Developing a holistic modeling approach for search-based system architecting

Renzhong Wang*, Cihan H. Dagli

Abstract

This paper proposes a holistic modeling approach that combines the capabilities of Object Process Methodology (OPM), Colored Petri Net (CPN), and feature model. The resultant holistic model not only can capture the structural, behavioral, and dynamic aspects of a system, allowing simulation and strong analysis methods to be applied, it can also specify the architectural design space. This modeling approach is developed to facilitate the implementation of search-based system architecting where search algorithms are used to explore design trade space for good architecture alternatives. Such architecting approach integrates certain model construction, alternative generation, simulation, and assessment processes into a coherent and automated framework. Both the proposed holistic modeling approach and the search-based architecting framework are generic. They are targeted at systems that can be specified by conceptual models using object-oriented or process-oriented paradigms. The broad applicability of the proposed approach is demonstrated with the configuration of reconfigurable manufacturing systems (RMSs) under multi-objective optimization as an example. The test results showed that the proposed modeling approach could cover a huge number of architecture alternatives and supported the assessment of several performance measures. A set of quality results was obtained after running the optimization algorithm following the proposed search-based architecting framework.

* Corresponding author. Tel.: +1-573-308-9439.
E-mail address: rwkb4@mail.mst.edu.

Keywords: Object oriented modeling; Simulation; System analysis and design; System architecting

1. Introduction

Traditional system architecture design, analysis and development approaches and the modeling, analysis and simulation tools developed for them usually only focus on a single system model or very limited design alternatives. Trade-off studies, as a separate process, are only conducted on simplified system model using partial system information. On the other hand, architecture design space is usually vast since fewer constraints have been identified in this stage of design. Architecture design shapes the final form and function of a system and therefore is crucial to the success of the system. Overlooking potential architecture alternatives means loss.

This paper presents a holistic modeling approach that supports automatic generation of design alternatives such that they can be integrated into various search algorithms to find the optimum solutions thus converting architecture
development into a search process. Such search-based architecting is an emerging field in the systems engineering. Search-based approaches developed for architecture development in general are still relatively rare. A Smart Systems Architecting framework is proposed in [1]. It highlights the tasks of applying computational intelligence to architecture trade-off space exploration but provides few implementation details. A generic framework for constructing an evolutionary design model for the design of complex systems is presented in [2]. This framework identifies the architecture modeling tasks for various design states and a set of existing technologies applicable to each design task. Still no implementation is developed. Search-base algorithms applied to software architecture designs have been relatively well-studied. A software development paradigm known as generative programming (GP) is first proposed in the dissertation of Dr. Czarnecki [3] and later become an active research topic in software engineering. GP builds on system-family engineering (also referred to as product-line engineering). It is “about designing and implementing software modules which can be combined to generate specialized and highly optimized systems fulfilling specific requirements” [3]. However, GP focuses on a class of systems within a domain, not necessarily exploring all possible variants. Its major application is software systems.

The evolutionary algorithms and other metaheuristic based algorithms have been broadly applied to many architecture related designs [4]. Most of such applications use no explicit system models or use only very simple system description. Rather, the focus is to develop problem specific chromosome representations and crossover/mutation operators. For example, the Genetic Algorithm (GA) is applied to software architecture design in [5]. In this work, the chromosome is comprised of a list of supergenes, each which corresponds to a responsibility in the system. Each responsibility is described by a set of attributes, has a set of responsibilities depend on it, and is associated with a class. Such type of chromosome encodes the complete information of a system into a chromosome eliminating the need of both architectural models and extra alternative generation mechanism (since mutation and crossover operators can be used instead). The disadvantage is that such chromosome encoding cannot generalize well for use in non-software systems. Another problem-specific application of GA in architecture related problem is presented in [6], where GA was applied to dynamic and multiple criteria web-site optimizations. Again, no system model is used. The chromosome representation is simply a sequence of web-objects described by a look-up table.

Effective implementation of the search-based system architecting framework requires a holistic system model that both captures all design information and supports system analyses. Such kind of modeling approach has not been well-studied yet. A meta-language for systems architecting called object-process network (OPN) was developed by Koo in [7] with software implementation. It is a Petri net like executable language that utilizes a small set of linguistic primitives, i.e., objects and processes that transform objects. Koo [7] suggested three usage of OPN: (i) as a declarative language to specify the space of architectural options, (ii) as an imperative language to create architectural option instances and to compute the performance metrics for those instances, and (iii) as a simulation language. The rationale behind usage (i) and (ii) is an analogue of defining classes and creating instances. Therefore, its variability generation mechanism, like that in object-oriented analysis/design, is limited to the intra-application variability (i.e., creating object variants only) as pointed out by [3]. It is not as capable of modeling both variations and constraints as feature models and the domain engineering [3], [8]. Thus, although OPN is effective in creating element instances, it still lacks an effective way to automatically generate entire architectures as alternatives.

This paper presents a holistic modeling approach that is achieved by combining the capabilities of object-process methodology (OPM), colored Petri net (CPN) and feature models. OPM developed by Dori [9] is a visual modeling language with a single diagrammatic view and a small set of symbols consists of objects, processes, and a variety of relational links connecting them. OPM can be used to specify both structural and behavioral aspects of a system. A Petri net [10] is a directed bi-partite graph that uses tokens to mark the state of a system with passive nodes called places to store tokens and active nodes called transitions to move tokens between places. Feature models [8], [11] are widely used in software product line engineering. A feature model is represented as a hierarchically arranged set of features composed by: (1) relationships between a parent (or compound) feature and its child features (or subfeatures); (2) cross-tree constraints that are typically inclusion or exclusion statements. A basic feature model has the following relationships among features [8]:

- Mandatory: the child is included in all products in which its parent feature appears.
- Optional: the child can be optionally included in all products in which its parent feature appears.
- Alternative: only one feature of the children can be selected when its parent feature is part of the product.
- Or: one or more of children can be included in the products in which its parent feature appears.

The remaining of the paper is structured as follows: Section 2 presents the search-based architecting framework.
Section 3 first investigates the drawbacks and open issues in current modeling languages and then a holistic modeling approach is presented. Section 4 briefly describes the generic implementation of the proposed approach. Section 5 presents a case study along with test results on the design of a reconfigurable manufacturing system (RMS). Finally, Section 6 outlines some concludes.

2. Search-based Architecting Framework

There are four distinctive tasks in search-based architecture development: (i) developing an architectural model, (ii) generating architecture instances, (iii) assessing architectural instances, and (iv) validating design and/or further refining design. Figure 4.1 depicts these processes using an OPD. The architecture development cycle follows the requirements analyses. A generative class model that can describe a collection of systems is first developed. An architecture generation mechanism is then applied to generate all of the architecture alternatives within the design space specified. Next, the architecture assessment process proceeds with the three key tasks, analysis, selection, and optimization, each of which is facilitated by a specific type of model.

The analysis process derives the behavioral properties and/or performance measures from the generated architecture alternatives using various analysis methods (or models in general) and/or simulations. Depending on the modeling language used, a system model (e.g., a Petri net model) can sometime double as an analysis model. A design problem usually involves multiple domains, each of which can have one or more analysis models developed. The selection process is facilitated by a decision-making model, which is used in conjunction with the optimization model to select good designs that constitute a desired trade-off between conflicting objectives. The optimization process is primarily a search process in this context. The design variable space, comprised of architecture instances, is discrete in nature and usually is subject to constraints.

The solution from the optimization is subject to verification and validation to ensure the selected architecture alternative(s) can conform to the constraints, perform the intended functionality, generate desired behavior, and satisfy the performance requirements. If further refinement is needed, another round of the design cycle can proceed. Such development process is intended to proceed automatically as the design space might be vast.

3. Holistic Modeling Approach

In the search-based system architecting, the design space is comprised of architectural models, which are actively involved in the assessment and search process. Hence, an integrated architecture model that contains all aspects of information needed for both design and analysis is preferred. Such an architecture development process also requires both a class model to represent the design space and a set of instance models to participate in the computation. Thus there is a need for holistic modeling. Particularly, the concept of holistic modeling in this context is fivefold:

- One integrated model for system specification instead of multiple disjoint diagrams,
- Capture structural, behavioral, and dynamic aspects of the system of interest,
- Capture design space (or constraints),
- Can be used as both static presentation and dynamic simulation, and
- Support system analysis.
3.1. Developing a holistic modeling approach

Existing modeling languages emphasize and excel at only certain aspects of system modeling as the comparison summarized in Table 1 [12]. Based on the literature review [7], [13], a holistic modeling language as introduced in last section has yet to be designed. Instead of developing such a modeling language from scratch, this research proposes the integration and combinational usage of some existing modeling languages, i.e., OPM, CPN and feature model. Such integration not only allows users to benefit from their familiarity with and the advantage of these individual languages but also allows existing software tools and analysis methods developed for them to be reused.

Table 1. Comparison of UML/SysML, OPM, and CPN

<table>
<thead>
<tr>
<th>Aspects</th>
<th>UML/SysML</th>
<th>OPM</th>
<th>Petri Nets</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model format</td>
<td>Graphic</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Text</td>
<td>X</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mathematics</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Model Singularity</td>
<td>X</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>Model Coverage</td>
<td>Structure</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Behavior</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Dynamic</td>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td>Mathematics</td>
<td>O(OC1)1</td>
<td>X</td>
<td>By programming</td>
</tr>
<tr>
<td>Model Capability</td>
<td>Presentation</td>
<td>Good</td>
<td>Excellent</td>
<td>Poor</td>
</tr>
<tr>
<td></td>
<td>Specification</td>
<td>Excellent</td>
<td>Good</td>
<td>Excellent</td>
</tr>
<tr>
<td></td>
<td>Communication</td>
<td>Excellent</td>
<td>Excellent</td>
<td>Poor</td>
</tr>
<tr>
<td></td>
<td>Simulation</td>
<td>Poor</td>
<td>By extension</td>
<td>Excellent</td>
</tr>
<tr>
<td></td>
<td>Analysis</td>
<td>Poor</td>
<td>Poor</td>
<td>Excellent</td>
</tr>
<tr>
<td>Model Notation</td>
<td>Compactnessa</td>
<td>Poor</td>
<td>Good</td>
<td>Excellent</td>
</tr>
<tr>
<td></td>
<td>Usability and convenienceb</td>
<td>Poor</td>
<td>Excellent</td>
<td>Poor</td>
</tr>
<tr>
<td></td>
<td>Advanced expressionc</td>
<td>Excellent</td>
<td>Good</td>
<td>Poor</td>
</tr>
</tbody>
</table>

Fig. 2 illustrates the way that these languages can be integrated. The formal system model is to be specified by OPM for its closeness to holistic modeling. OPM also serves as the hub for integrating other modeling formalisms as it contains a smaller set of modeling primitives and notations that are both easy to comprehend and easy to extend while still maintaining good specification quality. A UML (or SysML) model consisting of multiple diagrams can be generated by either using OPCAT [15] or following some other proposed mapping schemes [16], [17]. A standard OPM model, however, still lacks well-documented execution semantics, the ability to capture dynamic aspects of system behavior, certain numeric properties (e.g., time), and constraints. This research proposes utilizing CPN to formally define the execution semantics of OPM such that the simulation capability and analysis methods developed for CPN can be utilized. Additionally this research proposes incorporating feature model concepts and domain engineering into OPM modeling so that OPM can be used to develop a class model that represents a collection of instance models. The OPM with such extensions is called OPM/H hereafter. Accordingly, this research developed the mapping between these modeling languages. The basic ideas and rationales are summarized as follows.

In software engineering, domain analysis and feature models are used to define product lines. Such concepts can be incorporated into OPM modeling to define the architectural design space. For example, the concept of features can be applied to any model element in an OPM model because features are higher level abstractions. As defined in [18], a feature is a prominent or distinctive user visible aspect, quality, or characteristic of a software system or system. Accordingly, design elements in an architectural model can be categorized as either common or variable
elements. Common elements are always part of a system and, therefore, can be modeled as mandatory features and they are not relevant to the decision making. Variable elements are part of only some systems and, therefore, can be modeled as either optional, alternative, or OR-relationship features. Extended with the concepts of the feature model, OPM can be used to develop the generative class model. A cardinality concept needs to be defined first:

Cardinality is an interval denoted as \([\text{min}, \text{max}]\) applied to an OPM element, where \(\text{min}\) is the lower bound and \(\text{max}\) is the upper bound. Two types of cardinality exist: participation cardinality (corresponding to the feature cardinality in the feature model) and group cardinality (corresponding to the group cardinality in the feature model).

An OPM can be extended with the feature model concepts by following the rules below:

1. A set of alternative things can be grouped and represented by one OPM object (or process, whichever applicable). Fill the value field of this object (or process) with a Boolean expression that is constructed by connecting the values representing alternatives with “XOR” operators. Alternatively, a notation representing a generative function to be implemented can be used in the value field if too many alternatives exist. Optionally, OPM things representing the alternatives can be created and then connected with the parent object (process) using the classification-instantiation links of OPM (necessary if any of the alternatives needs to connect to other OPM things).

2. A set of things with “OR” relationships can be modeled similarly except that the “OR” operators should be used instead and child objects (processes) can be connected to their parent with any applicable OPM structural links.

3. The group cardinality of a feature can be captured by adding a multiplicity attribute to each OPM thing.

4. The mandatory and optional relationships of a feature model can be represented by participation cardinalities in an OPM. Particularly, add a participation constraint attribute to the structural links of OPM. Then apply the above defined cardinality concept to each terminal end of the link. It is known as participation cardinality here. Participation cardinality is a generalization of the mandatory ([1, 1]) and optional ([0, 1]) relationships.

5. The “requires” relationship of a feature model can be expressed by various OPM procedure links or OPM tagged structural links depending on the relationships between these entities in OPM semantics.

6. Other cross-tree constraints between things are represented by OPM tagged structural links.

7. The “OR” and “XOR” relationships between OPM procedure links can be expressed directly using the “OR” or “XOR” notations of OPM.

8. A root node representing the entire system is optional if all of its child nodes have a participant cardinality of \([1, 1]\).

9. Other extended features and constraints can be added to corresponding OPM elements as feature attributes.

Fig. 3 shows a sample feature model for a mobile phone system (adopted from [8]) along with the corresponding OPM/H model. The dimensions of this system, along with their domains, are (i) GPS: \{True, False\}, (ii) Screen: \{Basic, Color, High resolution\}, (iii) Media: \{Camera, MP3, Camera AND MP3, False\).

Fig. 3. (a) A sample feature model ([8]); (b) An OPM model (created by OPCAT) extended with feature model concepts

An OPM/H model also contains extended information to support the construction of a CPN model. Such additional information (link conditions, guard conditions, code segments, time delays, and markings) can be viewed as annotations added to a regular OPM model. Their semantics is pure CPN semantics. The details of such extended information have been omitted for brevity (refer to [12] for details). Only the rational of mapping OPM to CPN is presented briefly as follows. Map OPM processes to CPN transitions. Map OPM attribute objects (objects connected to their parent object using exhibition-characterization links) to CPN color sets. Such color sets thus define the set of class attributes for the OPM object being connected by those attribute objects. Map non-attribute objects that have no states and object states of OPM to CPN places. Map the value(s) of an OPM object to CPN token(s). One or a set
of tokens on a CPN place represents either the existence of an object or an object being at the state represented by that place. The former corresponds to the case that the place is mapped from an OPM object with no state and the token(s) on that place represent alternative objects. The latter corresponds to the case that the place is mapped from an OPM state. An object in the object-oriented modeling is defined by three parts, states, attributes and services (or method, function or process). By following the mapping scheme discussed above, a CPN token can capture the attribute and state part of an object definition. The service part of an object definition can be inferred if the CPN model created from the OPM follows certain naming conventions. For example, an object’s service can be modeled as an OPM process connected to the corresponding OPM object with an exhibition-characterization link. When such process is mapped to a CPN transition, the transition can be named by prefixing the corresponding OPM process name with the corresponding OPM object name. In doing so, the ownership relation between the object and the process can then be inferred. OPM structural links that have no effect on the system dynamics are not mapped to CPN. The details of the procedure for mapping an OPM/H model to a CPN model have been omitted for brevity. Interested readers can refer to [12] for details.

3.2. Architecture generation

An architecture alternative generation mechanism is needed to generate all instance models that covers the entire design space as defined by the generative class model. Such generation operations work on three levels. The most fundamental level of operations apply to a single element. Structural generation operations work on a set of related elements. The system level operations form a complete architecture alternative. Additionally, there is an automatic generation mechanism that enumerates all possible instances dictated by the design space specification.

(1) Generating element instances. This is the most fundamental variant generation operation that generates object instance (from object-oriented sense) for a single model element, which can be any OPM/H construct. This operation is fundamental because it is used in all other variant generation operations proposed in this paper.

(2) Generating structural variants. The second level variant generation operations are based on two primary operations: (i) adds/removes/modifies links between distinct entities (objects, processes, or states) in the system (Operation 1) and (ii) adds/removes/modifies entities (objects, processes, or states) in the system (Operation 2).

The above defined Operation 1 and 2 should be applied in an appropriate order to handle side effects, which include cleaning up both isolated objects and dangling links and removing child objects when removing parent object if they are connected by structural links. Some advanced variant generation operations can also be constructed using these two primary operations, for example (a) object decomposition (adding a sub-system), (b) process decomposition, (c) aggregation (grouping existing system components to create a new subsystem), (d) de-aggregation, (e) breakout (split responsibilities) and (f) merge. With the primary operations, the system can be expanded or shrunk horizontally by adding or removing entities or links whereas these advanced operations achieve vertical scalability (i.e., either refinement or its reverse) by connecting/disconnecting things to a root thing.

(3) Generating full architecture alternative. The above defined variant generation operations should be applied to each applicable dimension of the design space. The entire variable part of an architecture alternative can then be generated by applying the operation of generating element instances defined above to the entire variable part. Finally, the generated variable part is combined with the common part to form a complete architecture alternative.

An automatic mechanism that enumerates the entire set of architecture alternatives is needed when the design space is either very large or very complicated. In the research of automated analysis of feature model, several operations of analysis on feature models have been proposed. These operations can be utilized for both generating architecture alternatives and analyzing a class model. For example the “all products” (or “all valid configurations”, “list of products”) operation can be utilized to generate all alternatives defined by the class model.

4. Generic implementation

A software implementation (written in Python 2.7.3) of the proposed approaches is developed to integrate the development of holistic system models, the generation of architecture alternatives, the calculation of performance metrics, and the search for optimum solutions into one coherent process. It employs (with extensive modifications) two open source libraries: the SNAKES [19], for its CPN support, and the Inspyred [20], for its GA support. The workflow of related activities in applying this implementation is illustrated in Fig. 4 using OPM notations.
5. Application Demonstration

The software implementation presented in last section is generic except for the data pre-processing part which must be problem-specific. Such implementation is applied to the design of RMS to demonstrate its usage.

5.1. Problem definition

The RMS considered here is the flow-line configuration as proposed in [21]. Such an RMS is comprised of a set of stages each of which contains multiple identical stations/machines arranged in parallel with identical operation assignments. The demand scenario under consideration is multiple products with mid-to-large production volumes. In order to facilitate the benchmark comparison this research adopted the case study used in [22]. The same problem definition is used except that multi-objective optimization is assumed here. A second objective, minimizing unit production time (or equivalently maximizing production rate), is added in addition to the minimizing capital cost objective. Readers are encouraged to refer to the original paper for detailed problem definitions. Fig. 5 shows an example of a selected configuration capable of producing two different types of parts.

Fig. 5. Example of a Selected RMS Configuration ([22])

5.2. Architecture modeling

Following the proposed holistic modeling approach, a generative class model is first developed using OPM/H as shown in Fig. 6. The execution semantics of this OPM/H model can be precisely specified using CPN as shown in Fig. 7 (instance model). Since all stages of the RMS share the same structure, only one representation is needed in
this model. Information regarding the configuration of each particular stage is reflected by the instance values (token values) of both Machine object and Part object, which are the only variable elements in this system model. Note that the variable elements in this particular model only involve OPM objects, no processes or links. Modeling a system in such a way is encouraged because it makes generating architecture alternatives much easier.

Fig. 6. OPM/H Model for a RMS (Zoom-in into Manufacturing Process)  
Fig. 7 CPN Model for the RMS

The CPN model is worth a closer look. Initially, tokens representing parts are all at the place P_ready simulating that they are ready to be moved to the next stage and tokens representing machines are all at the place M_Idle. Within each stage, a part goes through the M_Mount, M_Process, M_Unount, and M_HE_Transport processes and iterates like this for all stages that the part should be processed. A part is finished when the corresponding part token reaches the place P_Arrived and when its osseq and osdistr lists are both empty (the head value of the lists are removed when the MHE_Transport fires), signaling no further processing is needed. Although there is only one set of transitions shared by all stages of the RMS in this CPN model, they still can model concurrent behavior because tokens on a place can be currently enabled. The binding of a part to a machine is reflected by mtp and mid attributes. The transition M_Mount has a guard inscription that ensures a part is bound to a machine at the right stage. mbf attribute keeps counting how many spaces left for mounting parts. A machine token is moved from the place M_Idle to the place M_Working when its buffer is full simulating that a machine is fully loaded and begins to process parts. A time delay is added after the transition M_Process is fired representing the time needed to process a part. The difference of a part’s total time stayed in the RMS and its total processing time is the time that a part stayed in the RMS and its total processing time is the time that a part

The CPN model is worth a closer look. Initially, tokens representing parts are all at the place P_ready simulating that they are ready to be moved to the next stage and tokens representing machines are all at the place M_Idle. Within each stage, a part goes through the M_Mount, M_Process, M_Unount, and M_HE_Transport processes and iterates like this for all stages that the part should be processed. A part is finished when the corresponding part token reaches the place P_Arrived and when its osseq and osdistr lists are both empty (the head value of the lists are removed when the MHE_Transport fires), signaling no further processing is needed. Although there is only one set of transitions shared by all stages of the RMS in this CPN model, they still can model concurrent behavior because tokens on a place can be currently enabled. The binding of a part to a machine is reflected by the value of the corresponding part token’s mtp and mid attributes. The transition M_Mount has a guard inscription that ensures a part is bound to a machine at the right stage. A machine token’s mbf attribute keeps counting how many spaces left for mounting parts. A machine token is moved from the place M_Idle to the place M_Working when its buffer is full simulating that a machine is fully loaded and begins to process parts. A time delay is added after the transition M_Process is fired representing the time needed to process a part. The difference of a part’s total time stayed in the RMS and its total processing time is the time that a part spent in waiting. The smaller this time is the more efficient the RMS system is. The idle time (it) and operation time (ot) attributes of a machine token keep tracking the accumulated idle time and operation time, respectively, of the machine represented by the token. These attributes, therefore, can be used to measure resource utilization.

The dimensions of the design space of the RMS configuration, according to the model in Fig. 6, can be expressed as (Machine × Part). The sub-dimensions of Machine are (machine_type(mtp) × machine_configuration(cfg) × stage_assignment(stg) × number_of_machine). The sub dimensions of Part are (part_type(ptp) × os_sequence(osseq) × os_stage_distribution(osdistr)). The remaining attributes for Machine and Part are transit (or dynamic) attributes that should not be counted in the dimensions of the design space since they only make sense when the system is running.

5.3. Architecture assessment

Two quantitative objects are considered here. One is the capital cost of the configuration, which is the sum of the cost of all machines involved in the RMS. The other one, average unit production time, is derived from simulating the CPN model and computed according to \( PT_u = T_{sys} / NP_f \), where \( PT_u \) is unit production time (in seconds), \( T_{sys} \) is the model time of the CPN model when the simulation end, and \( NP_f \) is the number of parts finished, i.e., the number
of tokens at the place $P_{\text{Arrived}}$ with their $\text{osseq}$ and $\text{osdistr}$ lists both empty when the simulation ends. As concluded in [23] and referenced in [22], a special case of this optimization problem with fixed machine configurations, fixed order of operations and no consideration of capacity requirements was proven to be NP-hard. Therefore the GA (particularly, the non-dominated sorting genetic algorithm II (NSGA-II) [24]), as a meta-heuristic global optimization algorithm, is selected as the optimization model.

5.4. Results and discussion

The CPN model for the RMS was initialized with 24 tokens for part A and 36 tokens for part B. Fig. 8 shows the Pareto-front obtained from an optimization run. The user can select one of them as the final solution based on more detailed analyses. This example is only intended to provide such reduced solution space. The string representations of these solutions are provided in Fig. 9. One of them has already been illustrated graphically by Fig. 5.

![Fig. 8. Pareto front obtained for the RMS after running GA](image)

![Fig. 9. String representations of the near-optimum solutions](image)

A huge variety of techniques have been proposed [22], [25]–[31] for solving the RMS configuration problem. All these approaches developed some problem-specific models particularly for RMS, which cannot (or are very hard to) be generalized and applied to other systems. Moreover, all these approaches can only take into account very limited aspects in the objective space and limited factors and design variables in the design space due to the lack of a comprehensive (holistic) system model. An optimization covering limited dimensions of the objective space while ignoring other, potentially critical, objectives tends to be biased. Developing multiple models, each of which having certain aspect(s) of the system optimized, may produce multiple designs that need extra efforts of integration. The optimality of the integrated design is not guaranteed. The holistic modeling approach, along with the search-based architecture development framework proposed in this research allows comprehensive information to be captured and maintained using one integrated system model.

6. Conclusion

The proposed holistic modeling approach combines the full features of OPM, CPN and feature models and utilizes them in a complementary way. Therefore, its expressiveness is the sum of these individual languages. The OPM provides both object-oriented and process-oriented modeling capabilities. Such OPM is supplemented by CPN which provides state-transition-based execution semantics supporting discrete-event system simulation. The simulation capability is an indispensable means to derive certain performance metrics and to conduct behavior analyses. The incorporation of CPN into the architecture modeling allows the developed system model to be also used as an analysis model. A large collection of analysis methods and tools developed for CPN can be utilized for strong model analysis, verification, and validation. The extension of OPM with feature model concepts allows the specification of design space.

A holistic model provides a common foundation to integrate various design activities. By using a holistic model, various design aspects and knowledge from multiple domains can be integrated and represented in one single system model that can be used in multiple design activities. Such integration thus eliminates both the need to transform models between design activities and the efforts to maintain model consistency.

The development of a generative class model and the generation of all instance models enable architectural models to be used as design alternatives in various search algorithms. With such design approach, vast design space
can be explored and evaluated before commitment to more detailed design, thus reducing time, cost, and risks and improving design quality.

All components of the proposed approach are domain independent and problem neutral. Hence, they should be applicable to a broad range of systems that can be specified by conceptual models with object-oriented or process-oriented paradigms. More case studies are needed to further examine the capabilities of the proposed approach.

References


