Sygaldry: DMI Components First and Foremost

Travis J West, Stéphane Huot, Marcelo M. Wanderley

Sygaldry: DMI Components First and Foremost

Travis J. West
Input Devices and Music Interaction Laboratory (IDMIL), Centre for Interdisciplinary Research in Music Media and Technology (CIRMMT), McGill University, Montréal, Canada, and Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
travis.west@mail.mcgill.ca

Stephane Huot
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Marcelo M. Wanderley
Input Devices and Music Interaction Laboratory (IDMIL), Centre for Interdisciplinary Research in Music Media and Technology (CIRMMT), McGill University, Montréal, Canada

ABSTRACT

Motivated by challenges involved in the long-term maintenance of digital musical instruments, the frustrating problem of glue code, and the inherent complexity of evaluating new instruments, we developed Sygaldry, a C++20 library of digital musical instrument components. By emphasising the development of components first and foremost, and through use of C++20 language features, strict management of dependencies, and literate programming, Sygaldry provides immediate benefits to rapid prototyping, maintenance, and replication of DMIs, encourages code portability and code-reuse, and reduces the burden of glue-code in DMI firmware. Recognising that there still remains significant future work, we discuss the advantages of focusing development and research on DMI components rather than individual DMIs, and argue that a modern C++ library is among the most appropriate realisations of these efforts.

Author Keywords

DMI development, maintainability, longevity, replication, reuse, components, evaluation

CCS Concepts

•**Applied computing** → **Sound and music computing;** Performing arts;

1. WHY WE NEED DMI COMPONENTS

1.1 Clear and Generalisable Evaluation

The NIME community has its roots in the field of human-computer interaction (HCI), and the importance of evaluation is inherited from that heritage. Recent discussions have, however, called into question the extent to which evaluation can be a useful goal in NIME research, the forms NIME evaluation may take, and the kinds of contributions that evaluation can generate.

Wanderley and Orio [1] establish (if not explicitly, then by implication) that the aims of evaluation are to enable the incremental improvement of our designs, the development of a body of empirically validated design insights that can inform such improvements, and to realise the scientific progression of this design research. If, for the sake of discussion, we assume these goals (which does not preclude us from holding other simultaneous and even conflicting aims, or from recognising the Western/colonial ideological assumptions present in these goals) we are confronted with immediate challenges. Wanderley and Orio ask: "How can we rate the usability of an input device if the only available tests were done by few—possibly one—expert and motivated performers? [...] How do we evaluate an input device without taking into account a specific aesthetic context?".

Rodger et al [2] offer a logical and perhaps inevitable response. They recognise that, given a specific musical instrument's interrelationship with deeply complex, even chaotic, socio-musical ecologies, it is wildly unreasonable to expect empirical observations of anything in particular about a specific musical instrument in a specific ecology to have any hope of generalising scientifically to another instrument situated in its own specific ecology.

We can extend this further. We recognise that the basic materials of computer music technology are deeply modular, interconnected, and network-like [3], [4]. Without even considering the entangled socio-musical ecology in which it is inextricably situated, we see that a musical instrument unto itself is already a deeply complex interconnected assemblage of complex components. Should we expect anything that we learn by evaluating the T-Stick [5], a rigid cylindrical bar with an array of capacitive touch strips, one magnetic-inertial sensor, and a position-insensitive pressure sensor, to generalise to the Sponge [6], a deformable rectangular prism with discrete buttons, two inertial sensors, and pressure sensors in two specific locations? And that's considering only the sensors. To what extent should an evaluation using the T-Stick played with one synthesiser (e.g. [7]) be reasonably expected to provide general insight when the same T-Stick is later played with a completely different synthesiser (e.g. [8])?

Rodger et al emphasise considering the whole instrument-musician-ecology system, with all of its specificities, when considering evaluation. "Any complex system is not reducible to its parts and the tightrope between precise analy-

NIME'24, 4–6 September, Utrecht, The Netherlands.

sis and distorted representation is ever present." They argue that evaluations' goals and methods should be as specific as these systems. They further argue that reductive evaluation methods, such as an evaluation of system latency, may not always be relevant to the specificities of a system, and that in some cases goals other than evaluation may be more appropriate.

We rather emphasise the usefulness of the reductive approach. Although it does not erase the importance of considering specificities, there are definite benefits to deconstructing the complexity of a DMI into simpler parts and evaluating these components. This is what naturally goes on when we study sound synthesis, sensors, or mapping strategies, especially when we consider them without a particular application in mind. There are two important advantages to evaluation studies facing reusable components of DMIs rather than assemblages of components in the context of fully realised instruments: clarity and generalisability.

First, there is relative freedom from the combinatorial complexity that arises when numerous components are interconnected. This means that evaluation of a component is likely to have clearer targets than evaluation of an assemblage of components. For example, if we ask "what makes one magnetic-inertial sensor fusion algorithm better than another?" we can readily bring to mind clear and unobjectionable engineering factors that could be evaluated and compared, such as accuracy, computational performance, code size, and runtime latency. It is comparatively opaque what should be evaluated when asking "what makes one T-Stick better than another?"

Secondly, evaluations facing components are directly as generalisable as the components are reusable. While the implications of changes to a given set of engineering factors must still be considered on a case-by-case basis within the scope of the specificities of a particular design project, it is reasonable to expect that information about these factors gained through evaluation may be useful to many such projects. Evaluation studies facing components can thus provide benefit to all designs that potentially make use of those components. For example, because magnetic-inertial motion sensors are used in many DMIs, studies facing these sensors and their related algorithms and mappings can generate empirically validated insights into the properties of these components, with the potential to incrementally improve many DMIs using these sensors, or at least inform the decision of the appropriate components for a given project based on its unique aims. When a component implements a functionality that is broadly reusable across numerous DMI designs, evaluation studies of such a component are broadly generalisable to the same extent that the component is reusable.

1.2 Better Maintainability and Replicability

Furthermore, it's very obvious that there are numerous components of DMIs that can very directly be implemented as reusable software components. The T-Stick consists of a button, an analog force sensor, a capacitive touch sensor, and a motion sensor. The mubone [9] consists of several buttons and a motion sensor. There's no particular reason why these two instruments should not share the same software components implementing the necessary procedure for reading buttons and motion sensors, and performing motion sensor fusion.

If there were a shared library of such components, or an agreed-upon convention for compatibility between independent libraries of components, this could drastically reduce the burden of DMI firmware maintenance by favoring reuse; e.g. rather than maintain two wholly seperate implementations for the mubone and T-Stick, with 0 code reuse, these two projects could overlap by a significant amount. 100% of the mubone's sensors are used by the T-Stick, so the former's firmware could be essentially a subset of the latter. If many instruments were implemented by merely combining components, then the maintenance burden of all of these instruments could be consolidated in the shared library of components. Similarly, efforts required to replicate one DMI on a new hardware platform could generalise to all DMIs making use of the library, consolidating the effort required to port and replicate instruments into the library.

This potential is in stark contrast with the observed reality of ongoing DMI maintenance projects. In the maintenance lifecycle of the T-Stick, multiple complete rewrites of the firmware have taken place, as seen in its firmware archives[1]. Similarly, in the lifecycle of the mubone the firmware of the instrument has been rewritten from scratch twice. Despite their significant functional overlap as DMIs, these projects have zero shared code in common beyond the non-DMI-specific Arduino libraries on which they are based. In our experience developing and maintaining DMIs, reuse at the level of DMI-specific functionality is not prevalent, and there is significant and wasteful repeated effort within and across DMI development and maintenance efforts.

A component-oriented development style also has the possibility to be significantly declarative. Whereas Arduino code is highly procedural (e.g. "To read a button: set up the pins. Read the pins. Send data over OSC."), firmware based on high-level components could simply consist of a declaration of those components used in the firmware (e.g. "To make a T-Stick: use a button, a force sensor, a touch sensor, and a motion sensor."). A high-level declarative style such as this would favor reuse and replicability, potentially further reducing code size and thus improving maintainability.

2. WHY DMI COMPONENTS SHOULD BE IMPLEMENTED IN MODERN C++

2.1 DMI Development is Heterogeneous

If there are such clear benefits to maintenance, replication, and research impact to be derived from having a shared library of reusable DMI components, why has the research community not already galvanised around a common library? Among numerous other factors, it is challenging to write code that is sufficiently portable as to be useful to all concerned developers.

DMIs are most commonly implemented using a combination of heterogeneous computing platforms and coding languages, such as a microcontroller unit (MCU) that acquires sensor data and communicates with a laptop computer that runs mappings and media synthesis. C++, especially using Arduino, is found ubiquitously in MCU firmware development for DMIs. However, each different hardware platform tends to introduce its own unique APIs. Without very careful attention to portability, even when using a hardware-abstraction layer such as Arduino, it is very difficult to write truly portable code.

Software developed to run on the laptop is even more diverse in terms of development environments. Max/MSP, Pure Data, Supercollider, ChuCK, Faust, Unity, various digital audio workstations, Touch Designer, Python, the

[1] `https://github.com/IDMIL/T-Stick-Archive`

Web, and various other languages, environments, and platforms are common. A library developed on any one of these platforms is almost always mutually incompatible with any other. For example, a Max/MSP library of DMI components, such as the Digital Orchestra Toolbox, cannot be used in Pure Data or any other environment.

2.2 Tyranny of Glue

Almost all of the environments just enumerated have a low-level API in C or C++ that allows the basic system to be extended through compiled plugins, and on embedded platforms it is entirely possible that a substantial amount of core DMI functionality could be implemented in a way that is hardware independent. Herein lies a potential solution to the problem of portability: a DMI component could be implemented in C or C++ once, and then wrapped/glued to the API for each language and hardware platform, thus enabling the components to be reused across all these environments. Unfortunately, this only leads to a different problem. If glue code must be written manually, as is generally the case, then the amount of glue code grows quadratically. For M components and N environments, there must be written and maintained $M * N$ wrappers [10].

Some languages, notably Faust, resolve this problem by enabling the glue code to be generated automatically. In Faust, an architecture file is used to automatically generate the glue code to wrap a given processor for a certain environment. Thus only N architectures must be written and maintained instead of $M * N$ wrappers, which are generated automatically. However, Faust is explicitly geared towards sample-oriented sound synthesis. It is not appropriate for the development of hardware-specific components such as sensor acquisition, nor for sensor fusion and other mappings, and generally any DMI components not related to synthesising and processing audio.

As of C++20, it is possible to implement automatic glue code generators in pure C++. Celerier [10] describes a reflectable component design pattern that enables this. Although a full review of this design pattern is outside the scope of this article, we note that Celerier's approach provides best-possible type-safety, runtime performance, and code size, and components can be implemented using only plain portable C++ without including any library, and is highly readable and easy to implement. It is arguably the most portable possible component design pattern, as it enables binding classes (analogous to Faust architectures) to generate glue code automatically, such that a single component implementation can be automatically wrapped for a variety of runtime and hardware environments. Wherever modern C++ is supported, this pattern enables a component to be ported without having to maintain quadratic glue code.

Because the underlying language is plain C++, all kinds of components can be implemented, including low-level hardware components, mappings, and sound synthesis and processing. Although Celerier describes the pattern in the context of sound processors being bound to different DAW plugin APIs, it is highly relevant for DMI all other parts of a DMI. For example, a component such as sensor fusion for magnetic-inertial sensors could be implemented once and automatically glued to various MCU hardware platforms, Max/MSP, Pure Data, other interpreted languages, and anywhere else such a component may be useful. We argue that these affordances, along with its portable ubiquity, make modern C++ the most appropriate language upon which to build shared DMI components.

3. SYGALDRY

Sygaldry is a new C++20 library meant as a proposal and provocation for a way to organise DMI development to provide better research impact, maintainability, replicability, code-reuse, and freedom from glue and boilerplate. It implements a shared library of functionally complete DMI firmware components, providing the benefits described above. It suggests one form that such components could take. Several important characteristics differentiate Sygaldry: a component-oriented architecture [11], use of C++20 compile-time reflection and metaprogramming [10] to reduce glue code and enable high-level declarative reuse of components, explicit management and delineation of portable vs platform-specific code to further favor reuse as hardware changes, and use of literate programming practices to enhance documentation [12].

Firmware for a DMI implemented in Sygaldry consists of a declarative high-level specification of the components used by the instrument, such as the following functionally complete implementation of a T-Stick in less than 15 substantive lines of code:

```
// ... license premble and includes ...

using namespace sygaldry;

struct TStick
{
    sygse::Button<GPIO_NUM_21> button;
    sygse::OneshotAdc<syghe::ADC1_CHANNEL_5> adc;
    sygsa::TrillCraft touch;
    sygsa::ICM20948<sygsp::ICM20948_I2C_ADDRESS_0
                    > mimu;
    sygsp::ComplementaryMimuFusion<decltype(mimu)
                                   > mimu_fusion;
};

sygbe::ESP32Instrument<TStick> tstick{};
extern "C" void app_main(void) { tstick.app_main(); }
```

The components of the instrument are listed in a simple structure definition, which is passed to a runtime class specific to a particular MCU. The runtime automatically iterates over the components of the TStick structure at compile time and generates boilerplate code to set up each component and call their main subroutines in a loop. The runtime also provides bindings that automatically generate glue code so that the endpoints of the TStick components are exposed over all of the control protocols available on that hardware platform.

As well as completely eliminating glue code and drastically reducing boilerplate for DMI firmware developers, the library strongly favors code reuse. For example, the firmware for the mubone orientor is substantively similar to that of the T-Stick:

```
// ... license preamble and includes ...

using namespace sygaldry;

struct Mubone
{
    sygse::Button<GPIO_NUM_21> button;
    sygsa::ICM20948<sygsp::ICM20948_I2C_ADDRESS_0
                    > mimu;
    sygsp::ComplementaryMimuFusion<decltype(mimu)
                                   > mimu_fusion;
};
```

```
sygbe::ESP32Instrument<Mubone> mubone{};
extern "C" void app_main(void) { mubone.app_main(); }
```

Sygaldry components are implemented using the reflectable component design pattern described by Celerier [10]. As well as enabling components to be glued to a hardware-specific runtime with hardware-specific transport protocols, the reflectable design pattern is extended in a novel way to enable the highly declarative implementation style, effectively eliminating boilerplate usually required to individually invoke each shared component in hand written setup and main loop subroutines.

The current typeset literate documentation of the library can be found at `https://sygaldry.enchantedinstruments.com`, including the full literate source code of all components in the library.

4. CONCLUSION

Sygaldry emphasises developing portable highly-reusable components. This emphasis is motivated by the potential to broaden and deepen the impact of DMI software development in the research context–DMI components provide a more generalisable foundation for DMI evaluation and study than whole DMIs, and enable bettern maintainability and replicability by favoring code reuse. A library of such components is especially needed in the context of DMI firmware development. Sygaldry suggests a particularly promising form in which these components may be implemented, enabling them to be automatically deployed to various hardware and software runtime environments without the DMI developer having to write any glue code or boilerplate.

Sygaldry is in an early stage of development. The pool of available components is small and limited to DMI firmware, and it remains as future work to begin the intended evaluations and component-oriented study. Nevertheless, the design and implementation of Sygaldry already provides immediate benefits: the library enables the trivial implementation of new DMIs based on the available pool of components, by high-level declaration; it is trivial to later replace a single component of a DMI, enhancing maintainability and simplifying incremental improvement; the emphasis on code portability facilitates porting DMI implementations to new platforms, futher enhancing maintainability; and the use of C++20 language features drastically reduces the burden of glue code, further enhancing portability, code-reusability, and rapid prototyping; all of these features favoring code-reuse within and across projects enabling the maintenance and replication burden of many instruments to be consolidated into a single library, with the potential to drastically reduce overall development effort. The NIME community is urged to consider the benefits of developing easily interconnected and mutually compatible DMI components first and foremost, allowing instruments to emerge as trivial assemblages of these components. Sygaldry is offered as an example of how this development might be approached, and a possible starting point for future collaboration.

5. ENVIRONMENTAL STATEMENT

Maintainable DMI development is more sustainable. Sygaldry encourages the reuse of electronic hardware in prototyping efforts. However, we acknowledge that all DMI prototyping efforts involve the consumption of electronic materials and resources, and that this consumption has a permanent impact on the environment and communities involved in the production of these materials and resources.

References

[1] M. M. Wanderley and N. Orio, "Evaluation of input devices for musical expression: Borrowing tools from hci," *Computer Music Journal*, vol. 26, 2002.

[2] M. Rodger, P. Stapleton, M. van Walstijn, M. Ortiz, and L. Pardue, "What makes a good musical instrument? A matter of processes, ecologies and specificities," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2020.

[3] J. Chadabe, "The limitations of mapping as a structural descriptive in electronic instruments," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2002, pp. 1–5.

[4] V. Goudard, "Ephemeral instruments," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2019.

[5] J. Malloch, "A consort of gestural musical controllers: Design, construction, and performance," M.S. thesis, McGill University, 2008.

[6] M. Marier, "The Sponge: A flexible interface," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2010.

[7] T. J. West, B. Caramiaux, and M. M. Wanderley, "Making mappings: Examining the design process," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2020.

[8] T. J. West, B. Caramiaux, S. Huot, and M. M. Wanderley, "Making mappings: Design criteria for live performance," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2021.

[9] T. J. West and K. Leung, "Early prototypes and artistic practice with the mubone," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2022.

[10] J.-M. Celerier, "Rage against the glue: Beyond runtime media frameworks with modern C++," in *Proceedings of the International Computer Music Conference (ICMC)*, 2022.

[11] J. Lakos, *Large-Scale C++: Process and Architecture*. Addison-Wesley Professional, 2019, vol. 1.

[12] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, 1984.