ABSTRACT
In this paper, we present a design space exploration (DSE) method for embedded systems, which represents the design space as a categorical graph product, in order to overcome the challenge of performing multiple DSE activities, such as task mapping, processor allocation, and software binding. Moreover, the method adopts a Model-Driven Engineering (MDE) approach, defining a design space metamodel to represent the categorical graph product and other DSE concepts, such as solutions, costs, and DSE activities. Furthermore, exploiting the MDE approach, we use model-to-model transformation rules to implement the design constraints, which guide and prune the design space. The method is applied to the design of a real-life application, and experiments demonstrate its effectiveness.

Categories and Subject Descriptors
I.6.5 [Model Development]: Modeling methodologies; J.6 [Computer-Aided Engineering]: Computer-aided design; C.3 [Special-purpose and application-based systems]: [Real-time and embedded systems]

General Terms
Design, Languages, Performance, Standardization, Verification

Keywords
Design Space Exploration, Model-Driven Engineering, UML.

1. INTRODUCTION
Modern embedded systems have increased their functionality through the composition of a large amount and diversity of hardware and software components, integrating complex Multi-Processor Systems-on-Chip (MPSoC). During the development of MPSoCs, a wide range of design alternatives arises from different design activities. The combination of alternative designs and stringent requirements unveils a complex design space, which the design team must explore under reduced time-to-market. At a system level, a Design Space Exploration (DSE) activity is performed, by which one looks for different solutions for the mapping between an application and an architectural platform, such that each solution corresponds to a different trade-off regarding design requirements and constraints [10].

From a broader point of view, DSE is performed always when an engineer must choose between multiple design alternatives, which arise at different levels. The DSE activity starts in the early development phases, when system requirements are used to define the earliest design artifacts. Distribution of responsibilities between components and definition of the interactions between them, as well as the algorithms to be executed, are some of the first design decisions. Most common design activities, which have automated support by some DSE tools, include definition of schedulable tasks, hardware/software partitioning, mapping of tasks to processors, binding of software to memories, and others. At a lower abstraction level, values for specific configuration parameters of system components, such as cache type and size, pipeline, bus width, and software precision, must be defined in order to tune the system implementation.

All those activities can be performed at different abstraction levels and have different levels of support by a wide range of DSE tools. Each tool requires its own evaluation method, input languages, and output format, so that setting up a tool chain to support all of those design activities is a complex task. Moreover, there is no common understanding about the execution order of those design activities, due to the strong interdependencies between them.

In order to cope with the complexity of the DSE process at earlier design steps, this work defines an abstract design space representation and the support of automated DSE activities by exploiting the Model-Driven Engineering (MDE) [17] approach. First, we define a design space abstraction as categorical graph products, dealing with the interdependence between design activities. We then specify a design space domain metamodel using the
Eclipse EMF [6] project tools, so that it can be used in well-adopted MDE tools, during the development process. Exploiting the MDE approach, we use model-to-model transformation rules to specify the constraints, which guide the exploration process and prune the design space.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 presents the definition of our design space abstraction and its metamodel. Section 4 describes how model-to-model transformation rules are used, in order to guide the design space exploration process. The supporting tools are introduced in Section 5. In Section 6 a case study illustrates the method, applying it to a real-life application design. Section 7 draws main conclusions and introduces future work.

2. RELATED WORK
After more than 20 years, DSE methods still represent an important research topic and many works can be found in the literature, as observed in reviews on DSE [8] and co-design [5].

The design space is the set of candidate designs, from where an engineer should select at least one candidate to implement the system. Each DSE approach has its own way to represent the design space, which depends on supported design activities and optimization algorithms. The representation tries to either expose some alternative properties or facilitate the optimization process.

The most common alternative to represent the design space is the definition of a direct mapping between graphs representing the application and architecture elements, as proposed by Blickle et al. [3]. Following [3], a specification graph consists of compositions of dependence graphs, which represent the application and the architecture models. In the specification graph, each dependence graph is mapped onto another one by mapping edges. The specification graph defines the design space and user constraints for allocation, binding, and scheduling. The term “activations” specifies the set of nodes and edges that are active in the specification graph. A set of active nodes and edges represents an implementation, i.e., a candidate or final solution. After its definition, the specification graph is codified in genes as an internal representation, such that a genetic algorithm may perform an optimized definition of the activation of nodes and vertices. Besides the mapping edges, equations define additional constraints usually related to the design candidate quality, which could be freely defined. However, to the best of our knowledge there is no tool implementing this method, providing either an open API or other mechanism to specify user constraints.

Erbas et al. [7] define the design space by equations, which are composed to form the function to be optimized. The variables are then encoded in genes for a Strength Pareto Evolutionary Algorithm (SPEA). As in [3], the constraints are equations that must be satisfied during the generation of candidates or after their evaluation. However, differently from [3], in [7] the constraints are fixed, according to the problem defined for the design space exploration tool, and restricted to define legal candidates.

A similar approach for direct mapping is implemented by MILAN [1], where the mapping model specifies the available mappings of each dataflow component in the application model onto components at the resource model (architectural model). The mapping is represented by one or more references between components in both the application and architecture models. These multiple references represent the design choices available during the design space exploration. Additionally, values for performance and power can be attached to the references in order to guide the exploration algorithm. The mapping model also determines the channel that implements the communication between dataflow tasks.

The DESERT tool [12], as our approach, also defines a design space metamodel. However, differently from ours, its metamodel represents concepts only for component composition. The design space metamodel consists of references to a library of Simulink components, each reference containing constraints with name and expression. The expressions are defined in OCL and can be freely specified by the user at the moment a component is inserted in the library. The DESERT tool focuses on the exploration of this library, and the constraints mainly impose restrictions on the composition.

The Koski tool [9] uses design constraints specified in the system UML model. Constraints related to real-time properties are defined as tagged values and used to remove unfeasible candidates. The constraints related to the architectural platform are defined as subclasses of a Constraints class, contained by the element to be constrained. Each subclass is a constraint associated to a field of the constrained class. The fields of the constraint class specify the bounds to be satisfied. The internal representation of the design space is not described. Mapping, allocation and scheduling can be directly defined in the UML system model and are used as a start point to the optimization process.

The DaRT project [4] proposes an approach based on MDA for the design of SoCs, where metamodels are defined to specify applications, architectures, and associations between hardware and software. Motel-to-model transformations are used to refine a Platform Independent Model into a Platform Specific Model expressed as a SystemC metamodel. It is possible to refactor an application model in order to better adequate it to a given architecture model. However, in DaRT there is no strategy for design space exploration based on these transformations, and the main focus is just the generation of code for simulation at TLM and RT levels from the models.

What all those methods have in common is the fact that the design space is restricted, according to the activity to be performed. Moreover, the generation of candidate designs is internally implemented, usually as a function that is programmed directly in the tool. As a result, no extension mechanisms are provided, requiring multiple tools to support each design activity. Moreover, except for the Koski and DESERT methods, for most approaches either the constraints set is restricted to previous constraints implemented by the tool or the method supports limited constraints constructs.

The method proposed in this paper overcomes those restrictions by defining a design space abstraction, as a categorical graph product [18]. Besides the automatic construction of the design space, performed by the product of graphs, this abstraction provides a common representation for multiple design activities. Moreover, the specification of a metamodel using a well-adopted technology allows us to exploit the MDE approach, as model-to-model transformation rules are used to implement any user constraints, improving the flexibility of the DSE tool.
3. DESIGN SPACE REPRESENTATION

3.1 Design Space Abstraction

Similar to most DSE and optimization approaches we explicitly define the design space as a mapping of graphs. However, differently from the common approach presented in [3], which is a manual mapping between semantically defined graphs, our approach uses the categorical graph product [18], automatically generating specific representations of these graphs. This abstraction is free of any specific semantics from the view of the DSE tools. In the following we define the representation of the design space.

Consider \( G = \langle V, E, \partial_0, \partial_1 \rangle \) as a graph, where \( V \) is the set of all vertices of \( G \); \( E \) is the set of all edges of \( G \); \( \partial_1 : E \rightarrow V \) is the source function of an edge; and \( \partial_0 : E \rightarrow V \) is the target function of an edge. Let \( S \) be the set of graphs, where \( G_i = \langle E_i, V_i, \partial_0, \partial_1 \rangle \in S \), \( i \in \{1..n\} \) and \( n \) is the number of graphs in \( S \). This set is formed from graphs that are extracted from the design information, such as a task graph, an architectural graph, the communication structure of buses, and others. How these graphs are extracted from design models will be explained later. However, the specific semantics of each graph is not considered during the generation of the design space abstraction. The specific semantics of these graphs is considered in the exploration problem, as explained in the next section.

The design space is a graph \( D \) resulting from the categorical graph product of the sequence of terms, which are all graphs in \( S \). In this fashion, \( D = G_1 \times G_2 \times \ldots \times G_n = \langle V_1 \times V_2 \times \ldots \times V_n, E_1 \times E_2 \times \ldots \times E_n, \partial_{01} \times \partial_{02} \times \ldots \times \partial_{0n}, \partial_{11} \times \partial_{12} \times \ldots \times \partial_{1n} \rangle \)

represents the graph product between \( G_1, G_2, \ldots, G_n \), where \( \{\partial_{0k} \times \partial_{0k+1} \times \ldots \times \partial_{0n} | k \in \{0, 1\}\} \) are unambiguously induced by the dot product between vertices and edges, considering that any two vertices \((u_k, u_{k+1}, \ldots, u_n)\) and \( (v_0, v_1, v_2, \ldots, v_n) \) are adjacent in \( D \), if and only if if \( u_i \) is adjacent with \( v_i \) in \( G_i \), \( u_{i+1} \) is adjacent with \( v_{i+1} \) in \( G_{i+1} \) and \( u_n \) is adjacent with \( v_n \) in \( G_n \), \( i \in \{1..n-1\} \), where \( n \) is the number of graphs in \( S \).

Each product of the sequence \( G_1 \times G_2 \times \ldots \times G_n \) that constitutes \( D \) represents a design activity, such as task mapping, processor selection, processor allocation, voltage scaling selection, etc., such that vertices in \( D \) are design decisions and edges in \( D \) are design alternatives available at a specific vertex of \( D \). The projection function \( \pi_x = \langle \pi_E, \pi_V \rangle : G_i \times G_{i+1} \rightarrow G_i \) is defined and returns the graph \( G_i \) involved in the product. Using this abstraction, a graph \( G' \) is a sub-graph of \( D \) and represents a candidate design.

Illustrating our approach, consider as a simple example two graphs \( T \) and \( P \). Graph \( T \) (Figure 1-a) represents a task graph where the vertices are tasks and edges specify the data dependences between them. Graph \( P \) (Figure 1-b) represents the processing units and the allowed communications between them. Graph \( T \times P \) in Figure 1-c is the categorical graph product between \( T \) and \( P \), representing a design space for the task mapping design activity. The vertices in \( T \times P \) represent design decisions (e.g. vertex \( \langle T1, P1 \rangle \) specifies that task \( T1 \) should be mapped into processor \( P1 \)), and edges identify the available design alternatives at a specific vertex (e.g. after selecting the vertex \( \langle T2, P2 \rangle \), which maps task \( T2 \) into processor \( P2 \)), the available design decisions are \( \langle T3, P1 \rangle, \langle T3, P2 \rangle, \langle T4, P1 \rangle \) and \( \langle T4, P2 \rangle \).

Using the categorical graph product abstraction, the design space exploration problem consists in searching for sub-graphs, which represent candidate designs, independent of the design exploration activities to be performed. Figure 1-d illustrates a selected sub-graph composed by vertices \( \langle T1, P1 \rangle, \langle T2, P2 \rangle, \langle T3, P2 \rangle, \) and \( \langle T4, P2 \rangle \). The procedure to select vertices in order to produce the sub-graph must be guided by an optimization algorithm and a set of exploration rules, which search on the design space graph for candidate sub-graphs.
multiple design decisions at the same time. This abstraction also exposes the communication (dependencies) between elements and is well suited to combine the communication in multiple hierarchies, such as classes, task, processors, and systems.

3.2 Design Space Metamodel

To implement the design space abstraction in a DSE tool and integrate this tool to an MDE environment, we have defined a metamodel to represent the design space abstraction and provide elements to support DSE in a more general approach. The metamodel was defined considering MDE concepts that help an engineer during the DSE phase, such as orthogonalization of concepts, abstraction, and automation. It is important to highlight that this metamodel is proposed to represent a design space in abstract fashion and gauge the DSE process, instead of direct representing a system design. Therefore, it must be used to complement a design language, such as UML, Simulink, or any DSL. This design space metamodel is shown in Figure 3.

Figure 3. Design space exploration domain metamodel

The root container in this metamodel is DSEDomain, which is a container for all elements related to DSE. It inherits properties from DSEModelElement as all other elements in this metamodel. The generalization was omitted to keep the diagram clear. DSEDomain contains DSEProblems, which define a DSE scenario. DSEProblem contains a list of DesignGraphs extracted from design models. A DesignGraph contains vertices and edges, where vertices are ExplorableElements, and Edge represents the dependencies between vertices. ExplorableElement is a reference to a design element from which the DesignGraph is generated. This reference is important to hook the DSE elements to the design model and allows the metamodel to be attached to multiple models, such as UML, Simulink, and others. Currently, this reference is implemented by holding the name of the design element as a field of ExplorableElement and using queries to find the instance of the design element in the design repository (e.g. UML model). This implementation could be improved, but it is important to evaluate factors such as performance, increase of dependence between metamodels, and traceability of design elements. DSEProblem also contains a list of Objectives, which are the values to be optimized, defined by their name and unit. DesignSpace represents the categorical graph product and contains the available DesignDecisions, Alternatives link the allowed DesignDecisions. DesignDecision is a tuple of n vertices from the DesignGraphs. It contains an instance of DesignGraph as a key and an instance of Vertex as a value, so that it can map a design decision to the ExplorableElements represented in the DesignGraphs. A list of DSESolutions represents candidate designs, which are subgraphs of the design space graph. A DSESolution must conform to ExplorationRules. A DSESolution has its costs defined in the ObjectiveToCostMap, acquired from an estimation/simulation process.

The list of ExplorationRules defines rules, which guide the DSE tool and prune the design space. These rules are expressed as in any graph grammar, to efficiently handle the design space model. Actually, these rules can be implemented in any model-to-model transformation language, as QVto, QVTr, Xtend, ATL, and others. As the current implementation of the metamodel uses the Eclipse EMF technology, the restriction is that the transformation engine/language must support Ecore based metamodels. However, EMF is largely supported nowadays. Because there are some mature or standard transformation languages, there is no reason to have a real instance of ExplorationRules. However, we are planning to use this element as a facade to configure transformation rules, depending on the DSEProblem configuration. Using model-to-model transformation rules, one can write constraints that directly handle the concepts of the design, such as processors, tasks, slots, voltage level, and others. Besides the constraints, these transformation rules are used to guide the DSE process, as they are used directly as a source to generate a candidate solution. This approach allows a generic use of the DSE tool for multiple DSE activities, where the user can freely define the transformations to be applied on the design space, which characterize the semantics of the graphs and the DSE activity.

4. Exploration Rules

In our approach, exploration rules are model-to-model transformation rules, which receive an instance of raw DesignSpace (i.e. unconstrained design space) as input and generate a constrained DesignSpace instance as output. These rules are constraints to guide and prune the available design space, to reduce the exploration time and ensure the feasibility of a candidate solution. To ease the identification of rules that must be applied during the DSE process, we classify them into three categories:

A) Structural: These rules are applied to avoid illegal designs, which could appear as a sub-graph of the design space. Typical rules avoid double assignments of an element, e.g. different processors assigned to the same slot in a given communication structure or the same task assigned to different processors. Other rules may ensure that all tasks must be mapped, or at least one processor must be allocated. These rules can specify integration issues, such as two components that could not be integrated in the same system because of incompatibility issues.

B) Non-Functional: Even if a design is feasible, it can be invalidated when checked against non-functional requirements, which must be satisfied by the system. This way, these rules avoid the violation of requirements such as task deadlines, maximum delays, and maximum energy consumption. One challenge to the DSE process is to deal with system metrics, which cannot be partially evaluated. As such, the DSE process may postpone the design space pruning to a second step, after system evaluation, when it could filter the candidates from the design space. This procedure avoids selecting the candidate again, by removing design alternatives that may cause requirements violation.
C) Pre-defined Design Decisions: Designs usually start with pre-defined design decisions and previously developed components, and the selected platform may impose restrictions, which an engineer must respect. Moreover, engineer's experience may influence how the automated DSE process proceeds. Therefore, these rules are specified in cases where one needs to interfere on the DSE process through specific design decisions. Typical rules define specific task mapping, processor allocation, specific task or processor execution frequency, and others.

Considering this classification, the user of the DSE method is expected to define some rules for each category, which apply to his/her specific DSEProblem. The rules can be specified in any model-to-model transformation language that supports ECORE metamodels. A small effort is required to migrate the DSE tool to use a different language. First one needs to configure the DSE tool to load the appropriate file containing the exploration rules in the desired language. Then, it may be required to implement the interface needed to call the transformation engine, for the case that the configured engine does not support the desired language.

To alleviate the user effort, a set of typical rules was implemented and is provided as a library to the user. The current available rules are:

- **Duplicated Processor Assignment:** Avoids assigning different processors to the same slot in a given communication structure.

- **Multiple Assignments of a Processor:** Avoids assigning the same processor to different slots in a given communication structure.

- **Multiple Assignments of a Task:** Avoids assigning the same task to different processors.

- **Map All Tasks:** Ensures that every task in the system is mapped to at least one processor.

- **Lower / Upper Performance / Power / Memory / Communication Value:** Defines the lower or upper values for performance, power, memory, or communication amount for a task.

- **Maximum Processor Occupation:** Defines an upper limit for occupation of a processor, so that a schedulability test can be satisfied.

- **Task Deadline Violation:** Verifies the timing properties of a task and removes the candidate from the population if there is a deadline violation.

- **Specific Task Mapping:** Defines that a task (active object/thread) must execute in a specific processor.

- **Specific Processor Allocation:** Defines that a processor must be allocated to a specific position/slot in a communication structure.

- **Specific Processor Selection:** Defines the processor type that must be selected to implement the candidate design.

- **Specific Task Execution Frequency:** Defines the frequency at which a processor must execute for a specific task.

- **Specific Processor Execution Frequency:** Defines the frequency at which a processor must execute.

Those rules are parameterized, and the user can pass the instance names, when calling a specific rule. Additional rules can be specified, referring to the instances of the DSE metamodel.

Furthermore, for the rules contained in that library, the DSE tool implements a generator, based on UML/MARTE [15] models, so that the user is not required to manually specify those exploration rules. To implement this generation, some UML/MARTE stereotypes and tags are applied in class, component, and sequence diagrams. Table 1 presents a simplified view of the mapping from UML/MARTE constructs to Exploration Rules. The <<nfp>> stereotype must be used to define the data type of the non-functional properties to be evaluated (including objectives to be optimized). <<nfpConstraint>> is required to define performance, power, memory, and communication data types used in the specification of a <<nfpConstraint>>. The tag type of the <<nfpConstraint>> stereotype must specify a required value for the property. These constructs can be applied to different elements, such as tasks and processors, and are mapped to the rules Lower/Upper Performance/Power/Memory/Communication and Maximum Processor Occupation. The Task Deadline Violation rule requires a task identified using the <<rtUnit>> stereotype, and its behavior identified using the stereotype <<rtf>> with its timing information, namely, type of occurrence, period, and relative deadline specified by the tags occKind, period, and relDL, respectively.

The rule Specific Task Mapping is mapped from the stereotype <<Allocated>>, which must be applied to an association between a task identified by the <<rtUnit>> stereotype and a processor, stereotyped with <<hwProcessor>>. The connection between a communication structure identified as <<hwBus>>, through its ports to a processor, defines the Specific Processor Allocation rule. To define a Specific Processor Selection rule, the user just needs to instantiate a class stereotyped with <<hwProcessor>>. The stereotype <<hwClock>>, together with the frequency tag, can be associated with a task or a processor, in order to define the Specific Task Execution Frequency or the Specific Processor Execution Frequency.

<table>
<thead>
<tr>
<th>UML/MARTE</th>
<th>Exploration Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;nfp&gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;nfpConstraint&gt;&gt;</td>
<td>Lower/Upper Performance/Power/Memory/Communication</td>
</tr>
<tr>
<td>(kind=required)</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;nfpConstraint&gt;&gt;</td>
<td>Maximum Processor Occupation</td>
</tr>
<tr>
<td>&lt;&lt;rtUnit&gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;rtf&gt;&gt;</td>
<td>Task Deadline Violation</td>
</tr>
<tr>
<td>(occKind, period, relDL)</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;Allocated&gt;&gt;</td>
<td>Specific Task Mapping</td>
</tr>
<tr>
<td>&lt;&lt;hwBus&gt;&gt;</td>
<td>Specific Processor Allocation</td>
</tr>
<tr>
<td>&lt;&lt;hwProcessor&gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>Port Connection</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;hwClock&gt;&gt;</td>
<td>Specific Processor Selection</td>
</tr>
<tr>
<td>(frequency)</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;hwClock&gt;&gt;</td>
<td>Specific Task Execution Frequency</td>
</tr>
<tr>
<td>(frequency)</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;hwClock&gt;&gt;</td>
<td>Specific Processor Execution Frequency</td>
</tr>
</tbody>
</table>
5. SUPPORTING TOOLS

The method presented in this paper extends the High-level Design Space Exploration tool (H-SPEX) [13], which implements the design space abstraction method as the DSE metamodel. The DSE metamodel was defined using the Eclipse EMF technology and is an ECORE model. The inputs to the DSE tool can be specified using the DSE metamodel editor generated using the EMF tools, or extracted from UML models and configuration files. The design information is extracted from UML models through model-to-model transformations, which extract the required information. Currently, this is performed by the MODES framework [11], which works as a design repository and provides the design information extracted from UML models.

The library of exploration rules is an extension in the Xtend language, from the openArchitectureWare [16]. The exploration rules can be specified using the Eclipse Modeling distribution, which contains the editor for Xtend.

The DSE tool also requires an optimization algorithm and an evaluation tool. We implemented two algorithms, in order to provide an optimization step during the candidates’ generation: Crowding Population-based Ant Colony Optimization for Multi-Objective (CPACO-MO) [2] and Random. Actually, the DSE tool is not limited to these algorithms, and we are planning to integrate this tool to some optimization library to improve the optimization support with analysis and graphical features. The optimization is observed as a black-box transformation, which uses an API to communicate information between the transformation engine and the optimization algorithm. As evaluation tool, we use SPEU [14], a static analysis tool based on UML models, as it provides a fast evaluation step, which is the bottleneck of the DSE process. However, any other evaluation tool could be used, since the evaluation and DSE tools can exchange data. Currently, this is done by assigning the costs for a DSESolution, which can be performed by model-to-model transformations or using the API generated by the EMF tool.

6. CASE STUDY

6.1 A Design Space Exploration Scenario

In order to illustrate the proposed DSE method, this section presents a DSE scenario for a real application, concerning the automated control of a wheelchair. The application model consists of one use case diagram, one class diagram, 18 interaction diagrams, and one deployment diagram. Figure 4 illustrates the wheelchair system and its functional components.

![Wheelchair system diagram](image)

Figure 4. Wheelchair control system overview

In the automatic DSE process performed in this scenario, the DSE tool was configured to perform following design tasks: (i) definition of which objects are active or passive (runnables), among the 17 behaviors defined in the Interaction Graphs; (ii) deployment of the active objects to selected processors (up to 6 processors); (iii) allocation of the selected processors into a hierarchical bus with two segments; and (iv) processor voltage scaling with 4 distinct voltage levels. Exploring all these activities simultaneously, the DSE tool was configured to optimize the system in terms of performance (cycles), power (µWatt), energy (µJoules), total memory (bytes), and communication volume (bytes, transmitted in the bus).

The candidate population was found after 5,000 evaluations and represents the non-dominated set of candidate designs. Figure 5 illustrates these results. The best overall candidate must be selected after a trade-off analysis between the obtained estimations and based on some criteria, such as weights for the optimized objectives, or any other design feature.

![Normalized exploration between objectives](image)

Figure 5. Normalized design space exploration results with five objectives: performance (+), power ( ), total memory (x), energy ( *), and communication (o).

The design space in this case study contains 2,064 alternative design decisions (vertices) and 334,080 edges, from where a set up to 17 (maximum active task distribution) vertices must be selected to define a candidate design solution (sub-graph). The unveiled design space presents more than $5.89 \times 10^{17}$ alternative designs, considering an unrestricted design space (fully connected graph). However, in this proposal, edges guide the available alternatives, and constraints, specified as model-to-model transformation rules, are locally applied between the current vertex and its neighbors, thus pruning the design space and speeding up the DSE process.

Let a task drawn from the wheelchair case study be identified as T15, which implements a stereovision function (in Figure 4, T15 corresponds to the “Correlation-based + Median Filters” vertex), presenting heavy image processing algorithms. Figure 6-A shows a diagram specifying that the DSE tool must map Task 15 into the DSP processor P0, benefiting from the DSP processor architecture. Figure 6-B shows the Specific Mapping constraint, which is automatically applied when a deployment diagram is specified.
DesignSpace specificTaskMapping{
    DesignDecision v1, DesignDecision v2, DesignSpace inDesignSpace
    String task,
    String processor:
    let t2 = v2.get("TASK"):
    let p2 = v2.get("PROCESSOR")
    ((t2 == getTask(task)) &&
    (p2 != getProcessor(processor)) ?
    inDesignSpace.removeEdge(v1,v2):
    null -> this;
}

Figure 6. Sample of design constraints

Let consider a vertex from design space graph be the tuple 
<T13, P1, C1, V2>, which specifies that task T13 must be
mapped to processor P1, while P1 must be allocated to com-
unication bus C1 and execute T13 with voltage level V2. There are
48 alternatives at this vertex. Figure 7 illustrates a partial graph,
representing the design space at this vertex, which is located at
the center. The shadowed vertices around the vertex <T13, P1,
C1, V2> in the centre are pruned nodes, and the white nodes are
the alternative designs that satisfy all constraints.

Figure 7. Sample of a partial design space graph

Applying the structural constraints presented in Section 4
and the sample design constraint here defined, the pruning process
has reduced the design space by 83 % on the specific vertex,
avoiding wasting time with unnecessary evaluations and unfeasi-
ble designs, thus focusing the search for an adequate solution on
the most relevant design points.

6.2 Design Space Exploration Evaluation

Experiments were obtained with an Intel® Core™ 2 Quad
2.4 GHz processor with 2 GB RAM, running Microsoft Windows
XP Professional with Service Pack 2 operating system. We per-
formed 100 iterations (i) of each experiment. The number of gen-
erated candidate designs (m) was set to 50, thus resulting in 5,000
evaluations for each experiment, and the population size (p) was
set to 20 (only the non-dominant candidates).

Figure 8 presents the total execution time for 100 iterations
of the optimization algorithm, obtained when the number of ob-
jectives being explored varies from 1 to 7. The number of activi-
ties (a) was fixed at 2. The activities are task mapping and proces-
sor allocation. The objectives to be optimized are: performance, energy, power, program memory, data memory, total memory, and communication volume.

For one single objective, the circle represents the average
value of the execution time for the seven distinct objectives, while
the top and bottom bars represent the maximum and minimum
execution time obtained in the exploration of these seven objec-
tives, respectively. For two to six objectives, the circle represents
the average value of the execution time of three randomly taken
combinations of n objectives, where n is the number of objectives
to explore (e.g. if n = 2, one possible combination is <performance, energy>), and the top and bottom bars represent the max-
imum and minimum execution time obtained for the exploration,
respectively. In the case of seven objectives, the indicated value
represents the execution time of the only possible single explora-
tion combination for those seven objectives.

The results, varying from 11.0 to 11.5 minutes for the execu-
tion time, show an almost constant value for the exploration ex-
cution time, no matter how many objectives are being explored.
It can be concluded that the proposed algorithm is robust regard-
ting the increase of objectives being explored. Actually, for each
additional objective, the exploration algorithm requires only four
additional if-then-else commands and one variable assignment.

Table 2 presents results obtained when we vary the number
of activities being explored with 100 optimization iterations. The number of objectives (k) was fixed at 2. The activities explored
are definition of runnables, deployment, allocation, and voltage
scaling, with their parameters equal to the previous DSE scenario
presented in Section 6.1. The fixed objectives are performance and power.

Table 2. Exploration with 2 to 4 activities (a)

<table>
<thead>
<tr>
<th>Activities</th>
<th>Exec.time</th>
<th># vertices</th>
<th># edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runnable and deployment</td>
<td>0.1478</td>
<td>172</td>
<td>2320</td>
</tr>
<tr>
<td>Runnable, deployment, and allocation</td>
<td>0.2175</td>
<td>344</td>
<td>9280</td>
</tr>
<tr>
<td>Runnable, deployment, allocation, and scaling</td>
<td>1.0</td>
<td>1376</td>
<td>148480</td>
</tr>
</tbody>
</table>

In the second column, execution times were normalized to
the longest one, i.e. when the DSE tool is executed with four ac-
tivities. The numbers of vertices and edges account for the size
of the graph product. Results in Table 2 show that design space
exploration is highly dependent on the number of exploration activi-
ties taken into account.
7. CONCLUSION
This work has proposed a new design space abstraction based on the categorical graph product. This abstraction overcomes the challenge to deal with interdependences between design activities and provides a flexible representation for multiple design activities. Moreover, this representation decouples the exploration algorithm from the design space, always keeping the same exploration problem view regardless of the optimization algorithm, namely to find a sub-graph that optimizes a set of objectives. This abstraction is well suited for automatic exploration tools, which can take advantage from the MDE approach, as the graphs are easily handled by model-to-model transformation rules. Considering this fact, a DSE metamodel was defined, so that the design space could be easy handled by MDE transformation engines using their transformation rules. These rules are used to implement design constraints that prune the design space and generate the candidate design, thus improving DSE results. Moreover, the non-functional requirements are used to generate the additional transformation rules, which remove unfeasible designs from the design space, thus saving time with unnecessary evaluations.

The proposed method integrates the core of the DSE steps, namely candidate solution generation and design space pruning, into an MDE methodology and opens new opportunities to improve the DSE process through the exploitation of MDE. In the future, we are planning to extend the DSE metamodel to be more integrated to some OMG metamodels, such as OCL, QVT, and/or MARTE, so that a user could use OCL expressions and a standard transformation engine, improving the specification of the exploration rules. These rules are used to implement design constraints that prune the design space and generate the candidate design, thus improving DSE results. Moreover, the non-functional requirements are used to generate the additional transformation rules, which remove unfeasible designs from the design space, thus saving time with unnecessary evaluations.

ACKNOWLEDGEMENTS
The work described herein was partly funded by the German Ministry of Education and Research through the SANITAS project(01M3088).

REFERENCES