XXxXxXx: Realizing Design Space Identification for Generic and Efficient Design Space Exploration

Omitted for blind review

***Abstract*—Automated design space exploration (DSE) is a key component of model-driven engineering (MDE) methodologies, enabling complex designs otherwise prohibitively challenging. Such complexity mitigation is achieved by carefully choosing the models abstracting the target embedded systems and implementing efficient DSE methods to explore the design space these models define. However, achieving both domain independence and performance in DSE frameworks remains challenging, especially if feasibility and optimality guarantees are desired.**

The design space identification (DSI) approach has been proposed as a generic and tunable DSE framework to tackle this challenge. DSI uses composable identification rules to identify different abstract decision models through a separation-of-concerns philosophy. This technique allows cross-domain and specialised DSE methods to seamlessly coexist and be used for the same input design model.

We propose in this paper enhancements to the DSI approach in order to make multi-modular implementations of DSI possible. In these lines, we present XXxXxXx, the first DSE based on the enhanced DSI approach. XXxXxXx follows a multi-modular implementation in order to push the mathematical extensibility of DSI into a generic DSE framework that supports modules connecting different MDE frameworks. XXxXxXx is openly available for download and extension.

We showcase how XXxXxXx achieves domain independence, exploration performance and guarantees of feasibility and optimality through illustrative case studies from the avionics domain and digital signal processing domain.

***Index Terms*—design space exploration, model-driven-engineering, design space identification**

I. Introduction

Automated design space exploration (DSE) is a key component of embedded system-level model-driven-engineering (MDE) methodologies, enabling embedded systems designs that would otherwise be unobtainable due to the complexity of designing such systems [1]. A non-exhaustive list of such methodologies is: the SDF^3 [2] toolset and design flow, the DAEDALUS methodology [3], and the ForSyDe methodology [4] with the DeSyDe DSE tool [5].

A successful *generic and automated* DSE tool must be cooperatively extendable by *design domain experts* and *optimization experts*. This cooperative extensibility is crucial so that the tool covers the target design domains without sacrificing *exploration peformance* and *guarantees* of optimality and feasibility. These three characteristics are antagonistic. A design domain expert typically is not proficient in optimisation to extend a DSE tool with new optimization models in a way that is both efficient and retains optimality and feasibility guarantees. Likewise, an optimisation expert is typically not proficient in the design domain to extend the tool with suitable design domain models for different design scenarios.

Recently, the design space identification (DSI) approach was proposed in [6] as a generic and tuneable systematic approach to constructing DSE methods and tools. The approach does not impose or fix a universal foundation to connect different DSE methods, as usual for *meta-DSE* or *domain independent*-DSE approaches (Section V), but enables multiple foundations to exist in tandem. This co-existence enables design domain experts and optimisation experts to cooperate within their respective expertise, resulting in generic and efficient DSE methods. Moreover, [6] proposed the DSI approach so that it can be implemented into a framework for automated DSE tools following a separation-of-concerns philosophy. However, the authors did not implement this approach in [6].

In this paper, we present XXxXxXx, a DSI-based extensible DSE tool that is the first DSI framework implementation. To successfully implement this framework, we have enhanced the DSI approach with clearly defined stages and well-defined interfaces between them.

The paper is structured as follows. Section II discusses the DSI approach as well as the proposed enhancements. Section III discusses XXxXxXx and its implementation considerations. Section IV showcases case studies performed with XXxXxXx; these are found in the companion repository[1]. Section V relates XXxXxXx to previous methods and tools, and Section VI concludes the paper and states possible directions for the future.

II. Design space identification and exploration

DSI is a *compositional and systematic* approach for creating DSE solutions that distinguishes between the *design domain* and the *optimisation and decision domain*. The design domain contains *design models* typically used by MDE tools that implicitly define design spaces. The optimisation and decision

Omitted for blind review.

[1]https://anonymous.4open.science/r/memocode-demonstrator-16-0DBA

domain contains *decision models* that are sets of parameters and associated functions that explicitly abstract design spaces and are potentially explorable. The bridge between both domains is achieved through the combination of *composable identification rules* and an *identification procedure* [6].

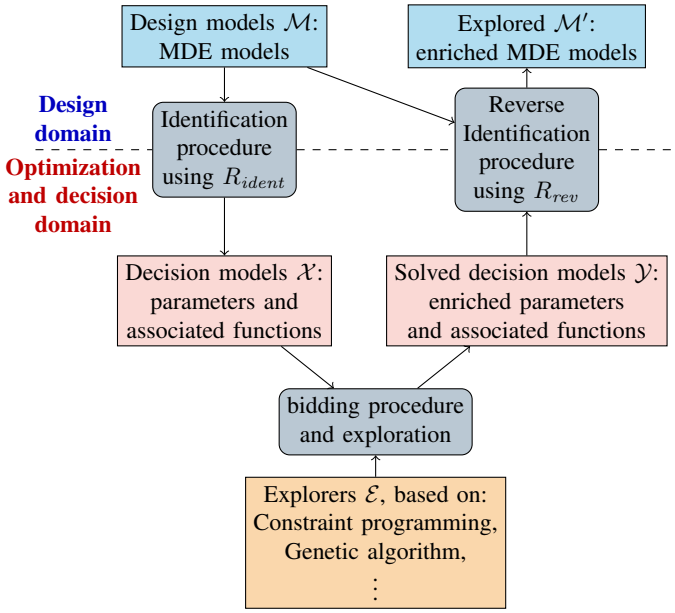

Fig. 1. Enhanced overview of the DSI-based DSE activity.

Identification rules map design models and other decision models onto new decision models, i.e. *partially identify* new decision models. The identification procedure is an automatic fix-point-based algorithm that incrementally produces decision models based on identification rules. Consequently, a DSE tool based on DSI does not require explicit dependencies between decision models but only the identification rules that partially identify them. For the same reason, backwards-compatibility after extensions is guaranteed by keeping identification rules existing before extension [6].

We enhance the DSI approach with *reverse identification rules* and *bidding*. By reverse identification, we mean the activity of *reverse identifying* explored design models out of the explored decision models resulting from exploration. Reverse identification is implicit in [6], by the "backwards" transformations taking exploration results in the decision domain to the design domain. We choose the word "reverse" instead of "inverse" as reverse identification rules are *not* inverse mathematical functions of identification rules. By bidding, we mean the process where different explorers share exploration characteristics and compete to explore a decision model. The proposed enhancements are more convenient for implementation purposes (Section III) as the explicit interfaces between identification, exploration and reverse identification can be implemented in general-purpose programming languages.

Therefore, the enhanced DSI approach (Fig. 1) has three consecutive stages, each with its automated procedure. Concrete examples of DSI elements, such as identification rules and decision models, are given in the evaluation (Section IV).

A. The identify-explore-reverse diagram

Flow runs can be visualised through *identify-explore-reverse diagrams*. A flow run is the sequential execution of the three procedures to obtain explored design models out of input design models (Fig. 1). Identify-explore-reverse diagrams show the interplay of design models, decision models, identification rules, reverse identification rules and explorers in a flow run. An example of this diagram is shown in Fig. 2.

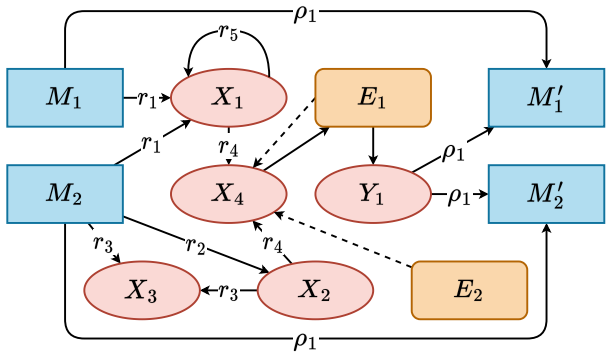

Fig. 2. An example identify-explore-reverse diagram.

We use Fig. 2 as the guiding example to describe the conventions of identify-explore-reverse diagrams, as it fully captures such conventions. The first convention is that design models, decision models and explorers must be visually distinct. In Fig. 2, this convention is applied by choosing different colours and shapes. The second convention is that (reverse) identification rules label the arcs, showing which (reverse) identification rule identified the target design or decision models based on the source design or decision models. For instance, the identification rule r_1 may identify the decision model X_1 out of the design models M_1 and M_2; and the reverse identification rule ρ_1 may reverse identify the explored design models M_1' and M_2' out of the explored decision model Y_1. The third convention is that arcs from and to explorers describe biddings or exploration and must be visually distinct. In the example, the biddings are shown as label-less dashed arcs, whereas exploration is shown as label-less solid arcs.

We make two important remarks on these diagrams. First, identify-explore-reverse diagrams are comprehension aids, and DSI-based DSE tools do not need to generate them automatically when performing a flow run. XXxXxXx does not generate such diagrams, for instance. Second, (reverse) identification rules do not explicitly specify their input design and decision models (Section II-B and III); identify-explore-reverse diagrams shown where they *may succeed* in partially identifying new models. This last observation is also valid for explorers in identify-explore-reverse diagrams.

B. First stage: identification procedure

In the first stage, a set of decision models $\mathcal{X}$ is identified from the input design models set $\mathcal{M}$ given a set of identification rules R_{ident}. In line with the beginning of this section,

each identification rule $r \in R_{ident}$ is a mathematical function from the input design models set $\mathcal{M}$ and any set of decision models $\mathcal{X}_i$ to a new set of decision models $\mathcal{X}_{r,i+1}$:

$$\mathcal{X}_{r,i+1} = r(\mathcal{M}, \mathcal{X}_i) \tag{1}$$

where the additional subscript r denotes that the decision models in the set $\mathcal{X}_{r,i+1}$ have been partially identified by r. The set $\mathcal{X}_{i+1}$ is obtained by:

$$\mathcal{X}_{i+1} = \mathcal{X}_i \bigcup_{r \in R_{ident}} r(\mathcal{M}, \mathcal{X}_i) \tag{2}$$

The requirement for every identification rule $r \in R_{ident}$ is that they are *monotonically increasing* concerning the number of the *partially identified* elements of $\mathcal{M}$. That is, an identification rule at step $i+1$ cannot partially identify *less* elements than it did at step i via (1). The monotonicity property of an identification rule is not restrictive; intuitively, it requires that if the partially identified part of the input models does not change, the identified decision model remains the same.

More precisely, consider $elems(M)$ to be a set representing the elements of $M \in \mathcal{M}$, and $elems(\mathcal{M})$ the unions of such sets: $elems(\mathcal{M}) = \cup_{M \in \mathcal{M}} elems(M)$. Let $part(X)$ be the set of partially identified elements of a decision model $X \in \mathcal{X}_i$ with $part(X) \subseteq elems(\mathcal{M})$; we also define $part(\mathcal{X}_i) = \cup_{X \in \mathcal{X}_i} part(X)$ in analogy to the design models. Then, the monotonicity requirement is:

$$part(\mathcal{X}_{r,i}) \subseteq part(\mathcal{X}_{r,i+1}) \subseteq elems(\mathcal{M}) \tag{3}$$

The identification procedure is performed until a fix-point $\mathcal{X}$ is reached; that is, $\mathcal{X} = \mathcal{X}_{i+1}$ when $part(\mathcal{X}_{i+1}) = part(\mathcal{X}_i)$, as outlined by (1) and (2). This fix-point exists due to the monotonicity requirement (3), and the resulting identified decision model set $\mathcal{X}$ is used in the next stage (Section II-C).

We remark that (1) is different to the mathematical definition in [6], as a set of identified decision models are returned instead of a single decision model. However, (1) is conceptually equivalent to the one definition of [6], since their identification rules also progressively "partially identify more elements of $\mathcal{M}$ by taking previously partially identified $X \in \mathcal{X}$" [6].

C. Second stage: bidding procedure and exploration

In the second stage, a set of explorers $\mathcal{E}$ bid for the identified decision models and the winning bid proceeds to exploration. Every explorer $E \in \mathcal{E}$ in our approach is based on a decision and optimisation method. These methods may be generic decision and optimisation frameworks such as constraint programming (CP). Consequently, an explorer can be incapable of solving a decision model if the decision model is not compatible with the explorer's underlying method. Different explorers may display different exploration characteristics when they can solve the same decision model. For example, one explorer is the fastest to find any feasible solution but the slowest to find the optimal solution. In order to obtain the possible combinations between explorers and decision models, the *bidding* procedure is performed after the identification procedure.

In the bidding procedure, every explorer $E \in \mathcal{E}$ bids for each identified decision model $X \in \mathcal{X}$, returning whether E can explore X *and* the characteristics of performing this exploration. A bid can *dominate* another, in the sense that it partially identifies more elements or has better exploration characteristics. Therefore, the bidding procedure results in one or more dominating bids for exploration $\mathcal{B} \subset \mathcal{E} \times \mathcal{X}$; if there is more than one, they are equally favourable in terms of exploration characteristics.

More precisely, if there are up to c exploration characteristics, the bidding characteristics are $bid_E(X) \in \mathbb{R}^c$ for any explorer $E \in \mathcal{E}$ and decision model $X \in \mathcal{X}$. If $bid_E(X)$ is the zero vector in $\mathbb{R}^c$, then E cannot explore the design space defined by the decision model M. For every two distinct (E, X) and (E', X'), we say that (E, X) *dominates* (E', X') if:

(1) $part(X') \subset part(X)$ or,
(2) $part(X') = part(X)$ but all entries of $bid_E(X)$ are equal or greather than $bid_{E'}(X')$.

The resulting set $\mathcal{B}$ is computed via the dominance relation by keeping only the dominant bids:

$$\mathcal{B} = \left\{ (E, X) \in \mathcal{E} \times \mathcal{X} \;\middle|\; \begin{array}{l} \nexists (E', X') \in \mathcal{E} \times \mathcal{X}, \\ (E', X') \neq (E, X), \\ (E, X) \textit{ dominates } (E', X') \end{array} \right\} \tag{4}$$

After bidding, a dominant bid is randomly chosen from $\mathcal{B}$ and explored, returning a set of explored decision models. That is, a bid $(E, X) \in \mathcal{B}$ is chosen and the exploration function of E, $explore_E(X)$, returns a set $\mathcal{Y}$ of explored decision models. The set $\mathcal{Y}$ is used for the last stage.

D. Third stage: reverse identification procedure

Despite their conceptual analogy, the reverse identification procedure is simpler than the identification procedure. This is because the input explored decision models $\mathcal{Y}$ are already dominant, which entails that all the elements required to obtain the explored design model are already present in $\mathcal{Y}$. Thus, one identification step is enough and a fix-point-based procedure like Section II-B is unnecessary. Each reverse identification rule $\rho \in R_{rev}$ is mathematical function from the input design models set $\mathcal{M}$ and the set of explored decision models $\mathcal{Y}$ to a new set of explored design models $\mathcal{M}'_\rho$:

$$\mathcal{M}'_\rho = \rho(\mathcal{Y}, \mathcal{M}) \tag{5}$$

The final explored design model set $\mathcal{M}'$ is given by the union of all reverse identification rules results:

$$\mathcal{M}' = \bigcup_{\rho \in R_{rev}} \rho(\mathcal{Y}, \mathcal{M}) \tag{6}$$

III. XXxXxXx overview

XXxXxXx follows a *multi-module architecture* (Fig. 3), where an *orchestrator* coordinates different modules to perform the three procedures (Section II). The modules are standalone executable programs and can be implemented in

different programming languages. Earlier tool versions followed a *single-module architecture* and required no external orchestrator. Both architectures are equally extensible from the DSI perspective; namely, both are extensible with identification rules, explorers, and reverse identification rules. We focus on the current multi-modular architecture and describe major differences with the previous single-module architecture where appropriate.

A. Multi-modular architecture

The multi-module architecture consists of an arbitrary number of loosely coupled *identification modules*, or I-modules; and, an arbitrary number of loosely coupled *exploration modules*, or E-modules; as well as the *orchestrator*. This architecture is shown in Fig. 3.

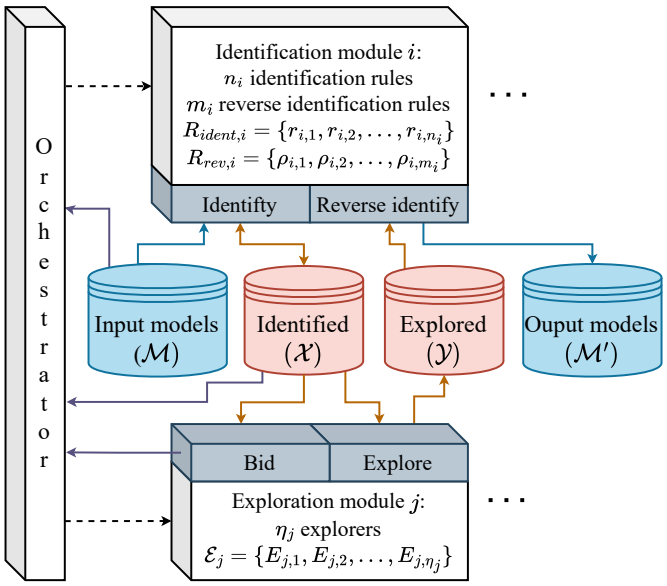

Fig. 3. Multi-module architecture diagram. An orange arc is an exchange of decision model bodies and headers; a blue arc is an exchange of design model bodies and headers; a purple arc is an exchange of decision model headers. Black dashed arcs from the orchestrator to the modules indicate coordination.

This architecture requires design and decision models to produce *headers* and decision models to produce optional *bodies*.

A design or decision model header is an interchangeable *data record* with at least three entries. First, a UTF8 identifier for the model category, e.g. `SDF` (Section IV-C); second, a UTF8 encoded path to the model body; third, a list of UTF8 strings that represent *elems* or *part*. This last entry may be implemented with multiple fields if *elems* or *part* requires it. XXxXxXx, for example, implements *elems* and *part* as both of elements themselves and relations between them, requiring a `covered_relations` to exist in header definitions. Listing 4 shows the Scala implementation of a decision model header as an example.

The body of a decision model is an interchangeable data record with all the parameters describing the model, in line with Section II. No associated functions are allowed in decision model bodies since there is no standard format to share arbitrary functionality between programs developed in different languages. However, for every decision model with associated functions, a new identification rule can be created that produces a new decision model without associated functions; this new decision model aggregates the fixed parameters of the original decision model with the result of its associated functions. Therefore, the lack of associated functions in decision model bodies is not a limitation of this architecture. On a side note, modules implemented in the same programming language can share their associated functions (Section III-B); consequently, the single-module *always* allows associated functions. XXxXxXx uses JSON[2] and MsgPack[3] as the format for interchangeable data records, the former being used only for debugging purposes.

```
case class DecisionModelHeader(
    val category: String,
    val body_path: Option[String],
    val covered_elements: Set[String],
    val covered_relations: Set[LabelledArcWithPorts]
) { /*...*/ }
```

Fig. 4. Scala decision model header from the Scala core library.

For the shared workspace between modules, XXxXxXx uses the operating system's filesystem with naming conventions. Namely, a *run path* is given to XXxXxXx at the beginning of every flow run; XXxXxXx creates one nested folder in this run path for each of $\mathcal{M}$, $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{X}'$ (Fig. 3), and orchestrates the modules in the flow run along these created folders. Future versions of XXxXxXx can use portable and multi-language databases such as SQLite[4] instead of files and folders for increased consistency during the three DSI procedures.

This architecture enables the combination of independently developed modules. The orchestrator can detect all modules available in the workspace before a flow run and use them to perform the three DSI procedures; thus, a problematic module can be updated or replaced in isolation from other modules or the orchestrator. More importantly, this architecture directly enables re-using MDE libraries and code in their target language to decrease the implementation effort. For example, a Java-based I-module is able to use ECore-based MDE as design models. In addition, a Python-based I-module could use SymPy for symbolic algebra to identify linear-algebra-heavy decision models like `SDF`. Both I-modules would be seamlessly coordinated during the identification procedure. In contrast, this interaction is not possible with the single-module architecture and requires language-specific techniques to interact with code written in other languages.

The multi-module architecture performs the three DSI procedures slower than the single-module architecture. The slower performance arises from two different overhead sources. First,

[2]https://www.json.org/json-en.html
[3]https://msgpack.org/index.html
[4]https://www.sqlite.org/index.html

the overhead involved in reading and writing the decision models in the shared workspace. Second, the overhead in independently executing I-modules and E-modules routines during any procedure. The latter factor includes not only operating system overheads of spawning new processes but also warm-up times of modules implemented in just-in-time compiled or interpreted languages; examples are Java-based or Python-based modules. A portion of these overheads can be mitigated if the modules are long-lived services instead of short-lived processes as in the current XXxXxXx version, at the cost of increased orchestration effort. We leave this architectural direction for future work.

B. Blueprints and core libraries

Both I-modules and E-modules build on *module blueprints*, which themselves build on *core libraries*. A core library for a programming language is the set of routines, data structures and interfaces that facilitates extending XXxXxXx. A module blueprint for a programming language is an additional set of routines, data structures and interfaces that enables the correct rapid creation of I-modules and E-modules as command-line applications. Creating new modules through the blueprints guarantees that the orchestrator uses the created modules reliably on the appropriate procedures. Listing 5 shows the function interface for identification rules in the core libraries for Scala and Rust as examples.

```
trait IdentificationModule {
    // ...
    def identificationRules: Set[
        (Set[DesignModel], Set[DecisionModel]) =>
    Set[? <: DecisionModel]
    ]
    // ...
}
```

```
pub type IdentificationRule =
    fn(&Vec<Box<dyn DesignModel>>, &Vec<Box<dyn
    DecisionModel>>) -> Vec<Box<dyn DecisionModel>>;
```

Fig. 5. Identification rule interfaces from the Scala and Rust core libraries.

Building an additional *common library* between modules written in the same programming language is good practice though not mandatory. This common library can hold the data structure for decision models so that modules consuming this library produce and consume their decision models correctly. All the four I-modules and the one E-module in Section IV follow this approach, as they are implemented in Scala.

For the sake of modularity, the creation of new I-modules in XXxXxXx ideally a one module per MDE framework ratio. This rule-of-thumb ratio facilitates the tracing of possible implementation mistakes and facilitates the deployment of the modules as standalone programs.

IV. Evaluation

We showcase the extensibility of XXxXxXx through three case studies. These three case studies result in the combined identify-explore-reverse diagram shown in Figure 6 that is used for reference in the following sections.

A. Avionics case study

The first case study is based on the demonstrators of the PANORAMA ITEA3 project [7]. It consists of an avionics safety-critical functionality that processes instrumentation data and displays it to the aeroplane pilot. The target platform type is a typical shared-memory-based multicore machine with asymmetrical processing, i.e. a heterogeneous platform.

The functionality of this case study can be described in terms of periodic workload models with extended precedences [9]. The platform can be described as a heterogeneous partitioned multi-core platform, where every core has a runtime that follows the fixed priority scheduling policy. This specification is given as a system graph in the YyyYyYyyYY meta-modelling framework [10], the design model for this case study. The implicit design space in this design model is the set of mappings of the functionality on the platform, respecting the real-time constraints of the functionality.

We create five new identification rules, r_1 to r_5, across two I-modules to identify this design space explicitly. These I-modules are the YyyYyYyyYY and the Common I-modules, both implemented in Scala.

Rule r_1 in the YyyYyYyyYY module partially identifies `PerWork` from the workload components in the system graph. r_1 guarantees that `PerWork` conforms to [9].

Rule r_2 in the YyyYyYyyYY module partially identifies `HW Topology` from the platform components in the system graph. r_2 guarantees that every core in `HW Topology` accesses at least one memory component to ensure correct data and instruction mappings.

Rule r_3 in the YyyYyYyyYY module partially identifies `Partitioned Runtimes` from the runtime components in the system graph. r_3 guarantees that every runtime component in `Partitioned Runtimes` has a one-to-one mapping with all core components in the system graph.

Rule r_4 in the Common module partially identifies `Partitioned Platform` from the identified decision models, by composing through aggregation `Partitioned Runtimes` and `HW topology` if they are compatible.

Rule r_5 in the Common module partially identifies `PerWork to PartPlat` from the identified decision models, by composing through aggregation `Partitioned Plat` and `PerWork` if they are compatible.

We create the Choco E-module to explore `PerWork to PartPlat` with the `Choco Solver` explorer. This explorer takes `PerWork to PartPlat` decision models and uses the CP formulation of [11] to explore the design space efficiently, resulting in explored `PerWork to PartPlat` decision models that are guaranteed to be feasible and optimal. In this case, the optimality is resource usage minimisation.

Lastly, we create one reverse identification rule, ρ_1, in the YyyYyYyyYY module. Rule ρ_1 enriches the input system graphs with the exploration results of the explored `PerWork to PartPlat` decision models.

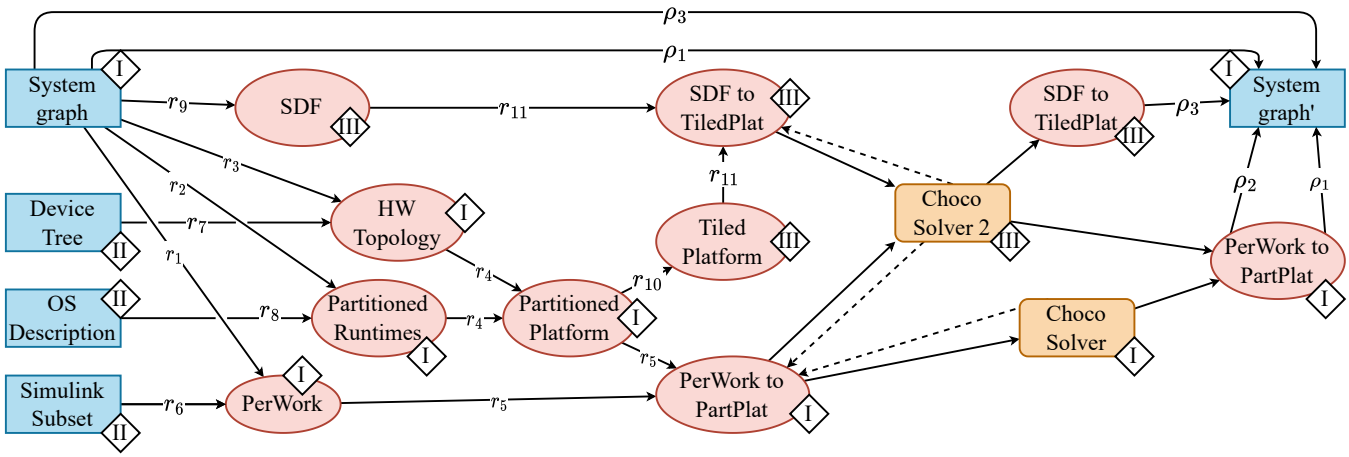

Fig. 6. Case studies combined identify-expored-reverse diagram. The additional badges numbered I, II, and III indicate in which case study the annotated decision model, design model or explorer was added.

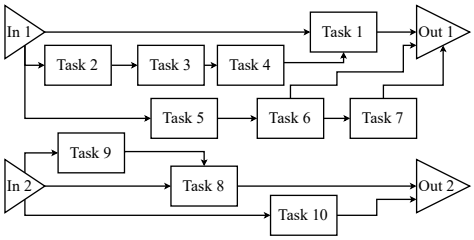

Fig. 7. Avionics case study functionality diagram. Arcs denote data propagation, not precedence. Adapted from [8].

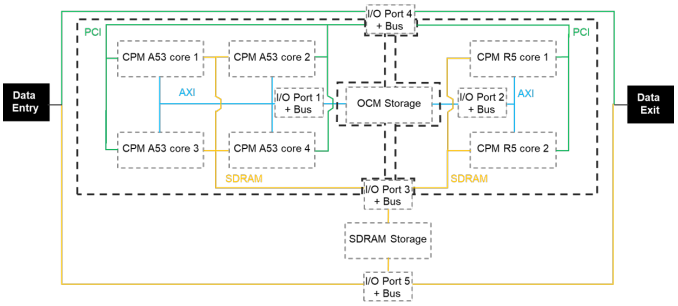

Fig. 8. Avionics case study multi-core platform diagram. [8]

These extensions were successfully tested with the demonstrator case study and a smaller variant of this demonstrator that can be solved manually. Both tests can be reproduced with the design models in the companion evaluation repository[5].

B. Simulink and DeviceTree subset case study

After the first case study success (Section IV-A), we wish to extend XXxXxXx to support other design models that are more commonplace than YyyYyYyyYY. Namely, we will extend XXxXxXx to support a proof-of-concept restricted subset of Simulink models and the DeviceTree specifications as inputs.

[5]https://anonymous.4open.science/r/memocode-demonstrator-16-0DBA

The proof-of-concept Simulink subset does not allow hierarchies and contains a few fundamental blocks from the Simulink common library. Specifically, these blocks are Gain, Relational Operator, (Unit) Delay and Discrete Integrator. The discrete systems described in this restricted Simulink subset can be interpreted in terms of the workload model of [9], and therefore the DSE solution of Section IV-A can be used with this design model. An example Simulink model complaint to this restricted proof-of-concept subset is shown in Fig. 9.

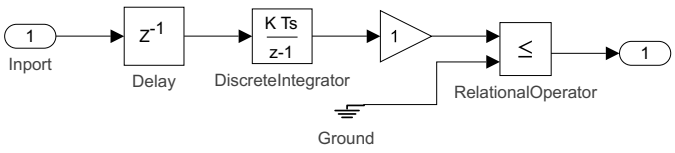

Fig. 9. Example of restricted proof-of-concept Simulink model.

We create three new identification rules, r_6, r_7 and r_8, across two new I-modules. These I-modules are the DeviceTree and Matlab modules, implemented in Scala with minor Matlab scripting. The identification rules are as follows.

Rule r_6 in the Matlab module partially identifies `PerWork` decision models out of `SimulinkReactiveDesignModel` design models. `SimulinkReactiveDesignModel` is an intermediate representation between XXxXxXx and MATLAB/Simulink and is simply shown as `Simulink Subset` in Fig. 6; its creation is necessary due to the integration limitations of Simulink. This extra intermediate representation is invisible to the XXxXxXx end-user, since the Matlab module executes the pre-intermediate conversions transparently during the identification procedure.

Rule r_7 in the DeviceTree module partially identifies `HW Topology` out of `Device Tree` design models. These design models are created directly from the input DeviceTree specification files. The labels in each DeviceTree specification file are assumed to be global so that devices can be shared

across files. Due to the memory-mapped nature of the specification, an implicit bus always connects all devices in each DeviceTree specification.

Rule r_8 in the DeviceTree module partially identifies `Partitioned Runtimes` out of `OS Description` design models. These design models are given in YAML files complementary to DeviceTree specification files, since each DeviceTree file provides only a local view of the HW in the platform. Thus r_8 provides required additional information on the partially identified runtimes, e.g. scheduling policies, and their platform-level relationships.

No explorer extensions are necessary since the new design models can be identified in terms of the decision models pre-existing from Section IV-A. However, with r_6, r_7 and r_8, there is potentially no input system graph for ρ_1 to consume and reverse identify explored system graphs. Therefore, a new reverse identification rule ρ_2 is created; ρ_2 reverse identifies a new system graph per the explored `PerWork to PartPlat`.

These extensions were successfully tested with small proof-of-concept models that can be solved manually. The test can be reproduced with the design models in the companion evaluation repository[6].

C. SDF-based digital signal processing case study

This case study consists in implementing synchronous dataflow (SDF)-based applications for tiled-based multicore platforms, optimising for throughput. The input design models closely follow Experiment III of [12] and are once more system graphs in the YyyYyYyyYY framework. Although this case study is different in nature from IV-A, the design models are similar; therefore we wish to re-use the extensions of Section IV-A as much as possible.

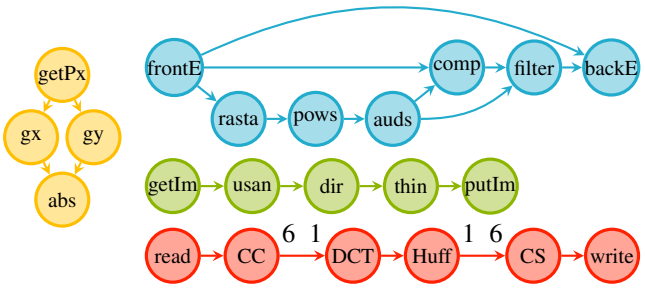

Fig. 10. Case studies SDF graphs. Sobel (yellow); RASTA (blue); SUSAN (green); JPEG encoder (red). Channels with unitary production and consumption rates are drawn as simple arcs.

The key observation in the re-usability direction is that shared-memory platforms where cores do not share memory elements are potentially tiled-based platforms. In this line, we create three new identification rules, r_9, r_{10} and r_{11}, by extending the YyyYyYyyYY and Common modules. These identification rules are as follows

Rule r_9 in the YyyYyYyyYY module partially identifies `SDF` decision models out of system graphs. The partially identified SDFs do not need to be consistent [13] as the parameters are used in other analysis methods.

Rule r_{10} in the Common module partially identifies `Tiled Platform` decision models out of `Partitioned Platform`. The platforms are not genuinely tile-based if there are mechanisms to decouple communication and computation [14], e.g. direct memory access. This check is not currently possible with the standard platform view of YyyYyYyyYY.

Rule r_{11} in the Common module partially identifies `SDF to TiledPlat` out of `Tiled Platform` and `SDF` models. r_{11} checks whether the SDF models are consistent before creating the resulting decision model so that exploration can be performed correctly.

In order to explore the design space defined by `SDF to TiledPlat` decision models, we extend the Choco Solver E-module to `Choco Solver 2`. Fig. 6 shows `Choco Solver` and `Choco Solver 2` co-existing for the sake of chronology and comprehension, but only `Choco Solver 2` exists after extension. This explorer employs the techniques of [12] to explore the explicit design space of `SDF to TiledPlat`.

Finally, we create a new reverse identification rule ρ_3 in the YyyYyYyyYY module that enriches input system graphs with the exploration results in the explored `SDF to TiledPlat` decision models.

These extensions were successfully tested with Experiments III of [12] and a smaller variant that can be solved manually. Both tests can be reproduced with the design models in the companion evaluation repository[7].

V. Related work and tools

Previous approaches and tool frameworks for generic or domain independent DSE lack one of three desirable characteristics: ease of extension, exploration performance and guarantees of feasibility and optimality (Section I). A previous survey on this topic can be found in [15], which includes a comparison of a portion of the tools described next. The authors in [15] also propose a DSE framework that potentially has all three characteristics but did not provide an implementation for evaluation.

Frameworks that use simulation and black-box optimisation techniques (e.g. Genetic algorithms) are typically 1) easy to extend, 2) acceptably performant 3) but give no guarantees of feasibility and optimality. We cite in this category MOST of COMPLEX [16], MULTICUBE [17], and Sesame [18] of DAEDALUS [3]. Only DAEDALUS[8] has a publicly available source repository.

MOST uses meta-heuristics and the response surface methodology with the simulation of UML/MARTE models to propose Pareto-suboptimal designs to the user [16]; the underlying response surface methodology does not give optimality guarantees for the proposed designs or that a design exists.

[6]https://anonymous.4open.science/r/memocode-demonstrator-16-0DBA

[7]https://anonymous.4open.science/r/memocode-demonstrator-16-0DBA

[8]http://www.teisa.unican.es/gim/en/scope/source.html

New design domain models must comply with COMPLEX and UML/MARTE to use MOST as an automated DSE tool. MULTICUBE uses the same DSE method with meta-heuristics and response surfaces but requires a direct representation of the design space; that is, instead of UML/MARTE models, MULTICUBE requires the configuration parameters that make up the design space. Both COMPLEX and MULTICUBE support multiple objectives due to the underlying meta-heuristics. Lastly, Sesame uses genetic algorithms (GAs) with simulation-in-the-loop to explore a variant of Kahn process networks (KPNs) and SystemC transaction-level models of platforms, simultaneously optimising for power, performance and design cost [18]. Extending this framework requires the new models to be reduced to the KPN variant and the SystemC transaction-level models.

Frameworks that use *back-of-the-envelope* models [19] as their foundation are typically 1) easy to extend, 2) give optimality and feasibility guarantees 3) but suffer from slower exploration performance. We cite in this category Tahmuras II [20], MILAN [21], [22], S.P.L.O.T [23], FAMA [24]. Only Tahmuras II[9] and S.P.L.O.T[10] have publicly available source repositories.

Tahmuras II uses the clock constraint specification language (CCSL) [25] as the underlying foundation and uses CP to solve these models. The design models are cleanly separated into a meta-model of applications, platforms and bindings; extending the framework includes adding new models complaint to this three-part meta-model. MILAN provides similar extension points to Tahmuras II, with a meta-model separated into applications, constraints and resources; however, the underlying foundation is ordered binary decision diagramss (OBDDs) which is used with the DESERT DSE tool [26] to obtain the exploration results. S.P.L.O.T and FAMA use binary decision diagrams (BDD) or boolean satisfiability (SAT) (or CP in the case of FAMA) to reason about feature models and propose valid configurations of such models. A design domain expert can extend the tools by creating new design models that comply with the meta-models provided by S.P.L.O.T and FAMA. We note that S.P.L.O.T and FAMA do not guarantee optimality, only feasibility.

Lastly, *specialised* frameworks are typically 1) efficient in terms of exploration performance and 2) provide optimality and feasibility guarantees 3) but cannot be easily extended. By specialised, we mean frameworks that can still be used for a few different design domains but require fundamental changes in implementation and methodology to be as generic as previous frameworks. In this category we cite SDF^3 [2] and DeSyDe [12], [27]. Both are publicly available[11][12].

SDF^3 connects a series of heuristics steps to implement SDF graphs for time-division multiplexing (TDM)-arbitrated tile-based multicore platforms, using statically ordered self-timed schedules. Additionally, SDF^3 provides similar solutions other dataflow models of computation (MoCs), namely cyclo-static dataflow (CSDF) and finite-state machine scenario-aware dataflow (FSM-SADF) [28]. Extending SDF^3 requires understanding these MoCs so that the new design models can be reduced to them; otherwise, new algorithms must be designed for the new design models. DeSyDe can be described similarly to SDF^3 in terms of extensions. However, it supports the SDF MoC, not CSDF or FSM-SADF, alongside the independent periodic task workload model [29] in addition to the SDF^3 platform model. In terms of DSE methods, DeSyDe relies on CP for feasibility and optimality; it also uses problem-specific techniques to accelerate the exploration, which is challenging to generalise.

Compared to these previous works, XXxXxXx combines ease of extension, exploration performance, and guarantees of optimality and feasibility through the enhanced DSI. This is possible because XXxXxXx enables multiple foundations to co-exist alongside several design models; these connect through identification rules that partially identify increasingly abstract but descriptive decision models. Therefore, XXxXxXx is able to identify and explore generic decision models that are likely computationally inefficient alongside specialised decision models with problem-specific techniques whenever possible. Moreover, XXxXxXx's points of extension for design domain experts and decision domain experts are loosely coupled and clear (Section III).

VI. Conclusion

We have presented XXxXxXx, the first design space identification (DSI)-based design space exploration (DSE) tool openly available with a permissive license. The DSI foundation enables XXxXxXx to achieve generality and efficiency in different DSE scenarios. This is achieved through the composition of identification rules that stepwise partially identify models with explict design space parameters out of models with implicit design spaces. This composition grants XXxXxXx extensibility following a separation-of-concerns philosophy.

We evaluated the tool's extensibility through three case studies that demonstrate how the identification rules can be composed to re-use previously existing code and techniques. The efficiency is highlighted by two different exploration solutions co-existing in XXxXxXx seamlessly after the case studies are performed; that is, XXxXxXx was extended to solve different DSE scenarios with problem-specific knowledge for higher efficiency. The XXxXxXx version and the case studies in this paper are available for complete reproduction of the demonstrators.

Future work includes two directions. First, extending XXxXxXx with universal-like formulations as commonly done in the related work, e.g. clock constraint specification language (CCSL). This could provide different fallback generic DSE approaches for problems that are too novel to have problem-specific solutions. Second, extending XXxXxXx with meta-herusitic solution frameworks, e.g. genetic algorithm (GA). This could provide large specifications with a faster exploration method before problem-specific solutions are found.

[9]https://github.com/shaniaki/TahmurasII
[10]http://www.splot-research.org/
[11]https://www.es.ele.tue.nl/sdf3/download/
[12]https://github.com/forsyde/DeSyDe

References

[1] A. D. Pimentel, "Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration," *IEEE Design & Test*, vol. 34, no. 1, pp. 77–90, Feb. 2017, ISSN: 2168-2364. DOI: 10.1109/MDAT.2016.2626445.

[2] S. Stuijk, M. Geilen, and T. Basten, "SDF$\hat{3}$: SDF For Free," in *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, Jun. 2006, pp. 276–278. DOI: 10.1109/ACSD.2006.23.

[3] T. Stefanov, A. Pimentel, and H. Nikolov, "DAEDALUS: System-Level Design Methodology for Streaming Multiprocessor Embedded Systems on Chips," in *Handbook of Hardware/Software Codesign*, S. Ha and J. Teich, Eds., Dordrecht: Springer Netherlands, 2017, pp. 983–1018, ISBN: 978-94-017-7267-9. DOI: 10.1007/978-94-017-7267-9_30. [Online]. Available: https://doi.org/10.1007/978-94-017-7267-9_30 (visited on 04/26/2023).

[4] I. Sander and A. Jantsch, "System modeling and transformational design refinement in ForSyDe," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 1, pp. 17–32, 2004.

[5] K. Rosvall and I. Sander, "A constraint-based design space exploration framework for real-time applications on MPSoCs," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2014, pp. 1–6. DOI: 10.7873/DATE.2014.339.

[6] R. Jordão, I. Sander, and M. Becker, "Formulation of Design Space Exploration Problems by Composable Design Space Identification," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Feb. 2021, pp. 1204–1207. DOI: 10.23919/DATE51398.2021.9474082.

[7] P. W. consortium, "D7.2 - Requirements and evaluation criteria," p. 50. [Online]. Available: https://itea4.org/project/workpackage/document/download/7129/D7.2%20-%20Requirements%20and%20evaluation%20criteria%20(improvement%20phase).pdf.

[8] P. W. consortium, "D7.3 - Demonstrators," p. 50.

[9] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti, "Scheduling Dependent Periodic Tasks without Synchronization Mechanisms," in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, Apr. 2010, pp. 301–310. DOI: 10.1109/RTAS.2010.26.

[10] R. Jordão, F. Bahrami, R. Chen, and I. Sander, "A multi-view and programming language agnostic framework for model-driven engineering," in *2022 Forum on Specification & Design Languages (FDL)*, Sep. 2022, pp. 1–8. DOI: 10.1109/FDL56239.2022.9925666.

[11] R. Jordão, T. Granlund, M. Becker, I. Sander, and I. Söderquist, "Design Space Exploration for Safe and Optimal Mapping of Avionics Functionality on Partitioned Integrated Modular Avionics Platforms," in *Accepted for Publication in the 42nd AIAA/IEEE Digital Avionics Systems Conference (DASC)*, Barcelona, Spain, 2023.

[12] K. Rosvall and I. Sander, "Flexible and Tradeoff-Aware Constraint-Based Design Space Exploration for Streaming Applications on Heterogeneous Platforms," *ACM Transactions on Design Automation of Electronic Systems*, vol. 23, no. 2, 21:1–21:26, Nov. 2017, ISSN: 1084-4309. DOI: 10.1145/3133210. [Online]. Available: http://doi.acm.org/10.1145/3133210 (visited on 11/12/2018).

[13] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, 1 Jan. 1987, ISSN: 0018-9340. DOI: 10.1109/TC.1987.5009446.

[14] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. Gulf Professional Publishing, 1999, 1056 pp., ISBN: 978-1-55860-343-1. Google Books: MHfHC4Wf3K0C.

[15] T. Saxena and G. Karsai, "A Meta-Framework for Design Space Exploration," in *2011 18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, Apr. 2011, pp. 71–80. DOI: 10.1109/ECBS.2011.21.

[16] F. Herrera, H. Posadas, P. Peñil, *et al.*, "The COMPLEX methodology for UML/MARTE Modeling and design space exploration of embedded systems," *Journal of Systems Architecture*, vol. 60, no. 1, pp. 55–78, Jan. 1, 2014, ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2013.10.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S138376211300194X (visited on 04/25/2023).

[17] C. Silvano, W. Fornaciari, G. Palermo, *et al.*, "MULTICUBE: Multi-Objective Design Space Exploration of Multi-Core Architectures," in *VLSI 2010 Annual Symposium*, N. Voros, A. Mukherjee, N. Sklavos, K. Masselos, and M. Huebner, Eds., ser. Lecture Notes in Electrical Engineering, Dordrecht: Springer Netherlands, 2011, pp. 47–63, ISBN: 978-94-007-1488-5. DOI: 10.1007/978-94-007-1488-5_4.

[18] A. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 99–112, Feb. 2006, ISSN: 1557-9956. DOI: 10.1109/TC.2006.16.

[19] B. Kienhuis, E. F. Deprettere, P. van der Wolf, and K. Vissers, "A Methodology to Design Programmable Embedded Systems," in *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation — SAMOS*, ser. Lecture Notes in Computer Science, E. F. Deprettere, J. Teich, and S. Vassiliadis, Eds., Berlin, Heidelberg: Springer, 2002, pp. 18–37, ISBN: 978-3-540-45874-6. DOI: 10.1007/3-540-45874-3_2. [Online]. Available: https://doi.org/10.1007/3-540-45874-3_2 (visited on 06/28/2022).

[20] S.-H. Attarzadeh-Niaki and I. Sander, "Automatic construction of models for analytic system-level design space exploration problems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, Mar. 2017, pp. 670–673. DOI: 10.23919/DATE.2017.7927074.

[21] A. Bakshi, V. K. Prasanna, and A. Ledeczi, "MILAN: A Model Based Integrated Simulation Framework for Design of Embedded Systems," in *Proceedings of the 2001 ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems*, ser. OM '01, New York, NY, USA: Association for Computing Machinery, Aug. 1, 2001, pp. 82–93, ISBN: 978-1-58113-426-1. DOI: 10.1145/384198.384210. [Online]. Available: https://dl.acm.org/doi/10.1145/384198.384210 (visited on 04/26/2023).

[22] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis, "Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation," 2002.

[23] M. Mendonca, M. Branco, and D. Cowan, "S.P.L.O.T.: Software product lines online tools," in *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09, New York, NY, USA: Association for Computing Machinery, Oct. 25, 2009, pp. 761–762, ISBN: 978-1-60558-768-4. DOI: 10.1145/1639950.1640002. [Online]. Available: https://dl.acm.org/doi/10.1145/1639950.1640002 (visited on 04/26/2023).

[24] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortes, "FAMA: Tooling a Framework for the Automated Analysis of Feature Models,"

[25] C. André, "Syntax and Semantics of the Clock Constraint Specification Language (CCSL)," 2009. [Online]. Available: http://journals.sagepub.com/doi/10.1177/075910639203700105 (visited on 04/25/2023).

[26] S. Neema, J. Sztipanovits, G. Karsai, and K. Butts, "Constraint-Based Design-Space Exploration and Model Synthesis," in *Embedded Software*, R. Alur and I. Lee, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2003, pp. 290–305, ISBN: 978-3-540-45212-6. DOI: 10.1007/978-3-540-45212-6_19.

[27] K. Rosvall, T. Mohammadat, G. Ungureanu, J. Öberg, and I. Sander, "Exploring Power and Throughput for Dataflow Applications on Predictable NoC Multiprocessors," Aug. 1, 2018, pp. 719–726. DOI: 10.1109/DSD.2018.00011.

[28] M. Geilen, J. Falk, C. Haubelt, T. Basten, B. Theelen, and S. Stuijk, "Performance Analysis of Weakly-Consistent Scenario-Aware Dataflow Graphs," *Journal of Signal Processing Systems*, vol. 87, no. 1, pp. 157–175, 1 Apr. 2017, ISSN: 1939-8018, 1939-8115. DOI: 10.1007/s11265-016-1193-7. [Online]. Available: http://link.springer.com/10.1007/s11265-016-1193-7 (visited on 03/25/2019).

[29] N. Khalilzad, K. Rosvall, and I. Sander, "A Modular Design Space Exploration Framework for Multiprocessor Real-Time Systems," in *2016 Forum on Specification and Design Languages (FDL)*, Sep. 2016, pp. 1–7. DOI: 10.1109/FDL.2016.7880377.