AOSEC 2023
Asia Oceania Systems Engineering Conference
Theme: Digitalization for engineering Complex Systems
11th - 14th October, 2023 | Bangalore, Karnataka India
Hosted by INCOSE India Chapter

Design toolchain for automotive E/E architecture exploration

LAURENT Jean-Baptiste
Forvia
jeanbaptiste.laurent.pro@protonmail.com

RICHEFEU Julien
Forvia
julien.richefeu@forvia.com

MONSUEZ Bruno
U2IS/ENSTA Paris, IP Paris
bruno.monsuez@ensta-paris.fr

Copyright © 2023 by Laurent Jean-Baptiste. Permission granted to INCOSE to publish and use.

• Abstract

Design space exploration (DSE) is an elegant way to tackle the ever-growing design complexity of automotive Electrical/Electronic (E/E) architectures. In this paper, a toolchain extension is presented and discussed. While still under development, it enhances the modeling with a graphical canvas and allows the model to be defined more naturally, and agnostically before being attached to the rest of the toolchain. It is, then, that more heavy-duty simulation frameworks such as Metropolis are put to use to perform guided design space exploration and converge toward an optimum.

We validated the extended toolchain with a proof of concept use case being challenged by the addition of a third camera, and the way the existing two-frontal cameras subsystem could be adjusted to fit it best. It is intended for this modeling toolchain to be reusable on a larger range of design use-cases.

Keywords: Design space exploration, toolchains, automotive E/E architectures, model base design, Metropolis, hypergraph

• Introduction

Automotive electrical/electronic (E/E) architectures are complex heterogeneous systems burdened by many constraints, such as cost, safety, performance, or cyber-security. Those characteristics combined make their design rather challenging. Toolchains could be used to ease part of this work that would otherwise be done manually, automating error-prone tasks which also are, most often, cumbersome. Unfortunately, the process of early design space exploration (DSE), architecting, engineering, testing, and finally showcasing spans over several competency fields. More often than not, the original model is redundantly expressed and this prevents the usage of a unified toolchain.

The objective of this paper is twofold: presenting the pros of expressing the architecture as a visual hypergraph, instead of text, and having a way to easily feed it then to relevant simulation and verification tools without redundancy.

Our illustration use-case will showcase DSE applied to an automotive camera subsystem, with the enhanced toolchain attached to the Metropolis framework. This is to follow a similar study also done within the Metropolis framework to perform DSE on an automotive use-case.

More recent research exists in the toolchain field for DSE, but they lack qualities needed for prototyping and are more heavy duty and electronically oriented (Gautier, 2022). It is rarely easy for a single person (or small team) to prototype the model, and explore solutions while staying focused on the validation of the concept, compared to the actual implementation work. To put it another way: remaining adaptable without the need for superfluous legacy code, while simultaneously being expandable with minimal effort if more features are to be tested. However, they do have features allowing for the automation of the exploration effort.

The work presented here is part of a larger research project to produce a methodology able to produce an automotive E/E architecture fitting the given product requirements (Cost, bandwidth, flexibility, safety, cybersecurity, …). A lot more remains to be done. However, the results are so far encouraging. It showed potential to be applied to other fields when prototyping an hypergraph like model within a mini graphical user interface (GUI) tool, and reuse it for further demonstration toward a non-technical audience. Limitations and ways to address them will be discussed in a dedicated section.

This paper is structured as follows: Section 1 introduces automobile E/E designs in general. Section 2 will discuss toolchains and why they are useful for easing conceptualization work. Section 3 outlines design-space exploration, which will be utilized in Section 4 to illustrate the results provided by the toolchain compared to the original approach. A discussion will be brought in Section 5. Section 6 presents potential topics for related future research and concludes.

Section 1: Automotive E/E architectures

Designing E/E architectures as distributed was an appropriate design choice for a time, when the number of components were kept minimal, simple, and were associated with a primarily bus-oriented topology. With time, cars became increasingly more than a cabin on wheels and are dealing with evolution such as electric propulsion, and the growing amount of features both for room comfort and driving assistance. On top of it, regulation also comes with its share of mandatory features.

Combined with the need for those new features to start cooperating and behaving as a whole, this leads to an exponentially increased overall complexity. In addition, as autonomous driving becomes more prevalent and vehicles become increasingly linked, safety and cyber-security concerns will need to be addressed much more seriously, resulting in yet another layer of complexity [2].

New architectures are being developed to address the above and its complexity, with solutions aimed around centralizing and segregating sections of the design to contain the amount of components. All in all, allowing parts of it to be abstracted and suppressed from the global picture.

Yet, the amount of new viable abstract candidates is likely high as well, and committing to one of these could also fall short in regard to their results to less intuitive but more optimal solutions. In the long run, this optimization issue is unlikely to be fully solved manually. Since automotive E/E architectures are in need to remain flexible to support future and innovative features. The ability for a car to be used in multiple contexts with little to no modification, such as private cars being rented by apps or plain autonomous, are good examples of applications. If we go further, more intrusive features could allow upgrades to be directly plugged to the car such as advanced driver assistance system (ADAS) sensors, or existing components being replaced by a more capable version.

Subsequently, those architectures would eventually need to consider themselves extended to components beyond the car with vehicle-to-everything (V2X) features. For example, to support dedicated cloud-based services allowing cars to operate within fleets, direct vehicle to vehicle (V2V), vehicle to infrastructure (V2I) communication or as a plain and remote processing unit. Therefore, a finely tuned design made today would either 1) need to be flexible to support those, or 2) be abstracted enough for its refinement to be applied on-demand based on the given constraints.

Producing such a methodology is an ideal target for a research project warranting the work presented in this paper. Illustrated in Fig 1, car-makers first emit a set of high level, functional specifications. Then, a system architect from original equipment manufacturers (OEMs) translates those into a more detailed version, ready for the modeling. Once there, the model gets refined iteratively until all the given constraints are respected. It is after a last simulation step that a set of architecture proposals able to run the detailed functional tasks given as input is finally produced.

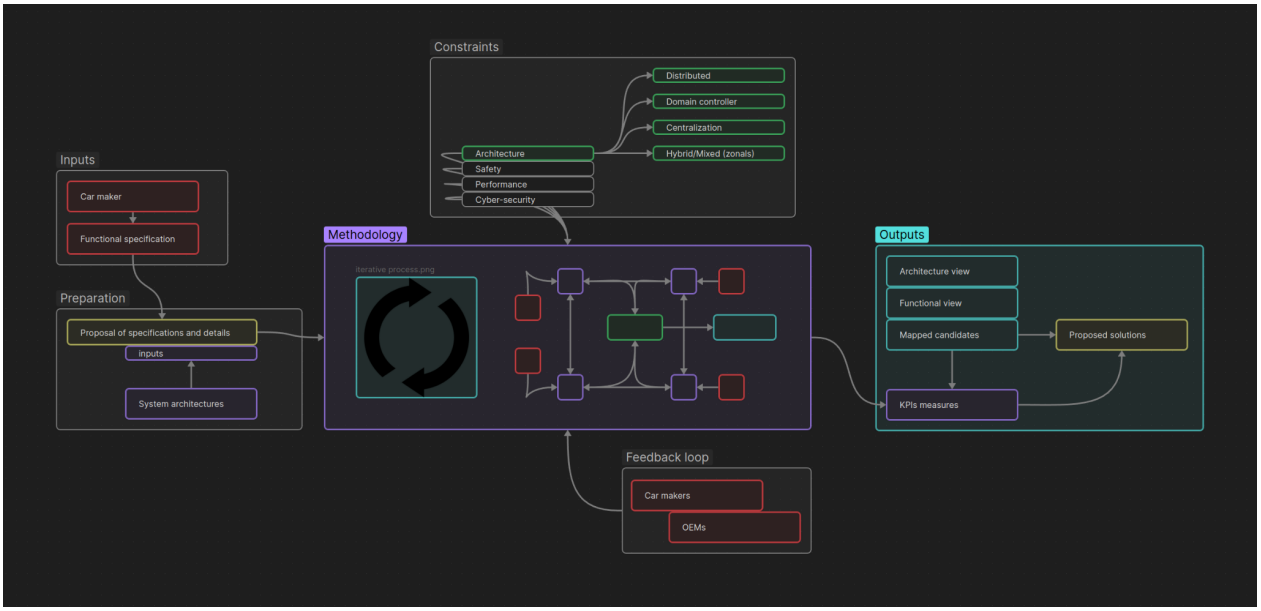


Fig 1: Automotive E⁄E architectures design methodology

It is worth mentioning that redoing the entire design work manually, in a timely manner, is to be excluded. Going toward a flexible design with hardware-agnostic tasks, such as local and cloud oriented services, would allow the development work to be more modular and reusable. Being agnostic may also open opportunities in regard to supporting V2X, distributed computing innovative features. Yet, it comes with a performance tradeoff as the underlying system is less finally optimized: a flexible versus finely tuned system.

Besides, if the design process is sufficiently defined and (semi)-automated, it could allow off the shelf products to be efficiently inserted in an existing one with minimal modification, for example by using predefined templates adjusted then. This approach focuses on reducing the traversal time of the solution space to search for an optimum, instead of evaluating and validating a solution based on a given concept architecture. The former implies the latter, since evaluating different candidates is required during the refinement process.

• Section 2: Toolchains

Trying to describe a graph exhaustively in a natural language, let alone lines of code mixed up with mandatory function calls, is a recipe for legacy and unmaintainable code. When shown visually as a colored graph, though, it becomes much easier to understand (Similarly to how illustrations are used to document text).

To model automotive E/E architectures, and make it more human friendly to explore alternatives, we are ideally looking forward to an extended toolchain with the following mandatory characteristics: The model is stored in plain text, in a parsable format as to be compatible with version control and parsing tools. Unfortunately, it is not uncommon for design frameworks to provide a proprietary or undocumented binary representation. It would also need to be:

1. Usable offline as a portable program. Online services come with their own constraints, including privacy, availability, support, and version compatibility over time. This would bring too much uncertainty to a toolchain aimed at being reliable and consistent.
2. Self-contained, ideally lightweight, and simple to set up, understand, and utilize. A program requiring an installation wizard, an online linked account or too many dependencies would be excluded. Downloading the project sources, and a quick installation, should be all that is needed before being able to work.

Issue is, describing the model as a graph directly could easily bring a handful of dependencies, and split the development into several sources of different forms, so it had to remain simple not only on its representation, but also all the way to its usage. DOT and UML like software programs such as Plantuml were candidates (Roques 2009), as both were having the required characteristics. However, this is only moving the original issue of having to describe the model as plain text in the first place elsewhere.

After filtering out online tools and programs lacking the above traits, the canvas feature of Obsidian came out as a good option. Even though the program sources are proprietary, only the produced JSON is needed. Also, not having to add it as a hard dependency to build the project was very welcome. Instead of the original text description, it is now allowed to express the model visually and let the toolchain deal with the format. Yet, to make it work out of the box a python module was written to ultimately: 1) parse the JSON and 2) deal with the boilerplate code generation and: declare, allocate, initialize, name the components, add the connections and finally perform some nice cleanup.

Currently, the developed python script only supports the transition from the JSON into C++, included into project sources. It is not excluded to later allow the resulting simulation to write live data back into it for Obsidian to display in real time for the experiment, such as seen in Fig 2.

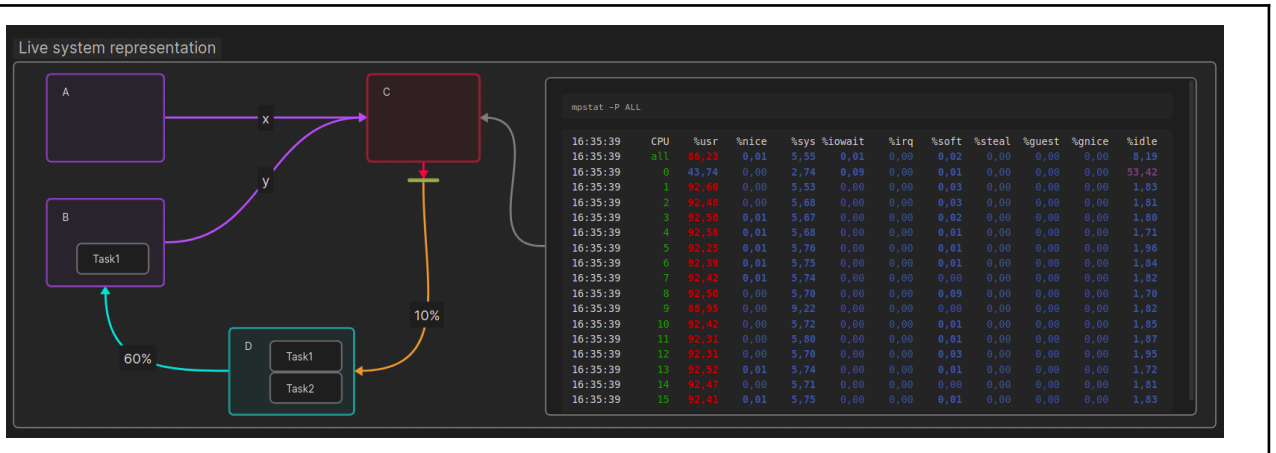


Fig 2: Proof of concept feedback from a node displayed to the canvas by writing back into it

To dig more into the technical details of the current implementation: a developer would develop architecture and functional components. Both will then be mapped together to form a complete system view and explicit which tasks run on which components, ready for simulation. All of it takes the form of a regular compiled project with its classic set of tools to: compile, link and execute. To enhance the expressivity of the model and reduce dependency on the simulation framework, the preprocessing was taken out and replaced by the gpp preprocessor (Miller, 2020). Chronologically, the canvas and translation Python script was added then, ahead of the preprocessor, to produce the relevant mapping code. If needed, the Python script will take simulation outputs and write it back to the canvas, either live, or once the simulation is over.

Section 3: Design space exploration

Design space exploration (DSE) is a methodology to efficiently browse many solutions for a given problem, by either pruning unwanted branches at higher abstraction levels or descend into promising ones. Metrics to discriminate candidates often consider cost and performance as a starting point, but depending on the issue at hand, any can be considered such as flexibility. In our use-case, flexibility refers to the ability of an automotive E/E architecture to support a wide range of tasks or if it can perform while sustaining failures (single point of failures (SPoF), HW failure due to impact or attrition, cybersecurity breaches, …).

Metropolis is an older DSE oriented C++ framework based on a platform-based design methodology. Other methodologies of interest, such as ForSyDe (Ingo S., 2003), Rapide (Luckham and Vera, 1995) and Spade (Lieverse P., 1999) were looked at but were lacking two aspects: a ready to use set of tools, and a precedent study in the automotive industry. It is the latter that validates, at least to some extent, the choice for Metropolis as the first one to evaluate within the list.

The platform-based design methodology can be represented within 3 views: A functional description of the feature, the physical architecture, and how to link the two: i.e., translating the functional description into tasks capable of being hosted by one of the components provided by the architecture. This can be seen as both performing a top-down and bottom-up approach, and meeting both in the middle for the mapping.

Role wise, the functional and architecture view are a set of components grouped up in “platforms”, using Metropolis terms. Those platforms, or development libraries made by a platform developer are then connected by a system developer (which can be the same person) which will perform the design space exploration, and determine the behavior of the entire system.

The result is a metamodel able to be verified by third party tools, and simulate an execution to produce arbitrary metrics. Finding deadlocks using the SPIN checker (Holzmann, 2008), or simply recording execution delays from one point to another to relatively compare two candidates for a given mapping.

Regarding the compilation to execution toolchain. In order to perform a simulation, we go through a couple of stages very similar to C/C++: Writing the model using your favorite editor, compiling it

and running it with a couple of modifications: The code is not purely directed toward the execution alone, but also toward a preparation phase called the “elaboration”.

The elaboration stage is where the “mapping” takes place before the simulation is allowed to proceed, along with a couple metamodel-related activities, such as resolving templates or checking static constraints (Metropolis Project Team, 2004).

To summarize, Metropolis builds and executes its own unique metalanguage before passing the produced code abstract syntax tree (AST), to be then compiled and run for simulation/verification/synthesis. It is, however, getting technologically behind in terms of programming practices.

From a developer point of view, Metropolis is lacking modern sweet features to express code in a clear manner. The metamodel language inhibits the use of C++ preprocessing features, standard templates and finally the use of the C++ standard template library (STL) itself. Even though a couple of simulation compatible containers are provided by the framework, boilerplate code has to be (re)done. Fortunately, the toolchain presented in section 3 alleviates most of the model legacy description, leaving time to develop platform packages, and perform the actual DSE.

Section 4: Experimentation

Context

The extended toolchain will be illustrated with fictive automotive E/E architecture subsystems aimed at presenting side by side the stream of frontal cameras, and elaborating it then with a rearview camera (RVC). Both frontal cameras are plugged to a convert unit whose job is to translate the raw data into a displayable stream. The tagged and annotated streams are then gathered by a high performance computer (HPC) unit, which ensures synchronization and concatenation to provide a single, unified stream for the display to render.

This configuration was initially created solely within the Metropolis framework. Two functional views and a middleware mapping view were developed. So far, we were able to map functions directly to the architecture, with a one to one ratio. The mapping view is mostly composed of glue-code. Simultaneously, we must remain careful to avoid transforming a simple model into a time-consuming and difficult to handle setup. For example, when refining the components or adding new ones, such as a support for a RVC connected with the ADAS.

Instead of writing code for each iteration that only a git/SVN will find easy to track, we can work our way more naturally. Moreover, due to the orthogonality of concerns provided by the platform-based design methodology, the functional and architectural views remain untouched while doing the exploration of design. This is technically allowing non-developers to handle this step and from there onward, it is no longer needed to understand the Metropolis framework to perform the exploration.

Design space exploration

The most straightforward solution to allow the model to support a RVC is to create a completely new chain of components. Like the other two cameras, it will be attached for processing into the HPC, with its own wiring and converter, even if located far from near the trunk. It is a rather classic approach for legacy automotive ecosystems, but in an effort to mitigate the complexity and amount of money it would cost, it needs to be made more efficient.

If the currency cost is the only concern to balance the design with, then it would make sense to plug the RVC higher in the chain to one of the existing converters, actually suppressing a converter to HPC wire. This change most likely necessitated an upgrade to the converter, as it is unlikely that it had some spare computational capacity for the extra stream, because it was designed for the one to which it was coupled. Depending on the upgrading cost, it would be valid to merge the third converter as well. The data flow is preserved, and changes so far only concern the scale of components and the overall topology.

If we continue with the upgrade, all converters could be removed and the processing delegated into the HPC unit. The HPC is designed for that and could possess hardware acceleration features. Despite the fact that it can handle the entire processing load, this is a more intrusive centralization that goes beyond the share of available resources.

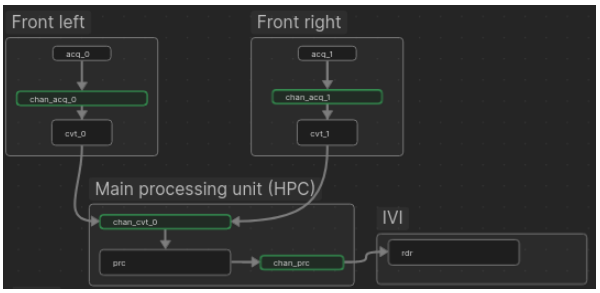
Now that the cost constraint was considered, we are no left with a couple of exploration paths:

- Refine the different candidates into a more detailed model by either: 1) detailing a component implementation, 2) consider scheduling explicitly as a task to be performed and benchmarked, 3) changing the logical flow or architectural composition.
- Insert more constraints and re-evaluate the design, such as safety or flexibility, in case we want to remain agnostic from part of the systems if larger changes are needed.

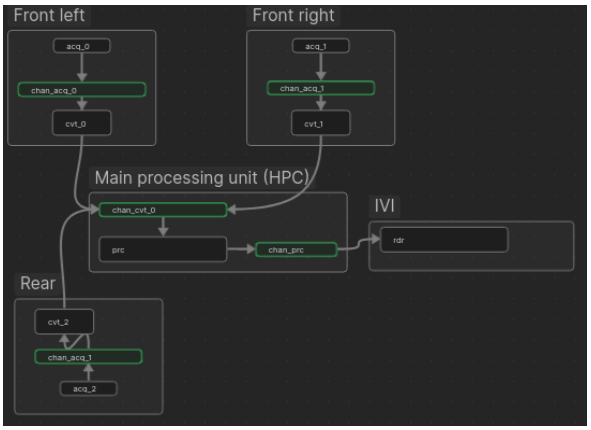
One of such changes could concern the ISO 26262 functional safety standard. If front cameras are to be used for ADAS processing at high speed, they will need to respect a higher level of constraints. The RVC might not need to be held to such a level of safety. Merging all three converters would effectively propagate the constraints, whereas leaving the RVC with its own separated, small converter could very well be less costly.

It is worth noting that even if the fully centralized solution was found after the distributed one, it shall not be seen as an evolution of it but rather a different one to be considered. With its own pros, cons, and simulation metrics. The reader is welcomed to follow the paper from K. Sri and Z. Haibo (2006) for a more complete listing of both designs, including a QoS metric oriented around flexibility.

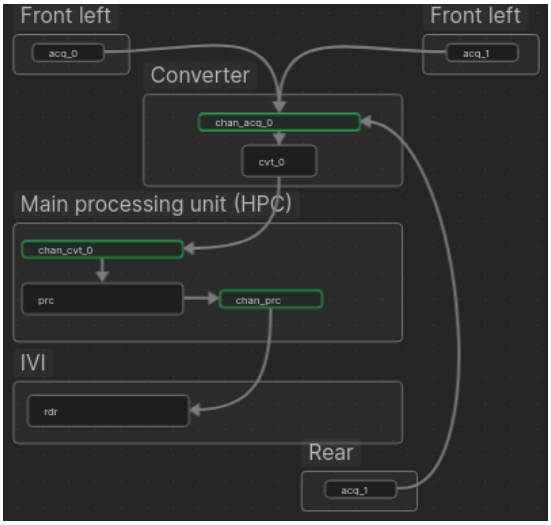
No code update was needed to simulate and gather results regarding the above example, outside changes to the canvas to describe the model content.



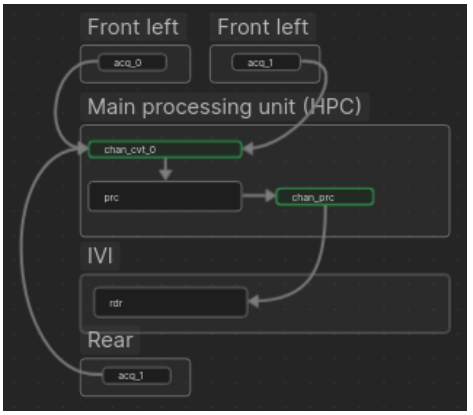
Initial setup: Two raw video streams as input



1st change: new RVC stream



2nd change: Centralizing converter



3rd change: Full centralization

Fig 3: System candidates

• Section 5: Discussion

At first, the goal of the work presented here was to suppress the necessity of having to deal with a significant amount of glue code present in Metropolis. For the matter, we took a couple tools to handle the redundant and error-prone parts, to only leave the creativity aspects of it. This change allowed us to perform the DSE and reach convincing automotive E/E architecture candidates. This was done with the addition to the simulation framework of: 1) a preprocessor, 2) a translation script and 3) a software allowing the description of the model onto a graphical canvas.

After putting everything together, we found out that the toolchain could be improved further. However, this would not work out of the box, and internal developments would be required for that to happen. This is, so far, out of scope as final requirements are yet to be formalized. Prototyping and the ability to pivot quickly remains the main qualities for the toolchain, as of today.

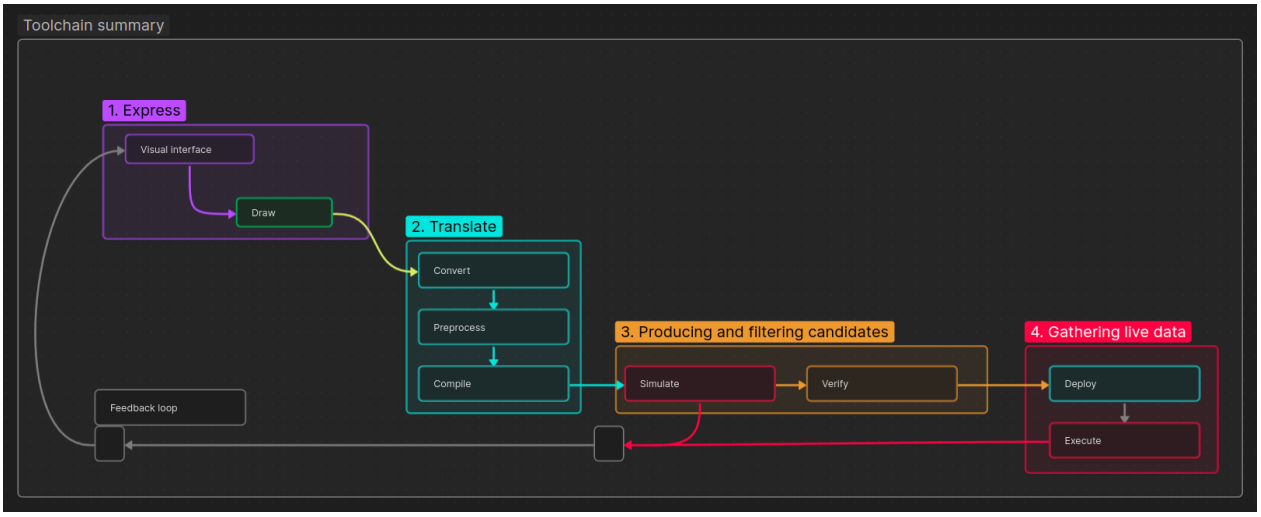


Fig 4: Toolchain content

The toolchain can be decomposed into four components, which would need to be: 1) Expressing the model concisely, even if not exhaustive, 2) Parsing the stored version of the model (from a JSON, comma separated values (CSV) or other plain text files), and generating code ready for the simulation. 3) and 4) aim at producing results. The former in a quick, ideally automated way, while the latter is used in complement to gather real benchmarks. Additionally, the simulation is the only part capable of testing different architecture topologies without manual intervention.

More generally, 1) fits into the set of visual programming languages. For that component, interoperability, and the ability to freely annotate metadata is key. Regarding the simulation and deployment parts, it can be considered to be a black box system and should be modular, to allow more verification tools to be used independently. It is worth noting that while we placed the verification part with the simulation component, it is also relevant for it to be working with the real deployed version. Ultimately, verification should be able to work from one or the other. As long as those keep a clean interface, the only component requiring extensive development shall be the translation part. In generality, having a dedicated translation, or “adaptor” unit is fairly common when dealing with different formats over time.

Obsidian came out as a good candidate for prototyping and fits the requirements stated earlier, but is fairly bare-boned for the given use-case. For comparison, Grasshopper (David R, 2014), a graphical algorithm editor, supports the development of third-parties components such as a dedicated DSE set of features (Brown N. 2019). Moreover, the fully visual features prevent raw text from being entered and wrongly parsed and render computed values later, which is always a bonus

(Fig 5). It is, however, dependent on Rhinos3D (Robert M, 2020), which is licensed based, and missing portability, and designed for 3D modeling in the first place rather than raw, weighted hypergraphs optimization. Although not usable, its means to express a model, and subsequently explore, are worth converging to.

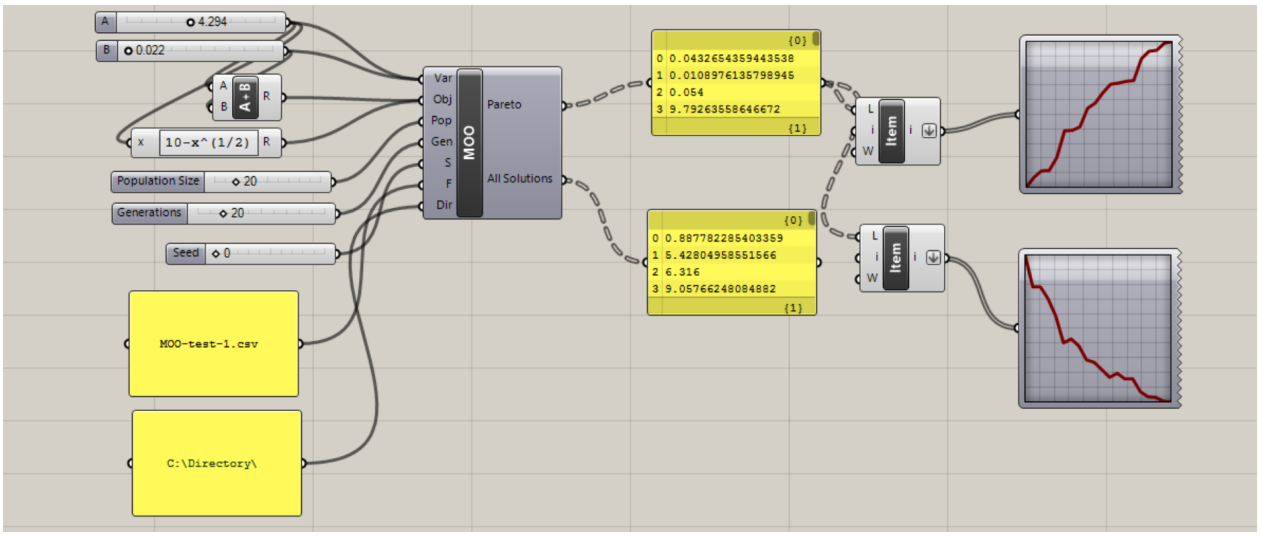


Fig 5: A DSE plugin for the grasshopper module of Rhinos3D, displaying input sliders and a visual output updated live.

To be complete, the toolchain would benefit from being extended one step further ahead with the addition of algorithms to explore and generate different kinds of exotic topologies. To not only restrict itself to sizing the model. For example, one could use Boost C++ libraries and produce random graph theories, use an Erdős–Rényi model, or use evolving networks based on constraints.

However, even though automotive use-cases tend to have a limited and small number of nodes, each produced topology would need to host a set of tasks of various kinds: comfort services, critical tasks with hard deadlines or dedicated programs using hardware acceleration capabilities not always capable of being load balanced. This mapping is likely where the bulk of the complexity will be at, but also the part which would benefit the most from toolchain components being discussed earlier.

Combined with exploration methodologies and different abstraction levels, such toolchain could be reused for a larger range of topics also dealing with the production of heterogeneous ecosystems.

It is important to note that the topic remains complex, and over engineering the whole would be an easy pitfall to fall for. An overly detailed canvas becomes a burden to maintain, unguided exploration would lead to countless candidates being checked, or too technical components slowing down the prototyping. The design will indeed need to be done carefully and remain as simple as possible.

Section 6: Future work & conclusion

Representing automotive E/E architectures within a graphical canvas proved to be an efficient way to explore different abstract designs more conveniently. Because the canvas is entirely custom, mutating the model to reflect a specific view of the system is natively supported. Using model of computations (MoC) from the Y-Chart methodology (Bart, Kienhuis. 2000), and refining one layer at a time with the addition of new constraints for DSE is a logical next step.

Furthermore, the usage of a canvas being internally represented by a JSON file, parsed then by a python script allows it to be plugged into more heavy-duty frameworks with minimal adaptation. It allows the prototyping work to be reused further down the development cycle. More extensions could profit the toolchain, either ahead to produce topologies or fuzz values automatically, or down the stream by implementing a modular verification framework. The verification could work both on an abstracted simulation to guide the exploration, or directly on the refined and deployed version, reducing further the amount of tools needed.

Although simple to use, the above extensions are yet to be tried and tested more thoroughly. Practice will either highlight showstoppers limitations pushing for a change of technology, or refine the existing toward a generic enough version applicable to elaborated use-cases. The toolchain lack of maturity shall be addressed in a future paper more closely linked to the original project regarding automotive E/E architecture design.

Finally, more general research could be conducted on accessible DSE enhancing algorithms, such as libraries enabling watersheds' distribution, neural networks, gaming theories or graph balancing theories.

References

Arnaud, R. (2009). Open-source tool that uses simple textual descriptions to draw UML diagrams. https://plantuml.com/

Bart, K. (2000). Y-chart methodology and Models of Computation and Architecture. https://ptolemy.berkeley.edu/~kienhuis/dacSlides/kienhuis.pdf

Boost C++ Libraries. (2023). [HTML]. Boost.org. https://github.com/boostorg/boost (Original work published 2013)

Brown, N. (2019). Digital-Structures/gh-design-space-exploration [C#]. Digital Structures. https://github.com/Digital-Structures/gh-design-space-exploration (Original work published 2017)

David, R. (2014). Grasshopper 3D. grasshopper3d.com

Gautier, Q., Althoff, A., Crutchfield, C. L., & Kastner, R. (2022). Sherlock: A Multi-Objective Design Space Exploration Framework. ACM Transactions on Design Automation of Electronic Systems, 27(4), 33:1-33:20. https://doi.org/10.1145/3511472

Haas, R. E., & Moller, D. P. F. (2017). Automotive connectivity, cyber attack scenarios and automotive cyber security. 2017 IEEE International Conference on Electro Information Technology (EIT), 635–639. https://doi.org/10.1109/EIT.2017.8053441

Hegedus, A., Horvath, A., Rath, I., & Varro, D. (2011). A model-driven framework for guided design space exploration. 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 173–182. https://doi.org/10.1109/ASE.2011.6100051

Holzmann, G. J. (2008). The spin model checker: Primer and reference manual (4th print). Addison-Wesley.

Ingo, S. (2003). ForSyDe. https://people.kth.se/~ingo/Papers/Thesis_Sander_2003.pdf

Jakovljevic, M., & Gatard, J. (2016). Advanced integrated embedded platforms for advanced TCAS, situation awareness and (semi-)autonomous piloting.

Lieverse, P., van der Wolf, P., Deprettere, E., & Vissers, K. (1999). Spade, a methodology for architecture exploration of heterogeneous signal processing systems. 1999 IEEE Workshop on Signal Processing Systems. SiPS 99. Design and Implementation (Cat. No.99TH8461), 181–190. https://doi.org/10.1109/SIPS.1999.822323

Luckham, D. C., & Vera, J. (1995). An event-based architecture definition language. IEEE Transactions on Software Engineering, 21(9), 717–734. https://doi.org/10.1109/32.464548

Miller, T. (2023). GPP. https://github.com/logological/gpp (Original work published 2015)

Robert, M. (2020). Rhinoceros 3D. rhino3d.com

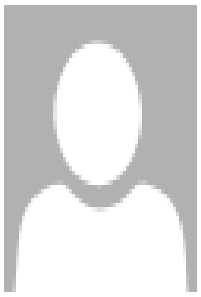
Sander, I., & Jantsch, A. (2004). System Modeling and Transformational Design Refinement in ForSyDe. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 23(1), 17–32. https://doi.org/10.1109/TCAD.2003.819898

Sri, K., & Haibo, Z. (2006). Exploring Trade-off's Between Centralized versus Decentralized Automotive Architectures Using a Virtual Integration Environment. https://ieeexplore.ieee.org/document/1656943

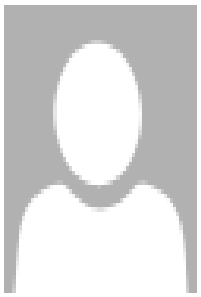
The Metropolis Project Team. (2004). The Metropolis META model—Version 0.4. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2004/4259.html

Zeng, H., Davare, A., Sangiovanni-Vincentelli, A., Sonalkar, S., Kanajan, S., & Pinello, C. (2006). Design Space Exploration of Automotive Platforms in Metropolis. 2006-01–1468. https://doi.org/10.4271/2006-01-1468

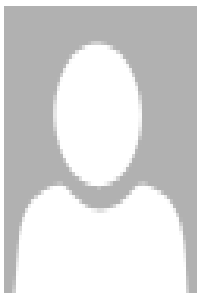
Biography



LAURENT Jean-Baptiste is a PhD student at the National School of Advanced Techniques, ENSTA University in Paris. His research interests include system design and computer science.



RICHEFEU Julien is an Innovation Manager at Forvia group, responsible for E/EA innovation topics on FCE, supervising Jean-Baptiste's work. He has a PhD on Artificial Intelligence obtained in 2006 at ENSTA.



Bruno MONSUEZ. Provide a short biography of the author.