The Liquid Computing Paradigm

Thomas Gabor and Matthias Hölzl

Ludwig-Maximilians-Universität München

Rapid technological progress by makers of computing hardware and the increased flexibility that software-intensive systems provide im comparison to purely hardware-based solutions has led to system designs in which software plays a more and more prominent role. This results in an increase in software complexity which is exacerbated by the ability of software to "absorb" real-life problems on top of more-or-less commodity hardware: Software plays such an integral part in the construction of a safe vehicle, a modern production line, or armed drones, that the tasks of driving a car, building a toy, or waging war have, to a large degree, become software problems.

As of now, this all too often means that when building a complex software project, developers are facing the worst of two worlds: They have to deal with the uncertainty, tolerances and fuzziness of the real world using the strict, precise and often brittle techniques of digital logic and software. The hardware of a self-driving car is designed with well-defined tolerances and margins of errors based on physical laws; the effects of weakening or aging parts on the system are understood by engineers. The software of the car should interact smoothly and appropriately with the physical world, but it has to achieve this based on a purely logical setting.

A common approach to this need to be "soft on the outside, but hard on the inside" is to attempt to gain more control over the software's internal behavior. This includes the usage of formal methods in the design of adaptive systems (as, among others, considered by the ASCENS project and discussed in Wirsing et al. (2013), e.g.) in order to verify its correctness. The appeal of formal methods is that they allow developers to only be concerned with a simpler model of their problem. However, the difficulty can strike back when translating the formal model to the basic behavior available in hardware anyway.

Another interesting approach is imposing restrictions on the software's abilities , e.g., in swarm intelligence, and benefit from the behavior emerging from interactions of multiple simple components. However, so far there seems to be no simple method to design a swarm that reliably performs complex tasks.

On an interesting note, while the intricate abilities of hardware to interact with its environment and other complex systems has increased drastically in the last few years, the programming techniques and languages have not held up at becoming more powerful.[1] This leads to a sort of "modeling gap" where the translation between models and reality becomes increasingly expensive. And while it is clear that nothing short of an artificial intelligence will be able to deliver programmers from "going all the way", i.e. implementing all models on concrete hardware, in this paper we propose a paradigm for developing collective adaptive systems (CAS) that aims to aid developers at automating or delegating low-level tasks more easily.

Continuity in Development and Adaptation

The first of two principles we make use of to tackle CAS design originates from the observation that for very complex systems in very complex environments, traditional methods of software testing tend to become bland. This is especially clear when we expect a complex system to adapt to unforeseen changes, which we cannot fully test for by definition. Likewise, new circumstances the system finds itself in often come with the need for drastic changes to its behavior, which—from a software engineering perspective—we would like to run through our test cycles. But since the system is online at that point, it is the adaptation process that needs to ensure the viability of the changes made and thus effectively performs software tests. In the long run, it is clear that we need to regard development and adaptation as two parts of a single process: We guide the development of a system until it meets certain "shipping requirements" and then the system develops itself a little more in response to its experience. On both sides of this process it only seems natural to employ the benefits of the respective other side: On the one hand, we may want to use a system's self-adaptation capabilities during human-driven development to aid the programmer; on the other hand, we may dynamically send external help (in the form of a human expert or an online cat-

[1] By "more powerful", we mean the ability to express a wider range of low-level behavior with less amount of high-level code.

alog of suggestions, e.g.) to aid a stuck system at runtime, when the system controls itself for the most part. Both of these courses of action should be included in the standard development process of a CAS. We include both under the term "adaptation" and think of a CAS as a program that continuously adapts from a very unspecific basis to more elaborate environments.[2]

In Hölzl and Gabor (2015), we have already proposed a development pattern called *continous collaboration*. It enables developers to control large amounts of agents indirectly via a designated subgroup of the agents called *teachers*: They are usually equipped with some plan of actions (or any plan-generating algorithm) and attempt to spread their plans among the other agents called *students*, who then choose the plan from the teacher they trust the most and execute it. Thus, instead of explicitly programming a specific collective behavior for all agents, the developers can include a variety of plans in the system and have the trust mechanic of the students adapt the students' behavior adaptively. By including mutation and recombination of plans (alongside the selective pressure induced by the need to gain the students' trust to spread plans), this pattern gives rise to an evolutionary mechanic. Developers can adjust the dynamic of that process by changing the teachers and their respective plans. However, all included plans are subjected to the evolutionary process and thus need to "fight" for their actual execution. Since students are able to choose their teachers individually, the evolutionary process is also presumed to be able to support the specialization of students towards certain teachers and thus towards a specific type of behavior.

Putting these results into a bigger perspective, we can envision to not only use this design pattern to specialize components in specific CAS but also to allow several only loosely related CAS to communicate acquired (meta-) knowledge whenever they can do so meaningfully, thus possibly extending the ability to self-adapt of every involved CAS beyond the scope of computational power accessible to that CAS alone.

Eventually, it seems both necessary and promising to blend the border between design time and run time of a CAS: Both represent a semi-autonomous learning process where humans may take some part in but can less and less be expected to fully grasp the whole structure of the CAS. The key improvement arising from that change of perspective is the pervasive use of automated learning and integration of expert knowledge throughout the whole lifetime of the CAS.

Spontaneity in Synthesis and Compilation

However, as already stated, we are far from achieving a general learning algorithm that can tackle a wide range meaningful problems on its own. Instead, we need to employ human intelligence in order to program complex algorithms, which is still far more efficient than making a machine reinvent the wheel. We can, however, attempt to augment human code writing abilities via automation, as discussed above. Adaptive learning techniques may deviate from the intended behavior to a certain extent. However, current programming languages and paradigms lack an easy way to describe such relationships between pieces of code.

The approach proposed here can be summed up as "software artifacts as first-class citizens" in that it includes a control system for different versions of and dependencies between pieces of code into the program code. That means that we expect our programming environment to allows us to explicitly state relationships between software artifacts and ideally also to infer additional relationships automatically. For instance, we might mark several optimizations of an abstract algorithm as having "equal behavior" so that the run time system knows they can be exchanged on the fly and may choose the most efficient one for the current dataset. Other such relationships may include "is compiled from", "implements interface of", or "has evolved from". A similar endeavor was already deemed necessary by Nierstrasz et al. (2008): They suggest that historical relations between software artifacts should be modeled explicitly in the program code using a suitable programming language and runtime environment.[3] These should enable software developers to more easily grasp the intertwined dependencies between several artifacts and to analyze the history of a specific code passage. Consequentially, this also means that large parts of the software project's previous versions need to be included in the shipped package, again resulting in a blend between design time and run time of a CAS, which is necessary to proactively develop so-called *eternal systems*, which are software systems that are never really shut down but continually adapted and adjusted to new circumstances by a wide range of developers over a long range of time. So far, a few of these systems have occurred "naturally" (like the Internet, e.g.) but almost none have been developed with their eternity in mind.

To go even further, we can again consider how multiple of these systems might interact. As argued by Nierstrasz et al. (2008), the language and platform of an eternal system must themselves be eternal. The same holds for the libraries and other tools used. It is only a small step to think about an "eternal software ecosystem", where libraries, code snippets or development environments evolve independently but can be brought together when needed and adapt accordingly (by, e.g., choosing suitable versions of the delivered package of variants). We can also augment this concept by not only including different versions with regard to the historic development of the software but also provide different versions with regard to the platform, the language, the level of opti-

[2]After all, what is classical software development but a system of a few programmers and empty text files adapting to get paid?

[3]These, however, are in large parts still to be developed.

mization or some properties of the environment. Effectively, every action performed in traditional software development results in a new artifact linked to the previous ones. This approach can then engulf the whole process from adding new models, writing new code to implement previous models, linking external resources and even compiling (parts of) the project to machine code.

Liquid Computing

From the introduction of these two future directions in the development of CAS, it is clear that the progress made in each direction can be helped by results in the respective other area. More precisely, evolutionary learning techniques provide a powerful tool to automated software adaptation but need an ample database of different solutions (i.e., a population of code) and a meaningful way to integrate several competitive solutions into a software project (i.e., augment without overwriting). In turn, a software platform fit for eternal systems is in need for easy and automated ways to generate promising new artifacts, test new additions and choose between different version. These can be provided by using an online evolutionary mechanic as discussed by Karafotias et al. (2011), i.e. execute an evolutionary algorithm while its current population of solution candidates is used to actively solve the problem at hand and use the results from deploying the solution candidates to guide the evolutionary process.

As the combination of the presented approaches results in the dissolution of the border between design and run time and enables software projects to grow and mix dynamically, we call the resulting development paradigm *Liquid Computing*.[4] This technique attempts to support the development of CAS via the pervasive use of automated learning techniques in the form of evolutionary algorithms, explicit expression of the history of the software project and its artifacts, and a global platform, which makes it easy to re-use the knowledge gained by already existing CAS. The development process of Liquid Computing is less centered on reaching a fixed set of requirements, but on endowing software with the ability to grow alongside a dynamic environment and adapt to ever-changing challenges. To achieve this, we give up on the human developers exercising total control over a software's behavior, which is not a realistic concept for large systems anyway, and instead leave room for machine learning to work on improving and adjusting the code. To sum it up briefly, we envision the future of software development to be less like architecture, but more like gardening.

Next Steps and Related Work

In the long run, it is obviously necessary to test the approach presented in this paper in a wide range of software products. However, before this is possible, we need to implement the features presented here in a *Liquid Computing Language*, which among other things needs to include a notion to designate singular artifacts and state the relations between them. Furthermore, we demand an intuitive way to write partial programs and to tell the learning algorithms running the background to fill in the gaps. As already hinted at in the text, it might be desirable to expand said language to a full-fledged development environment, supporting every single step in software development and being connected to a global backend for inter-system information exchange. The effectiveness of established and new learning patterns must be tested in this specific context; as must the viability and scope of relations between software artifacts.

Beyond the inspiring papers mentioned in the text, the usefulness of explicitly defining an artifact-centered network structure for software development has been analyzed by Seichter et al. (2010). In Hölzl and Gabor (2015), we have already raised the question how learning-based and thus inherently more fuzzy development approaches can be integrated with formal verification in fruitful way. Furthermore, just as complex software systems seem to absorb various problem domains, complex development schemes seem to absorb various areas of computing: Thus, the connections between an approach like Liquid Computing and topics like evolutionary game theory, guided self-organization and distributed learning are still to be researched.

[4]Furthermore, the approach was inspired by *liquid democracy* as a method to dynamically integrate a wide variety of actors of originally differing interests.

References

Hölzl, M. and Gabor, T. (2015). Continuous collaboration: A case study on the development of an adaptive cyber-physical system. In *Proc. of the International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS), Firenze, Italy*. to appear.

Karafotias, G., Haasdijk, E., and Eiben, A. E. (2011). An algorithm for distributed on-line, on-board evolutionary robotics. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 171–178, New York. ACM.

Nierstrasz, O., Denker, M., Gîrba, T., Lienhard, A., and Röthlisberger, D. (2008). Change-enabled software systems. In Wirsing, M., Banâtre, J.-P., Hölzl, M., and Rauschmayer, A., editors, *Software-Intensive Systems and New Computing Paradigms*, volume 5380 of *Lecture Notes in Computer Science*, pages 64–79. Springer Berlin Heidelberg.

Seichter, D., Dhungana, D., Pleuss, A., and Hauptmann, B. (2010). Knowledge management in software ecosystems: Software artefacts as first-class citizens. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pages 119–126, New York. ACM.

Wirsing, M., Hölzl, M., Tribastone, M., and Zambonelli, F. (2013). Ascens: Engineering autonomic service-component ensembles. In Beckert, B., Damiani, F., de Boer, F. S., and Bonsangue, M. M., editors, *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 1–24. Springer Berlin Heidelberg.