stateResult(result)
    }
    omniscientdebug listens trace
    depends on execution, debugadapter, tracemanager {
        recv stepBackIn -> send trace/state
            -> recv trace/stateResult(traceState)
            -> send setState(traceState.backInto)
            -> recv setStateDone
            -> send stepBacklnDone
    }
}
```

Figure 3.3 – Protocol Specification for *Step Back In* Service

CHAPTER 4

A PRINCIPLED APPROACH TO REPL INTERPRETERS

In this chapter, we provide an approach to build REPL interpreters. We start by motivating the need for such a principled approach based on language engineering principles (Section 4.1). Next, we discuss the knowledge obtained by analyzing existing REPL implementations of popular programming languages (Section 4.2). Then, we go into details on the methodology to obtain a REPL interpreter from language implementations (Section 4.3). Finally, we validate the approach through different REPL interpreters implementations (Section 4.4). We close this chapter discussing the limitations and threats to validity of the approach (Section 4.5). This chapter is based on our Onward!2020 publication ([15]).

4.1 Motivation

Read-Eval-Print-Loops (REPLs, also known as command-line interfaces, or interactive shells) are a popular way for programmers to interact with programming languages. They allow incremental definition of abstractions, testing out snippets of code with immediate feedback, debugging executions, and exploration of APIs.

Even if a REPL can, in practice, be defined for any kind of programming languages, some such as scripting languages or interpreted languages are more naturally compatible with the REPL mode of interaction, and the different styles of programming that it enables (and that programmers have come to expect). For example, a sequence of valid code snippets written in the REPL of Python is itself a valid Python program. This enables an exploration workflow where the programmer can incrementally write a fully working program in environments based on REPLs, such as Jupyter Notebook, before exporting the source code as-is and integrate it easily inside another code base. On the other hand, JShell, for instance, allows programmers to write expressions, statements, variable declarations and method declarations as code snippets, even though these constructs are not allowed at the top-level in Java programs.

Consider the following example JShell interaction (every line is a code snippet sent separately):

```
class Example {}
Example obj = new Example();
class Example { public int meth() { return var; } }
int var = 1;
```

This example raises several questions : Can classes be redefined? Is `obj` still accessible after `Example` has been redefined? Or would `obj` be migrated to the new class definition and, if so, what methods would it have? And if `meth` is still available, would a call to `obj.meth()` return `1`? Without giving answers here, this example shows that the relations between a programming language and the behavior of its REPL are sometimes not obvious at all. The above questions are fundamentally about language design: several sensible answers are possible, and the answers have a significant impact on programmer experience down the line.

In some sense, JShell can be seen to implement *its own* language, which, even though strongly reminiscent of Java, is markedly different. In this chapter, and more generally the rest of this manuscript, we take this observation and run with it: we assume that a REPL interpreter for $\mathcal{L}$ effectively defines its own language $\mathcal{R}$, often as an extension or a modification of $\mathcal{L}$, whose programs are sequences of valid code snippets according to the REPL.

To this end we identify and define the class of languages that drive REPL interpreters as *sequential languages*. The essence of sequential languages is that the *concatenation of two programs is again a program*. Or, to put it more precisely, a language is sequential if it features an associative sequencing operator $\fatsemi$, such that the following equation holds:

$$[\![p_1 \fatsemi p_2]\!] = [\![p_2]\!] \circ [\![p_1]\!]$$

The meaning of a sequence of program fragments is defined by composing the meanings of the individual fragments, including the possible side effects of executing these fragments.

This chapter discusses a methodology to obtain a sequential language from another existing language, which could in turn serve as the basis for a sound REPL interpreter. In particular, the contributions include:

— a feature-based analysis of the landscape of REPLs for a selection of the most popular programming languages (Section 4.2);

— a methodology for developing REPL interpreters by *sequentializing* languages with a definitional interpreter (Section 4.3);

— case studies to illustrate the feasibility of the approach (Section 4.4).

4.2 REPL Domain Analysis

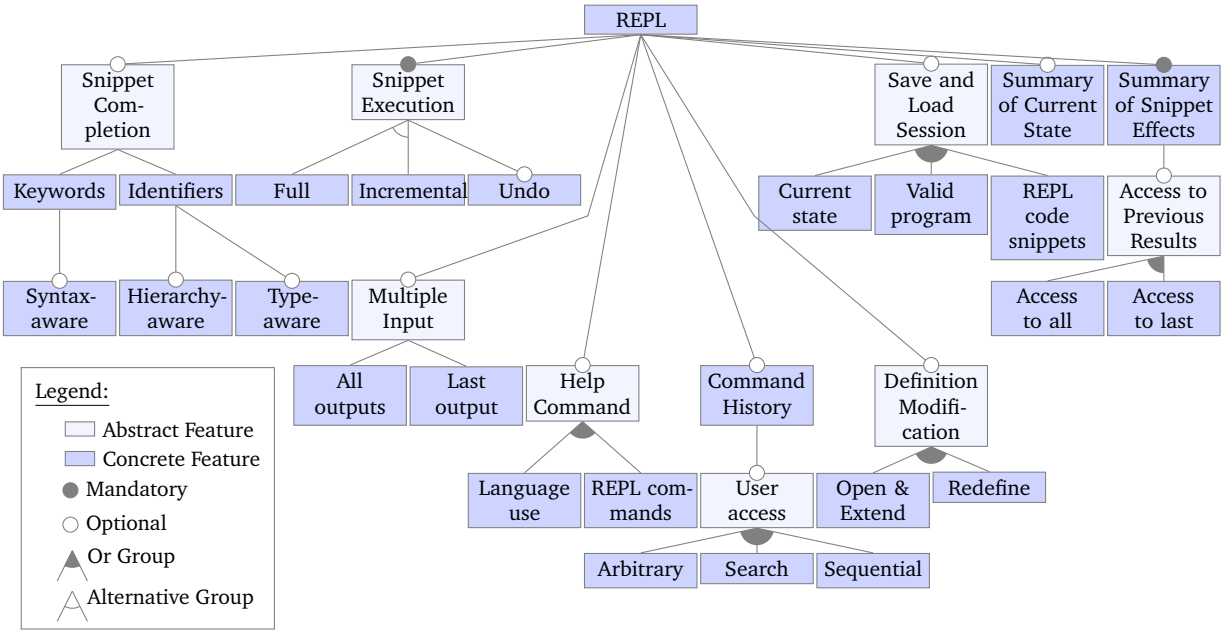

Figure 4.1 – Feature Model for REPL Interpreters

This section provides a study of existing REPL interpreters and their main features. We have studied freely available REPL implementations, listed in Table 4.1, for the 15 most popular languages from the TIOBE index [1], except for Visual Basic for which we could not find a freely available implementation. For MATLAB, we have selected GNU Octave as a substitute. We performed a feature-oriented domain analysis [43], resulting in the feature model presented as Figure 4.1. In the following we briefly describe the main mandatory and optional features we identified.

Mandatory Features An interpreter must have certain features to be considered a REPL. In particular, a REPL has the ability to execute multiple code snippets across

1. https://www.tiobe.com/tiobe-index/ (accessed May, 22nd, 2020)

Table 4.1 – Surveyed REPL implementations

REPL	Reference
CLing (C/C++)	https://cdn.rawgit.com/root-project/cling/master/www/index.html
JShell (Java)	http://openjdk.java.net/jeps/222
Python	https://docs.python.org/3/tutorial/interpreter.html
C#	https://mono-project.com/docs/tools+libraries/tools/repl/
Node.js (JavaScript)	https://nodejs.org/api/repl.html
PHP	https://php.net/manual/en/features.commandline.interactive.php
PsySH (PHP)	https://psysh.org/
SQLite (SQL)	https://sqlite.org/
R	https://r-project.org/
Swift	https://swift.org/lldb/
Gore (Go)	https://github.com/motemen/gore
GNU Octave	https://gnu.org/software/octave/
Rappel (assembly)	https://github.com/yrp604/rappel
iRB (Ruby)	https://github.com/ruby/irb

multiple interactions in a single session (as opposed to executing one full program per session). In most of the investigated REPL implementations, the REPL maintains a unique execution context and executes snippets incrementally (this is the behavior that we call “Incremental”, alternative of the “Snippet Execution” feature). Optionally, a REPL may provide a way to undo the execution of snippets (roll-back). An alternative to incremental execution is composing all the snippets into a single program and execute the program from scratch from a clean execution context (what we call the “Full” alternative). REPLs are expected to provide feedback after evaluating each snippet, showing at least the snippet’s printed output, and perhaps any changes in the runtime values or newly declared types (“Summary of Snippet Effects”).

Optional Features Next to these mandatory features, the investigated REPLs implement several additional features, such as providing auto-completion for the new snippets (“Snippet Completion”). This can target either language keywords or previously defined identifiers. Completion can take into account the syntactic context in which the user is typing, can be extended to fully qualified identifiers, and may also take into account the type of identifiers (through either static typing, or type hinting in the case of dynamically typed languages).

Even though the language itself might not support modifying an existing definition, most REPLs allow this behavior to some extent (“Definition Modification”). Common

ways include overriding the previous definition, either through accepting a snippet with a new definition or by editing the existing one from an external text editor. Other REPLs also allow opening up definitions (such as classes) for additions (“Open & Extend”).

Another common feature is the help (meta-)command (“Help Command”), which can document either the language, the REPL and its meta-commands, or both. The history of commands (including previously executed snippets) is usually made available to the user, in order to find and resubmit previous commands (“Command History”). It can be consulted sequentially through the arrow keys, but often includes a search facility as well. Some REPLs assign identifiers to commands in order to retrieve them arbitrarily. Some REPLs support saving and loading sessions (“Save and Load Session”). This may involve storing the execution context, or simply storing all user inputs to reproduce the execution context after loading. For some languages, the session can also be saved as a valid program executable outside the REPL.

REPLs behave differently when multiple code snippets are entered at once in a sequence (“Multiple Input”). Output is either provided for all the snippets from the sequence, or only for the last snippet (which could result in no output at all). Most REPLs allow the user to inspect the current execution context (“Summary of Current State”). And finally, some REPLs allow the results of previous snippets to be used in new snippets (“Access to Previous Results”), either for the last executed snippet or for all, for instance by assigning result values to special variables.

Feature Support of Existing REPLs Table 4.2 shows how the investigated REPLs support the features identified in the feature model of Figure 4.1. The table illustrates that no two REPLs share the same set of features. IPython supports most of the features, whereas PHP only supports a minimal set. Interestingly, PHP is the only REPL that does not print computed output values. The Go REPL (Gore) is the only REPL that simulates incremental execution by compiling a complete compilation unit in the background. Type-aware completion is not applicable to Node.js and R since the languages are dynamically typed and do not support type hinting. Sessions exported from SQLite and R include the snippets to reproduce data, but not the ones that query the data, so it is not possible to fully reproduce a session through this mechanism. Octave exports variables and their values, but not declared methods. Only three REPLs support exporting sessions as valid programs. Although IPython provides additional commands, they are all implemented in Python and can therefore be exported as well. As explained before, a valid Go program

Table 4.2 – REPL Interpreter Features (●= full, ◒ = partial, – = not applicable)

		CLing	JShell	Python	IPython	C# REPL	Node.js	PHP	PsySH	SQLite	R	Swift	Gore	Octave	Rappel	iRB
Snippet Execution	Incremental	●	●	●	●	●	●	●	●	●	●	●		●	●	●
	Full												●			
	Undo	●														
Summary of Current State		●	●		●	●			●	●				●	●	
Summary of Snippet Effects		●	●	●	●	●	●		●	●	●	●	●	●	●	●
Access to Previous Results	Access to last			●	●		●		●					●		●
	Access to all		●		●							●				
Multiple Input	Last output	●			●	●	●		●			●		●	●	●
	All outputs		●	●						●	●					
Snippet Completion	Keywords	●		●	●		●			●	●	●		●		●
	Syntax-aware	●			●							●				
	Identifiers	●	●	●	●	●	●	●	●	●	●	●	●	●		●
	Type-aware		●				–				–				–	
	Hierarchy-aware	●	●	●	●	●	●		●		●		●	●	–	
Definition Modification	Redefine		●[1]	●	●[1]	●					●[1]	●	●	●[1]	–	●
	Open & Extend														–	●
Help Command	REPL commands	●	●	●	●	●	●		●	●		●	●		●	
	Language use			●	●						●		●	●		
Command History (User Access)	Sequential	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	Search	●	●	●	●	●	●	●	●	●	●		●	●		●
	Arbitrary		●		●									●		
Save and Load Session	Current state									●	●			◒		
	REPL code snippets		●		●		●			◒	◒					
	Valid programs				●		●			◒	◒		●			

[1] The previous definition can be opened in an external editor for editing

is produced as part of every interaction with Gore.

The interactive interpreter for Swift also provides debugging facilities. This feature was observed but not discussed as a REPL feature because the two behaviors are accessed by running the interpreter in different 'modes'. Interestingly, the decision to provide both modes in a single tool was made from observing that they shared similar features, such as expression evaluation, data monitoring and step by step execution.

The wealth of features and diversity observed in REPLs motivated us to study the foundations of REPLs.

4.3 Methodology

In [15], we introduce and formalize the concepts of sequential language and exploring interpreter. Essentially, a sequential language is a language offering a "sequence operator", whose semantics follow the property that executing two code snippets delimited by this operator achieves the same results as executing these same two snippets independently

with a context preserved from one execution to the other. An exploring interpreter is a generic algorithm that creates and maintains an execution graph, where each node is a valid configuration, or a state of the execution context, and each edge corresponds to a program affecting the context and thus transitioning to a new state. Operations available to an exploring interpreter are executing a program (which potentially creates a new node and a new edge), reverting to a previous state, and displaying the current representation of the graph with an emphasis on the differences between the current state and the others.

In this section we propose a methodology for developing REPL interpreters based on these definitions. The methodology proposes to build a REPL on top of an exploring interpreter for a sequential language. In the event that the language targeted is not already sequential, a new sequential language can be defined as an extension of, or modification to, the base language. An exploring interpreter is essentially a bookkeeping device on top of a definitional interpreter and provides the "Incremental Snippet Execution" and "Undo" features directly (cf. Section 4.2). Additional motivation for using the exploring interpreter (for a sequential language) is that it promotes certain design principles while preserving the ability to implement many desirable features. These principles and their consequences are discussed in this section, together with a summary of the proposed methodology.

The core principles underlying our methodology are:

— the effects of a code snippet manifest as changes to an *explicit* state representation (a configuration),

— the effects of a code snippet are determined by the definitional interpreter used by the exploring interpreter,

— the effects of a sequence of code snippets is the composition of the effects of the individual snippets,

— and only code snippets change configurations.

For users of the REPL, the most important consequence of these principles it that an understanding of the definitional interpreter is enough to understand the precise behavior of the REPL for the language. In practical terms: to know the effects of code snippets, a user needs to understand the base language and the possible extension or modification introduced in its sequential variant. The extension or modification is made explicit by the definitional interpreter and should be communicated clearly (as precise documentation, a formal semantics, or an open-source implementation).

For engineers of the REPL, the most important consequence of the principles is that every feature (on top of "Incremental Snippet Execution" and "Undo") is implemented either:

— as a language extension (*e.g.,* the features "Definition Modification" and "Access to Previous Results"),

— as a series of interactions with the exploring interpreter (*e.g.,* "Multiple Input", explained below),

— based on information stored in the execution graph (*e.g.,* "Summary of Snippet Effects", "Summary of Current State" and "Snippet Completion"),

— or independently of the exploring interpreter, when the feature does not involve snippet execution (*e.g.,* "Help Command" and other meta-commands).

The methodology presented here is based on the hypothesis that many of the features of existing REPLs, including at least those in Figure 4.1, fall into the four categories listed above. This hypothesis is tentatively supported by the various feature implementations described across Section 4.4.

The methodology for developing a REPL for any base language $\mathcal{L}$ is formulated as the following steps:

1) Definitional Interpreter Formulate $\mathcal{L}$ as a language in terms of its concrete and abstract syntax, and a definitional interpreter that captures the effects of programs as a function over some set of configurations. If the language is sequential at this point, then steps 2–5 can be skipped.

2) *Phrase* Nonterminal To define a sequential variant $\mathcal{L}'$ of $\mathcal{L}$, reuse the syntax definitions of the previous step to define a new sort **phrase** with an alternate for each of the sorts of $\mathcal{L}$ that describe the syntax of a valid code snippet of the envisioned REPL. The syntax can also have other extensions or modifications, as long as **phrase** is the entry point of the syntax (the first syntactic component of a language).

3) *Phrase* Interpreter Define a definitional interpreter for $\mathcal{L}'$ to capture the semantics of phrases, reusing as much as possible the definitional interpreter of step 1, ideally by applying modular extension mechanisms (e.g., Object Algebras [61, 33], Rascal's `extend` [6]). Special consideration needs to be given to the effects of phrases to ensure

the next phrase is executed in the right context. For example, if the result value of a phrase needs to be available to the next phrase through a binding, this binding needs to be introduced as one of the effects of the first phrase.

4) ";"-Phrase Extend the sort **phrase** with an alternate that combines two valid phrases to form a phrase. For example, with the semicolon as a separator, let $p; q$ be a valid phrase if p and q are valid phrases.

5) Interpreter for ";" Extend the definitional interpreter of $\mathcal{L}'$ such that the effect of a phrase formed by combining two phrases is the composition of the effects of the combined phrases, *e.g.,* $I_{p;q} = I_q \circ I_p$. The language $\mathcal{L}'$ is sequential by definition as a result of this and the previous step.

6) Instantiate Explorer Obtain an exploring interpreter for $\mathcal{L}'$ by instantiating the generic exploring interpreter algorithm with the definitional interpreter for $\mathcal{L}'$. The implementation may be simplified compared to our definition, in that it maintains a simpler form of execution graph, if desirable. Instead of an exploring interpreter, the definitional interpreter for $\mathcal{L}'$ can also be used directly. In fact, any implementation that respects the semantics of the definitional interpreter can be used, *e.g.,* an implementation with real rather than simulated effects.

The interpreter can then be offered through various user interfaces, such as command-line interfaces, network services, or computational notebooks. The interface displays visualizations of the effects of phrases, *e.g.,* by showing output, computed values and new bindings, and can optionally implement additional REPL features.

4.3.1 Pragmatics

In the context of language workbenches [28] and DSLs [56], a common language implementation strategy is to define interpreters, consisting of functions traversing an abstract syntax tree whilst modifying a propagated configuration to express effects (following the Visitor design pattern). The case studies of the next section include such interpreters. The REPLs in these case studies are obtained through generic implementations of the exploring interpreter algorithm (in Java and in Haskell) that are easily specialized by providing the entry points of the abstract syntax and the interpreter. The

presented methodology is based on an exploring interpreter because it is a relatively natural and simple layer to add on top of the described definitional interpreters typically built with Rascal [47, 6]. Moreover, the generic exploring interpreter forms a suitable abstraction for reasoning about sequences of interactions between programmer and REPL – *e.g.,* saving and loading sessions and extracting base language programs – and for implementing advanced REPL and notebook features that support exploratory programming and live programming.

In theory, our approach can also be used for developing REPLs for (general-purpose) programming languages, as many languages can have their semantics expressed as a transition function. In practice, however, very few programming languages of this sort have an interpreter implemented as a pure function, or have a complete operational semantics from which such an interpreter can be derived. REPLs are not typically implemented with explicit state representation and few enable backtracking (in our survey only CLing supports "Undo"). However, an impure interpreter implementation can be used at step 6 (Instantiate Explorer) of the methodology. Although some advanced features – such as "Undo" – may then be harder to implement, the most important principles of our methodology still hold. In particular, the differences between the base language and its REPL should be formulated as extensions or modifications of the base language. This is achieved by updating the semantics of the base language such that repeated execution of its interpreter (*i.e.,* the composition of effects) gives the behavior expected of a REPL for this language. The details of how this can be achieved depend on the language and the techniques used to implement the language. Discussed next are the general patterns that have been observed in our survey.

4.3.2 Common REPL Language Extensions

Languages rarely provide directly an operator that corresponds precisely to the REPL top level. For example, a snippet with an uncaught exception is not expected to prevent subsequent snippets from being executed, whereas termination is expected when an exception occurs within a sequence of (;-separated) statements. Of the surveyed REPLs, only Gore prevents subsequent snippets from executing once a previous snippet raises an exception (a consequence of its "Full" execution model). In the other languages, the REPL top level catches any otherwise uncaught exceptions and presents them to the programmer after which a subsequent snippet can be executed. In languages with

constructs for catching and handling exceptions, one might explain or implement this feature with a top-level catch and a handler that prints the exception. For example, a snippet `{System.out.println(1); (1/0);}` can be considered as implicitly wrapped in a `try/catch` block in JShell as follows:

```
try {{System.out.println(1); (1/0);}} catch (Exception e) {
  ... // print the exception in a helpful format
}
```

This clarifies, in reference to the Java semantics, that any effects produced by a snippet before an exception is thrown, but not after, are preserved. However, the translation is inaccurate as a JShell snippet is not an isolated block, unlike a `try`-block. Bindings produced by top-level declarations are active when subsequent snippets are executed, *i.e.,* all snippets are in the same scope and the top-level catching exceptions does not change this. In the next JShell fragment, the meta-variable `$1` is available to subsequent snippets despite the exception.

```
jshell> 5; (1/0);
$1 ⇒ 5
|  Exception java.lang.ArithmeticException: / by zero
|        at (#2:1)
```

This example also highlights the importance of presenting new bindings, assignments, and any other effects to the programmer, providing the information required by the programmer to update their mental model of the REPL's execution state.

Another common example of a modification to the base language is the "Access to Previous Results" feature available in several REPLs of the survey (demonstrated by the variable `$1` in the above fragment). JShell and IPython ("Access to All") implement this feature as follows: whenever a code snippet produces a result value (other than void), this result value is assigned to a fresh variable. For example, if the *second* snippet sent to IPython produces result value `5`, then the variable `_2` is assigned `5`. The behavior differs between JShell and IPython when a code snippet contains multiple statements. In IPython ("Last Output"), the result of a sequence of statements is the result of the last statement [2], *e.g.,* the snippet `print(1);2;print(3)` prints `1` and `3` but has no result value. In JShell, the result of a sequence of statements is the result of each statement with a (non-void) result. If a snippet has multiple results, each result is assigned to a fresh variable. For example, if `3;2;System.out.println(1);` is sent as the first snippet to JShell,

2. Even when void. A possible alternative is to use the last non-void result.

then the variables `$1` and `$2` are assigned the values `3` and `2` respectively and `1` is printed. In Node.js (“Access to Last”), a statement such as `console.log(1)` produces `undefined` as a result, which is then assigned to the variable `_`. PsySH also assigns the last uncaught exception to the variable `$_e`. This feature is helpful in situations where the exception is not easily reproduced, *e.g.,* when caused by a (rare) non-deterministic, pseudorandom or timed event.

Most languages of the survey enable definitions to be overridden (“Definition Modification”), with only iRB also allowing extensions to existing definitions (“Open & Extend”). The main challenge to redefining or modifying existing definitions is checking whether an updated definition is consistent with definitions that depend on it. This is particularly challenging for statically typed languages such as Java. In JShell, any inconsistencies are reported when a (now incorrect) definition is used, as shown by the following interaction:

```
jshell> class B {int mymethod(){return 0;}}
|  created class B
jshell> class A {int mymethod(){return new B().mymethod();}}
|  created class A
jshell> class B {long mymethod(){return 0;}}
|  replaced class B
jshell> int x = 4; int y = new A().mymethod(); int y = 5;
x ⇒ 4
|  attempted to use class A which cannot be instantiated or
|  its methods invoked until this error is corrected:
|   possible lossy conversion from long to int
|   class A { int mymethod() { return new B().mymethod(); }}
y ⇒ 5
```

Note that the last snippet is not type-checked nor rejected as a whole, and that the error does not keep the other statements from being executed. Statements appear to be type-checked individually, with any errors causing only the individual statement to be rejected. However, the following JShell interaction shows that this is a simplification:

```
jshell> int x = 1; new A(); int y = 2;
x ⇒ 1
|  Error:
|  cannot find symbol
|    symbol:  class A
```

A downside of showing inconsistencies just before they cause problems is that a

menial mistake can cause a cascade of avoidable mistakes to go undetected, perhaps requiring tedious efforts to resolve. A downside of reporting inconsistencies as soon as they arrive is that they may be considered redundant and a nuisance when a programmer is aware and about to resolve the inconsistencies.

The C# REPL does not update method definitions affected by an update to another class. So when, in the example above, `mymethod` is called on a new instance of `A`, the behavior is that of the old `mymethod` of class `B`. (A similar example using fields rather than methods causes the C# REPL to hang.)

A general theme in the discussed language extensions is that they relate to the effects of code snippets on their successors. A REPL engineer should consider all the different kinds of (side-)effects code snippets can produce and decide for each effect whether it should propagate and, if so, how the programmer is informed of the effect, enabling them to update their mental model of the REPL's state. To help the programmer further, the ability to request an overview of the currently active bindings is desirable, especially together with a mechanism for inspecting (modified) type definitions.

4.4 Case Studies

This section discusses several REPL implementations for a number of languages with different user interfaces. The section is structured according to three case studies for the Rascal-defined languages MiniJava and QL, and the Haskell-defined language eFLINT. The case studies implement novel sequential variants of these languages.

4.4.1 A Jupyter Notebook for MiniJava

The MiniJava language is a subset of Java that retains the essential object-oriented features of Java [5, 23]. The semantics of a MiniJava program is given by its interpretation as a Java program. The specific implementation discussed here is implemented as a definitional interpreter in the Rascal language workbench [47]. The extension to a sequential MiniJava uses Rascal's modular extension mechanisms and demonstrates the methodology of the previous section.

The first part of the extension is choosing the top-level constructs of the language. As for **JShell**, these are expressions, statements, variable, class, and method declarations, and their associative composition. The syntax of MiniJava is extended by adding the

`Phrase` construct:

```
syntax Phrase
  = Expression ";" | Statement
  | VarDecl | ClassDecl | MethodDecl
  | assoc Phrase Phrase;
syntax Statement
  = ...
  | "throw" "new" StringLiteral ";";
syntax Expression
  = ...
  | Identifier "(" ExpressionList? ")";
```

The extension also includes a new method call variant, enabling (global) methods to be called without a receiver. The `throw`-keyword is added to demonstrate an implementation of handling uncaught exceptions. Exception values are simplified to string literals rather than arbitrary objects.

The definitional interpreter of extended MiniJava is defined in Rascal as the function `Config eval(Phrase, Config)`, shown[3] in Figure 4.2. The type `Config`, shared by both MiniJava interpreters, is defined as the following tuple type:

```
alias Config = tuple[
  Env env, Sto sto,
  int seed, Out out,
  Val given, MaybeFailure failed,
  Val result
];
data MaybeFailure
  = failure(FailureType e)
  | no_failure()
  ;
data FailureType
  = failed()
  | exception(str msg)
  ;
```

Configurations have the following fields: the current execution environment (`env`), the store (`sto`), a seed (`seed`), the output of all executed phrases represented as a list of strings[4] (`out`), a given value (`given`) of type `Val` used for passing arguments, the field `failed` to indicate if and why the execution got 'stuck', and a value with the execution's

3. The notation `(NT)‘...‘` is used to pattern match against or construct concrete syntax trees of type `NT`, where `NT` is some nonterminal defined in Rascal's native grammar formalism; the parts between fish-angle brackets represent typed holes of the pattern.

4. The implementation converts the integers printed by MiniJava to strings and inserts a newline, corresponding to Java semantics.

result (`result`). The `Val` ADT (not shown) defines constructors for references, integers, booleans, vectors (arrays), environments, lists, closures, classes, objects, and `null`. The alternative `failed()` of `FailureType` indicates the execution got stuck because the evaluated program is invalid (*e.g.,* due to unbound variables). The alternative `exception(str msg)` indicates an exception has been thrown with exception value `msg`.

The cases of Figure 4.2 that handle declarations (class, variable, or method) first produce an environment by calling the respective functions `declareClass`, `declareVariables` and `declareGlobalMethod`. These functions also produce output that informs the programmer of the successful binding of the respective class, variable or method. If a class is redefined, the programmer is also informed. The `collectBindings` function (not shown) adds the bindings in the computed environment (`result`) to the execution environment (`env`). The function `catchExceptions` (not shown) checks whether a phrase has failed or raised an exception. If so, the failure or exception is reported and removed, ensuring that the next phrase executes normally. Note that a MiniJava code snippet of the form `1;(2/0);3;` is parsed as a sequence of three phrases and not a code block consisting of three statements. Since the division by zero error is removed, the next phrase (`3;`) is executed normally. So, contrary to JShell, there is no distinction between phrases executed as separate code snippets or as a single, semicolon separated code snippet. This arguably makes the language more consistent. The behavior of statements separated by a semicolon in code blocks is unaffected, and an exception will terminate the execution of a code block when it arises.

The first two cases of Figure 4.2 deal with expression and statement phrases, reusing the original interpreters for expressions and statements (`eval` and `exec` respectively). A statement, which may be a code block consisting of multiple statements, either computes `null` or an environment that contains the bindings for all variables that have been assigned a (new) value. The function `setOutput` (not shown) inspects the computed bindings, if any, and prints the variable and its assigned value, matching the behavior of JShell. An expression computes a value such as an integer, a boolean or an object reference. The function `createBinding` (not shown) assigns the computed value to a fresh variable, using the `seed` field of the current configuration, and binds the fresh variable to the identifier `$<i>`, where `<i>` is generated from the seed. The applications of `setOutput` and `collectBindings` ensure that the new binding is reported to the programmer and is active when the next phrase is executed, matching the behavior of JShell.

The final case confirms that two consecutive phrases are evaluated by function-

```
Config eval((Phrase)`<Expression e> ;`, Config c)
  = catchExceptions(collectBindings(
      setOutput(createBinding(eval(c, e)))));

Config eval((Phrase)`<Statement s>`, Config c)
  = catchExceptions(collectBindings(
      setOutput(exec(s, c))));

Config eval((Phrase)`<ClassDecl cd>`, Config c)
  = catchExceptions(collectBindings(
      declareClass(cd, c)));

Config eval((Phrase)`<VarDecl vd>`, Config c)
  = catchExceptions(collectBindings(
      declareVariables(vd, c)));

Config eval((Phrase)`<MethodDecl md>`, Config c)
  = catchExceptions(collectBindings(
      declareGlobalMethod(md, c)));

Config eval((Phrase)`<Phrase p1> <Phrase p2>`, Config c)
  = eval(p2, eval(p1, c));
```

Figure 4.2 – Interpreting MiniJava phrases

composition. The implementation of method calls without receiver expression is not given.

The definitional interpreter of the extended language forms the interface to language services such as REPLs and computational notebooks. The connection between the definitional interpreter and Rascal's notebook framework Bacatá is discussed next.

Exploring Interpreters in Bacatá Bacatá [55] is a generic Jupyter [5] kernel generator for languages developed within the Rascal Language Workbench. Bacatá is extended to support notebooks based on exploring interpreters. The generic implementation of the exploring interpreter maintains a full execution graph (in accordance to the previous definition). Bacatá relies on the definition of a language `repl`, a value of the `REPL` ADT shown below:

```
data REPL[&T]
= repl(&T initConfig, &T (str, &T) handler,
   Completion (str, int, &T) completor,
   Content (&T, &T) printer);
```

5. cf. http://jupyter.org/

A value of `REPL` contains all the information required to build a REPL command-line interface for a language, or, together with Bacatá, a computational notebook. The type parameter `&T` represents the configuration (*e.g.,* `Config` of MiniJava). The `handler` takes a line of input and a configuration and produces a new configuration. The completor can be provided for tab-completion services. Finally, the `printer` produces (HTML) content from the previous and/or current configuration.

Bacatá is used as an interface between a Jupyter server and the language's REPL. The workflow that describes the communication among these components is as follows: Jupyter takes the user's code snippets and sends them to the language's interpreter through Bacatá. Bacatá takes the user's code and calls the language's `handler` (defined in the `repl` value), which is responsible for calling the parser and then the interpreter of the language. Finally, the `handler` produces a result, which is then displayed to the user, using the `printer`.

Figure 4.3 shows a simple notebook for MiniJava, produced with Bacatá. The right shows the execution graph for exploring the user's interaction with the notebook. The active node is colored green. The user can click any other node, to make it active. The next cell will then be executed in the context of that very configuration, resulting in a split in the graph if the resulting configuration differs from the activated one.

A Notebook Interface for MiniJava Obtaining a REPL-style command-line or notebook interface for MiniJava amounts to instantiating the `REPL` data type with the appropriate handlers, printers, and completors. In the case of a Bacatá-generated notebook Jupyter interface, the programmer has access to a visual representation of the execution graph of the exploring interpreter, as shown in Figure 4.3.

The handler for MiniJava parses the incoming input as a `Phrase` and calls the extended definitional interpreter, which returns a new configuration. The printer takes the old and new configuration and prints relevant output. After a successful execution, the differences between the `out` components of the new and old configuration is shown. In the case of a declaration, the difference between the two `env` components gives the new bindings. The completor uses the bindings in its input configuration to suggest possible completions for identifiers.

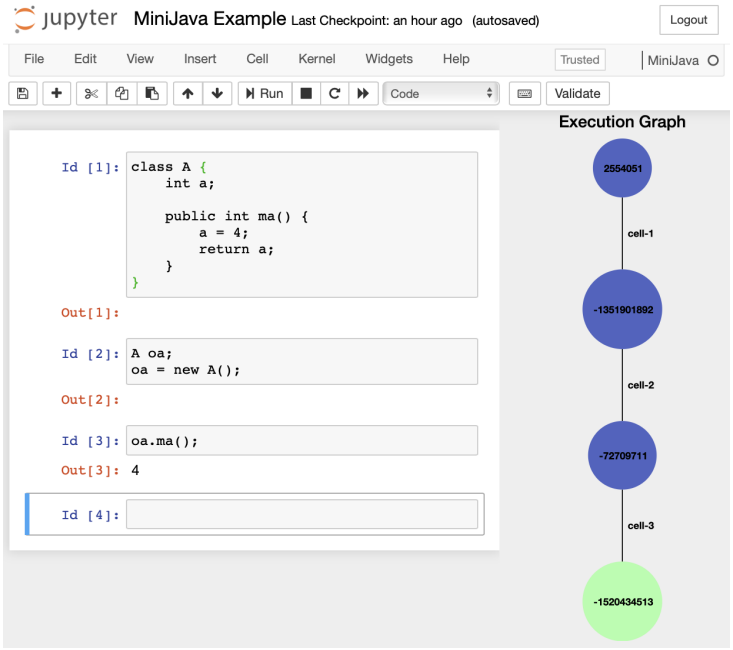

Figure 4.3 – Example of MiniJava in a notebook environment

4.4.2 QL: A DSL for Questionnaires

QL is a small language for defining interactive questionnaires [29, 28], like tax filing forms or online surveys. A QL form defines a sequence of questions, where each question has a label, an identifier, a type (boolean, integer, or string), and an optional expression if the question is computed. Expressions contain the usual arithmetic and comparison operations, and allow referring to the current value of another question. Furthermore, questions can be made conditional using if-then and if-then-else constructs.

The goal of a QL program is to be rendered as an interactive GUI program, where the user enters values for the (non-computed) questions. Depending on this input, conditional questions may be shown or hidden, and the value of computed questions may be recomputed, similar to a spreadsheet. A simple example is shown in Figure 4.4, including its rendering as an interactive UI.

From a REPL perspective, QL is interesting, because a form specifies a conditional data-flow network rather than a program consisting of instructions. Nevertheless, in this section we introduce a prototype REPL for QL, both as an instructive thought experiment, and to stress the concept of sequential language.

Abstractly, the semantics of QL can be described with the following (Rascal) function signature:

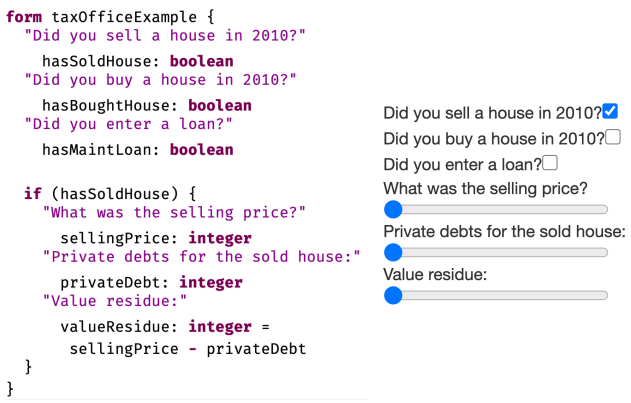

Figure 4.4 – A QL questionnaire (left) and its rendering (right)

```
tuple[UI, Env] eval(Form form, Env env, Event evt);
```

Given a form, an environment mapping question identifiers to values (`Env`), and a user event (`Event`), the function `eval` produces a rendering (`UI`) and an updated environment. Running a QL questionnaire then amounts to constructing an initial rendering, and then updating the current environment and redrawing the UI after every user action.

To provide a REPL interface for QL, we extend the language with a new start nonterminal, `Cmd`, the definition of which is shown in Figure 4.5. Commands are the snippets that the user can enter at the command line.

The first four alternatives of `Cmd` capture constructs to manipulate forms. The user can define complete forms, append or prepend individual questions to the current form, and replace questions arbitrarily nested in the form using a positional reference mechanism (`Addr`).

The last two alternatives can be used to evaluate expressions, which shows the result,

```
syntax Cmd
  = Form                // define form
  | Question            // append a question
  | Question "..."      // prepend question
  | "@" Addr Question   // replace question
  | Expr                // evaluate expression
  | Id "=" Value;       // perform user action

syntax Script
  = Cmd* commands       // batch perform commands
```

Figure 4.5 – Language extension for ReplizedQL

or update the value of a (non-computed) question, if the current state of the UI allows it. The update-value action simulates a user interaction if the form would have been rendered as a proper UI. Finally, Figure 4.5 defines a `Script` nonterminal to combine multiple commands in sequence.

The interpreter for commands is a function that takes a command and the current configuration and returns a new configuration:

```
Config eval(Cmd cmd, Config cfg) { ... }
```

The `Config` type captures the current environment, the current form, and a list of output values (UI renderings and expression evaluation results).

That our definition of QL is sequential can be seen from the definition of the interpreter for `Script`s:

```
Config eval(Script scr, Config cfg) =
  ( cfg | eval(cmd, it) | Cmd cmd ← scr.commands );
```

This function simply composes the `eval` function for commands for every command in the script[6]. This follows the definition of sequential language introduced earlier.

A Sample Interaction The above interpreter for commands can be hooked to Rascal's standard REPL infrastructure to obtain a command-line interface for QL. We illustrate the semantics of sequential QL below, using a sample user interaction. The code snippets use `>` as prompt, the output of a command is shown directly below.

First, let's define a simple form:

```
> form simple { }
  ·
```

The result is the empty rendering of the UI, indicated by ·. Then we append a (computed question), labeled "A", of type integer:

```
> "A" a: integer = c + b + 1
  A ·
```

The `a` question is not conditional, so it is shown in the UI rendering; note however that the value of the question is still undefined because questions `c` and `b` have not yet been defined.

The `b` question could be defined as follows:

6. The notation `(init | ... it ... | gen)` is Rascal syntax to write a reduce operation.

```
> if (a < 20) "B" b: integer = c + 1
  A ·
```

Since `b` is still undefined (because `c` is), `a` remains undefined as well, and as a result, the visibility condition of `b` evaluates to false. This all changes, however, after defining `c`:

```
> if (a > 20) "C" c: integer
  A 2
  B 1
```

The question `c` is not computed, so it receives an initial default value (in this case 0). Both `a` and `b` can now be computed, as well as the condition of `b`, causing `b` to be shown in the UI. Now let's change the value of `c`:

```
> c = 10
  A 22
  C [10]
```

Setting `c` to 10 disables `b`, but changes the visibility condition of `c` to true, making it appear in the UI. The square brackets around the value of `c` indicate it is editable.

Changing the value of `c` to 5 updates the UI accordingly:

```
> c = 5
  A 12
  B 6
```

Now `b` becomes visible, and `c` is hidden again.

It is possible to add questions to the beginning of the form:

```
> "D" d: integer = 3 * a...
  D 36
  A 12
  B 6
```

Or using the path-based address notation:

```
> :form
form simple {
 [0] "D" d: integer = 3 * a
 [1] "A" a: integer = c + b + 1
 [2] if (a < 20)
  [2.0] "B" b: integer = c + 1
 [3] if (a > 20)
  [3.0] "C" c: integer
}
> @2.0 "c + 1 is:" b: integer = c + 1
  D 36
  A 12
  c + 1 is: 6
```

The `:form` meta-command pretty-prints the current form annotated with addresses for every question. Using the `@`-notation, the user can replace any question in the form, in this case to change the label of the `b` question.

Note that the append-, prepend-, and position-based adding and replacement of questions can be considered a rather low-level (maybe even pathological) way of editing a program (reminiscent of the line-based editors of the past). Nevertheless, without necessesarily claiming this is a realistic way of evolving programs, it does illustrate a kind of REPL "completeness", where every program and program change can be realized using commands at the prompt.

4.4.3 eFLINT: Executable Normative Specifications

eFLINT is a DSL for developing executable normative specifications used to reason about compliance with regulations, contracts and/or policies [11]. eFLINT programs are used to simulate or verify normative decision-making processes. The methodology of Section 4.3 has been applied to develop two REPLs on top of one exploring interpreter for eFLINT. The implementation of eFLINT is available at GitLab [10].

REPL Interfaces The first REPL is a command-line tool for exploring compliant and non-compliant behaviors. Figure 4.6 shows an example session where the user explores the norm "children can ask their parents for help". As a meta-command, a user can choose actions and events to trigger from a given list of options. Choosing an action or event has the effect of updating a database of 'facts', representing the state of the world at a particular moment in time. A fact is said to 'hold true' if it is present in the database. Some facts are reifications of actions and correspond to acceptable behaviors when they hold true and when they are enabled by their pre-conditions. Disabled actions can be executed in order to explore non-compliant behaviors (although causing a violation). Other facts represent duties, which need to be 'terminated' before one of their violation conditions holds true. The phrases of the language are declarations of fact-, act-, event- and duty-types, action or event triggers, insertion and removal of facts, and queries on the database. After a phrase is executed, the user is presented with the changes in the database, newly defined types, any violations and a new list of options.

The second REPL is a TCP server that listens on a chosen port for incoming phrases and responds with the same information as the command-line REPL (in JSON form). The

```
#0 > Fact person. Placeholder parent,child For person
new fact-type person
no enabled actions or events
#3 > +person(Alice). +person(Bob) // introduce persons
+"Alice":person
+"Bob":person
no enabled actions or events
#5 > Fact parent-of Identified by parent * child
new fact-type parent-of
no enabled actions or events
#6 > +parent-of(Alice,Bob)
+("Alice":person,"Bob":person):parent-of
no enabled actions or events
#7 > Act call-for-help Actor child Recipient parent
                       Holds when parent-of()
new fact-type call-for-help
+("Bob":person,"Alice":person):call-for-help
enabled actions & events:
1. ("Bob":person,"Alice":person):call-for-help
#8 > :choose 1    // Bob asks Alice for help
enabled actions & events:
1. ("Bob":person,"Alice":person):call-for-help
#9 > :revert 7    // to before the action was declared
+("Alice":person,"Bob":person):parent-of
#7 > :current     // show the current set of facts
"Alice":person
"Bob":person
("Alice":person,"Bob":person):parent-of
#7 > ?Enabled(call-for-help(Bob,Alice)) // query
undeclared type: call-for-help
```

Figure 4.6 – A session with the eFLINT command-line REPL

TCP server is used as a general method for connecting other languages with eFLINT to benefit from the normative specification written in eFLINT. For example, a program can send queries to the eFLINT server to check whether certain actions are enabled before actually performing them. In this way, it is possible to develop software that is 'compliant by design'.

The REPLs are developed on top of an exploring interpreter for eFLINT, briefly explained next.

Execution Tree The type *Explorer* is an alias for functions that receive an *Instruction* and return a *Response* in the *IO* monad (Haskell's mechanism for input and output).

$$\begin{aligned}
&\textbf{type } \mathit{Explorer} &&= \mathit{Instruction} \to \mathit{IO\ Response} \\
&\textbf{data } \mathit{Instruction} &&= \mathit{Execute\ CPhrase} \mid \mathit{Revert\ Int} \mid \mathit{Display} \\
&\textbf{data } \mathit{Response} &&= \mathit{Success\ Node\ CPhrase\ Node} \mid \mathit{ExecError\ Error} \\
&\textbf{type } \mathit{Node} &&= (\mathit{Int}, \mathit{Config})
\end{aligned}$$

The values of $\mathit{Instruction}$ correspond to the actions of the generic exploring interpreter algorithm. There are two types of response, for successful executions and failing executions respectively. One of the values of Error indicates that the integer given as part of some **revert** action does not correspond to a known configuration. The success response contains the elements of an edge in the execution graph: two nodes and a label (phrase). The edge gives the effects, in terms of an input and output configuration, of the last phrase executed by the exploring interpreter. A node is a configuration and an integer that uniquely identifies the node. The label is a value of type $\mathit{CPhrase}$, a phrase that has been compiled.

A configuration contains information about declared types (a type environment), a database of facts and a list of output holding any reported violations:

$$\textbf{data } \mathit{Config} = \mathit{Cfg}\ \{\mathit{tyenv} :: \mathit{TyEnv}, \mathit{state} :: \mathit{Set\ Fact}, \mathit{out} :: [\mathit{String}]\}$$

The algorithm maintains a tree rather than a graph, and does so in a way that makes it very simple to find the path from the root to any given configuration in the tree. The type SIDMap is an alias for a map mapping integers to the configurations with which they form a node. The type $\mathit{History}$ represents a tree as a collection of edges.

$$\begin{aligned}
&\textbf{type } \mathit{SIDMap} &&= \mathit{IntMap\ Config} \\
&\textbf{type } \mathit{History} &&= \mathit{IntMap}\ (\mathit{Int}, \mathit{CPhrase})
\end{aligned}$$

If x maps to (y, p) in the $\mathit{History}$ map, this means that there is an edge $\langle \gamma, p, \gamma' \rangle$ in the tree where y is the integer identifying γ and x is the integer identifying γ'.

REPL Features The function $\mathit{getPath} :: \mathit{Int} \to \mathit{SIDMap} \to \mathit{History} \to [\mathit{CPhrase}]$ receives an integer identifying a node and uses the maps to compute the sequence of phrases labeling the path from the root of the tree to the node. The function is used to save a session by pretty-printing and storing the returned phrases in a file.

The definitional interpreter of eFLINT receives compiled phrases ($\mathit{CPhrase}$s) as input. The tool-set for eFLINT contains a compiler that translates from Phrase to $\mathit{CPhrase}$.

The compiler checks whether a $Phrase$ is well-typed and applies conversions to make explicit certain implicit operator applications. Compilation is performed by the function $compile : TypeEnv \to Phrase \to CPhrase$, receiving as input the type environment of the current configuration held by the exploring interpreter.

When the command-line or TCP server REPL receives a $String$ for execution, the string is parsed as a $Phrase$. If successful, the $Phrase$ is type-checked and compiled to a $CPhrase$. The $CPhrase$ is sent as an **execute** action to the exploring interpreter, which invokes the definitional interpreter and responds either with an error or with the edge of its graph representing the latest execution. This edge is given to a function called $effectsOf$ to compute the effects of executing the phrase. The function $effectsOf$ finds any new bindings by computing the difference between the two type environments of the input configurations, finds any created or terminated facts by computing the difference between the two state components, and finds new violations by computing the difference between the two output components.

4.5 Discussion

Limitations & Future Work The techniques described in this chapter are applicable to languages that can be implemented by deterministic interpreters with explicit state representations. Moreover, if an execution graph is not needed, then state does not have to be represented explicitly (see Section 4.3.1), as long as the effects of top-level phrases still compose and are communicated clearly to REPL users. This requirement does not necessarily rule out concurrent, non-deterministic, compiled or data flow languages. In some cases it is possible to *model* the complicating aspects of these languages, *e.g.,* with thread models, data flow graphs and lists to capture non-deterministic results.

Purely functional interpreters with explicit state representation are, however, further removed from actual implementations and may be less suitable for developing practical REPLs. For instance, a definitional interpreter for C can model memory (pointers) rather than providing real memory access. A REPL for C can also be based on an interpreter that invokes a C compiler, wrapping current and previous code snippets in `int main() {…}`, before compiling and executing the resulting program (similar to the Go REPL discussed in Section 4.2). It is possible to obtain a REPL interface in this way, but it would not be based on a sequential language and the explorative quality of exploring interpreters would be lost. The applicability of our approach in the context of such compilation-based

REPLs is to be investigated further.

The interpreters discussed in this chapter are all implemented in functional programming languages (Rascal and Haskell) with immutable data. Maintaining the execution graph is therefore easy to implement, but it may come at a cost of performance and memory footprint. Further research is needed to represent the graph more efficiently, for instance by maximizing sharing, caching intermediate results, or selectively culling the graph. The pragmatics of a REPL (small snippets, immediate feedback, etc.), however, suggest that such optimization might be premature.

Although not shown here, exploring interpreters can also be used to realize additional features not typically found in REPLs by performing sequences of **execute** and **revert** actions in response to a single user action. For example, if a user edits a cell in a notebook, this could cause the exploring interpreter to revert to the configuration in which that cell was originally executed, keeping track of all cells undone this way, re-executing the (now modified) cell, and executing all the remembered cells in the order they were first executed. Further research is needed to establish how this relates to live programming [77, 75]. The QL language described in Section 4.4.2 has a live programming environment and forms a natural starting point for this study.

The MiniJava notebook discussed in Section 4.4.1 displays the execution graph of the exploring interpreter, allowing arbitrary rollbacks to explore alternative execution paths. In future work we will explore the ability of the exploring interpreter to support exploratory programming. More generally, we aim to describe algebraic operations over execution graphs for both live and exploratory programming.

The methodology of Section 4.3 starts from a single base language. The methodology is easily generalized to take multiple languages as a starting point and defining a single sequential language as an extension of all of them, which is then used as the basis for a so-called *polyglot* REPL. The definitional interpreter for the sequential extension may however not be easy to define when the effects of the phrases of the different base languages are not easily reconciled. In a future study we hope to formulate and demonstrate the more general methodology and to show its benefits to developing polyglot REPLs and notebooks.

Conclusion REPLs provide programmers with a direct interface to a programming language, supporting exploration, testing, and incremental development. All mainstream languages have REPL interfaces, which indicates the value they represent to programmers.

However, the actual language that is accepted by the REPL is often not well-defined, and engineering REPLs lacks solid design principles.

In this chapter we have surveyed existing REPLs in a feature-oriented domain analysis, showing a wide diversity in feature support. To make the relation between a REPL and its language precise, we have defined and formalized the notion of sequential language, and used it as the basis of a methodology to construct REPL interpreters. The versatility of the approach has been demonstrated in three case studies, one based on MiniJava, and two based on DSLs (QL and eFLINT). The case studies show notebook, command-line, and client-server REPL interfaces, developed using the methodology by extending base languages and reusing existing interpreters.

The concept of sequential language and its associated language design and engineering guidelines may provide better insight into the essence of REPLs, and promote a principled approach to the construction of REPLs.

CHAPTER 5

FROM DSL SPECIFICATION TO INTERACTIVE COMPUTER PROGRAMMING ENVIRONMENT

In this chapter, we explore how to integrate language services of executable DSLs into interactive computer programming environments through a generative approach. We start by motivating the need for interactive computer programming environments for DSLs users (Section 5.1). Next, we give an overview of our generative approach to use existing DSL specifications in order to derive a REPL interpreter (Section 5.2). Then, we go into more details about the technical aspects, and we provide an implementation based on the language workbench "GEMOC Studio" (Section 5.3). Finally, we evaluate our generative approach on two DSL specifications (Section 5.4). We close this chapter by discussing perspectives opened by this contribution (Section 5.5). This chapter presents the work achieved in our SLE'19 publication ([39]).

5.1 Motivation

It has been recently recognized that the different tasks performed with a given language, possibly by different stakeholders, would require specific language support (*e.g.,* dedicated environments with the right facilities) [2]. For instance, while it can be convenient to have a comprehensive editor for editing complex logo programs, one would like access to other kinds of environment such as interactive environments that allow to immediately get the result of the program at the time it is being edited. Such an environment would be very useful for education purposes, such as learning about the language or evaluating existing libraries. Generally speaking, it becomes way easier to introduce a programming language to beginners if they don't have to conform to the structure of an entire program when they want to write their very first instructions [31]. *Java* is a perfect example of this [34, 3], as this is a language that requires to follow a rather complex process to simply print 'Hello World!' in a terminal emulator: the

programmer needs to define a public class, declare a method both public and static and give it a specific name, while also specifying that it will require a certain type of arguments, before finally writing the very first statement of a program. Of course, the syntax makes perfectly sense, and understanding it will be important in order to learn how to use the language, but it can be detrimental to introduce it during the first glimpse at Java programming. An interactive computer programming environment can be beneficial to simply learn how specific language statements work in practice. In the case of *Logo*, introduced in Section 2.2, interactive computer programming environments would provide immediate feedback from any statement of the language, mainly through an updating graphical representation of the canvas, and thus help to learn complex programming concepts. Note that interactive computer programming environments can be also beneficial for experienced language users, in order to test the behavior of code snippets at any time. It makes possible to run processes in an arbitrary order while having access to all the intermediary results, which can be very useful when experiencing a new API. Beyond learning the different facilities provided by the API, it may also help to get experience with specific protocols required in the use of the API (*e.g.,* feedback on advanced behaviors of HTTP when learning how to use a web framework).

Interactive computer programming environment can take various forms, including a basic language shell where single statements can be executed sequentially, based on a global context, and possibly with the history of the intermediate states of the built program. Alternatively, a notebook interface provides a virtual notebook environment used for literate programming. It supports the definition of a sequence of pieces of code, with intermediate results, and possibly word processing to document the program (Jupyter notebooks for example let the user embed both Markdown and LaTeX in-between their code snippets).

In all cases, an interactive computer programming environment requires a language interpreter in the form of a read–eval–print loop (REPL), which is able to execute a program piece by piece. As addressed in Chapter 4, this requires a language supporting different execution entry-points, and a specific management of the execution context and flow. The context is usually global (though some scoping rules can still apply), and the execution flow is sequential, beginning with the starting point defined by the programmer. Other strategies may exist, in particular regarding the execution flow where a specific order may be imposed to keep reproducible executions. REPL execution engines offer facilities such as: history of inputs and outputs, input editing and context specific

completion over symbols, path names, class names and other objects, as well as help and documentation for commands.

Nowadays, most general purpose languages take these considerations into account, and ship their own REPL implementation: *Python* and *C#* run in interactive mode by default, *Node.js* can execute the "repl" module out of the box, *PHP* can be run as an interactive shell with the switch "-a", *Swift* offers a REPL directly integrated within Xcode, *Ruby* is shipped with the executable "IRb", and even *Java* includes its own REPL "JShell" since the version 9 of JDK.

While interactive computer programming environments have been specifically developed for general purpose languages, there is currently no approach that helps to turn an existing DSL specification in a way that a complementary interactive environment can be automatically generated. Instead, a new specification, either extending another specification or built from the ground up, must be established for the specific purpose of driving the development of an interactive environment. Hence, the research question (RQ) we address in this chapter is the following:

> RQ: Is it possible to automate the transformation of an existing textual and interpreted DSL specification in such a way that an interactive computer programming environment can be automatically derived?

We consider a DSL specification that includes:

— a syntax described as a grammar defined using a rule language based on "Extended Backus-Naur Form Expressions",

— static semantics built from a set of first order logic rules,

— and operational semantics based on a *pure* interpreter pattern. *Pure* means that each *interpret* method of the interpreter accesses only the current node's and its children's attributes, and the attributes from the context object, but cannot go back to the node's parent.

Our approach provides the required abstractions for complementing an existing DSL specification with the minimal information needed for generating a REPL (*i.e.*, the expected execution entry points and their associated documentation and output messages), and automates the transformation of the DSL specification so that an interactive computer programming environment can be derived. In particular, we automate the transformation of the syntax to parse separate pieces of code, and the semantics to support the execution of single statements according to a specific execution context

and flow. We also propose a unified REPL interface in order to derive different kinds of environment, *e.g.,* a language shell and a notebook interface.

5.2 Approach Overview

The main objective of our approach is to automatically transform an existing DSL specification (cf. upper left part in Fig. 5.1) initially used to drive the development of a comprehensive integrated development environment (cf. lower left part in Fig. 5.1), into a new one (cf. upper right part in Fig. 5.1) that can be used to automate the development of interactive computer programming environments (cf. lower right part in Fig. 5.1).

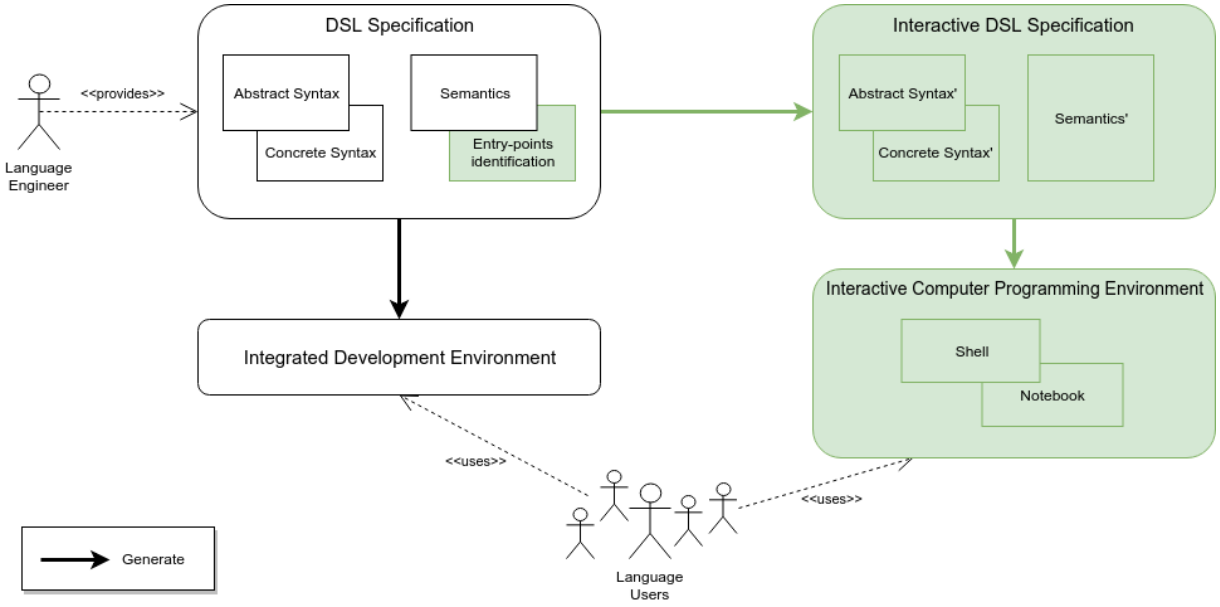

Figure 5.1 – Programming Environment Generation from DSL Specifications

Since we are aiming for a systematic transformation process, we put some restrictions on the supported DSLs: they need an extended BNF grammar and operational semantics that follow a pure interpreter pattern. We believe that these characteristics are ones of the most common, and thus that these are acceptable limitations.

To reach this objective and address the RQ specified in the previous section, we identified four challenges:

C1 Identification of the different execution entry points that are meaningful for the corresponding REPL, and the expected outputs and help messages given to the

user.

C2 Transformation of the syntax so that we can parse and load partial programs corresponding to the identified execution entry points.

C3 Definition of a sound yet flexible execution context and flow management,

C4 Transformation of the semantics so that we can execute pieces of code corresponding to the identified execution entry points.

Entry-points Identification (C1) We define as language entry-points the constructs that a language user can use and that can be executed outside of any other context. With most traditional DSLs, the execution can only handle a complete program and builds a context for it, that will later be used by all the statements and expressions. The only entry-point is as such the *complete program*. Here, we want to provide several entry-points with a granularity lower than a regular program.

However, the granularity of the new execution entry-points cannot be inferred automatically. The choice is up to the language engineer. In practice, they correspond to any expression to be considered as an executable statement within the interactive environment. It is therefore necessary to provide within our approach means of specifying these new entry points, and the underlying framework for loading and saving single statements.

To report the execution entry-points, relevant abstractions must be provided to the language engineer for enhancing the initial DSL specification. Abstractions must support the identification of the relevant statements to be executed independently. Moreover, to give intermediate feedback to the language user, the new entry-points need to be also supplied with additional information about the expected outputs (*e.g.,* the user would expect to get an evaluation result when he inputs a *Logo* expression), and possibly an help message.

In addition to the identification of the new execution entry-points in the DSL specification, a corresponding framework must be provided to save and load partial programs corresponding to the possible entry-points. For such a purpose, we transform the syntax specification in order to make partial programs valid for the parser and the corresponding syntax tree. We refer to these partial programs as *instructions*.

Transformation of the Syntax (C2) On the basis of the identified entry-points, the existing syntax specification must be transformed to enable all of these entry-points as

valid instructions. A new root rule within the grammar specification and a new root node for the AST named *Interpreter* are integrated. The latter contains all the newly defined valid entry points, and possibly the definitions of additional behaviors to instrument the execution.

Execution Context and Flow Management (C3) We do not handle complete programs anymore but independent instructions. In order to keep a consistent execution through the different iterations of the REPL, a global execution context and its flow along the independently executed instructions must be managed. This is the role of the proposed *Interpreter*, that will instantiate a context then simply pass it to instructions before executing them. Execution results must also be stored in a specific variable, and an execution trace manager must be provided to offer a complete history.

We propose a generic interface to interact with (sequences of) instructions, used by generic interactive computer programming environments such as a language shell and a notebook interface.

Transformation of the Semantics (C4) The last step of our approach consists in transforming the DSL semantics, so that the instructions can be executed independently, over a global context, and according to the proposed interface. In order to automate this transformation, we make several assumptions about the form of the DSL specification. In our current approach, we consider operational semantics defined according to the interpreter design pattern, *i.e.* an operation associated with each node of the AST, and the same context object associated with this operation containing all the dynamic information related to the language semantics. We also assume that the context passed to each nodes can be instantiated and initialized from the *Interpreter* node. If the context cannot be properly initialized on its own, we still give the ability to a language engineer to include custom rules in the semantics, but we do not try to infer them during the REPL language generation. Finally, each operation associated to a node of the syntax tree declared as an entry point must not make assumptions about the execution context other than that related to the initialization, nor about the structure of the parent nodes. We defined this property as a *pure* interpreter pattern.

5.3 Technical Details and Implementation

This section describes the technical details of the different steps of our approach, and proposes a particular implementation [1].

The proposed implementation comes in the form of a prototype based on the GEMOC Studio [17]. The GEMOC Studio is an Eclipse package on top of the Eclipse Modeling Framework [74], which has been experienced in various industrial projects. Among others, it offers a language workbench that supports the modular specification of DSLs, using Ecore for the abstract syntax, Xtext for the textual concrete syntax, OCL or Xtend for the static semantics and ALE for operational semantics. Other alternatives are also proposed but not illustrated in the scope of this chapter.

We illustrate both the approach and the implementation using the simple but real-world *Logo* language introduced in Section 5.1.

5.3.1 DSL Specification Enhancement

As presented in Section 5.2, we first provide to the language engineer the relevant abstractions for specifying the multiple execution entry-points:

— Identification of the valid instructions to be executed independently,

— Definition of the expected outputs as intermediate results, and

— Definition of the help messages for the language user.

In practice, these information could be provided either in the syntax or in the semantics. However, we had to consider that the visitor can be augmented by additional helpers for a given Ecore object, and there is no way of deciding on which to use. Besides, the output needs to refer to dynamic information which is mainly available within the semantics.

In order to identify the required entry-points, the language designer could methodically:

1. take a look at each rule of the grammar and choose the ones to provide in the REPL
2. decide on the expected outputs for each of the chosen rules
3. factor them in to abstract parent rules if the outputs are the same

1. See our prototype at https://anonymous.4open.science/r/84105ce9-f47c-4dd2-936e-9eb2dd345ad0/

To let the language engineer define the required information, we introduce a new metamodel shown in Fig.5.2. It can be seen as a dedicated meta-language to modularly complement the initial DSL specification with information related to the REPL. The core element of this metamodel is the *Instruction* meta-class. It defines three main information required to generate the REPL:

1. the new entry point in referencing a specific AST and the ALE method that defined the operational semantics for this node.
2. the help message to display if a user wishes to request help on this specific entry point.
3. the elements of the semantics to be used as textual outputs of the interpreter. These elements can be either attributes related to the execution (*i.e.,* attributes of the operational semantics), calls to existing methods of these semantics, or calls to ALE methods defined by the language engineer (*e.g., evalResult* and a set of *evalParams* that could target an *ALE Expression*).

The second main meta-class is *Interpreter*. It allows language engineers to specialize, among other things, the initialization of the execution context of the interpreter.

We provide two concrete syntax to populate the model conform to the *ReplDefinition* metamodel: a dedicated DSL, and additional annotations to ALE. This model represents the required information to drive the transformation of the DSL specification.

Using a New DSL The first concrete syntax is a new DSL built as an extension of ALE. Fig. 5.3 shows the definition of the REPL for the *Logo* language.

The first part defines the entry points, associated outputs and help messages. It defines two new entry points: Statement::execute and Expression::compute. It also defines their associated outputs to display: *logo_repl.turtle.toString()* and *output.toString()* (*output* refers to the actual value possibly returned by the entry-point). Finally we could define the help associated with these entry points (it has been done only partially in the Logo example).

The second part specifies the *Interpreter*, the specialization of its context of execution and its initialization. *Interpreter* will serve as the starting point of the execution of the REPL and will manage the future instructions. It will contain the same kind of runtime data as the entry-point of the base DSL, which will define the global context of the REPL. In the case of *Logo*, this means the turtle graphics, and a symbol table used to

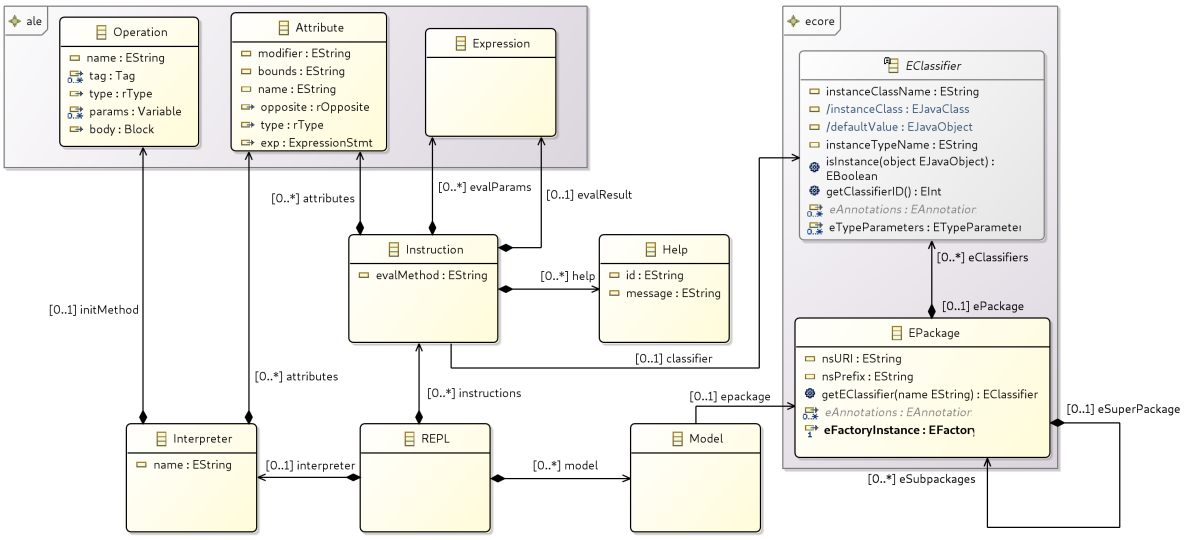

Figure 5.2 – ReplDefinition Metamodel

define procedures (using a symbol table for this is simply a design choice of the language engineer). In order to initialize this global context, the initialization method of the interpreter will also be the same as the base DSL.

Using Annotations The same kind of information can be defined directly within the existing ALE operational semantics using a set of annotations. We provided the language engineer with the following annotation:

```
@repl__outputtarget__outputcall__...
```

The output specification here is optional, and represents either an attribute read or a method call on a semantic object from the global context, or on the result of the operation being annotated. Note that the syntax is based on underscores because of limitations of the ALE language, which only supports identifiers as annotations. Using two underscores as the separator allows for compatibility with semanics using either camel case or snake case for identifiers.

```
1  extend http://www.gemoc.org/logo as logo
2
3  instruction logo.Statement:
4      help right "Turn turtle of 'p' degrees to the right"
5      help forward "Move turtle of 'p' units forward"
6      execute(logo_interpreter.turtle, logo_interpreter.st)
7          => logo_interpreter.turtle.toString();
8  instruction logo.Expression:
9      compute(logo_interpreter.st)
10         => output.toString();
11
12 interpreter logo_interpreter {
13     attribute Turtle turtle;
14     attribute SymbolTable st;
15     initmethod def void init() {
16         self.turtle := Turtle.create();
17         self.turtle.xpos := 0.0;
18         /* ... */
19         self.st := SymbolTable.create();
20         self.st.init();
21     }
22 }
```

Figure 5.3 – Example of ReplDefinition Model for Logo

The language engineer can also set the help message to display by using a *javadoc* like comment:

```
/**
 * keyword: Help message
 */
```

In the base semantics for *Logo*, the only additions besides the optional help messages were the two following annotations (cf. Github repository):

— `@repl_turtle_toString` for the `execute` operation of *Statement*

— `@repl_output_toString` for the `compute` operation of *Expression*

Fig 5.4 shows a code excerpt from the operational semantics of the Logo language extended with the proposed annotations to define the information required to complement the DSL specification with an interactive programming environment. The set of annotations is however less expressive than the DSL. Indeed a language engineer might want to use a more complex expression as the output of an instruction or specialize the execution context for the REPL, which could not be done through annotations. One could still choose to modify the behavior of the base semantics by adding a new ALE operation that could then contain any kind of ALE expression, and call it in the annotation. This

```
open class Statement {
    /**
     * right: Turn turtle of 'p' degrees to the right
     * forward: Move turtle of 'p' units forward
     */

    @repl__turtle__toString
    @step
    def void execute(Turtle turtle, SymbolTable st) {
        'ERROR: Unknown statement'.log();
    }
}

open class Expression {
    @repl__output__toString
    @step
    def Value compute(SymbolTable st) {
        'ERROR: Unknown expression'.log();
        result := null;
    }
}
```

Figure 5.4 – ReplDefinition Annotations Used on Logo

new operation could not, however, have access to the global context of the interpreter, nor to the intermediary results.

Based on this information, the DSL specification can be transformed so that a REPL can be derived and used by interactive computer programming environments. It is defined in three steps I) Abstract Syntax Tree transformation, II) Concrete Syntax transformation, III) Operational semantics transformation. The next subsections detail these three transformations applied on the original DSL specification. Finally, we present the generic REPL interface provided and the clients defined as interactive computer programming environments: a language shell and a notebook interface.

5.3.2 Abstract Syntax Transformation

During the DSL specification transformation process, we first complement the abstract syntax with additional concepts.

We first define *Interpreter* as the REPL entry-point. It owns a reference to the abstract class *InterpretableInstruction* which will be set during the execution to always target the current instruction.

InterpretableInstruction also has a containment to itself, which creates a linked list of the previously executed instructions, hence keeping the whole execution history in a single resource. For each instruction *I* defined in the *ReplDefinition* model, the new

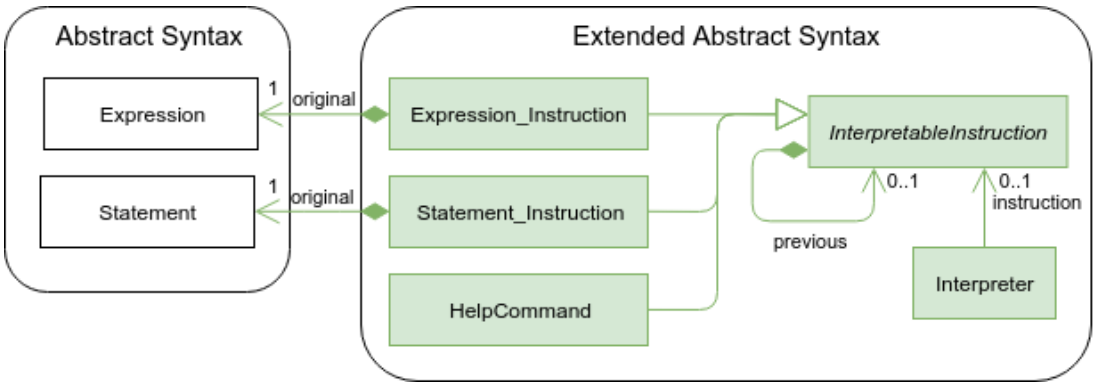

Figure 5.5 – Abstract Syntax Extension for Logo

abstract syntax will include an adapter *I_Instruction* extending *InterpretableInstruction*. Another instruction is the *HelpCommand*.

An example of the additions made for *Logo* can be seen in Fig. 5.5. The instructions that correspond to the new entry-points are *Statement* and *Expression*.

5.3.3 Concrete Syntax Transformation

The second step in our approach is to extend the existing concrete syntax to parse alternatives corresponding to the newly defined instructions. As such, for each instruction *I*, we retrieve the corresponding rule from the base grammar of the DSL and we reuse it for the newly defined adapter *I_Instruction*. The parsing rule *InterpretableInstruction* manages this part.

Another rule is created in order to instantiate an interpreter. Then, the grammar entry-point will be the parsing rule *EntryPoint* that will call either *Interpreter*, *InterpretableInstruction* or *HelpCommand* if the user asks for help on a specific subject.

We also add a custom scope provider in order to resolve the cross references between the previously executed instructions and the current one. When trying to resolve a cross reference, this scope provider will browse through the linked list of the previous instructions. If nothing was found, it will finally turn to the resolution mechanisms of the base grammar.

One of the limitations of this specific implementation is that we use the default Xtext parser to parse single statements. As such, we do not support non context-free grammars. If the language has two semantically different instructions that use the same notation, only one of them can be made into an entry-point. Some possible ways to support non

context-free grammars would be:

— to use a custom parser that could build a context from the previously executed instruction (which might add unwanted side effects)

— or to allow the language designer to modify the keywords used by some grammar rules, in order to remove potential conflicts

Fig. 5.6 depicts an organization of the different artifacts related to the concrete syntax extension of the language *Logo*, while Fig. 5.7 details the corresponding Xtext production rules.

5.3.4 Semantics Transformation

Last, we transform the DSL semantics to incorporate the new execution context and flow management, and to enable the new instructions to be executed.

Here, we handle operational semantics written in ALE. ALE is a language that allows to re-open classes from Ecore metamodels to statically introduce fields and operations at design time. By using the *open class* syntax, we can define the behavior for the classes we added in the syntax, and drive the execution with *@init* and *@main* annotations on operations.

We define the runtime data of the execution context and the initialization of the *Interpreter* entry-point as described in the *ReplDefinition* model. When executed, the interpreter will call the operation `interpret` on the instruction it is currently referencing.

Every instruction adapter takes care of the mapping defined in the *ReplDefinition* model: they become a wrapper that will call the original execution method of the statement or expression, possibly with the right parameters (the interpreter's execution context) and retrieve and display the expected outputs as described in the model.

Fig. 5.8 describes the overall execution flow for the language *Logo*, while Fig. 5.9 shows the generated ALE code corresponding to this execution flow. The Interpreter, its initialization method and specialized execution context are derived from the information provided by the language engineer in the *ReplDefinition* model (see section 5.3.1). For each new entry point, an operation is added to manage the semantics. A new operation is also added to the new *HelpCommand* meta-class.

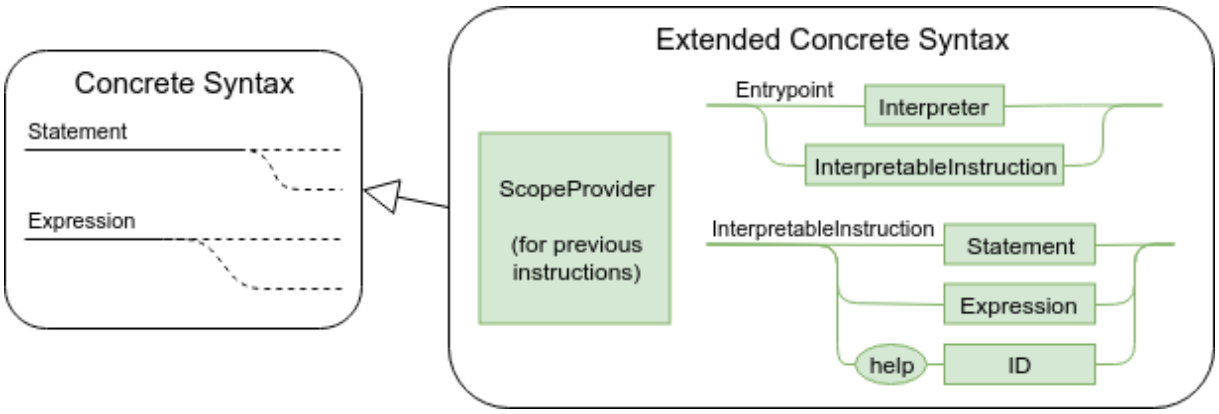

Figure 5.6 – Concrete Syntax Extension for Logo

```
1  // Import existing Logo definition.
2
3  EntryPoint returns ecore::EObject:
4      InterpretableInstruction | Interpreter;
5
6  InterpretableInstruction:
7      {Statement_Instruction} original=Statement
8      | {Expression_Instruction} original=Expression
9      | {HelpCommand} 'help' command=ID;
10
11 Interpreter:
12     {Interpreter}
```

Figure 5.7 – Generated Extended Grammar Definition for Logo

5.3.5 REPL Interface and Interactive Environments Examples

Having applied the aforementioned transformation process, the DSL is complemented with a multi entry points parsing of interpretable instructions. In order to build an interactive environments on top of it, we provide a generic REPL interface protocol and its underlying systematic execution framework (cf. Fig. 5.10):

1. Create an interpreter and initialize it

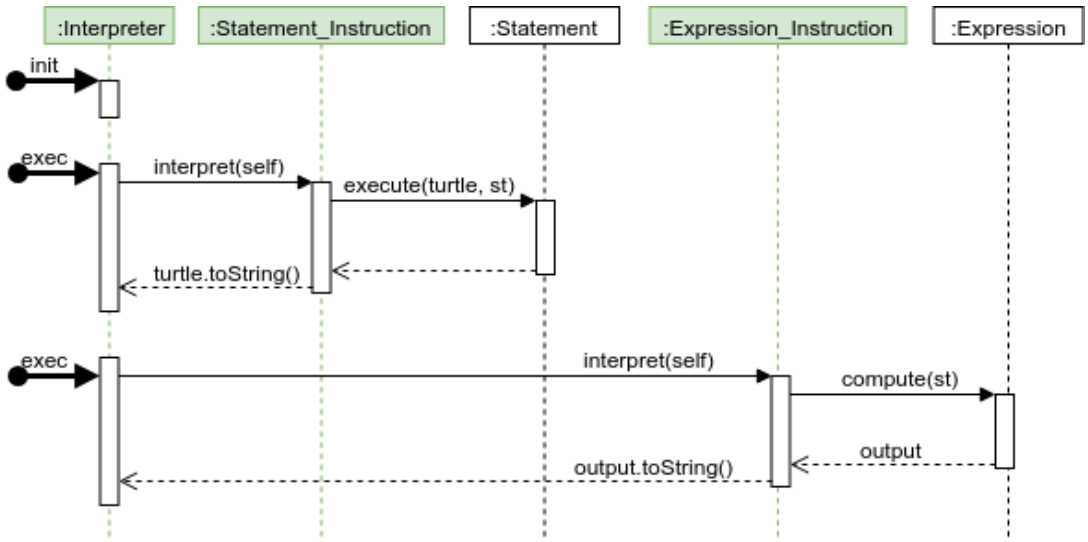

Figure 5.8 – Overall Execution Flow for Logo

2. Read the user input
3. Parse it as an instruction
4. Retrieve the previous instruction and store it
5. Swap the instruction of the interpreter for the new one
6. Run the interpreter
7. Print the relevant output
8. Go back to step 2

From the new generated DSL specification, we automatically generate the entire GLUE code for integration with two technical environments: Eclipse and Jupyter. For the first one, we provide a plugin including an eclipse view hosting a shell to communicate with the Interpreter. This generic view declares an Eclipse extension point type including among others the name of the REPL language and the qualified name of the interpreter class. Each REPL language declares this extension point. The generic view allows REPL users to select the desired DSL and then start an interactive session. This session keeps track of the executed instructions and offers the ability to reset the interpreter or cancel the last instructions thanks to the environment provided by Gemoc. The Language Server Protocol (LSP [2]) support provided by Xtext enables intelligent completion within the

2. https://langserver.org/

Shell. Figure 5.11 shows a screenshot of this integration within Eclipse for the Logo language.

For Jupyter, an editor specific to ipynb files (Jupyter notebook) has been created and manually integrated into Jupyter. The purpose of this editor is to replace the default code cell editor (ace editor[3]) with the monaco text editor[4]. The latter has the advantage of natively supporting LSP in order to allow completion, error reporting, etc. A generic glue code has been added to adapt between Jupyter's Kernel concept and GEMOC's execution engine to control the execution of interpreters. Thus for each new REPL, a connection file defining the connection URL to the GEMOC execution engines of this REPL is generated. A descriptor is also generated for Jupyter to register this new kernel.

On the GEMOC side, a class allowing to interface with a ZeroMQ message oriented middleware is created and makes the link with the execution interface of the GEMOC engine and the current REPL. The main advantages of the GEMOC integration is to leverage its execution trace management, debugging facilities and concurrency model management (e.g., to start from any cell or finely control the flow of execution of the cells.).

5.4 Evaluation

To address the four challenges identified in Section 5.2, we propose an approach to automatically generate an interactive computer programming environment from a DSL specification and an identification of the execution entry points for this REPL. A first level of validation consists in applying our approach on other DSLs, namely *MiniJava* and *ThingML*, and to reflect on the lessons learnt.

MiniJava is a subset of the general purpose language *Java* that was created for teaching purposes, since *Java* was considered too intimidating for students on various aspects [67]. The first implementation of *MiniJava*, released in 2001, was also shipped with a REPL. This DSL offer a good support to learn Java and test APIs as introduced in Section 5.1.

We started from an existing implementation in EMF/Xtext/ALE[5]. This specific implementation is a large one, with 80 meta-classes and 200 attributes in the abstract syntax,

3. Cf. https://ace.c9.io/
4. Cf. https://microsoft.github.io/monaco-editor/
5. Cf. https://github.com/manuelleduc/ale-lang/tree/master/languages/minijava

170 lines of Xtext for the grammar, and more than 1140 lines of ALE as operational semantics. In order to use real life APIs, we decided to add a support for native *Java* calls through the Java JSR-223 API [6]. JSR-223 is a standard API for calling scripting frameworks in Java. It is available since Java 6 and aims at providing a common framework for calling multiple languages from Java.

We selected nine execution entry-points: Type declarations, Method definitions, Statement blocks, Variable declarations, Assignments, For loops, While loops, Conditions, and Expressions. This means that we added nine `@repl` annotations, including one with a specific output for the expressions. Considering the initial size of the DSL specification, these additions are only nine more lines in the semantics, which can be estimated as a modification of 0.6% to generate a REPL for the existing DSL.

We also added both a Xtext *ScopeProvider*, to manage the scoping, and a Xtext *Validator*, to enforce access rights, to the base definition of *MiniJava*. Having these two new elements written with the pure interpreter pattern inside the DSL definition was not an issue, and they both work as intended for the generated REPL.

The second DSL is an ALE implementation of *ThingML*. ThingML [7] is a domain specific modeling language, that combines well-proven software modeling constructs for the design and implementation of distributed reactive systems:

— statecharts and components (aligned with UML) communicating through asynchronous message passing,

— an imperative platform-independent action language,

— specific constructs targeting IoT applications.

The ThingML language is supported by a set of tools, which include editors, transformations (*e.g.,* export to UML) and an advanced multi-platform code generation framework, which supports multiple target programming languages (C, Java, Javascript). Recently a simulator has been designed to emulate the distributed system behavior. The abstract syntax contains 88 meta-classes, for a total of 240 model elements. The grammar definition is more than 450 lines long, and writing the operational semantics in ALE require more than 1800 lines.

Being a dataflow language, it was an interesting case study for us since our approach was mainly aimed at imperative DSLs.

6. Cf. https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/prog_guide/api.html
7. Cf. https://github.com/TelluIoT/ThingML

In a *ThingML* program, a language user can define types and protocols, with some of those types being "things" that can declare functions and state machines, before instantiating them inside configurations. A configuration of connected things is a dataflow, and the operational semantics of *ThingML* execute it for as long as they have steps to execute.

The syntax of the DSL offers a way to describe complete dataflows, and our approach enables the definition of partial programs. However, the concept of partial dataflows is debatable. Since ThingML uses named elements in a configuration, we could use them as valid execution entry-points, but a smaller granularity would not make sense:

— We could have instantiations and connections between things outside of configurations, but this would mean either that we execute a partial flow from the beginning after each change, or that *ThingML* defines an `execute` instruction (and it does not). Running several configurations is honestly just as good.

— The statements used in functions could be available, but at best they could only be used to interact with completely executed configurations (and their instances if the concrete syntax provided a definition for qualified names), which would be of limited use.

— With the two above, we could actually end up with a complete imperative language, but it would be too far away from the original *ThingML* to stay in the scope of what we want to provide.

As such, we decided to use the following 3 entry-points: Type definition, Protocol definition, and Configuration declaration (and execution). They correspond to 3 more lines in the semantics, which represent 0.12% of the total DSL specification. Our *ThingML* REPL makes it possible to split a program between elements definition and several configuration executions. However, there are a lot more interesting aspects to an interactive environment for a dataflow language: altering an already existing dataflow with new nodes and transitions, or controlling the execution step by step for example.

Lessons Learned This experiment on two DSL specifications defined by other language engineers allowed us to verify several points, and to evaluate our initial RQ and associated four challenges.

The first lesson learned is the ability of our approach to be used on different DSLs as long as these specifications conformed to a certain number of expectations. The language

engineer can specify the new entry points of the DSL, the associated outputs and the associated help messages in an expressive way. The effort to define these entry points remains low compared to the level of reuse of the abstract syntax, the grammar and the operational semantics. Such an approach of automated transformation allows a language engineer to have significant confidence in the semantics preservation of the original DSL. Focusing all the tools associated with a DSL only on its specification is an important way to facilitate the evolution of all these artifacts, and in particular the associated REPL. The assumptions made about the form of the implementation of static semantics, the operational semantics, and the different scoping rules are a bit strong. This means that two things are required at the moment: The definition of new entry points must be done by a language engineer and it is required to be able to access the DSL specification in case of issues to correct the parts that do not fully respect the assumption of a pure interpreter design pattern. Finally, if the approach perfectly fits the generation of interactive computer programming environments for imperative languages, many perspectives are opening up in the case of declarative or dataflow languages, and they lead to considering new opportunities for the interactive parts of these kinds of languages.

5.5 Discussion & Perspectives

We described in this chapter an approach to automatically transform an existing specification of a textual and interpreted DSL, into a new specification that drives the development of an interactive computer programming environment. From additional information about the allowed entry points and the expected outputs when executed, we described how to transform the grammar specification and the operational semantics specification so that we can have multiple execution entry points, and a sound and extensible management of the execution context and flow. We also defined a unified interface to be used from different interactive environments such as a language shell and a notebook interface. The implementation and the evaluation have been done in the GEMOC Studio, but the proposed approach could be implemented in other language workbenches.

This approach opens up various perspectives. While our approach is currently expecting operational semantics in the form of an interpreter, we would like to extend it in the future to also cover translational semantics in the form of a compiler. We would also inves-

tigate the support of a seamless interoperability [26] between the interactive computer programming environments and the initial environment. In the long term, we would like to investigate polyglot interactive environments offering a seamless integration of heterogeneous languages.

```
1  open class Interpreter {
2    logolang::Turtle turtle;
3    logolang::SymbolTable st;
4
5    @init
6    def void init() {
7      self.turtle := logolang::Turtle.create();
8      self.turtle.xpos := 0.0;
9      self.turtle.ypos := 0.0;
10     self.turtle.direction := 0.0;
11     self.turtle.pendown := false;
12     self.turtle.canvas := logolang::Canvas.create();
13     self.turtle.canvas.segments := Sequence{};
14     self.st := logolang::SymbolTable.create();
15     self.st.init();
16   }
17
18   @main
19   def void run() {
20     self.instruction.interpret(self);
21   }
22 }
23
24 open class Expression_Instruction {
25   def void interpret(Interpreter logo_repl) {
26     ecore::EObject output := self.original.compute(logo_repl.st);
27     output.toString().log();
28   }
29 }
30
31 open class Statement_Instruction {
32   def void interpret(Interpreter logo_repl) {
33     self.original.execute(logo_repl.turtle, logo_repl.st);
34     logo_repl.turtle.toString().log();
35   }
36 }
37
38 open class HelpCommand {
39   def void interpret(Interpreter logo_repl) {
40     // Call help method of the node
41   }
42 }
```

Figure 5.9 – Generated Extended Operational Semantics for Logo

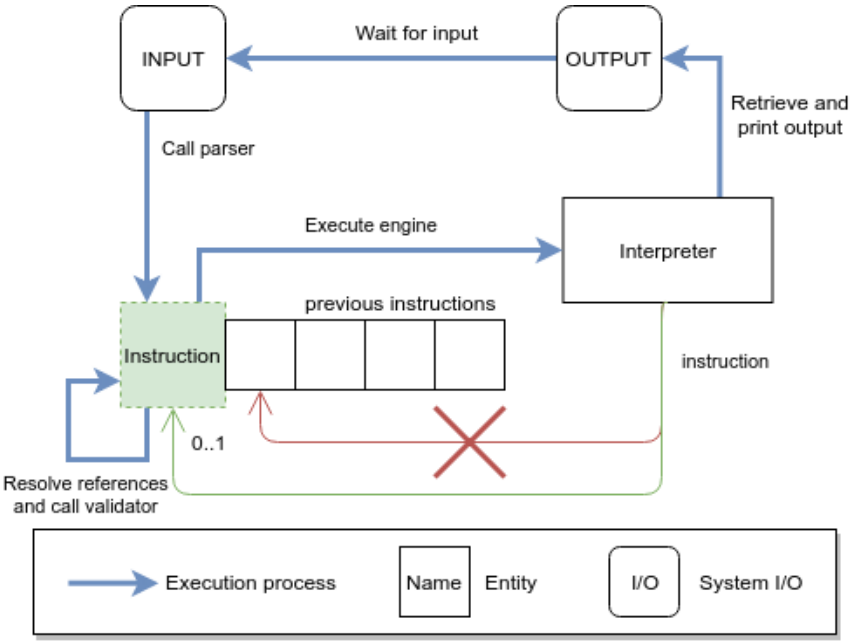

Figure 5.10 – REPL Execution

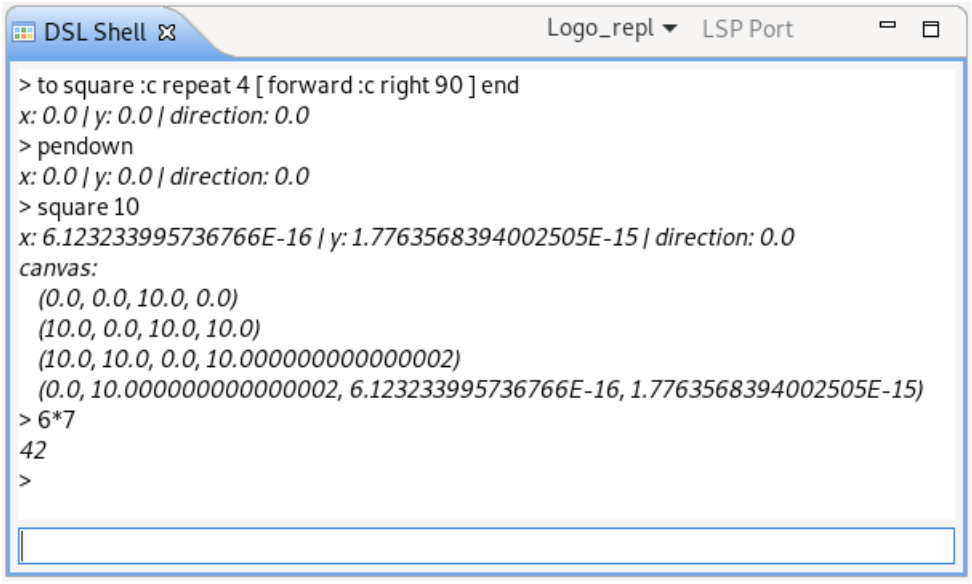

Figure 5.11 – Eclipse Shell Running with Logo REPL

CHAPTER 6

A PROTOCOL FOR DECOUPLING EXECUTION SERVICES FROM LANGUAGE RUNTIMES

In this chapter, we analyze different language execution services in order to organize them into a hierarchy, to leverage on the most minimal language execution component. We start by motivating the importance of a minimal execution component in order to ensure semantics consistency between different execution services (Section 6.1). Next, we detail the protocol to communicate with such a component, and formalize possible implementations (Section 6.2). Then, we evaluate its usability through two environments, one based on a generative approach to obtain compatible operational semantics and the other using operational semantics made specifically for this use-case (Section 6.3.2). Finally, we discuss future perspectives opened by this work (Section 6.4).

6.1 Motivation

The famous vision that "software is eating the world" is getting more realistic every day [4]. The increasing digitalization of education, industry, social and political life is leading to the increasing appearance of digital data and of programs to create, process, understand and visualize these data. A natural corollary to the need to create new programs efficiently is the emergence of many programming languages and the need for a domain expert to build abstractions related to their domain. These abstractions could be built within frameworks for existing programming languages or even better by defining new languages (Domain Specific Languages, or "DSLs") manipulating these abstractions as first class entities. Experts could benefit from specialized tools (*e.g.,* rich text or graphic editor, linter, interpreter) for their abstractions.

To build these specialized tools, the community is looking for a way to define a generic core that can be specialized from a rich specification of these languages. This investigation is obviously a trade-off to be found between a rich core that has limited specializations vs.

a small core with a high degree of specializations. In this trend, approaches such as Debug Adapter Protocol (DAP) [1] or Language Server Protocol (LSP) [2] have made it possible to build great core tools for assisted text editing or for advanced debugging. These two protocols had a real impact on the development of modern Integrated Development Environments (IDEs) but also on language definition workbenches [30, 47].

In the last few decades, the emergence of notebooks [48] (programs mixing documentation, code and output visualization cells) to explore and analyze data has created new needs in terms of debugging, but also in terms of interaction with the execution environment of a language. In order to capitalize on a small core that could be specialized in the definition of a new programming language, this chapter explores the idea of leveraging from a single operational semantics implementation, by formally describing possible implementation patterns to follow in order to benefit from a rich execution environment tooling (such as the integration within a notebook or an advanced debugger within the notebook). We push the idea to revisit the way we implement an operational semantics for a programming language in order to inherit a rich execution environment adapted to an integration in a notebook-like framework. This includes a protocol on top of the semantics implementation to communicate with the rest of the environment. In order to demonstrate the pertinence of our formal models and our protocol, we show that it is possible to implement these rich execution tools, and we propose a generative process to convert existing operational semantics implementations into ones that follow the formalization to prove that it is possible to integrate our approach in the current practices.

The contributions of this chapter include:

— The identification of the minimal set of services required to support three different execution tools: an execution engine, a REPL interpreter and a debugger,

— The specifications of a protocol that bridges operational semantics and execution tools,

— And a formalization of three operational semantics styles that can be accessed through this protocol.

We first motivate in depth the need for our approach in Section 6.1. We then detail how we identified the minimal set of services required in our protocol in Section 6.2. Section 6.3 proves the applicability of this approach by formally describing three operational

1. cf. https://microsoft.github.io/debug-adapter-protocol/
2. cf. https://microsoft.github.io/language-server-protocol/

semantics styles that can support the presented protocol, and presenting two language implementations. Finally, we conclude the chapter with Section 6.4.

To illustrate the process of driving multiple generic execution tools from a single operational semantics implementation, we use *MiniJava*. *MiniJava* is a subset of the *Java* language, first designed for teaching purposes [67]. Contrary to *Java*, this language is usually fully interpreted instead of compiled to bytecode.

Given its teaching nature, we argue that a *MiniJava* environment should offer offer similar tools and features as a *Java* one. If we consider the different ways to run Java programs, this means that such an environment requires at least three execution tools:

— An interpreter for complete programs that produces the same result as compiling an equivalent *.java* source code file using *javac* and running it on a *Java* virtual machine;

— A REPL implementation that behaves similarly to *JShell*, for integration in notebook-like tools for example;

— And a debugger that offers the same execution granularity and information as a JPDA-compliant debugger such as *Eclipse JDT Debug* [3].

Originally, in the *Java* ecosystem, these tools were developed independently, but in practice they share the same language constructs. The main difference between an interpreter executing a complete *Java* program and a *Java* REPL is the entrypoint: in the first implementation, the semantics require finding the main method defined in the class executed at top level, and run it with the input arguments after the definition of all the classes, while in the second one code snippets are interpreted directly and immediate feedback is given to the user. The constructs that can be used at the top-level in the REPL, such as method declarations and expressions, need to be given semantics, *e.g.,* the values of top-level expressions are assigned to fresh variables in JShell. Using an operational semantics for *MiniJava* defined around the concept of entrypoints such as in [39] or [15], it is possible to also support the execution of a complete program in a REPL interpreter as long as it also supports the injection of the input arguments (*e.g.,* through context modification or a specific entrypoint). But these approaches do not offer the necessary granularity for debugging the execution. This would require managing the execution at the level of an “execution step”, as defined in the approach of [17].

Consequently, an operational semantics for *MiniJava* that exposes both entrypoints

3. cf. https://www.eclipse.org/eclipse/debug/index.php

and execution steps could be leveraged to execute a complete program, provide an interactive REPL session, and offer a debugging process without modifications. Importantly, this approach ensures consistency in the behaviors of the language constructs when using the language through the different tools. The goal of this chapter is to derive from these approaches the minimal protocol able to support the different usages, and to formally describe the requirements for operational semantics implementations in order to fit this protocol. In the end, the different tools are driven by a single common core component, which could even be defined as generic and reusable for a family of similar languages since the capabilities of the protocol are fixed.

6.2 A Minimalist Execution Protocol

6.2.1 Services Identification

This contribution expands on the work achieved by Bousse et al. in [17]. Several tools are based on it such as an implementation of an *Execution Engine* for *Henshin* ([84]), a generic omniscient debugger ([18]), or an efficient runtime monitoring approach ([52]). This contribution is sound, but too restrictive to properly implement REPLs to realize exploratory programming styles. It was designed to support the execution of complete programs but not multiple code snippets in the same execution context, *i.e.,* it only offers a single possible entrypoint. Our protocol should thus be more fined-grained, allowing the definition of an *Execution Engine* whilst allowing a multitude of entrypoints.

In this section, we identify the language services necessary to support each execution tool:

Execution Engine An *Execution Engine* is a language-agnostic tool that manages the execution of a complete program, given the language's definition with its operational semantics. Based on the aforementioned work, its interface describes two execution methods:

— **initialize** to load the language definition and the program, and to prepare the execution context,

— **execute** to run the program, which consists in launching a specific method from the operational semantics for a given element.

An Execution Engine can also manage *Engine Addons*, which are execution tools that can observe and alter the execution at any "execution step". Execution steps are explicitly defined in the operational semantics of the language to represent the observable steps of an execution in the context of a specific language. Supporting addons requires the following events:

— **engineStarted** to notify of the start of the execution, and the need to start the addons,

— **engineStopped** to notify of the end and thus clean the addons,

— **aboutToExecuteStep** before the execution of a step, after which the addons should each be handed over control on the execution,

— **stepExecuted** after a step finishes.

This notion of execution step is essential to coordinate the different addons with the runtimes of different languages. The language designer himself defines what an observable execution step should be. A debugger implemented as an addon (for example) could take this information into account to pause the execution on parts that might be of interest to language users.

REPL A REPL runs model elements at a finer granularity than the ones managed by an *Execution Engine*. Another difference is that it is an interactive component, whereas an engine is single-goal driven: running a full program to completion. When they are handed over control of the execution, *Engine Addons* can technically pause it for an arbitrary time, which is how a generic debugger can be implemented on top of an engine. However, in the case of a REPL it is necessary to keep user interactions as much as possible, so an interpreter paused on a step should still be able to receive and interpret code snippets. In [15], the authors also discuss the importance to notify the users of every change in the environment introduced by new executed snippets. This feature requires access to the execution context in order to compute the differences.

As such, the necessary services to implement a REPL can be reduced to:

— **interpret**, a request that can take as input a code snippet and execute it,

— **modified**, an event that notifies the environment that parts of the context was modified, with details on the exact changes.

Debugger Whether the debugging target is an execution engine or a REPL, a debugger needs to have control of the execution at the granularity of an execution step. When paused, the execution stack is visible and all defined symbols are both readable and writable. This can be done through multiple approaches, one being locking the thread of the running program whenever a step is reached. However, like mentioned earlier, this is a bad fit for interactive environments relying on REPLs since it becomes impossible to interpret new code snippets while the execution is locked.

For this reason, we favor an implementation where it is possible to directly manage the execution of steps, which boils down to the following services:

— **step** is a request that executes the next available step of a program,

— **stepEnded** is an event that notifies that a step has ended, which is necessary to support multiple stepping strategies with inner steps,

— **read** is a request that enables fetching values from the execution context,

— **write** is a request to modify context values.

The next subsection describes the specifications for a protocol that supports all the three previously described execution tools as demonstrated by the implementation and the use cases described in the next section. Although not demonstrated by the case studies, our experiments revealed the protocol can also directly support *omniscient* debugging (as described earlier).

6.2.2 Protocol Specification

Considering the nature of the identified services, a protocol to manage them all needs to expose a “stepwise interpreter”, as formally introduced in section 6.3.1.

We define the step data-structure, that will be exchanged between the interpreter and the rest of the environment, with the following:

— a unique ID,

— the location of the corresponding syntax element from the program currently running.

Taking into account the services previously defined, the requirements for the different tools can be factored into the following messages:

interpret is a request that can take as a parameter either a complete program or a code snippet and parse it. It does not start the execution, however. It returns a step data structure corresponding to the very first observable step available.

step is the request that manages the execution by running the semantics until it reaches a new step (since observable steps can be defined recursively). It returns a step data structure corresponding to the newly reached step.

stepEnded is an event notifying that a step has ended, with a reference to that specific step through its ID.

finished is an event indicating that there is no more steps to run, that also specifies the kind of termination, *e.g.,* natural or error-related.

read is a request that returns the current state of observable values stored in the execution context.

write is a request to modify editable context values.

modified is an event notifying that an observable value changed during execution, unrelated to the use of **write**.

To realize an execution engine tool, the **initialize** service connects to the backend corresponding to the right language through the protocol, and use *write* requests to set up the environment. The **execute** service requires setting the program input, which can also be done through *write* requests, before using *interpret* to send the program for stepwise execution. Completing an execute request is then a matter of repeating and managing the execution of steps through *step* requests and *stepEnded* events until a *finished* event is received.

For a REPL, no initialization is required, and the code snippets can be sent freely through *interpret* while managing the completion of steps. By listening to the *modified* event, feedback for the user can easily be generated for each snippet by making use of the *read* request.

And finally, for a debugger, managing the observable execution steps is quite straightforward, and *read*, *write* and *modified* allow to perform any necessary operation on the

execution context (*e.g.,* managing conditional breakpoints logic and manually editing values). As an added benefit, the debugger can be safely separated from the tool managing the execution itself, meaning that an execution engine could be told to no longer loop through the steps and instead give control to a debugger at any point, while also allowing a REPL to run code snippets for *e.g.,* exploratory programming and omniscient debugging purposes.

In order to prove the usability of the protocol in actual scenarios, the next section will describe several operational semantics formalizations and implementations that can expose the necessary services.

6.3 Implementation & Evaluation

6.3.1 Operational Semantics Formalization

Considering the minimalistic nature of the expressed protocol, we are aware that it can be implemented in a lot of different ways. As of now, we do not provide guidelines to obtain the "best" implementation. Each DSL is different, and language engineers are various and used to different technologies and styles of semantics implementations. Evaluating the best implementation for a given DSL is not part of the scope of this thesis, so instead this section describes formally multiple possibilities. One of them, "Procedural style", also comes with the specifications of a generative approach that can be applied to existing operational semantics, but this does not mean that we want to enforce this one in particular. The generative approach should be seen mainly as an applicability evaluation, since we consider that implementing the interface should be a design decision above all else.

In [15], a formal model is developed that characterizes interpreters that naturally exhibit REPL behavior. In this model, interpreters are transition functions over configurations – deterministic transition relations in the style of Plotkin's Structural Operational Semantics (SOS) [64]. The notion of a *sequential language* is defined, capturing the class of languages [4] in which a sequence of programs is itself a valid program such that the effect of the composite program is that of the individual programs executed one-by-one in order. This behavior naturally corresponds to the behavior of a REPL, in which a larger program is developed by repeatedly submitting program fragments. In

4. Not necessarily equal to the class of languages that execute non-concurrently.

addition, the paper proposed a methodology in which, in order to obtain a REPL for some language L, the language L is extended to a sequential variant of L. The method involves selecting the 'entry points' of the language L – the syntactic categories of which program fragments should be accepted at the REPL's prompt – and defining the syntactic category of "phrases" as the union of the entry points. An interpreter (or operational semantics) for L can then be reused to define an interpreter (or operational semantics) for the phrases of L. The interpreter for phrases should express the effects of phrases as modifications to a given configuration (representing runtime state). To obtain a REPL with the desired behavior, any effects of a phrase that affect subsequent phrases should manifest as a modification to the input configuration (*e.g.,* introducing new bindings in an environment or assignments in a store). The result is an extended language L' with phrases as programs and an interpreter that can be used as the basis of a REPL. Here we show that if the interpreter for L is a *stepwise* interpreter, executing or evaluating program fragments step-by-step, then L' automatically gives rise to an L'' supporting the basic features of debugging on top of REPL-style programming.

A stepwise interpreter is defined as follows.

Definition 6.3.1. Given a set of configurations Γ and a set of computations C, a stepwise interpreter is a function $\rightsquigarrow : C \times \Gamma \to C^{\mathbf{done}} \times \Gamma$, with $C^{\mathbf{done}} = C \cup \{\mathbf{done}\}$. The iterative closure of $\rightsquigarrow$ is the function $\dashrightarrow : C^{\mathbf{done}} \times \Gamma \to \Gamma$ defined as:

$$\langle \mathbf{done}, \gamma \rangle \dashrightarrow \gamma \tag{6.1}$$

$$\frac{\langle c_1, \gamma_1 \rangle \rightsquigarrow \langle c_2, \gamma_2 \rangle \quad \langle c_2, \gamma_2 \rangle \dashrightarrow \gamma_3}{\langle c_1, \gamma_1 \rangle \dashrightarrow \gamma_3} \tag{6.2}$$

The iterative closure $\dashrightarrow$ captures the full [5] evaluation of a computation where $\rightsquigarrow$ captures step-by-step evaluation.

In [15], a language is considered to define a set of phrases P, a set of configurations Γ, an 'initial' configuration $\gamma^0 \in \Gamma$ and a (definitional) interpreter in the form of a family of functions I, assigning to each $p \in P$ a function $I_p : \Gamma \to \Gamma$. Consider that our hypothetical language L has m syntactic categories, defined via grammar/production rules, with S_i denoting the set of sentences generated by the i-th category for each $1 \leqslant i \leqslant m$ (making

5. For simplicity, we ignore that computations may diverge, in which case the iterative closure is not well-defined.

no assumptions about whether the sentences are strings, parse trees or otherwise). If the language is extended to the sequential variant L' according to the method proposed in [15], then the categories can be ordered such that there is a $1 \leqslant n \leqslant m$ for which holds that the first n categories are the entry points of the language. That is, the syntax of phrases is defined by the following abstract syntax definition with $s \in S_1 \cup \ldots \cup S_n$:

$$p \in P ::= s \mid p_1; p_2$$

(The alternate $p_1; p_2$ denotes the sequential composition of phrases, following the methodology of [15].)

Next, we are going to assume the existence of a function $sem : S \to C$ that converts sentences into computations expressing the semantics of sentences and a stepwise interpreter for evaluating these computations. Under these assumptions, it is possible to extend L' with the core debugging constructs $\mathbf{step}$, $\mathbf{finish}$, and $\mathbf{debug}(s)$. The syntactic part of this extension is as follows, defining the set of phrases P':

$$p \in P' ::= s \mid p_1; p_2 \mid \mathbf{debug}(s) \mid \mathbf{step} \mid \mathbf{finish}$$

The extended language L'' is then given by the structure $\langle P', \Gamma \times C^{\mathbf{done}}, \langle \gamma^0, \mathbf{done} \rangle, I \rangle$, with the definitional interpreter I_p (for all $p \in P'$) defined by the following equations:

$$I_s(\langle \gamma_1, c \rangle) = \langle \gamma_2, \mathbf{done} \rangle \textbf{ where } \langle sem(s), \gamma_1 \rangle \dashrightarrow \gamma_2 \tag{6.3}$$

$$I_{p_1;p_2}(\langle \gamma_1, c_1 \rangle) = I_{p_2}(I_{p_1}(\langle \gamma_1, c_1 \rangle)) \tag{6.4}$$

$$I_{\mathbf{debug}(s)}(\langle \gamma_1, c_1 \rangle) = \langle \gamma_1, sem(s) \rangle \tag{6.5}$$

$$I_{\mathbf{step}}(\langle \gamma_1, c_1 \rangle) = \langle \gamma_2, c_2 \rangle \textbf{ where } \langle c_1, \gamma_1 \rangle \rightsquigarrow \langle c_2, \gamma_2 \rangle \tag{6.6}$$

$$I_{\mathbf{finish}}(\langle \gamma_1, c_1 \rangle) = \langle \gamma_2, \mathbf{done} \rangle \textbf{ where } \langle c_1, \gamma_1 \rangle \dashrightarrow \gamma_2 \tag{6.7}$$

The main intuition behind these definitions is that the extended language L'' has an additional component in its configurations to keep track of (possibly done or otherwise intermediate) computations. In other words, the evaluation of a phrase can be performed via more than one step, with remaining steps represented by the computation component of the configuration (if any). The computation component corresponds to the step data structure described in the previous section, with the **debug** phrase serving a similar purpose as the **interpret** message of our protocol, readying a program/sentence for future stepwise execution. In the formalizations of the upcoming subsections, the computation

component is a stack of procedure calls, a continuation function or a program term as used in Plotkin's SOS.

Via the approach described here, intermediate computations are also reflected in the *execution graph* underlying the *exploring interpreter* proposed in [15] for the purposes of exploratory programming. An exploring interpreter is a bookkeeping device on top of a definitional interpreter (as defined in [15]) that records the configurations reached by executing phrases with the definitional interpreter in an execution graph. By combining these approaches, omniscient debugging [18, 53] is achieved. Strategies for efficient omniscient debugging with an exploring interpreter are to be investigated in future work.

The connection between the implementations discussed in Section 6.3.2 and the theory developed here is demonstrated next by discussing both procedural (stack-based) stepwise interpreters, functional (continuation-based) stepwise interpreters, and interpreters derived from SOS specifications.

Procedural style

An operational semantics can be given as a collection of procedures (or methods in object-oriented languages) with each procedure expressing the effects of evaluating a code fragment of a particular syntactic category given some (implicit or explicit) context. The MiniJava description of Section 6.3.2 provides an example of such an operational semantics in the ALE meta-language. Executing a program corresponds to calling one of these procedures (depending on the syntactic category to which the program belongs), which may then call further procedures in a mutually-recursive fashion. Such an operational semantics can be explained as an instance of a stepwise interpreter by revealing the call-stack handling the procedure calls and handling all instructions in a procedure's body as if they were procedure calls as well. This is the approach taken in Section 6.3.2 and formalized in the subsequent paragraphs.

In the formalization of the approach, the function sem assigns to every sentence s, the stack of procedure calls that demand execution in order to evaluate s, with all necessary information propagated as part of the context between procedure calls. The translation procedure described in Section 6.3.2 thus forms the basis of sem. If for some hypothetical language the set of possible procedure calls is $\mathcal{P}$ and the context is captured by the set of configurations Γ, then we can define a stepwise interpreter by taking sequences of procedure calls as the computations of the stepwise interpreter, *i.e.,* $C = \mathcal{P}^+$, **done** represents the empty sequence, and $\rightsquigarrow_{\text{STACK}}$ is defined as an instance of $\rightsquigarrow$ as follows

(with $\rho_i \in \mathcal{P}$ and '[' and ']' delimiting sequences):

$$\langle [\rho_1, \ldots, \rho_k], \gamma_1 \rangle \rightsquigarrow_{\text{STACK}} \langle s \cdot [\rho_2, \ldots, c_k], \gamma_2 \rangle \quad \textbf{where } \langle \gamma_2, s \rangle = exec(\rho_1, \gamma_1) \tag{6.8}$$

The situation where $s \cdot [\rho_2, \ldots, \rho_k]$ is the empty stack (*i.e.,* the computation is **done**) arises when $k = 1$ and the procedure call ρ_1 does not add new elements to the stack (s is empty). The situation where $s \cdot [\rho_2, \ldots, \rho_k]$ is the empty stack arises when $k = 1$ and the procedure call ρ_1 does not add new elements to the stack (*i.e.,* s is the empty stack). The relation $\dashrightarrow_{\text{STACK}}$, derived from $\rightsquigarrow_{\text{STACK}}$ through equations 6.1 and 6.2, is operationalized by the following (purely functional) pseudocode:

```
1 loop(stack,context) {
2     if (empty(stack)) {
3         return context;
4     } else {
5         let (c,stack'')       = pop(stack);
6         let (context',stack') = exec(c,context);
7         return loop(stack' ++ stack'',context');
8     }
9 }
```

The implementation described in Section 6.3.2 demonstrates how a procedural operational semantics (in ALE) can be transformed into stack-based, stepwise interpreter. The transformation adds specific types of stack-elements that enable user-control over whether a procedure call halts the current execution and groups certain instructions together to reduce stack activity for efficiency.

Continuation-passing style

An operational semantics can also be given as a collection of (pure) functions in continuation-passing style. In this style, functions receive an additional 'continuation function' as an argument to be called with a return value 'at the end' of a computational step, rather than actually returning the value. The continuation function of a called function thus explicates the next step of the computation following the called function. Continuation-passing style is commonly employed in compiler optimization steps to reduce stack activity through tail recursion. However, it can also be used as a (functional)

alternative to explicit stack-handling, as has been done in functional implementations of generalized parsing [41, 38, 13]. Similarly, the continuation-passing style can be used to define a stepwise interpreter, reminiscent of the procedural, stack-based, solution discussed previously.

In this approach, a continuation function is a function k that given a configuration returns a configuration and either a continuation function or otherwise **done**. A continuation function plays the role of 'computation' in a stepwise interpreter and is (recursively) defined as yielding the subsequent computation to be performed (if any), *i.e.,* C is the set of functions $k : \Gamma \to \Gamma \times C_{\mathbf{done}}$. The relation $\rightsquigarrow_{\text{CPS}}$ defines a stepwise interpreter for continuation-based operational semantics.

$$\langle k_1, \gamma_1 \rangle \rightsquigarrow_{\text{CPS}} \langle k_2, \gamma_2 \rangle \textbf{ where } \langle \gamma_2, k_2 \rangle = k_1(\gamma) \tag{6.9}$$

Figure 6.1 shows (a simplified version of) the QL interpreter discussed further in Section 6.3.2 in which `stepAll` operationalizes the relation $\dashrightarrow_{\text{CPS}}$ derived from $\rightsquigarrow_{\text{CPS}}$. The function `stepAll` is called with the current program (`Form`), user input `inp` and the initial environment. The `step` function (not shown) evaluates the form to produce the initial configuration, which, if not done immediately, contains the context `ctx` and a continuation function of type `K`. As such, `step` implements a combination of sem (turning a form into a continuation function) and a single transition (step) via $\rightsquigarrow_{\text{CPS}}$. The while-loop then repeatedly calls consecutive continuations until the result is a `done` configuration. The result of `stepAll` is the final context.

Since QL programs represent interactive forms, the actual implementation (see Section 6.3.2) runs in the event-loop of the UI, where calling of the `step` function is managed by the breakpoints and step/continue actions of the user.

Small-step SOS style

An operational semantics can also be given as a small-step SOS specification [64] in which a collection of logical inference rules defines a transition relation $\longrightarrow$. In a typical SOS specification, the transition relation is over structures of the form $\langle p, e_1, \ldots e_n \rangle$, with p the program term under evaluation and $e_1, \ldots, e_n$ auxiliary semantic entities (such as environments, stores, control signals) that provide context to the evaluation of p. The transition relation $\longrightarrow$ captures the (small) steps via which the program term evaluates to either a term on which further steps can be taken, a fully evaluated term (often a

```
1 data Conf = conf(Ctx ctx, K cont) | done(Ctx ctx);
2
3 Ctx stepAll(Form f, Input inp, Env env) {
4     Conf c = step(f, inp, env);
5     while (!(c is done)) {
6         c = c.cont(inp, c.ctxt);
7     }
8     return c.ctx;
9 }
```

Figure 6.1 – Executing a QL form `f` for user input `inp` in environment `env` by sequentially executing all steps.

value) or a stuck term (no further steps due to unexpected or undesireable behavior). If the specification is deterministic, then for every combination of program term and semantic entities, there is at most one transition, i.e. the transition relation is a (partial) function.

Under these assumptions, an arbitrary SOS specification giving rise to $\longrightarrow$ defines a stepwise interpreter as follows:

$$\langle p, \langle e_1, \ldots, e_n \rangle \rangle \rightsquigarrow_{\text{sos}} \langle p', \langle e'_1, \ldots, e'_n \rangle \rangle \textbf{ where} \tag{6.10}$$
$$\quad \langle p, e_1, \ldots, e_n \rangle \longrightarrow \langle p', e'_1, \ldots, e'_n \rangle$$
$$\langle p, \langle e_1, \ldots, e_n \rangle \rangle \rightsquigarrow_{\text{sos}} \langle \textbf{done}, \langle e_1, \ldots, e_n \rangle \rangle \textbf{ otherwise} \tag{6.11}$$

For practical reasons it may be desirable to make a difference between stuck program terms and fully evaluated program terms. This can be achieved by syntactically defining the notion of fully evaluated terms (or values) as part of the specification. Alternatively, (attempted) transitions to (or from) stuck terms can be specified to raise errors.

In the component-based semantics approach of [12], a library of reusable fundamental constructs (funcons) is defined to reduce the effort of providing formal operational semantics to programming languages. Both the funcons and the object language are defined in the meta-language CBS. The `funcons-tools` project [14] implements a framework in which so-called 'micro-interpreters' implement the behavior of individual funcons having been generated from their definition written in a modular variant of SOS within CBS. Applying the ideas expressed formally in this section, the `funcons-tools` framework was easily extended to support both incremental (REPL-style) execution of funcon terms and stepwise debugging. The extension of the original interpreter involved in

the order of 50-150 lines of Haskell code, most of which are related to the interactions between the REPL and its user, e.g. reading input, providing feedback in the form of output and by describing the meta-commands under 'help'.

6.3.2 Languages Implementations

This section presents two language implementations that expose the services requires for the protocol, each based on one of the approaches formalized in Section 6.3.1. The two follow different strategies, with the first one following a generative approach to respect the protocol, and the second one being implemented from the start with protocol support in mind.

Generated Stack-based Semantics

Here we evaluate the support of the proposed protocol through a generative approach able to transform existing operational semantics written in ALE [6] into a procedural, stack-based, stepwise interpreter, following the formalization defined in Section 6.3.1.

The supported semantics are of the same nature as the ones used in [39] to generate REPLs from existing DSL specifications:

— operational semantics written following a pure interpreter pattern in an imperative action language,

— and operations corresponding to observable execution steps explicitly annotated.

Such a definition can have multiple observable steps called in the same operation body, which need to be separated in the execution stack. So the general idea is to *cut* operation bodies into smaller operations of three different types:

— a **big-step-start** is an operation that will directly execute a single operation annotated as an observable step,

— a **big-step-end** serves as an indicator that a started step has ended and can also be used to obtain return values following the start call,

— a **small-step** defines every other operation of the operational semantics that is not an observable step.

6. https://gemoc.org/ale-lang/

Since we are cutting operation bodies, we need to explicitly define their context and handle it externally. So additions to the generated semantics include a new runtime class "Context" for every cut operation, as well as a context instantiation method.

Figure 6.2 illustrates this process using the operator "+" defined as part of the *MiniJava* (cf. Section 6.1) ALE semantics:

— The **eval** method of class **Plus** is annotated as an observable step, but more than that, two other step methods are called in its body: **left.eval()** and **right.eval()**. As such, it is necessary to cut the method body around these calls and re-implement the normal execution flow through the execution stack.

— **Plus_Eval_Context** is created to store the context of the cut method. It consists of both the parameters of the method and the variable definitions in its body. The method **newContext** can instantiate this new class and initialize the values for the previous parameters (in this case, **s**).

— Every reference to the parameters and variables definitions are replaced with references to the context.

— The items pushed in the execution stack are typed as big steps for the calls to **eval**, and small steps for the other inner calls that are not observable.

— Managing operations with return values (such as **eval**) requires accessing the context of this operation after it has finished executing. This is why the contexts instantiated in **eval_1** and **eval_3** are also stored in **c** and accessed in the "big-step-end" methods.

In the case of conditionals, such as a traditional if-then-else structural statement with bodies that are blocks of statements, the same cutting needs to be done both around the complete conditional, and around the step calls in the bodies. The reason is that, just like with a call step, we have to make sure that the statements pre-conditional and post-conditional are pushed to the stack to keep the expected control flow. If the step call is in the expression of the conditional, it needs to be extracted with its value stored in the context of the current method, and the corresponding big-step-end needs to fetch this value, test it and push on top of the stack a small-step corresponding to the body to execute. It is not possible to keep loop constructs with this approach, so the transformation rewrites them using conditionals and recursive functions.

This whole process is generalizable to any operational semantics written in ALE and using step annotations, by recursively cutting any method calling a step, then any method

calling a method calling a step, etc.

The execution stack, as shown in Figure 6.3, manages execution items that contain:

— **target**, the target of the original operation,

— **operationName**, the name (post cutting) of the operation to execute,

— **context**, the runtime context corresponding to the operation,

— **type**, the type of operation as defined earlier.

This requires a method lookup mechanism for every target that will end up being pushed in this stack. In our example, we automatically added a method "executeOperationFromName" that takes care of executing the right part of cut operations from their name post cutting.

With this, it is possible to define an operation in the interpreter to manage the execution of a big-step, as shown in Figure 6.4. This operation will start by popping an execution item from the stack and execute it. Then the behavior depends on the type of the execution item currently at the top of the stack: there is a need to execute every small-step until the next big-step, so small-steps are simply run inside a loop; if an operation is a big-step-end, it will also be executed and the environment notified that a step has finished; if the stack is empty, then the environment is notified that the execution is finished.

Figure 6.5 shows a current prototype of a notebook leveraging the protocol to offer a generic debugging framework. As of now, only stepping through cells and displaying the stack of the current cell are available. We can see that it is possible to run cells in any order, even when the debugger is paused on another cell (in this case, the 6th one is run while the 5th one is being debugged). For the REPL, every executed cell offers an output listing the changes made in the context, even if it is still quite naive and would require to be specialized with the concepts of the language.

QL: a DSL Questionnaires

QL is a simple DSL for specifying interactive questionnaires, supporting data entry, computed values, and conditional logic. An example execution of a prototype IDE for QL is shown in Figure 6.6. The left shows a debugger view ("State") with widgets for each field of the questionnaire, affordances for setting break points and stepping through the execution. The area below the debugger view ("Application") shows the actual running questionnaire. The middle shows an editor with an example QL program, inspired by a

```
open class Plus {
    @step
    def IntValue eval(State s) {
        IntValue leftVal := self.left.eval(s);
        IntValue rightVal := self.right.eval(s);
        IntValue resultVal := IntValue.create();
        resultVal.intVal := leftVal.intVal + rightVal.intVal;
        result := resultVal;
    }
}
```

```
open class Plus {
    def Plus_Eval_Context newContext(State s) {
        Plus_Eval_Context c := Plus_Eval_Context.create();
        c.s := s;
        result := c;
    }

    def void eval_1(ExecutionStack s, Context c) {
        Context newContext := self.left.newContext(c.s);
        c.newContext := newContext;
        s.push(self, 'eval_2', c, 'big-step-end');
        s.push(self.left, 'eval_1', newContext, 'big-step-start');
    }

    def void eval_2(ExecutionStack s, Context c) {
        c.leftVal := c.newContext.result;
        s.push(self, 'eval_3', c, 'small-step');
    }

    def void eval_3(ExecutionStack s, Context c) {
        Context newContext := self.right.newContext(c.s);
        c.newContext := newContext;
        s.push(self, 'eval_4', c, 'big-step-end');
        s.push(self.right, 'eval_1', newContext, 'big-step-start');
    }

    def void eval_4(ExecutionStack s, Context c) {
        c.rightVal := c.newContext.result;
        s.push(self, 'eval_5', c, 'small-step);
    }

    def void eval_5(ExecutionStack s, Context c) {
        c.resultVal := IntValue.create();
        c.resultVal.intVal := c.leftVal.intVal + c.rightVal.intVal;
        c.result := resultVal;
    }
}

class Plus_Eval_Context extends Context {
    State s;
    IntValue leftVal;
    IntValue rightVal;
    IntValue resultVal;
    IntValue result;
    Context newContext;
}
```

Figure 6.2 – Example of Operation Cutting for an Existing *Plus* Binary Operator

```
1  class ExecutionItem {
2      EObject target;
3      String operationName;
4      Context context;
5      String type;
6
7      def void execute(ExecutionStack stack) {
8          self.target.executeOperationFromName(self.operationName, stack, self.context);
9      }
10 }
11
12 class ExecutionStack {
13     Sequence(ExecutionItem) items;
14
15     def void init() {
16         self.items := Sequence{};
17     }
18
19     def void push(EObject target, String operationName, Context context, String type) {
20         ExecutionItem item := ExecutionItem.create();
21         item.target := target;
22         item.operationName := operationName;
23         item.context := context;
24         item.type := type;
25         self.items += item;
26     }
27
28     def ExecutionItem peek() {
29         if (self.items->size() > 0) {
30             ExecutionItem item := self.items->last();
31             result := item;
32         } else {
33             result := null;
34         }
35     }
36
37     def ExecutionItem pop() {
38         if (self.items->size() > 0) {
39             ExecutionItem item := self.items->last();
40             self.items -= item;
41             result := item;
42         } else {
43             result := null;
44         }
45     }
46 }
```

Figure 6.3 – Execution Stack Implementation

```
1  open class Interpreter {
2      ExecutionStack s;
3
4      def void init() {
5          self.s := ExecutionStack.create();
6          self.s.init();
7      }
8
9      def void step() {
10         boolean done := false;
11         self.s.pop().execute(self.s);
12         while (not done) {
13             ExecutionStack peeked = self.s.peek();
14             if (peeked = null) {
15                 done := true;
16                 self.notifyFinished();
17             } else if (peeked.type = 'small-step') {
18                 self.s.pop().execute(self.s);
19             } else if (peeked.type = 'big-step-start') {
20                 done := true;
21             } else if (peeked.type = 'big-step-end') {
22                 self.s.pop().execute(self.s);
23                 self.notifyStepEnd();
24             }
25         }
26     }
27 }
```

Figure 6.4 – Big-Step Loop Execution

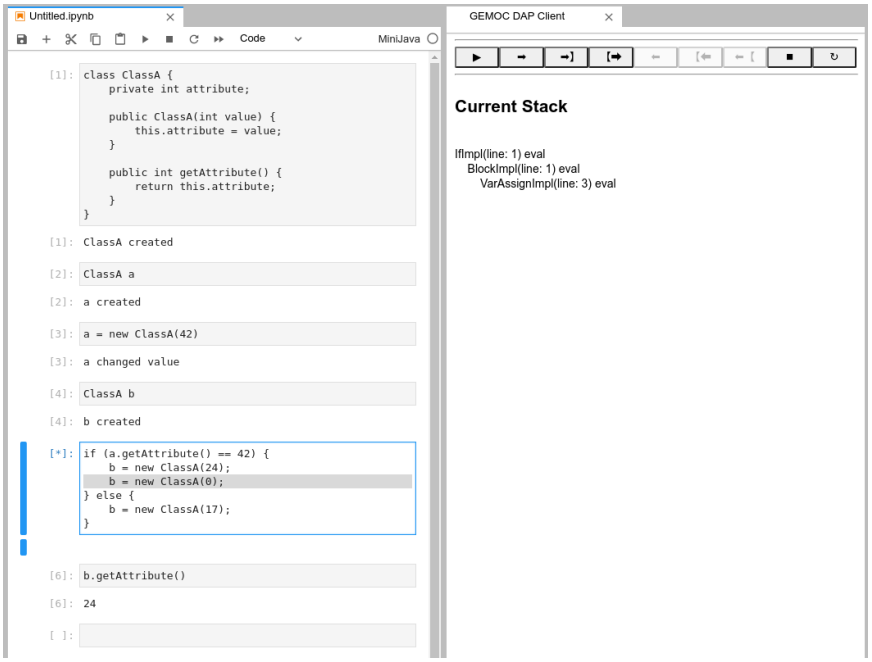

Figure 6.5 – Prototype Generic Notebook with Debugger (Running MiniJava)

tax filing application. Finally, the right shows a REPL for QL as sequential, numbered cells.

The REPL allows developers to enter commands, *e.g.,* for evaluating expressions (1), updating fields (2), or navigating a debug session (4–11). Moreover, edit actions in the center are reflected in the REPL as commands as well, updating the semantic representation of the running application (3) [75, 70], in this case changing the label of the `privateDebt` question. This latter feature supports live programming: the running questionnaire dynamically adapts to source changes without restarting.

The semantics of QL dictates a spreadsheet-like evaluation model: after every interaction of the user with the questionnaire (or modifying the run-time state in the debugger view or REPL), the conditions and values of the computed questions (*e.g.,* `valueResidue` are recomputed in a fixpoint loop. This is required because QL supports "use before define": the value of a question can be referenced before the question is defined in the questionaire. This is the reason that the debugger halts at the `valueResidue` breakpoint twice (cells 8 and 10).

QL is originally implemented as a definitional (big-step) interpreter. However, to support pausing the execution, the fixpoint and conditional logic have been converted into continuation-passing style as described in Section 6.3.1. Unlike the case-study of Section 6.3.2, the interpreter was not automatically derived from the original direct-style interpreter of QL, although the CPS interpreter reuses the evaluator of QL expression sub-language.

The key point of this demonstration is that all interaction with either debugger, code, or running application are reflected and recorded as commands in the REPL. As a result, the REPL is a syntactic and historic representation of interacting with the generic execution protocol.

The QL interpreter complies with the basic protocol of Section 6.2.1, restricted to the part that does not involve call-backs. The basic building blocks are **interpret**, **step**, **read**, and **write**. The first two enable full program evaluation (the running questionnaire, always on in this case) and fine-grained stepping through the evaluation; breakpoints and continue are implemented on top of **step**. The latter two requests deal with state modification, either via the debugger view, or the questionnaire itself.

Furthermore, because at each point the internal data structures (state vector, source code, continuations, etc.) are saved, the REPL becomes a vehicle for "undo" (the button with the up-arrow), essentially integrating versioning and back-in-time debugging in a

unified framework. We leave it as future work to investigate techniques to implement this behavior efficiently in order to scale to bigger and more realistic languages.

These two implementations showcase how a language designer can benefit from implementing, at least partially, the proposed protocol to provide an interactive environment with support for debugging. Even if the shown environments may not fully support some real-life scenarios, providing program execution, a REPL interpreter, and debugging facilities already covers a significant portion of programming activity. Future works still need to take into account more advanced execution tools, and study how they would interact with the protocol. But right now, the method from which we derived the protocol, the formalization of the styles of operational semantics implementations, and the preliminary results shown in this section demonstrate the worth of this approach, and we hope that it will serve as a basis for future work on this topic.

6.4 Discussion

In this chapter, we motivated the need for an execution protocol that all operational semantics should consider in order to support the creation of a generic tool associated with the execution environment of a language. We defined precisely the boundaries of such a protocol, and we promoted a formal model to characterize the behavior of an interpreter supporting it. We highlighted the applicability of the approach to two different languages and the ability to obtain a REPL that can be integrated in a notebook, but also the ability to benefit from a rich debugger for these languages. This work also gives a path to different perspectives. Firstly, as the state of the execution environment is saved, it is natural to consider this protocol as a vehicle to support undo in an execution context within a REPL or a notebook. In this context, it is necessary to think about the efficient implementation of this mechanism. Next, we can also evaluate the addition of optional extension points within this minimal protocol to enable tool specialization. In the community of Software Language Engineering, it exists a permanent quest for designing generic and a highly specializable core for each language services. In this chapter, the idea is to show that it is possible to obtain completely generic tools as long as the operational semantics of a language respect a minimal generic protocol. No doubt that it could be required to specialize some of these tools for a specific domain, it will then be necessary to evaluate the impact on the proposed protocol. Another

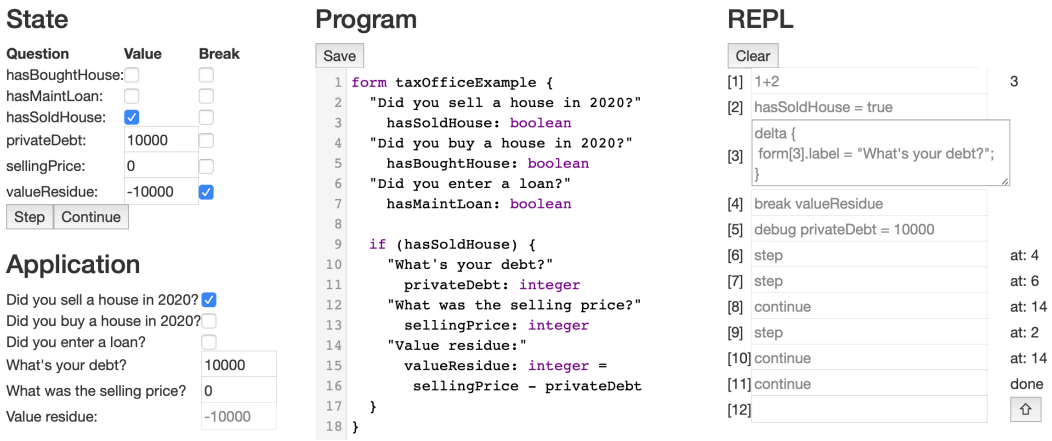

Figure 6.6 – Prototype IDE for QL: run-time view (left), source code (center), REPL (right)

research direction relates to the semantics accessed through this protocol. While the set of exposed services is minimal, they might not all be useful for every scenario. Exploring the concept of self adaptable semantics ([42]) could provide substantial performance gains by, *e.g.*, lowering the granularity of steps when debugging is not necessary, while being invisible to the execution tools that would only interact with the protocol. Finally, it would be interesting to revisit MSOS [57] to evaluate the effort required to implement an MSOS operational semantics specification within an interpreter that respects the protocol proposed in this chapter.

CHAPTER 7

CONCLUSION & PERSPECTIVES

7.1 Conclusion

The adoption of DSLs relies primarily on their integration in the workflows of their target users. With the advent of language workbenches, designing, implementing and maintaining an external DSL has never been as easy as it today. However, the tools provided to the end users end up being highly dependent on a specific environment (*i.e.,* modeling workbench) that provides features similar to the environment they are already used to. At the same time, supporting multiple environments is an arduous process, as illustrated by previous research works mentioning the *IDE portability problem*.

In this thesis, we explored the use of language protocols such as LSP and DAP as a potential solution, and concluded that these approaches were not good fit for the plethora of peculiar features expected from DSLs. As such, we propose the vision of *IDE as Code*, that relies on providing language features as independent services with explicit dependencies between them. For each language service, a sequence of operations can be defined to be executed after the reception of a particular message through a DSL, and the overall communication protocol of the environment can be derived from these specifications. This answers the first challenge mentioned in Section 1.2. Instead of either providing multiple environments for each specific language and require the user to master them all, a fully integrated environment bloated with features they do not use, or a generic environment missing essential specific features, development environments should be customizable to integrate and support only the services required for a specific DSL, both understood and actually used by the target users. In the case of Web IDEs, this vision also enables the deployment of services on multiple machines, and provides the benefits of microservices architectures such as on demand scaling and fast upgrade cycles.

We then applied this ideology to language runtimes. First we defined *REPL Interpreters*, which are essential artifacts to drive new programming paradigms such as literate and exploratory programming, through a generative approach. This relates directly to the

third challenge mentioned in Section 1.2. Then, we researched the most basic interpreter required to support multiple runtime tools such as *Execution Engines* (interpreters supporting the execution of a complete program while providing an interface to connect addons) and *Omniscient Debuggers*. With an interpreter of this sort, exposing a very minimalist protocol, other execution services can be implemented generically, which ensures semantics consistency between all the different execution services. As generic services running on top of this protocol, we considered a generic REPL interpreter, a generic execution engine, and a generic debugger. These generic services should be able to integrate seamlessly in multiple environments as long as they also expose their own protocol. This hierarchy established between language components answers the second challenge mentioned in Section 1.2.

However, the work actually done on the topic of *IDE as Code* is still at an early stage. First of all, while we have proven the advantages of the approach for the subset of runtime components, others have not yet been explored. The advantages of the microservicization of textual editing services has been researched by Coulon et al. in [25], and graphical editing services are also starting to be considered ([7]), but others such as visual representations during animation (like the animation framework presented in [17]) are still to be addressed. The actual integration of all these services in a popular IDE such as *VS Code* would be a valuable contribution to evaluate the overall approach discussed in this manuscript.

In the rest of this chapter, we will list perspectives for future works on this topic which we deem necessary before we can safely envision any adoption.

7.2 Perspectives

Our vision, summarized as *IDE as Code*, places the language protocol specification in the adaptation loop of the IDE, where language services are packaged and deployed dynamically, and provided as new capabilities to the user. From our preliminary explored implementations, we identify and discuss concrete challenges in the following section.

Unified representation of language services specifications As mentioned in Section 3.2, this work done on this topic focused mainly on the specification and deployment of language packages, and their interactions. We do not target a specific representation of the specifications of languages and their services. In Chapter 6, we consider interactions

made through the use of JSON-RPC protocols, but it is still too early to suggest this technology for every language service. At the very least, we require a way to specify the interactions between services completely independent of any specific technology, and a generative approach to derive the language service packages from these. A first step in this direction would be a formalization of the generic concepts offered by multiple data transport technologies, such as JSON-RPC, GraphQL, and REST.

Domain-specific IDEs concepts Chapter 6 proposed a first categorization of IDE-specific concepts, such as workspaces and UI services. While we consider these first concepts sound for full-fledged IDEs, it is not yet clear how traditional UI services translate to other development platforms such as Notebooks, or Terminal UIs like (neo)vi(m) or Emacs. Subsequent work is required to properly categorize these concepts and introduce them as first-class constructs to specify language protocols.

Stateless vs stateful trade-off Microservice architectures efficiently support resources scaling. But in practice, it is unlikely that every language package will be stateless. For instance, a package to manage the workspace definitely requires a concept of session, and possibly access to a database. To support web-based IDEs in practice, it is necessary to provide a way to specify the stored context of the different packages, and still be able to support scaling even with stateful microservices, probably through synchronization mechanisms that language designers should not have to manage.

Automation of the deployment Our vision revolves around an automatic deployment of the "language service packages" and their integration into IDEs, based on properly defined constraints. For "backend" services, this requires either a generative approach, or an interpreter (possibly with an adaptation loop to automate scaling depending on the available resources and the demand) to process the protocol specifications. Further considerations need to be taken into account for UI services, as their deployment would end up being quite specific to the targeted environment.

Measuring the impact on performances A direct consequence of microservicizing the different language services will be a multiplication of the number of messages exchanged to complete any action, and numerous asynchronous waits. Beyond flexibility, scaling and collaboration, the hit in performances could be significant at the level of a fully-featured

IDE. A proper study of the possible impacts of this approach to the user experience is required. It is worth investigating alternatives, such as approaches to automatically compile different language service packages (possibly dynamically) back into bigger components when they are deployed on the same machine. Other technical choices for the serialization format (*e.g.*, XML, protobuf) and the transport layer (*e.g.*, pipes, WebRTC) should also be considered and chosen dynamically depending on the proximity with the deployed services.

BIBLIOGRAPHY

[1] Ibrahim S. Abdullah and Daniel A. Menascé, « The Meta-Protocol framework », *in*: *Journal of Systems and Software* 86.*11* (2013), pp. 2711–2724, ISSN: 0164-1212, DOI: https://doi.org/10.1016/j.jss.2013.05.096, URL: http://www.sciencedirect.com/science/article/pii/S0164121213001386 (cit. on p. 37).

[2] Mathieu Acher, Benoit Combemale, and Philippe Collet, « Metamorphic Domain-Specific Languages: A Journey Into the Shapes of a Language », *in*: *Onward! Essays*, Portland, United States: ACM, Sept. 2014, pp. 243–253, DOI: 10.1145/2661136.2661159, URL: https://hal.inria.fr/hal-01061576 (cit. on p. 75).

[3] Eric Allen, Robert Cartwright, and Brian Stoler, « DrJava: A Lightweight Pedagogic Environment for Java », *in*: *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, SIGCSE'02, Cincinnati, Kentucky: ACM, 2002, pp. 137–141, ISBN: 1-58113-473-8, DOI: 10.1145/563340.563395, URL: http://doi.acm.org/10.1145/563340.563395 (cit. on p. 75).

[4] Marc Andreessen, « Why software is eating the world », *in*: *Wall Street Journal* 20.*2011* (2011), p. C2 (cit. on p. 97).

[5] Andrew W. Appel and Jens Palsberg, *Modern Compiler Implementation in Java*, 2nd, Cambridge University Press, 2003, ISBN: 052182060X (cit. on p. 59).

[6] Bas Basten, Jeroen van den Bos, Mark Hills, Paul Klint, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Tijs van der Storm, and Jurgen Vinju, « Modular language implementation in Rascal – experience report », *in*: *Science of Computer Programming* 114 (2015), pp. 7–19, ISSN: 0167-6423, DOI: https://doi.org/10.1016/j.scico.2015.11.003, URL: http://www.sciencedirect.com/science/article/pii/S0167642315003524 (cit. on pp. 54, 56).

[7] Romain Belafia, Pierre Jeanjean, Olivier Barais, Gurvan Le Guernic, and Benoit Combemale, « From Monolithic to Microservice Architecture: The Case of Extensible and Domain-Specific IDEs », *in*: *MODELS 2021: ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems*, Virtual, Japan:

IEEE, Oct. 2021, pp. 1–10, URL: https://hal.inria.fr/hal-03342678 (cit. on pp. 25, 122).

[8] Federico Bergenti, Eleonora Iotti, Stefania Monica, and Agostino Poggi, « Interaction Protocols in the JADEL Programming Language », *in*: *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2016, Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 11–20, ISBN: 9781450346399, DOI: 10.1145/3001886.3001888, URL: https://doi.org/10.1145/3001886.3001888 (cit. on p. 37).

[9] Valentin Besnard, Matthias Brun, Philippe Dhaussy, Frédéric Jouault, David Olivier, and Ciprian Teodorov, « Towards One Model Interpreter for Both Design and Deployment », *in*: *Proceedings of MODELS 2017 Satellite Event: Workshops (ModComp, ME, EXE, COMMitMDE, MRT, MULTI, GEMOC, MoDeVVa, MDETools, FlexMDE, MDEbug), Posters, Doctoral Symposium, Educator Symposium, ACM Student Research Competition, and Tools and Demonstrations co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, USA, September, 17, 2017*, ed. by Loli Burgueño, Jonathan Corley, Nelly Bencomo, Peter J. Clarke, Philippe Collet, Michalis Famelis, Sudipto Ghosh, Martin Gogolla, Joel Greenyer, Esther Guerra, Sahar Kokaly, Alfonso Pierantonio, Julia Rubin, and Davide Di Ruscio, vol. 2019, CEUR Workshop Proceedings, CEUR-WS.org, 2017, pp. 102–108, URL: http://ceur-ws.org/Vol-2019/exe_4.pdf (cit. on p. 32).

[10] L. Thomas van Binsbergen, *eFLINT implementation on GitLab*, https://gitlab.com/calculemus-flint/eflint-tools/eflint, [Online, accessed 12 October 2020], 2020 (cit. on p. 68).

[11] L. Thomas van Binsbergen, Lu-Chi Liu, Robert van Doesburg, and Tom van Engers, « eFLINT: A Domain-Specific Language for Executable Norm Specifications », *in*: *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2020, ACM, 2020, DOI: 10.1145/3425898.3426958 (cit. on p. 68).

[12] L. Thomas van Binsbergen, Peter D. Mosses, and Neil Sculthorpe, « Executable Component-Based Semantics », *in*: *Journal of Logical and Algebraic Methods in Programming* 103 (Feb. 2019), pp. 184–212, DOI: 10.1016/j.jlamp.2018.12.004 (cit. on p. 110).

[13] L. Thomas van Binsbergen, Elizabeth Scott, and Adrian Johnstone, « Purely functional GLL parsing », *in*: *Journal of Computer Languages* 58 (2020), p. 100945, DOI: 10.1016/j.cola.2020.100945 (cit. on p. 109).

[14] L. Thomas van Binsbergen and Neil Sculthorpe, *The Haskell Funcon Framework*, Hackage, [Online, accessed on the 13th of April 2022], 2018, URL: https://hackage.haskell.org/package/funcons-tools (cit. on p. 110).

[15] L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais, « A Principled Approach to REPL Interpreters », *in*: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2020, Virtual, USA: Association for Computing Machinery, 2020, pp. 84–100, ISBN: 9781450381789, DOI: 10.1145/3426428.3426917, URL: https://doi.org/10.1145/3426428.3426917 (cit. on pp. 25, 30, 47, 52, 99, 101, 104, 105, 106, 107).

[16] Meinte Boersma, *Are language workbenches dead?*, 2017, URL: https://medium.com/@dslmeinte/are-language-workbenches-dead-4b05d1698d3c (cit. on p. 34).

[17] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale, « Execution Framework of the GEMOC Studio (Tool Demo) », *in*: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 84–89, ISBN: 9781450344470, DOI: 10.1145/2997364.2997384, URL: https://doi.org/10.1145/2997364.2997384 (cit. on pp. 18, 33, 81, 99, 100, 122).

[18] Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry, « Omniscient Debugging for Executable DSLs », *in*: *Journal of Systems and Software* 137 (Mar. 2018), pp. 261–288, DOI: 10.1016/j.jss.2017.11.025, URL: https://hal.inria.fr/hal-01662336 (cit. on pp. 15, 16, 23, 42, 100, 107).

[19] Mark van den Brand, B. Cornelissen, Pieter A. Olivier, and Jurgen J. Vinju, « TIDE: A Generic Debugging Framework - Tool Demonstration », *in*: *Electron. Notes Theor. Comput. Sci.* 141.*4* (2005), pp. 161–165, DOI: 10.1016/j.entcs.2005.02.056, URL: https://doi.org/10.1016/j.entcs.2005.02.056 (cit. on p. 34).

[20] Mark van den Brand, Arie van Deursen, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser, « The Asf+Sdf Meta-Environment: a Component-Based Language Development Environment », *in*: *Electron. Notes Theor. Comput. Sci.* 44.*2* (2001), pp. 3–8, DOI: 10.1016/S1571-0661(04)80917-4, URL: https://doi.org/10.1016/S1571-0661(04)80917-4 (cit. on p. 34).

[21] Hendrik Bünder., « Decoupling Language and Editor - The Impact of the Language Server Protocol on Textual Domain-Specific Languages », *in*: *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,* INSTICC, SciTePress, 2019, pp. 131–142, ISBN: 978-989-758-358-2, DOI: 10.5220/0007556301310142 (cit. on pp. 9, 17, 18, 34).

[22] L. Burgy, L. Reveillere, J. Lawall, and G. Muller, « Zebu: A Language-Based Approach for Network Protocol Message Processing », *in*: *IEEE Transactions on Software Engineering* 37.*4* (2011), pp. 575–591 (cit. on p. 37).

[23] João Cangussu, Jens Palsberg, and Vidyut Samanta, *The MiniJava Project*, https://www.cambridge.org/us/features/052182060X, [Online, accessed 12 October 2020], 2002 (cit. on p. 59).

[24] Andrei Chis, Tudor Gîrba, and Oscar Nierstrasz, « The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers », *in*: *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, ed. by Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, vol. 8706, Lecture Notes in Computer Science, Springer, 2014, pp. 102–121, DOI: 10.1007/978-3-319-11245-9FFF_6, URL: https://doi.org/10.1007/978-3-319-11245-9_6 (cit. on p. 35).

[25] Fabien Coulon, Alex Auvolat, Benoit Combemale, Yérom-David Bromberg, François Taïani, Olivier Barais, and Noël Plouzeau, « Modular and Distributed IDE », *in*: *SLE 2020 - 13th ACM SIGPLAN International Conference on Software Language Engineering*, Virtual, United States, Nov. 2020, DOI: 10.1145/3426425.3426947, URL: https://hal.archives-ouvertes.fr/hal-02964806 (cit. on pp. 13, 21, 42, 122).

[26] Fabien Coulon, Thomas Degueule, Tijs Van Der Storm, and Benoit Combemale, « Shape-Diverse DSLs: Languages without Borders (Vision Paper) », *in*: *SLE 2018 - 11th ACM SGIPLAN International Conference on Software Language Engineering*,

Boston, United States: ACM, Nov. 2018, pp. 215–219, DOI: 10.1145/3276604.3276623, URL: https://hal.archives-ouvertes.fr/hal-01889155 (cit. on p. 94).

[27] Peter Deutsch, *PDP-1 LISP*, tech. rep., http://www.bitsavers.org/pdf/mit/rle_pdp1/memos/Deutsch_PDP-1_LISP.pdf, MIT Research Laboratory for Electronics, 1964 (cit. on p. 35).

[28] Sebastian Erdweg, Tijs van der Storm, Markus Volter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriel Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning, « Evaluating and comparing language workbenches: Existing results and benchmarks for the future », *in*: *Computer Languages, Systems & Structures* 44 (2015), pp. 24–47, ISSN: 1477-8424, DOI: 10.1016/j.cl.2015.08.007 (cit. on pp. 30, 55, 64).

[29] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning, « The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge », *in*: *Proceedings of the 6th International Conference on Software Language Engineering (SLE'13)*, 2013, pp. 197–217, DOI: 10.1007/978-3-319-02654-1_11, URL: https://doi.org/10.1007/978-3-319-02654-1_11 (cit. on p. 64).

[30] Moritz Eysholdt and Heiko Behrens, « Xtext: implement your language faster than the quick and dirty way », *in*: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, 2010, pp. 307–309 (cit. on p. 98).

[31] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen, « DrScheme: A pedagogic programming environment for scheme », *in*: *Programming Languages: Implementations, Logics, and Programs*, ed. by Hugh Glaser, Pieter Hartel, and Herbert Kuchen, Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 369–388, ISBN: 978-3-540-69537-0, DOI: 10.1007/BFb0033856 (cit. on p. 75).

[32] Martin Fowler, *Language Workbenches: The Killer-App for Domain Specific Languages?*, 2005, URL: http://martinfowler.com/articles/languageWorkbench.html (cit. on p. 33).

[33] Maria Gouseti, Chiel Peters, and Tijs van der Storm, « Extensible Language Implementation with Object Algebras (Short Paper) », *in*: *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE'14)*, Västerås, Sweden, 2014, pp. 25–28, DOI: 10.1145/2658761.2658765, URL: https://doi.org/10.1145/2658761.2658765 (cit. on p. 54).

[34] Kathryn E. Gray and Matthew Flatt, « ProfessorJ: A Gradual Introduction to Java Through Language Levels », *in*: *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA'03, Anaheim, CA, USA: ACM, 2003, pp. 170–177, ISBN: 1-58113-751-6, DOI: 10.1145/949344.949394, URL: http://doi.acm.org/10.1145/949344.949394 (cit. on p. 75).

[35] Object Management Group, *Semantics of a Foundational Subset for Executable UML Models, v1.5*, 2021, URL: https://www.omg.org/spec/FUML/1.5/About-FUML/ (cit. on p. 32).

[36] Sahar Guermazi, Jérémie Tatibouet, Arnaud Cuccuru, Ed Seidewitz, Saadia Dhouib, and Sébastien Gérard, « Executable Modeling with fUML and Alf in Papyrus: Tooling and Experiments », *in*: *EXE@MoDELS*, 2015 (cit. on p. 33).

[37] Juhana Harmanen and Tommi Mikkonen, « On Polyglot Programming in the Web », *in*: *Modern Software Engineering Methodologies for Mobile and Cloud Environments*, IGI Global, 2016, pp. 102–119, DOI: 10.4018/978-1-4666-9916-8.ch006, URL: https://doi.org/10.4018/978-1-4666-9916-8.ch006 (cit. on pp. 9, 17, 28).

[38] Anastasia Izmaylova, Ali Afroozeh, and Tijs Van Der Storm, « Practical, General Parser Combinators », *in*: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2016, ACM, 2016, pp. 1–12, ISBN: 978-1-4503-4097-7, DOI: 10.1145/2847538.2847539, URL: http://doi.acm.org/10.1145/2847538.2847539 (cit. on p. 109).

[39] Pierre Jeanjean, Benoit Combemale, and Olivier Barais, « From DSL Specification to Interactive Computer Programming Environment », *in*: *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*, SLE

2019, Athens, Greece: Association for Computing Machinery, 2019, pp. 167–178, ISBN: 9781450369817, DOI: 10.1145/3357766.3359540, URL: https://doi.org/10.1145/3357766.3359540 (cit. on pp. 25, 75, 99, 111).

[40] Pierre Jeanjean, Benoit Combemale, and Olivier Barais, « IDE as Code: Reifying Language Protocols as First-Class Citizens », *in*: *14th Innovations in Software Engineering Conference (Formerly Known as India Software Engineering Conference)*, New York, NY, USA: Association for Computing Machinery, 2021, ISBN: 9781450390460, URL: https://doi.org/10.1145/3452383.3452406 (cit. on pp. 25, 39).

[41] Mark Johnson, « Memoization in Top-down Parsing », *in*: *Computational Linguistics* 21.*3* (1995), pp. 405–417, ISSN: 0891-2017 (cit. on p. 109).

[42] Gwendal Jouneaux, Olivier Barais, Benoit Combemale, and Gunter Mussbacher, « Towards Self-Adaptable Languages », *in*: *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2021, Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 97–113, ISBN: 9781450391108, DOI: 10.1145/3486607.3486753, URL: https://doi.org/10.1145/3486607.3486753 (cit. on p. 119).

[43] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson, *Feature-oriented domain analysis (FODA) feasibility study*, tech. rep., Carnegie-Mellon Software Engineering Institute, 1990 (cit. on p. 49).

[44] Sven Keidel, Wulf Pfeiffer, and Sebastian Erdweg, « The IDE portability problem and its solution in Monto », *in*: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, ed. by Tijs van der Storm, Emilie Balland, and Dániel Varró, ACM, 2016, pp. 152–162, URL: http://dl.acm.org/citation.cfm?id=2997368 (cit. on p. 34).

[45] Mary Beth Kery and Brad A Myers, « Exploring exploratory programming », *in*: *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2017, pp. 25–29 (cit. on pp. 15, 23).

[46] Paul Klint, A. Taeke Kooiker, and Jurgen J. Vinju, « Language Parametric Module Management for IDEs », *in*: *Electron. Notes Theor. Comput. Sci.* 203.*2* (2008), pp. 3–19, DOI: 10.1016/j.entcs.2008.03.041, URL: https://doi.org/10.1016/j.entcs.2008.03.041 (cit. on p. 34).

[47] Paul Klint, Tijs van der Storm, and Jurgen Vinju, « Rascal: A Domain Specific Language for Source Code Analysis and Manipulation », *in*: *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, IEEE Computer Society, 2009, pp. 168–177, ISBN: 978-0-7695-3793-1, DOI: 10.1109/SCAM.2009.28 (cit. on pp. 56, 59, 98).

[48] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing, « Jupyter Notebooks – a publishing format for reproducible computational workflows », *in*: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, 2016, pp. 87–90, DOI: 10.3233/978-1-61499-649-1-87 (cit. on pp. 30, 36, 98).

[49] Donald Ervin Knuth, « Literate programming », *in*: *The computer journal* 27.*2* (1984), pp. 97–111 (cit. on pp. 15, 23).

[50] Jan Koehnlein, « Beyond LSP: Getting Your Language into Theia and VS Code », EclipseCon 2020, 2020, URL: https://www.eclipsecon.org/2020/sessions/beyond-lsp-getting-your-language-theia-and-vs-code (cit. on pp. 10, 19).

[51] Yoann Laurent, Reda Bendraou, and Marie-Pierre Gervais, « Executing and Debugging UML Models: An FUML Extension », *in*: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, Coimbra, Portugal: Association for Computing Machinery, 2013, pp. 1095–1102, ISBN: 9781450316569, DOI: 10.1145/2480362.2480569, URL: https://doi.org/10.1145/2480362.2480569 (cit. on p. 33).

[52] Dorian Leroy, Pierre Jeanjean, Erwan Bousse, Manuel Wimmer, and Benoit Combemale, « Runtime Monitoring for Executable DSLs », *in*: *J. Object Technol.* 19.*2* (2020), 6:1–23, DOI: 10.5381/jot.2020.19.2.a6, URL: https://doi.org/10.5381/jot.2020.19.2.a6 (cit. on pp. 26, 100).

[53] Bil Lewis, « Debugging Backwards in Time », *in*: *Computing Research Repository* cs.SE/0310016 (2003), URL: http://arxiv.org/abs/cs/0310016 (cit. on p. 107).

[54] J. McCarthy, R. Brayton, D. Edwards, P. Fox, L. Hodes, D. Luckham, K. Maling, and D. Park S. Russell, *Lisp I programmer's manual*, http://history.siam.org/sup/Fox_1960_LISP.pdf. [Online, accessed 12 October 2020], Computation Center and Research Laboratory of Electronics (MIT), 1960 (cit. on p. 35).

[55] Mauricio Verano Merino, Jurgen J. Vinju, and Tijs van der Storm, « Bacatá: Notebooks for DSLs, Almost for Free », *in*: *Programming Journal* 4.*3* (2020), p. 11, DOI: 10.22152/programming-journal.org/2020/4/11, URL: https://doi.org/10.22152/programming-journal.org/2020/4/11 (cit. on pp. 36, 62).

[56] Marjan Mernik, Jan Heering, and Anthony M. Sloane, « When and How to Develop Domain-specific Languages », *in*: *ACM Computing Surveys* 37.*4* (2005), pp. 316–344, ISSN: 0360-0300, DOI: 10.1145/1118890.1118892 (cit. on p. 55).

[57] Peter D. Mosses, « Modular Structural Operational Semantics », *in*: *Journal of Logic and Algebraic Programming* 60–61 (2004), pp. 195–228, DOI: 10.1016/j.jlap.2004.03.008 (cit. on p. 119).

[58] Gunter Mussbacher, Benoit Combemale, Jörg Kienzle, Silvia Abrahão, Hyacinth Ali, Nelly Bencomo, Márton Búr, Loli Burgueño, Gregor Engels, Pierre Jeanjean, Jean-Marc Jézéquel, Thomas Kühn, Sébastien Mosser, Houari Sahraoui, Eugene Syriani, Dániel Varró, and Martin Weyssow, « Opportunities in intelligent modeling assistance », *in*: *Software and Systems Modeling* 19.*5* (Sept. 2020), pp. 1045–1053, ISSN: 1619-1374, DOI: 10.1007/s10270-020-00814-5, URL: https://doi.org/10.1007/s10270-020-00814-5 (cit. on p. 26).

[59] P. S. Newman, « Towards an Integrated Development Environment », *in*: *IBM Syst. J.* 21.*1* (Mar. 1982), pp. 81–107, ISSN: 0018-8670, DOI: 10.1147/sj.211.0081, URL: https://doi.org/10.1147/sj.211.0081 (cit. on p. 27).

[60] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld, « Example-Based Live Programming for Everyone: Building Language-Agnostic Tools for Live Programming with LSP and GraalVM », *in*: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2020, Virtual, USA: Association for Computing Machinery, 2020, pp. 1–17, ISBN: 9781450381789, DOI: 10.1145/3426428.3426919, URL: https://doi.org/10.1145/3426428.3426919 (cit. on pp. 10, 19).

[61] Bruno C. d. S. Oliveira and William R. Cook, « Extensibility for the Masses - Practical Extensibility with Object Algebras », *in*: *ECOOP 2012 – Object-Oriented Programming*, Springer Berlin Heidelberg, 2012, pp. 2–27 (cit. on p. 54).

[62] Fernando Perez and Brian E. Granger, « IPython: A System for Interactive Scientific Computing », *in*: *Computing in Science and Engineering* 9.*3* (2007), pp. 21–29, DOI: 10.1109/MCSE.2007.53 (cit. on p. 36).

[63] F. Plasil and S. Visnovsky, « Behavior protocols for software components », *in*: *IEEE Transactions on Software Engineering* 28.*11* (2002), pp. 1056–1076 (cit. on p. 36).

[64] Gordon D. Plotkin, « A Structural Approach to Operational Semantics », *in*: *Journal of Logic and Algebraic Programming* 60–61 (2004), Reprint of Technical Report FN-19, DAIMI, Aarhus University, 1981, pp. 17–139, DOI: 10.1016/j.jlap.2004.05.001 (cit. on pp. 104, 109).

[65] Jonas Kjær Rask, Frederik Palludan Madsen, Nick Battle, Hugo Daniel Macedo, and Peter Gorm Larsen, « The Specification Language Server Protocol: A Proposal for Standardised LSP Extensions », *in*: *CoRR* abs/2108.02961 (2021), arXiv: 2108.02961, URL: https://arxiv.org/abs/2108.02961 (cit. on pp. 9, 17).

[66] Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore, « The Evolution of Forth », *in*: *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, Cambridge, Massachusetts, USA: ACM, 1993, pp. 177–199, ISBN: 0897915704, DOI: 10.1145/154766.155369, URL: https://doi.org/10.1145/154766.155369 (cit. on p. 35).

[67] Eric Roberts, « An Overview of MiniJava », *in*: *Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '01, Charlotte, North Carolina, USA: Association for Computing Machinery, 2001, pp. 1–5, ISBN: 1581133294, DOI: 10.1145/364447.364525, URL: https://doi.org/10.1145/364447.364525 (cit. on pp. 90, 99).

[68] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot, « Towards a Language Server Protocol Infrastructure for Graphical Modeling », *in*: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '18, Copenhagen, Denmark: Association for Computing Machinery, 2018, pp. 370–380, ISBN: 9781450349499, DOI: 10.1145/3239372.3239383, URL: https://doi.org/10.1145/3239372.3239383 (cit. on pp. 9, 10, 17, 18).

[69] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej, « How Do Professional Developers Comprehend Software? », *in*: *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, Zurich, Switzerland: IEEE Press, 2012, pp. 255–265, ISBN: 9781467310673 (cit. on pp. 13, 21, 28).

[70] Riemer van Rozen and Tijs van der Storm, « Toward live domain-specific languages – From text differencing to adapting models at run time », *in*: *Softw. Syst. Model.* 18.*1* (2019), pp. 195–212, DOI: 10.1007/s10270-017-0608-7, URL: https://doi.org/10.1007/s10270-017-0608-7 (cit. on p. 117).

[71] Adam Rule, Aurélien Tabard, and James D. Hollan, « Exploration and Explanation in Computational Notebooks », *in*: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, New York, NY, USA: Association for Computing Machinery, 2018, pp. 1–12, ISBN: 9781450356206, URL: https://doi.org/10.1145/3173574.3173606 (cit. on p. 30).

[72] J. C. Shaw, « JOSS: A designer's view of an experimental on-line computing system », *in*: *AFZPS Conference Proceedings, vol. 26, 1964 Fall Joint Computer Conference*, 1964, pp. 455–464 (cit. on pp. 35, 36).

[73] Janet Siegmund, « Program Comprehension: Past, Present, and Future », *in*: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, 2016, pp. 13–20, DOI: 10.1109/SANER.2016.35 (cit. on p. 28).

[74] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro, *EMF: eclipse modeling framework*, Pearson Education, 2008 (cit. on p. 81).

[75] Tijs van der Storm, « Semantic Deltas for Live DSL Environments », *in*: *Proceedings of the 1st International Workshop on Live Programming*, LIVE '13, IEEE, San Francisco, California: IEEE Press, 2013, pp. 35–38, ISBN: 9781467362658, DOI: 10.1109/LIVE.2013.6617347 (cit. on pp. 72, 117).

[76] Walid Taha, « A Gentle Introduction to Multi-stage Programming », *in*: *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*, ed. by Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 30–50, ISBN: 978-3-540-25935-0, DOI: 10.1007/978-3-540-25935-0_3, URL: https://doi.org/10.1007/978-3-540-25935-0_3 (cit. on p. 29).

[77] Steven L. Tanimoto, « A perspective on the evolution of live programming », *in*: *1st International Workshop on Live Programming (LIVE'13)*, IEEE, 2013, pp. 31–34, DOI: 10.1109/LIVE.2013.6617346 (cit. on p. 72).

[78] Warren Teitelman, « History of Interlisp », *in*: *Celebrating the 50th Anniversary of Lisp*, Nashville, Tennessee: ACM, 2008, ISBN: 9781605583839, DOI: 10.1145/1529966.1529971, URL: https://doi.org/10.1145/1529966.1529971 (cit. on p. 35).

[79] Warren Teitelman, « PILOT: A Step Toward Man-Computer Symbiosis », PhD thesis, MIT, 1966, URL: http://hdl.handle.net/1721.1/6905 (cit. on p. 35).

[80] Federico Tomassetti and Marco Torchiano, « An Empirical Assessment of Polyglot-Ism in GitHub », *in*: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, London, England, United Kingdom: Association for Computing Machinery, 2014, ISBN: 9781450324762, DOI: 10.1145/2601248.2601269, URL: https://doi.org/10.1145/2601248.2601269 (cit. on pp. 9, 17).

[81] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney, « How Data Scientists Use Computational Notebooks for Real-Time Collaboration », *in*: *Proc. ACM Hum.-Comput. Interact.* 3.*CSCW* (Nov. 2019), DOI: 10.1145/3359141, URL: https://doi.org/10.1145/3359141 (cit. on p. 30).

[82] Stephen Wolfram, *The Mathematica Book (4th Edition)*, USA: Cambridge University Press, 1999, ISBN: 0521643147 (cit. on p. 36).

[83] Ziv Yaniv, Bradley C Lowekamp, Hans J Johnson, and Richard Beare, « Correction to: SimpleITK Image-Analysis Notebooks: a Collaborative Environment for Education and Reproducible Research », en, *in*: *J Digit Imaging* 32.6 (Dec. 2019), p. 1118 (cit. on p. 30).

[84] Steffen Zschaler, « Adding a HenshinEngine to GEMOC Studio: An experience report », *in*: *Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018*, ed. by Regina Hebig and Thorsten Berger, vol. 2245, CEUR Workshop Proceedings, CEUR-WS.org, 2018, pp. 429–431, URL: http://ceur-ws.org/Vol-2245/gemoc_paper_5.pdf (cit. on p. 100).

UNIVERSITÉ DE RENNES 1

Titre : Environnements de Développement Programmables : Réification des Protocoles de Langage en Objets de Première Classe

Mot clés : Génie logiciel, Générateurs (logiciels), Langages dédiés, Interpréteurs (logiciels)

Résumé : L'utilisation de langages de programmation modernes et complexes nécessite des environnements de développement dédiés, capables d'assister les programmeurs. Les environnements de développement intégrés (IDE) sont les environnements les plus utilisés aujourd'hui, fournissant tous les outils nécessaires pour utiliser efficacement les langages qu'ils ciblent. Fournir un IDE complet pour un langage spécifique est cependant très coûteux, ce qui conduit leurs mainteneurs à ne se concentrer que sur quelques langages pour chaque IDE, divisant ainsi leurs utilisateurs. Afin de rester pertinents, les petits langages tels que les langages dédiés (DSL) doivent être correctement intégrés dans l'environnement de leurs utilisateurs, ce qui nécessite de vastes ressources et ne peut pas prendre en compte la fragmentation entre les IDEs. Dans cette thèse, nous explorons l'idée de déployer des environnements de développement adaptés aux besoins de leurs utilisateurs, et de tirer parti d'outils de langages complètement séparés de tout IDE spécifique. Nous commençons par considérer les protocoles de langage, tels que LSP, et concevons une alternative modulaire et extensible qui correspond mieux aux spécificités des DSLs. Ensuite, nous nous concentrons sur les interpréteurs REPL, des interpréteurs de langage interactifs qui ont une grande valeur pour l'éducation et l'exploration, mais qui ne sont pas facilement dérivés à partir des techniques d'ingénierie des langages existantes. Nous proposons une approche formelle pour définir les REPLs, ainsi qu'une approche générative, et discutons de leur intégration dans les environnements de développement. Enfin, nous unifions la sémantique de plusieurs outils d'exécution (moteurs d'exécution, REPLs et débogueurs) par la spécification du protocole minimal entre eux.

Title: IDE as Code: Reifying Language Protocols as First-Class Citizens

Keywords: Software engineering, Generators (Computer programs), Domain-specific programming languages, Interpreters (Computer programs)

Abstract: The use of modern and complex programming languages requires dedicated development environments to support programmers. Integrated Development Environments (IDEs) are the most used environments today, providing all the necessary tools to use efficiently the languages they target. Providing a complete IDE for a specific language is however very costly, which lead to their maintainers to only focus on a few languages for each IDE, splitting their users. In order to stay relevant, small languages such as domain-specific languages (DSLs) need to be properly integrated in the environment of their users, which requires vast resources and cannot scale to the fragmentation of IDEs. In this thesis, we explore the idea of deploying development environments customized to the

needs of their users, and leveraging language tools completely separated from any specific IDE. We start by considering language protocols, such as LSP, and designing a modular and extensible alternative that is more in line with the specificities of DSLs. Then, we focus on REPL interpreters, interactive language interpreters which offer great value for education and exploration but are not easily derived from existing language engineering techniques. We provide a formal approach to define REPLs, as well as a generative approach, and discuss their integration in development environments. Finally, we unify the semantics of several execution tools (execution engines, REPLs and debuggers) through the specification of the minimal protocol between them.