Abstract—Software development in academia is usually done by master or Ph.D. students. These students stay on a project for a very limited period of time. As a result, the software development process can become an instance of the lava flow anti-pattern. In such a scenario, researchers have to rely on half-finished tools propagating their bugs through the toolchain until a step is reached where further extension becomes infeasible.

Based on experience with Bauhaus, we will share our approach of facing the lava-flow phenomenon in toolchain-development and explain why it diminishes the risks of development stall.

I. INTRODUCTION

The Lava Flow pattern commonly¹ refers to a development style where code is created and maintained for a short period of time. The resulting pieces of code become eternal truths afterwards which will cause the surrounding ecosystem to either forget about their existence immediately or to rely on them blindly. In this paper we will share our experience with the development of toolchains in a lava-flow style.

In academia, this phenomenon is usually caused by many people contributing to a common research project over a limited period of time. This is especially a problem if no maintainer is responsible for the overall code base of the research project. People working on a project for a limited period of time is usually a dictate of the environment, such as rules concerning the maximal timespan for the completion of a master or Ph.D. thesis. Thus, a solution has to be found that allows such a system to grow without eventually running into a development stall caused by lava flow.

This paper is written on the assumption that a maintainer who is reading through all changes on a master branch does not exist. This can be the case because either the code is too complicated to judge its quality or because there are no resources to pay for a maintainer at all. Further, we assume that code is split into a toolchain of many tools, each of which was created and maintained by a very small number of developers.

We will present findings obtained from about two decades of development of the Bauhaus toolchain[2]. These findings led to a toolchain design methodology that relies on a novel specification and serialization mechanism called SKiLL². This methodology is an attempt of turning the good parts of the Bauhaus development methodology into a general concept.

We believe that the problem can be mitigated by creating more and smaller tools. Our very concrete approach involves strict modularization of tools produced by individuals, using formal and, more importantly, complete specification of input and output. In order to get researchers to follow this common advice, we provide them with an easy-to-use specification language. If the specification of interfaces of a tool provides a short-term benefit, there is a realistic chance that people will create one. The incentives of our approach are twofold: The specification will provide the developer with a free choice of programming language – something that, in our experience, is especially motivating for master students. This is the case because bachelor students usually mastered only the language they were taught while Ph.D. students will adapt to whatever is used in a project.

Furthermore, the specification is used to generate an API for each tool. Therefore, the interaction with the remaining parts of the toolchain is limited to data structures determined to be important by the provider of the specification.

II. EXAMINATION OF SUB-PATTERNS

We will now discuss three categories of situations that lead to loss of productivity. Also, we will examine the influence of these findings on our enhanced tool-building methodology.

A. The Quitter

An interesting observation is that developers seem to break their tools when running out of time. Two reasons for this behaviour were observed. The first one is that people are simply running out of time and leave the project in a state where they were halfway through a larger change in their code base. The second very frequently observed reason is the understandable desire to collect the final data required for their thesis. These developers are sometimes willing to oversimplify the implementation of their own model to make it work on the selected test data. The second reason is rare among PhD students, but rather common with master theses. Especially master students tend to cover their tracks by removing comments, test and versioning.

If a tool is developed by a single person and this person leaves the project, it becomes dead code. If there is no safe way to fork such a tool, the only reasonable way to deal with it is to either accept shortcomings of the implementation or to throw it away entirely. This is especially true for the second category, because developers tend to leave their code undocumented or to stop updating the documentation when fixing a bug. The latter results in code that contradicts its documentation – a maintenance nightmare.

¹A good introduction is given by [1]. Please note that they claim that lava flow will only work for "small-scale throwaway prototypes" (see p. 52).
²A brief introduction to SKiLL is given in [3], a more complete and technical one, including an extensive related work section, can be found in [4].
We observed that nearly 40% of the tools in the Bauhaus repository consist of a single source file only. This came to us as a surprise, because the current intermediate representation (IR) is not tailored for this kind of usage. In contrast, our serialization mechanism will encourage the creation of throw-away debugging tools. These tools should be collected separately from the main branch.

In consequence, we decided to make it easy to implement tools that do simple things. This will speed up the implementation of tools for debugging or statistics. As an example, consider a tool counting all analysed source lines. Although this may look like an oversimplified example, it is a very real one. There are at least three Bauhaus tools that can be used for this purpose. With SKiIL, it can be realized as easily writing a line of Scala saying: `println(SkillState.read(path) .Sourcelines.size)`

We expect that easing implementation of small tools will actually increase the amount of lava flow. This will allow people to develop small debugging tools, e.g. to create a dot file for a small subset of data in a file in order to get a visual representation. To stay in the lava flow picture, the stream will become more liquid and that is a good thing because it will allow everyone to get further in less time.

B. The Bypass

We found various forms of hidden communication paths. They relate to the lava flow problem because the presence or creation of files in central directories can be missed easily. This can lead a user to the conclusion that a tool is not usable any more because no one tells him that, for instance, a simple configuration file is missing.

In Bauhaus, this problem seems to be caused by two reasons. The first is lack of training, i.e. new developers do not know how to extend the IR for their purpose. On the other hand, creation of small text files and parsing them is considered easy. The second reason is that most specification changes will break the file format. Thus, old research data would be lost. Hence, we designed SKiIL to be usable without further training and we put a lot of effort into the creation of a change-tolerant file format. The format ensures that old data will be readable after arbitrary changes and that it will remain usable in the toolchain if changes do not threaten consistency.

Another aspect of bypasses in our IR are so-called built-in types. They can be used to create type definitions that are not visible for the specification and that use handwritten serialization code. The introduction of these types was obviously caused by the lack of expressiveness of the specification language. As a consequence, we introduced container types to SKiIL and made sure that all definitions of built-in types in our IR can be expressed by semantically equivalent SKiIL definitions.

C. The Black Box

If unmaintained code exists for some time, there will eventually be a situation where the internals of a piece of code are completely unknown. The complete loss of knowledge eventually creates myths. Myths lead to patterns that just work for the moment and create suboptimal ways of obtaining data from the dead part of a toolchain. This is a huge problem because it causes a self-accelerating loss of code quality.

The problem of black boxes can be mitigated by sufficient documentation and a clean structure. As mentioned before, SKiIL will provide a user with an API. In the SKiIL specification language, comments are first-class citizens. That way, the generated API will contain comments of the specification. Having an inaccessible intermediate representation with scattered documentation is obviously nurturing myths. If, on the other hand, the developers’ thoughts can be accessed by hovering the mouse over an API call in an IDE, this will obviously inhibit creation of myths and increase accessibility. Foremost, the stability ensured by SKiIL binding generators enables updates of specifications to be made without breaking client code. This will encourage users to store their knowledge in the common specification of data structures.

Furthermore, SKiIL empowers us to break down a toolchain into many small tools doing logically small steps. In such an environment, rewriting a single tool is not a problem. In fact, occasional rewriting can improve the overall behaviour of a toolchain. A rewrite can make use of algorithms and libraries that did not exist at the time of the tool’s initial creation. Also, changes in hardware architecture or a desire for a different representation of the output can be incorporated.

III. Related Work

Our analysis is motivated by the lava flow anti-pattern as described in [1]. While we share their insights and agree with them with respect to software developing companies, we try to provide a solution that can be applied by academic researchers.

In the course of project Bauhaus, similar attempts of minimizing the negative impact of omnipresent lava flow have been made. In summary, our solution can be seen as a second iteration taking issues into account that have accumulated since the last review of the IR. To the best of our knowledge, no papers have been published regarding lava-flow in academia.

IV. Conclusion

We explained lava-flow based toolchain development from the perspective of our own toolchain. In the future, we will report on the migration of Bauhaus from IML to SKiIL with a special focus on its influence on the overall toolchain development pace. A report on experience with SKiIL-based Bauhaus extensions will be part of a future publication.

References