WIP: Domain Specific Debugging by using RunStaR

Ryana Karaki, Ludovic Marti, Julien Deantoni
Université Côte d'Azur
CNRS/I3S/INRIA Kairos
Sophia Antipolis, France
Emails: (Ryana.Karaki | Ludovic.Marti)@inria.fr, Julien.Deantoni@univ-cotedazur.fr

***Abstract*—The following work in progress aims to assist language designers by giving them the ability to specify the domain decoration of concrete program representations while debugging, thanks to the RunStaR domain-specific language. Two addons are developed alongside in the Gemoc Studio, exploring the possibilities offered by the Xtext and KLighD frameworks.**

I. Introduction

Nowadays, writing a program is no more restricted to computer scientists and there are more and more stakeholders aiming to write specific programs in many fields, such as business, mathematics, or physics to name a few. General Purpose programming Languages (GPLs) are usually inappropriate for them, being too far from their domain of expertise. Usually, such stakeholders are using languages that are tailored syntactically and semantically to their domain of use; *i.e.,* they use Domain Specific Languages (DSLs). To help in the creation of DSL, Language Workbenches (LW) are tools that make the development of new languages and their IDE affordable, *i.e.,* where the creation of language tooling (*e.g.,* editor, parser, checker, interpreter, debugger) is partially automated, typically by using a tooled metalanguage [1]. Debuggers are a crucial part of the tooling and help the user to understand the runtime state of its program during its execution. However, they are supported only by few LWs [2], [3], [4], [5]; and consequently require extra work with few support of the LW.

The proposed debug services, when provided by LW, are often generic and not related to the concrete syntax defined by the DSL, *e.g.,* current values of variables are shown in a distinct view without customisable support. Thereby, the debugger displays information the same way no matter the DSL and these representations are orthogonal to the concrete syntax of the DSL. When compared to dedicated debuggers of GPLs, we can often notice that several interesting features are missing, specially the ones that render debugging information directly as annotation of the concrete syntax. For instance, during a debug session, the IntelliJ IDEA [6] IDE adds the current value of data in non-invasive comments at the currently executed line; also the Eclipse Java Development environment allows inspecting data in a program by hovering it directly in the program editor. It is also frequent to see helpful graphical animation of diagrams for graphical languages [7], [8]. However, all these services are encoded without real support of LWs.

To make these services available as a service of the LW, we study the use of a new meta language that defines how the concrete representation of a program should be annotated to represent the runtime state of the program. By using our meta language, a language designer can define the representations of the runtime state that fits best its language. Our meta language is tooled so that the representations are automatically applied to the concrete representation(s) of a program during a debugging session. We experimented this approach based on the debugger offered by the Gemoc Studio [3] and realized an ongoing integration with two different concrete syntax technologies: textual concrete syntaxes written in Xtext [1] and graphical concrete syntaxes written with KLighD [9].

II. Proposition

Since language designers define domain specific languages, the proposed approach offers them the ability to specify the domain specific way the runtime state of a program should be represented, directly on the concrete representation of a program, during a debugging session. A language designer can thus decide what are the available representations in the generated IDE to help a user of its language to understand what is happening during the execution of a program. For instance, considering a language that defines the concept of `Variable`, the language designer can decide to put focus on it when its current value changed. Finally, depending on the framework used to specify the concrete syntax of its language, the language designer may be more precise and decide to display, for instance, the current value as a comment in the text, or in a hover as a graph of its evolution along the ten last values.

We explored the realization of this goal by defining a new DSL, named RunStaR, to specify the representations of the runtime state of a program during its execution; and by developing an application that, given a program and a specification in RunStaR, modifies the representation of a program during an execution. To be able to be used on different DSLs, we need RunStaR and the application to be independent of the DSL under development. Also, to ease the adoption of our approach with existing languages, the proposed approach should not require to modify the original

[1] http://eclipse.org/xtext

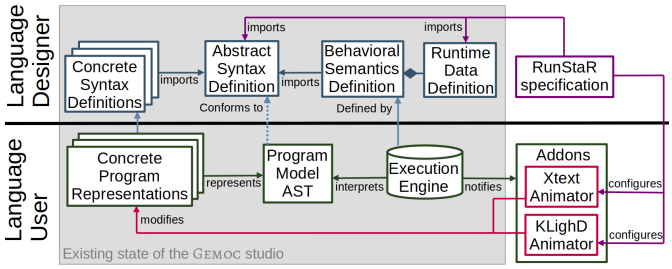

Fig. 1. Integration of our approach into the GEMOC Studio LW

definition of the DSL (at least not manually or in a non-revertible way).

From a technological point of view, we integrated our approach with the GEMOC studio LW [3]. It provides support for the definition of the semantics for languages written in the Eclipse Modeling Framework and consequently integrates well with different concrete syntax technologies. Additionally, it proposes an *addon* mechanism allowing an application to register to the execution of a program and to be notified each time a step is realized. Concerning the concrete syntax technologies, we chose to work with a textual one, Xtext and a graphical one, KLighD. We consequently defined two distinct GEMOC addons, both configured with a RUNSTAR specification.

A. RUNSTAR, *a DSL to define the runtime state representation of a program*

In the context of GEMOC Studio, the language designer defines explicit artifacts for the definitions of the *abstract syntax* and the *runtime data*. These artifacts need to be imported in a RUNSTAR specification to be able to explicitly refer to the runtime data and their context of definition, as shown in Figure 1.

A `RUNSTAR` specification contains a list of runtime `Variable` which specifies *what* property of a concept from the language under development is tracked and *when* the update of its value should occur. For instance, to track the `currentState` of a `StateMachine` concept each time the `update` event occurs, a language designer can write :

```
let theCS be StateMachine::currentState when update
```

The same variable can have multiple `Representations`, such as text or JavaScript code inside a hover, and/or a dynamic label next to your graphical representation. For instance:

```
create Hover on theCS
create Label on theCS
```

Representations can be customized as wished by the language designer, For instance with colors, images, comments or shapes in order to create the most appropriate runtime representation. Default decoration parameters are used if they are not specified.

B. Current Tooling Status

Xtext proposes many customizable features to enhance your DSL such as syntax coloring, highlighting and hovers. After investigating how these features work, it became possible to cherry-pick parts of their behavior and create the right components at the right time. For instance, RUNSTAR hovers are customized SWT widgets. We used these workarounds to avoir modifying the existing DSL definition. Instead we provision our components at runtime, when a debug session starts, and remove them once finished.

The KLighD framework provides a graphical representation of the program [9]. The graphical representation is itself a model, and it is possible to modify it at runtime. Additionally, KLighD gives access to most of the user interface functionalities, allowing many modifications of an existing representation. These modifications can be directly done by navigating through the established model and by adding or changing the representations of language concepts. Many possibilities have already been implemented, but more may come with new case studies.

the RUNSTAR development is still work in progress but the language definition and the GEMOc addons are already available here: https://gitlab.inria.fr/kairos/gemocbackends.

III. CONCLUSIONS AND FUTURE WORK

This paper presents a DSL named RUNSTAR that allows a language designer to specify the representation of the runtime state of a program during a debug session. This is done by giving modification intentions, which are used to configure technology-specific addons, which in turn modifies the concrete representation of a program. This is currently a work in progress, but early demos are convincing. Many future works are envisioned, for instance to allow registering to external services like graph or time series representations.

REFERENCES

[1] S. Erdweg, T. Van Der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh *et al.*, "The state of the art in language workbenches," in *International Conference on Software Language Engineering*. Springer, 2013, pp. 197–217.

[2] A. Chiş, T. Gîrba, and O. Nierstrasz, "The moldable debugger: A framework for developing domain-specific debuggers," in *Software Language Engineering*, B. Combemale, D. J. Pearce, O. Barais, and J. J. Vinju, Eds. Cham: Springer International Publishing, 2014, pp. 102–121.

[3] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale, "Execution framework of the GEMOC studio (tool demo)," in *Proc. ACM SIGPLAN Int'l Conference on Software Language Engineering (SLE'16)*, 2016, pp. 84–89.

[4] E. Bousse, D. Leroy, B. Combemale, M. Wimmer, and B. Baudry, "Omniscient Debugging for Executable DSLs," pp. 261–288, 2018. [Online]. Available: https://hal.inria.fr/hal-01662336/document

[5] D. Pavletic, S. A. Raza, M. Voelter, B. Kolb, and T. Kehrer, "Extensible debuggers for extensible languages," *GI/ACM WS on Software Reengineering*, 2013.

[6] "IntelliJ IDEA," https://www.jetbrains.com/idea/, JetBrains.

[7] "Yakindu Statechart Tools," https://www.itemis.com/en/yakindu/state-machine/, Itemis.

[8] S. Prochnow, "Efficient development of complex statecharts," Ph.D. dissertation, 2008.

[9] C. Schneider, M. Spönemann, and R. von Hanxleden, "Just model!—putting automatic synthesis of node-link-diagrams into practice," in *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. IEEE, 2013, pp. 75–82.