Model-driven Self-optimization Using Integer Linear Programming and Pseudo-Boolean Optimization

Sebastian Götz, Claas Wilke, Sebastian Richly, Christian Piechnick, Georg Püschel and Uwe Aßmann
Technische Universität Dresden, Software Engineering Group
Dresden, Germany
Email: {sebastian.goetz1,claas.wilke,sebastian.richly,christian.piechnick,goerg.pueschel,uwe.assmann}@tu-dresden.de

***Abstract*—The development of self-optimizing software systems usually requires developers to apply optimization techniques manually, which is time consuming and prone to error. The application of model-driven software development combined with models at runtime takes this burden from developers by generating optimization problems using model transformations. In this paper, we present two such approaches applying integer linear programming and pseudo-boolean optimization. Furthermore, we provide a scalability analysis of both approaches showing their feasibility for pipe-and-filter applications.**

Keywords*—*self-adaptive systems; integer linear programming; pseudo-boolean optimization; MDSD

I. Introduction

The future of software systems is predicted to be characterized by ubiquitous, interconnected software components, running on several heterogenous resources that are subject to frequent changes and optimize themselves w.r.t. their non-functional behavior [1], [2].

In this paper, we address a particular problem of such self-optimizing software systems: the burden of developers to apply optimization techniques manually, which is time consuming and prone to error. We propose to generate optimization problems from models, being more natural to the developers. Thus, we propose the application of model-driven software development, especially model transformations, and the models at runtime paradigm [3] to develop self-optimizing software systems.

We present two approaches: an Integer Linear Programming (ILP)-based and a Pseudo-Boolean Optimization (PBO) [4]-based solution. Both techniques belong to combinatorial optimization [5]. They are suited, because the system configurations, amongst which the best is searched, are combinations of decisions (e.g., which implementation to choose). We compare both approaches and evaluate them w.r.t. scalability, showing their applicability despite their high complexity.

The core contributions of this paper are:

- A runtime optimization approach using ILP.
- A runtime optimization approach using PBO.
- A scalability analysis of both approaches.

The remainder of this paper is structured as follows. In Sect. II a model-driven architecture for self-optimizing software systems is presented being the basis for both the ILP-based solution discussed in in Sect. III and the PBO-based solution discussed in Sect. IV. The scalability of both approaches is examined in Sect. V. Finally, in Sect. VI we outline related work and conclude the paper in Sect. VII.

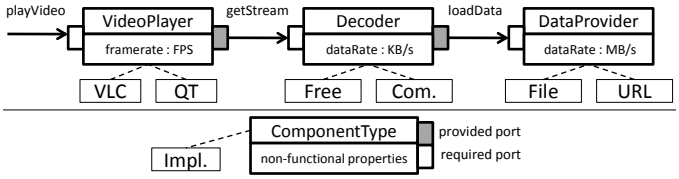


Fig. 1. VideoPlayer SW components and implementations.

II. A Contract-based Architecture for Self-Optimizing Systems

To design and model self-optimizing systems, we developed the non-functional property (NFP)-aware Cool Component Model (CCM) [6] and the Quality Contract Language (QCL) [7]. The CCM provides concepts to model hierarchical system architectures, covering both software components and hardware resources, because most NFPs base on the software's interaction with hardware resources (e.g., execution time and energy consumption). QCL provides concepts to express dependencies between CCM components based on NFPs. This implies dependencies between software components as well as software and hardware components. In the following, we introduce CCM and QCL, by means of a video application scenario.

A. Capturing Software and Hardware Components

The CCM distinguishes between modeling the system structure of hardware resource types, software components and variants of both. In the scope of this paper, variants of resource types are concrete hardware resources; variants of software components are concrete implementations. The system structure defines how a system looks like and, thus, represents type declarations for specific variants. For instance, consider the upper part of Figure 1 showing the types for a video application. It consists of three software components, namely a `Player`, a `Decoder` and a `DataProvider`. Each component may have one or more port types representing interfaces of the component. Port types can be used to connect different components. A set of connected components describes a software system.

Concrete implementations of software components (cf. Figure 1) have to correspond to their type. In the given example two variants of `Player`s, the `VLC` (Video LAN Client) and Quicktime (`QT`) implementation exist.

 ISBN: 978-1-61208-274-5

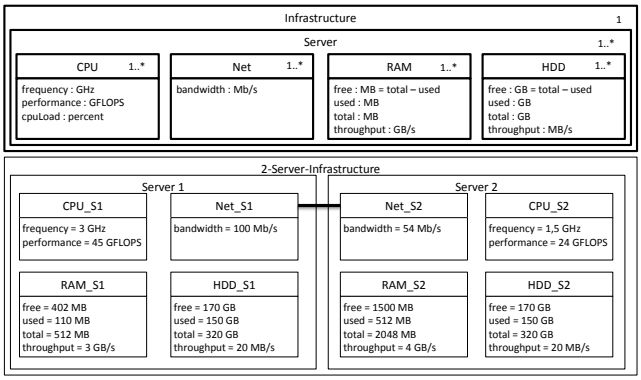


Fig. 2. Top: CCM Structure Model for hardware landscapes. Bottom: CCM Variant Model for a hardware landscape of 2 servers.

```
1  contract VLC implements VideoPlayer {
2    mode highQuality {
3      //required resources
4      requires resource CPU {
5        max cpuLoad = 50 percent
6        min frequency = 2 GHz
7      }
8      requires resource Net {
9        min bandwidth = 10 MBit/s
10     }
11     //dependencies on other SW components
12     requires component Decoder {
13       min dataRate = 50 KB/s
14     }
15     //what is provided in turn
16     provides min frameRate 25 Frame/s
17     provides min resolution 1080 p
18   }
19   mode lowQuality { ... }
20 }
```

Fig. 3. Example Contract for VLC Video Player.

For `Decoder`s, a free (`Free`) and a commercial (`Com.`) implementation are available. Finally, the `DataProvider` is implemented as a local file reader (`File`) and a remote URL reader (`URL`).

To capture types available in the hardware landscape, resource types are specified. The upper part of Figure 2 defines resource types, on which our video application shall be executed. The `Infrastructure` consist of one or more `Server`s, whereas each server contains one or more `CPU`s, network interfaces (`Net`), `RAM` chips and hard disks (`HDD`). For reasons of simplicity, we omit port types of resource types in the given example.

For each component type (software and hardware), NFPs can be defined. For instance, the `Player` type defines a property `framerate` in fps (frames per second) whereas the `HDD` type defines a property `used` (disk space) in GByte.

The lower part of Figure 2 shows a hardware instantiation of the resource type system mentioned above. It consists of two servers with specific resources according to the definitions at the type level. NFPs defined at type level, are available at variant level with concrete values. Each resource variant can provide behavior models to further specify its NFPs (e.g., the correlation of energy consumption and CPU utilization).

B. Specification of QCL contracts

QCL is used to define dependencies between CCM components as contracts, specified for each variant. Therefore, a QCL contract represents a specific view on a variant regarding its dependencies to other types. A contract defines one or more `mode`s, whereas each `mode` indepently (from other modes) defines dependencies to other components. Software components can depend on other software components as well as hardware resources, whereas resources can depend on other hardware resources only. Each dependency relates to a component type and defines bounds for required values of NFPs at runtime. In addition to property requirement constraints, provided NFPs are specified as well.

Figure 3 shows a contract for the *VLC* video player as a concrete implementation of the `VideoPlayer` component. As defined, the player can be used in two modes: `high-` and `lowQuality`. For `highQuality` the contract specifies requirements for a `CPU` and a `Net` device. The `CPU` needs to be utilized at most to 50% and to have a frequency of at least 2GHz. The `Net` device has to offer at least a 10 MBit/s bandwidth. Furthermore, another component is required—a `Decoder`. Any implementation of this type, which is able to provide a data rate of at least 50 KB/s can be used. Finally, the contract defines that in the `highQuality` mode a minimum framerate of 25 fps and a resolution of 1080p is provided.

To determine the hardware requirements, micro-benchmarks written by the component developer, evaluating the non-functional properties of interest, are used. A more detailed discussion on how to create these contracts has been published in [8].

In summary, a system modeled with CCM and QCL is highly variable in terms of multiple implementations of component types, multiple quality modes per implementation and, according to resource requirements of each quality mode, multiple possible mappings of implementations to hardware resources.

III. Self-Optimization with ILP

The central task of self-optimizing systems is to determine optimal system configurations. In this section, a model-based approach using an exact optimization technique called ILP is shown. In contrast to many existing approaches to self-optimization, the presented approach does not require the developer to apply the optimization technique itself, but generates the formulation of the optimization problem using the existing development artifacts. In this work, a system configuration denotes a set of software component implementations deployed on component-containers, which run on servers (or, more general, computing entities). Thus, the optimization problem is, which implementations of which component types need to be mapped onto which containers in order to reach the optimal trade off between user satisfaction and execution costs.

To solve this problem, a variety of information is re-

 ISBN: 978-1-61208-274-5

quired. Namely, variant models of hard- and software representing the currently running system, structure models of hard- and software representing the architecture of the system, QCL contracts characterizing the non-functional behavior of the software and a user request with the user's Quality of Service (QoS) demands.

As ILP is a mathematical formalism with its own language, a transformation from the structure and variant models as well as the contracts and the request to ILP is required. Figure 4 depicts the general approach of this ILP generation, which can be characterized as a model-to-text transformation in terms of Model-Driven Architecture (MDA) [9].

On the left upper side of Figure 4, the structure of the optimization problem formulated as an ILP is shown. An ILP comprises a set of objective functions, a set of decision variables and a set of constraints (as highlighted by the dashed lines). The objective functions depend on the user request (declaring objectives important for the user, e.g., response time) and on the variant (i.e., runtime) model. Decision variables depend on QoS contracts and variant models, too. Finally, the constraints of the ILP depend on all available input information. In the following, the generation of decision variables, objective functions and constraints is discussed in more detail.

A. The Rational of Decision Variables

In this work, the decision variables directly follow the characterization of system configurations: they denote which implementation is to be run in which quality mode on which container. They encompass a selection problem (i.e., which implementations to select) and a mapping problem (i.e., to which container the selected implementations shall be mapped). This information is comprised by the name of the variable seperated by hashtags, whereas the type of the decision variables is of boolean nature (meaning each of these variable being solved having the value `1` denotes the deployment of a specific component implementation in a specific mode on a specific resource). Equation 1 shows the general form of decision variables as used in the presented approach. The prefix `b#` is meant to highlight the boolean type of this variable.

$$b\#implementation\#mode\#container \in \mathbb{B} \quad (1)$$

Additional variables express the resource utilization and resulting NFP values implied by a certain implementation (as specified in QCL contracts). These variables have a real value (i.e., are in $\mathbb{R}$) and have the prefix `u#` for utilization. The naming of these variables fully qualifies an NFP of a certain resource of a component container (i.e., server). As resources are hierarchically structured, subresources, subsubresources, etc. can be specified. For example, a resource `CPU1` can comprises the subresources `Core1`, `Core2` and so on. Equations 2 and 3 denote the general forms of these variables.

$$u\#container\#resource\#subres.\#...\#NFP \in \mathbb{R} \quad (2)$$

$$implementation\#NFP \in \mathbb{R} \quad (3)$$

Solving an ILP working on these variables, leads to an assignment of values, representing the optimal system configuration. Thus, the solution provides information about the optimal selection of implementations, modes, their mapping to containers and additional information on the resulting NFPs of the participating components.

B. Generation of Objective Functions

All objective functions in the context of this work, base on assessment functions of system configurations. That is, an objective function assesses system configurations in terms of the respective objective (e.g., energy, performance or reliability) and aims to determine that configuration, which is assessed to be minimal or maximal w.r.t. the current objective.

A straightforward objective function based on the variables explained in the previous subsection is resource minimization as shown in Equation 4. However, this objective function does not consider the interplay between selected components instances there, required and provided NFPs and the available resources. Thus, constraints are used to restrict the ILP to correct solutions. Anyhow, the objective function of Equation 4 does not lead to the intended result (i.e., minimum resource consumption), because the units and the semantics of each resource usage variable are not considered. For example, the formula does not differentiate between utilizing 10 MB of main memory in contrast to utilizing 10 MB of hard disk drive (HDD) space.

$$\min : \sum u\#container\#res.\#subres.\#...\#NFP \quad (4)$$

A practical solution towards more sophisticated objective functions is the application of utility theory to map each variable to a utility expressed as a real value between zero and one. In the case of resource usage the utility functions can reflect the difference between using space of main memory and an HDD by putting the requested amount of space in relation to the totally available space. The general form of objective functions according to utility theory is shown in Equation 5. The objective is to maximize the overall utility.

$$\max : \sum utility(var_{decision}) \quad (5)$$

An even more sophisticated objective is the combination of the previous two types of objectives (i.e., cost minimization and utility maximization): *efficiency* maximization. The general form of this type of objective is depicted in Equation 6.

$$\max : \eta(var_{decision}) = \frac{utility(var_{decision})}{cost(var_{decision})} \quad (6)$$

Finally, an important aspect of objective functions in the context of reconfigurable systems is the need to consider the **reconfiguration** and **decision making** itself.

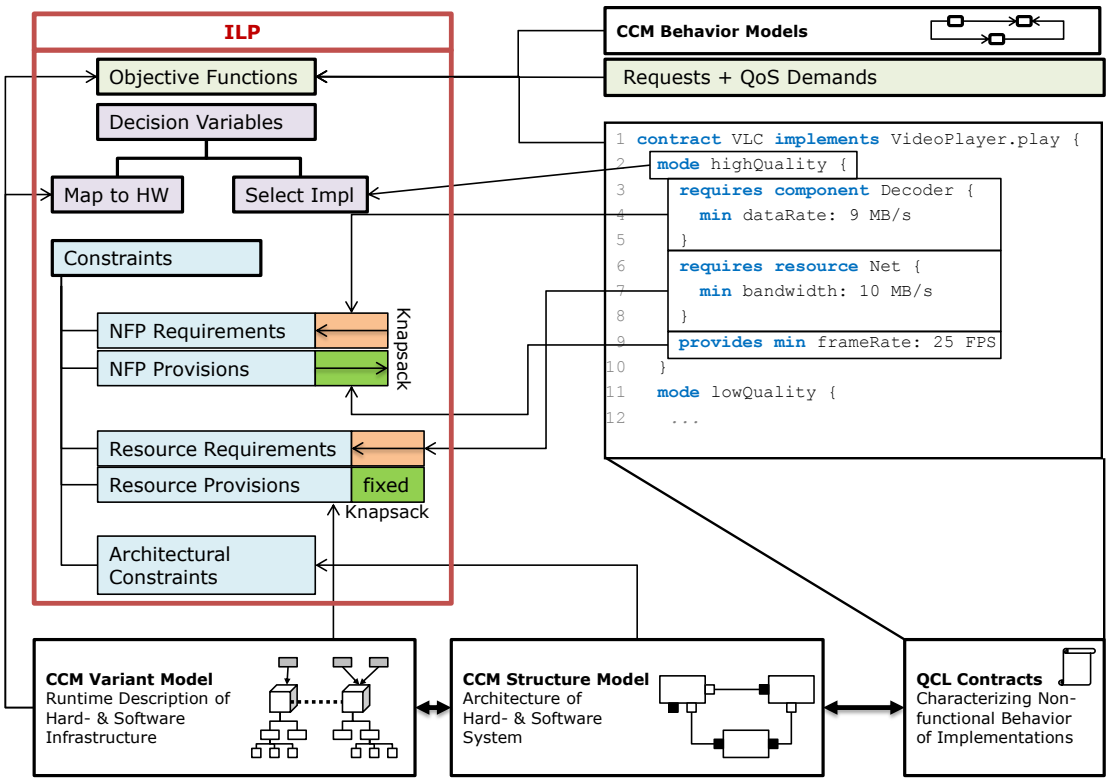


Fig. 4. Overview of ILP Generation.

This is, because both processes require time and resources and, thus, effect the objective functions. Hence, a general objective function should look as depicted in Equation 7, which aims for maximum efficiency. Notably, the costs implied by reconfiguration and decision making need to be assessed w.r.t. the quality of interest for the respective objective function.

$$\max : \sum_{i=1}^{n} \frac{util_i}{cost_i} * \frac{util_{reconf} + util_{decide}}{cost_{reconf} + cost_{decide}} \quad (7)$$

The assessment of $cost_{reconf}$ as well as $cost_{decide}$ (and the negative utilities) is a non-trivial task, which has to be investigated for each NFP individually. A closer discussion exceeds the scope of this paper.

C. Constraint Generation

As mentioned above, the ILP must be constrained by a set of constraints to allow the computation of sensible solutions only. In general, three classes of constraints are generated in the presented ILP approach: constraints negotiating software NFPs, constraints negotiating resource requirements, and architectural constraints. In the following each class is explained in more detail.

Software NFP Negotiation: The negotiation of software NFPs covers the interdependencies between NFPs of different software components expressed by their provisions and requirements in QCL contracts. The selection of an implementation of a software component type in a certain quality mode induces a set of NFP provisions (by the implementation) and requirements (to other components). To determine an optimal selection includes to find a balance between provided NFPs and required NFPs across all required components to fulfill the user's request. This problem is reflected in the ILP by two types of constraint clauses, which are generated for each NFP. First, NFP provisions are expressed as equality constraints as depicted in Equation 8. Depending on the assignment of the decision variables, the available amount of the respective NFP results.

$$NFP_i = \sum^{c} prov_a^b * b\#impl_a\#mode_b\#container \quad (8)$$

Second, the NFP requirements implied by selecting a certain implementation in a quality mode are expressed as inequality constraint as depicted in Equation 9. Notably, the relation between NFP_i and the aggregated NFP requirements is only "less or equal", if the respective NFP is of ascending order. For example, the NFP `memory` is of ascending order, whereas the NFP `response_time` is of descending order. Thus, for `response_time`, the inequality is of "greater or equal than" type.

$$NFP_i \leq \sum^{c} req_a^b * b\#impl_a\#mode_b\#container \quad (9)$$

 ISBN: 978-1-61208-274-5

Resource Negotiation: Besides NFPs provided and required by implementation, the selection of implementations implies resource requirements, too. In contrast to software NFPs, the provision of resources is fixed by the available hardware. Similar to software NFP negotiation, two types of constraints are generated for resource negotiation. First, the provision of resources as depicted in Equation 10. The provided property naturally needs to be greater or equal than zero and less or equal than the maximum offered by the resource. In addition, the granularity of the resource can be restricted. For example, the amount of disk space can only be utilized in blocks of a certain size (e.g., 4KB). In the constraints of Equation 10, the terms *maximum* and *granularity* are replaced by the respective concrete values. The term x remains a free variable, which is to be found by the solver of the optimization problem. The restriction of x to be binary (i.e., $\in \mathbb{B}$) enforces the restriction of the resource property to be a multiple of *granularity*.

$$\begin{aligned} ResourceProperty_i &\geq 0 \\ ResourceProperty_i &\leq max \\ ResourceProperty_i &= granularity * x \\ x &\in \mathbb{B} \end{aligned} \tag{10}$$

The second type of constraint covers the resource requirements by selecting an implementation. The resulting constraint is depicted in Equation 11.

$$ResourceProperty_i \leq \sum^{c} req_a^b * b\#impl_a\#mode_b\#container \tag{11}$$

For each resource property, constraints of these types are generated, whereby the search for valid assignments to the decision variables is further restricted.

Architectural Constraints: Finally, constraints based on the knowledge about the software's architecture and the requested software component are translated into constraints of the ILP. The simplest possible constraint of this type is the necessity to select exactly one implementation of a component type whose port is requested by the user. The corresponding constraint is depicted in Equation 12.

$$\sum b\#impl_a\#mode_b\#container = 1 \quad \forall a \in T \wedge b \in modes_of(a) \tag{12}$$

For all modes b of all implementations available for the component type T, the sum of the corresponding decision variables needs to be exactly one. This constraint suffices, if no other component types exist or the requested component type does not use any other component type. If another component type is used, the need to select an implementation of this type needs to be expressed, too. In the general case, constraints have to be generated, which express that the selection of an implementation of component type T_1 implies the need to select an implementation of type T_2. Equation 13 depicts this kind of constraint.

$$\begin{aligned} \sum b\#impl_a\#mode_b\#container &= \\ \sum b\#impl_c\#mode_d\#container & \\ \forall a \in T_1 \wedge b \in modes\_of(a) \wedge c \in T_2 &\ \wedge \\ d \in modes\_of(c) \wedge depends(T_1, T_2) & \end{aligned} \tag{13}$$

The above described three types of constraints, restricting the possible assignments to the decision variables, so only valid configurations are investigated for their optimality. In addition, corresponding to the decisions, values are assigned to the resource usage and NFP variables. In the next section we describe a lean variant of this approach, which avoids the use of resource and NFP variables.

IV. Self-Optimization with PBO

The key aspect of the configuration problem expressed for the ILP-based solution presented above, is denoted by boolean decision variables, encompassing the decision to select certain implementations and their mapping to certain resources. The remaining variables used in the ILP-based solution comprise resource usage and resulting NFP values. In this section, the ability to omit non-boolean variables is shown. The intended goal is to apply more efficient solving techniques to the generated optimization problems, which leverage on the restriction to use boolean variables only. Due to the exclusive use of boolean variables in an ILP a special type of ILP results: a 0-1 ILP. This type of ILP can be handled by PBO, which applies techniques used to solve satisfiability problems in propositional logics (e.g., DPLL [10]). These techniques have polynomial complexity, whereas algorithms used to solve ILPs (e.g., simplex) have exponential complexity. Hence, we investigate PBO for self-optimization.

A. Reformulation of the Configuration Problem in PBO

To apply techniques of PBO to the configuration problem to be generated, all non-boolean variables from the ILP solution need to be expressed in a different way. Namely, a new way to express resource negotiation, NFP negotiation including user requirements and the objective function is required. Notably, the architectural constraints defined for the ILP solution can remain unchanged, because they only refer to decision variables. In the following, a solution for each of the constraints subject to adjustment is given.

Resource Negotiation without Usage Variables: The expression of resource negotiation in ILP with usage variables has been shown in the previous section. All these constraints, except for the granularity restriction, can be expressed by a single PBO constraint, which implicitly represents the respective resource property (i.e., $ResourceProperty_i$) as shown in Equation 14. The term ${}^c req_a^b$ denotes the implied resource requirements by the respective implementation a in the specified mode b on a given container c.

$$\sum {}^{c}req_a^b * b\#impl_a\#mode_b\#container_c \leq max \quad (14)$$

NFP negotiation: Alongside with the restriction of resource usage, the dependencies between offered and required NFPs has to be expressed by constraints. In the ILP solution explicit variables for each NFP have been used to connect separate constraints for their provisions and requirements. The same principle, as for resource usage negotiation can be applied for NFP negotiation, too. Namely, the implicit expression of each NFP variable. Thus, for PBO, the provision and requirement constraints are merged into a single constraint which implicitly represents the possible expressions of the NFP. For the generation of these constraints the user request needs to be considered, too. This can be accomplished by incorporating the respective NFP requirements expressed by the user in his request as minimum value of the respective NFP. The resulting constraint is shown in Equation 15.

$$\begin{aligned}\sum (req_{user} +^{c} req_a^b * b\#impl_a\#mode_b\#container_c) \quad (15)\\ \leq \sum {}^{c}prov_a^b * b\#impl_a\#mode_b\#container_c\end{aligned}$$

Reformulation of Objective Function: The objective function in PBO can only rely on decision variables, because only these variables exist. In contrast, the ILP solution allowed for the application of resource usage and NFP variables. Thus, the minimization or maximization of resource usage and NFPs cannot be directly expressed in PBO. The general form of the objective function in PBO is shown in Equation 16.

$$min : \sum weight * b\#impl_a\#mode_b\#container_c \quad (16)$$

Thus, the major issue to generate meaningful objective functions in PBO is the computation of the *weight* constants for each decision variable. This *weight* has to express the impact the decision represented by the decision variable on the overall objective.

V. Evaluation

The above described approach, applying generated ILP- or PBO-based optimization for self-adaptive systems, is *not likely to scale*. This is, because solving an ILP or PBO is known to be an NP-hard problem, where the processing time required by the solver grows exponentially with number of decision variables of the ILP/PBO. In the following, we show how ILP/PBO generation and solving perform and that both approaches are feasible for typical pipes-and-filter applications. We compare the performance of both approaches, showing that the ILP solution outperforms the PBO solution.

A. Generation of Test Systems for Empirical Evaluation

To empirically evaluate the performance of the approaches, a set of test systems has been generated. As the ILP/PBO generation relies on the models of the system only (and not the system itself), it is possible to evaluate the approaches against a variety of system types without the need for their physical presence. Thus, we developed a parameterizable system generator, which is capable of generating models as usually derived by the runtime environment. This includes hard- and software structure models, hardware and software variant models and QoS contracts.

The generator is configured with several parameters:

- Number of servers S,
- Number of resources per server N_{res},
- Number of properties per resource $N_{resProp}$.
- Number of component types, C
- Maximum depth of dependency chains δ,
- Number of NFPs defined per component type N_{nfp},
- Number of implementations per component type N_{impl},
- Number of modes per implementation N_{mode},
- Number of hardware requirements per mode,
- Number of NFPs required per software dependency per mode and
- Number of provided NFPs per mode.

Note that it is impossible to generate systems leading to worst case execution times of the solver as the solvers internally use heuristics. But, for proper evaluation results using generated systems, each generated system should allow at least one valid configuration. Randomly generating NFP provisions and requirements obviously leads to infeasible systems in most cases. To ensure the existence of at least on feasible system configuration, a random request is generated, which serves as reference request for which a feasible system configuration is to be ensured. The process of system generation keeps track of how the random request transforms between dependent software component types. For the directly requested component type at least one (randomly chosen) quality mode Q_1 is selected to fulfill the request. Then, for all dependent component types at least one quality mode is chosen to fulfill the requirements of Q_1. The same process is performed for all dependent types of the dependent types and so on. The generated user request to ensure feasibility is reusedlater as test request for evaluation.

B. Measurements for Pipe-and-Filter Style Systems

The pipes-and-filter architectural style is common across many data processing applications. For example, in early detection of Alzheimer's disease, magnetic resonance tomography (MRT) pictures are processed by a

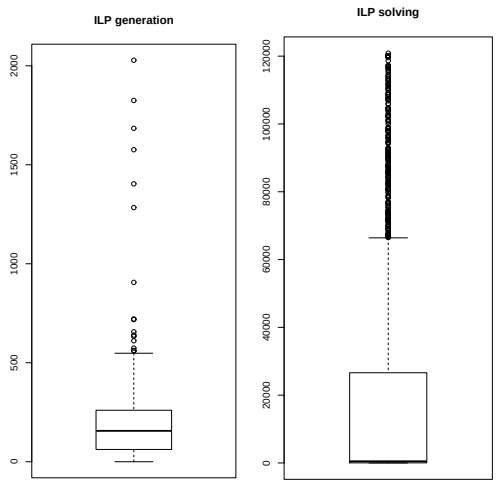
pipes-and-filter architecture comprising data preparation, Alzheimer's indicator search and data postprocessing. Another example is audio processing as performed, e.g., by auphonic (http://auphonic.com/), where audio files are processed by a chain of processing steps. In general, the pipes-and-filter style is characterized by a chain of component types. There are n component types, where the first component type requires the second, which in turn requires the third component type and so on. For this architectural style the parameter n, denoting the number of component types or the depth of the chain is of interest. In addition, the number of containers c is to be considered, because the number of possible configurations grows exponentially with the number of available containers. Thus, a test system in pipes-and-filter style is characterized as an $n \times c$ system having n component types and c containers.

For the server landscape, we assume each server (i.e., container) to have one central processing unit (CPU), one random access memory (RAM) module, on HDD and a network device. Each software component type has one provided and one required port type, except the last component type in the chain, which only has a provided port type. Moreover, each software component type has two NFPs and two implementations, having two modes, which each have four resource requirements, two software requirements and two provisions.

To assess pipes-and-filter type applications, all variants of C *x* S systems for $C = [2..100]$ and $S = [2..100]$ have been generated and the generation and solving time of the respective ILPs and PBO formulations has been measured. For each generated system, two measurements were taken. First, the time required to generate the respective ILP or PBO problem. Second, the time required to solve the problem using a standard solver. For the ILP solution the solver *LP Solve* [11] (version 5.5.20) has been used. For PBO the *OBPDB* [12] solver (v. 1.1.3) has been used. All measurements were taken on a DELL Alienware X51 desktop PC running Windows 7 64bit and containing a solid state disk, 8 GB DDR1600 RAM and an Intel Core i7-2600 CPU running at 3.4 GHz having 4 physical cores and 8 virtual cores by hyperthreading.

Analysis of the ILP Solution: Figure 5(a) shows a boxplot of the ILP generation time and Figure 6 depicts this generation time in relation to the number of software components. The median generation time is 156 ms and 75% of all ILPs were generated in at most 260 ms. The longest generation took 2028 ms. Notably, the 99%-quantile is 437 ms, meaning that in 99% of all cases, the maximum generation time is less or equal to 468 ms. A natural hypothesis is that the number of components and servers correlates to the generation time; which indeed exists: $T_{gen}(C) = 0.0291C^2 + 1.4429C + 5.3851$ with an R^2 of 0.8956.

Figure 5(b) depicts a boxplot of the ILP solving time. It reveals the random nature of worst and best case runtime of ILP solving, depicted by the vast amount of outliers. For only 121 out of 9801 generated ILPs the solver was not able to return any solution within two minutes for all measurements taken (i.e., in 1.2% of all cases). Please note, that the solving time had an upper limit at 2 minutes, as



(a) Generation time (b) Solving time

Fig. 5. ILP generation and solving time in milliseconds.

for a some ILPs the solving time can increase to multiple hours or even days, which is the worst case, where the complete problem space has to be explored. Surprisingly, the median solving time is only 478.5 ms. Thus, half of the ILPs could be solved in less than a second. The third quantile (i.e., 75%-quantile) is at 26.58 s. 79% of all ILPs were solved in less than a minute. Notably, if the (manually configured) timeout of two minutes was reached, the ILP solver returned the best solution found so far.

In the following, a closer investigation of the solving time is presented. The aim is to investigate if and how

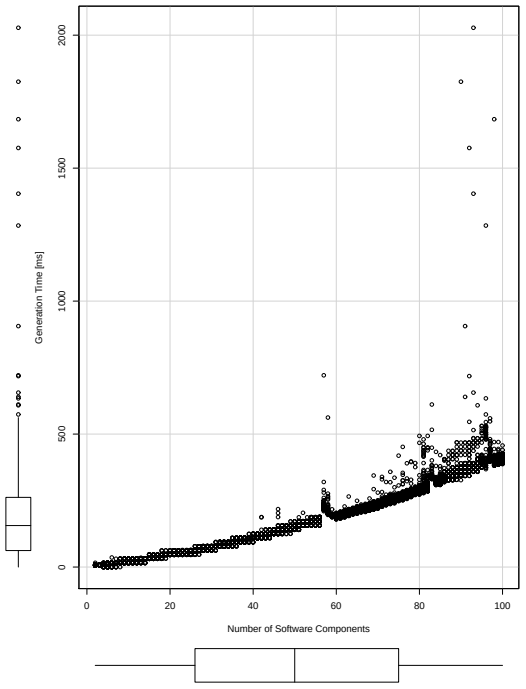


Fig. 6. ILP generation time in relation to number of components.

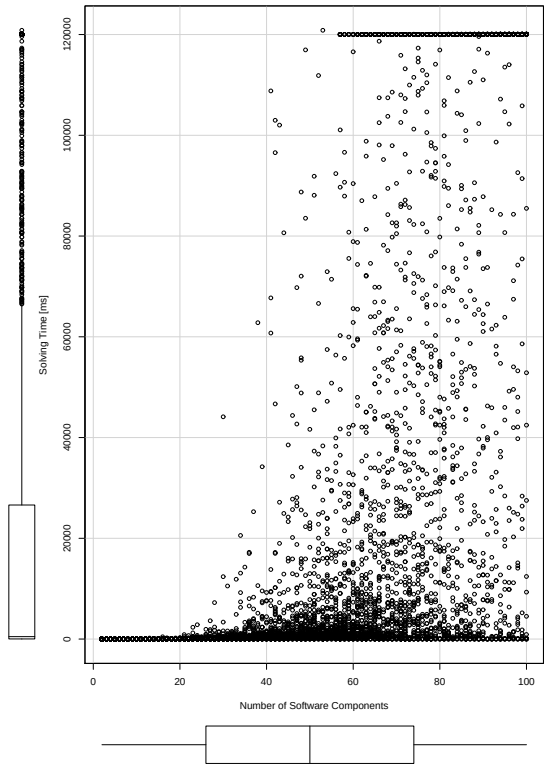


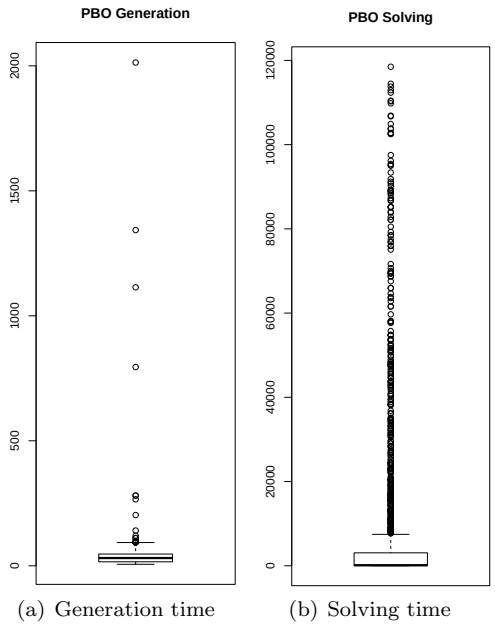
Fig. 7. ILP solving time in relation to number of components.

the system parameters correlate with the solving time of the generated ILP. The hypothesis is, that there is a correlation between the number of component types and the solving time. Figure 7 depicts this correlation. On each axis a boxplot of the corresponding variable is shown to highlight the high density of solutions in low solving times. An interesting conclusion from this figure is, that the predictability of solving time decreases at approximately 25 component types. Most ILPs are solved in a few seconds, though the more component types, the more likely are longer solving times.

The correlation between the number of component types and the solving time for scenarios is statistically poor. The linear regression has an adjusted R^2 of only 0.4286. Exponential regression (i.e., $T_{solve} = f(Components) = a \cdot e^x$) reveals a residual standard error of $1.911 \cdot 10^{13}$. Thus, there is no statistically significant correlation between the number of components and the solving time. The reason is the random nature of ILP solving, i.e., solving random ILPs can randomly lead to worst and best case situations.

Analysis of the PBO in Comparison to the ILP Solution: The same measurements have been done for the PBO solution. Figure 8 depicts the PBO generation and solving times. In addition, Figure 9 depicts the generation time in relation to the number of components. Please note that for the PBO solution only systems with up to 30 component types have been measured, because the approach does not scale beyond this number of component types.

In comparison to the ILP solution, the generation of



(a) Generation time (b) Solving time

Fig. 8. PBO generation and solving time in milliseconds.

PBOs looks comparably performant at a first glance. ILPs for systems of up to 100 component types are generated in up to 2 seconds. PBOs for systems of only up to 30 component types require up to 2 s, too. But, whilst the median for ILP generation was at 156 ms, for PBO generation it is at 31 ms. Also for the 99%-quantile, the ILP solution looks worse than the PBO solution, as for ILP the 99%-quantile is at 468 ms, whereas for PBO it is at 94 ms.

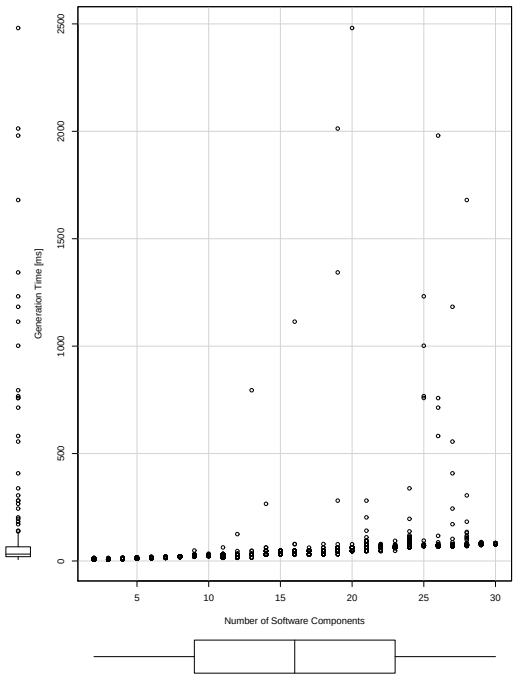


Fig. 9. PBO generation time in relation to number of components.

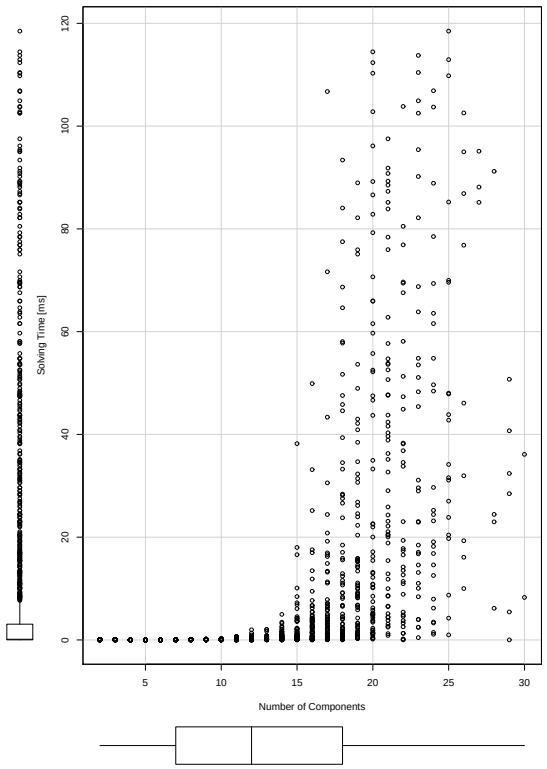
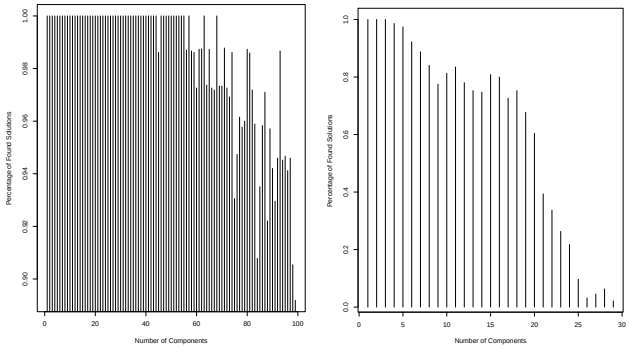


Fig. 10. PBO solving time in relation to number of components.

But, because for the PBO solution only systems of up to 30 component types have been used, the ILP generation times for up to 100 components cannot be used as a reference. An investigation of ILP generation for systems of up to 30 component types reveals a median of 32 ms and a 99%-quantile of 78 ms. Thus, PBO generation is even slower than ILP generation, although less constraints and less variables have to be generated. The reason for the better performance of ILP generation is the required program format for the applied solvers. The ILP solution can use long variables names to encode information (i.e., the decision variables encode their meaning in their name), whereas the PBO solution has to use enumerated variables (i.e., x_n). In consequence, the PBO generation has to handle the mapping of decision variables to their short versions, which consumes time and, hence, leads to the measured performance loss.

An investigation of the solving time of the PBO solution reveals surprising results. The rational for using PBO instead of ILP was the hypothesis that PBO performs better in many cases as it has polynomial complexity only. But, for the optimization problem discussed in this paper it apparently does not as Figure 10 depicts. The solving time has been limited to two minutes, too. But, in contrast to the ILP solution, the PBO solver does not deliver a suboptimal solution if the timeout is reached. Starting at 15 component types (compared to 25 in the ILP solution) the predictability of the solving time drastically decreases. Most interesting is the lack of fast solutions starting at 25 component types. In comparison, the ILP solution was able to deliver solutions in a few milliseconds even for



(a) ILP approach (b) PBO approach

Fig. 11. Percentage of found solutions for ILP and PBO within 2 minutes of solving time.

systems with more than 80 component types. Moreover, the number of determined solutions in a timeframe of less than 2 minutes significantly reduces starting already at 20 component types. For example, the measurements taken for 28 component types and 2 to 100 servers (i.e., 99 measurements) only lead to 4 solutions. For the remaining 95 systems the PBO solver could not find a configuration. For this reason, only measurements for up to 30 component types have been collected.

The measurements of the solving time indeed benchmark the used solver. Unfortunately, for PBO only one solver could be investigated, because all other solvers do not support the specification of equalities with variables on both sides of the equation. Therefore, OPBDP [12] was the only publicly available, working solver which could be investigated.

Thus, in conclusion, the PBO solution performs much worse than the ILP solution for the optimization problem at hand. This is because, *in the average case* the algorithms used to solve ILPs perform better than those used for PBO. Whereas the ILP solution is feasible for systems of up to 100 component types, the PBO solution can handle at most 25 component types. The generation of ILPs is below 437 ms for up to 100 component types and PBO generation takes less than 100 ms in 99% of all cases for systems of up to 30 component types. Solving of ILPs is below 500 ms in 50% of all cases and below 27 s in 75% of all cases. PBO solving is below 2.6 s in 50% of all cases, but reaches the timeout of 2 minutes already in 70% of all cases. Most notably, the PBO solution is not able to find configurations starting already at 25 component types in most of the cases, whereas the ILP solution is able to determine configurations even for 100 component types. Figure 11 depicts the percentage of solutions found by the ILP and PBO approach in correlation with the number of component types. The execution time of the ILP solution is predictable up to 25 component types, whereas the execution time of the PBO solution only up to 15 component types.

The reason for the decreasing predictability in the ILP solution is the growing span between best and worst case execution time of the solver. The best case execution time grows linearly with the number of variables, whereas the

 ISBN: 978-1-61208-274-5

worst case execution time grows exponentially. Both curves are very near to each other in the beginning, but depart more and more (exponentially) from each other the bigger the systems are. Notably, the numbers presented above are specific to the machine used for measurements. Thus, also the number of component types where predictability starts to worsen has to be determined for each machine.

VI. Related Work

The application of model-driven software development to self-adaptive systems has been studied by various groups throughout the last years.

In [13], Zeller et al. address the problem of determining a valid mapping of software components to electronic control units (ECU) by SAT solving and simulated annealing. The solving techniques do not guarantee optimality, but ensure the determination of a valid system configuration. The SAT problem is generated from the system's models. Zeller et al. evaluated their approach and showed that the runtime of their SAT solving approach is below 4s for 40 ECUs, 60 Sensors, 60 Actuators and 2000 Functions. For less than 1600 functions SAT solving takes less than 2 seconds. The biggest setup evaluated by Zeller et al. comprised 100 ECUs, 120 Sensors, 120 Actuators, 2500 Functions and took 18 seconds [13].

Another approach using model-driven software development for self-adaptive systems has been developed in the DiVA research project and presented in [14], where Fleurey and Solberg introduce a quality grading framework for software variants. The developer rates available implementations in different quality dimensions (e.g., energy or performance) using a model-based framework. To identify valid system configurations these models are transformed to Alloy [15]. Unfortunately, no empirical study of the approach's scalability has been published.

In contrast to the approaches presented in this paper, the previously discussed approaches do not search for optimal configurations, the constraints of the optimization problem are not generated, the notion of contracts is not utilized and only the mapping [13] or the selection problem [14] is considered, respectively.

VII. Conclusion and Future Work

In this paper, we propose the application of model-driven software engineering and generation of optimization problems to automatically compute optimal system configurations for self-adaptive software systems. We presented two approaches, an ILP-based and a PBO-based solution. The rational for investigating the applicability of PBO was the hypothesis that PBO could perform better than ILP for the investigated optimization problem. However, based on a large set of systems to be optimized, our scalability analysis rejected this hypothesis and revealed the feasibility of the ILP solution for systems of up to 100 component types to be distributed on 100 servers. As typical data processing applications like audio processing usually comprise a few tens of processing steps, the ILP-based approach has been shown to be applicable. For future work, we plan to investigate the feasibility and scalability of further optimization techniques.

Acknowledgment

This research has been funded by the ESF and Federal State of Saxony within the project ZESSY #080951806, and within the Collaborative Research Center 912 (HAEC), funded by the DFG.

References

[1] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in Future of Software Systems, 2007, pp. 259–268.

[2] B. H. Cheng, R. Lemos, and H. Giese et al., "Software engineering for self-adaptive systems: A research roadmap," in Software Engineering for Self-Adaptive Systems, ser. LNCS. Springer, 2009, vol. 5525, pp. 1–26.

[3] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg, "Models@Run.time to Support Dynamic Adaptation," Computer, vol. 42, no. 10, 2009, pp. 44–51.

[4] E. Boros and P. L. Hammer, "Pseudo-Boolean Optimization," Discrete Applied Mathematics, vol. 123, no. 1–3, November 2002, pp. 155–225.

[5] G. L. Nemhauser and L. A. Wolsey, Integer and Combinatorial Optimization. Wiley Interscience, 1999.

[6] S. Götz, C. Wilke, M. Schmidt, S. Cech, and U. Aßmann, "Towards Energy Auto Tuning," in GREEN IT 2010. GSTF, 2010, pp. 122–129.

[7] S. Götz, C. Wilke, S. Cech, and U. Aßmann, Sustainable ICTs and Management Systems for Green Computing. IGI Global, 2012, ch. Architecture and Mechanisms for Energy Auto Tuning, pp. 45–73.

[8] S. Götz, C. Wilke, S. Richly, and U. Aßmann, "Approximating quality contracts for energy auto-tuning software," in GREENS 2012. IEEE, 2012, pp. 8–14.

[9] J. Miller and J. Mukerji, "MDA Guide Version 1.0.1," OMG Document, 2003.

[10] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," CACM, vol. 5, no. 7, Jul. 1962, pp. 394–397.

[11] K. Eikland and P. Notebaert, "LP Solve 5.5 Reference Guide," http://lpsolve.sourceforge.net/5.5/ (access in Nov. 2012).

[12] P. Barth, "A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization," Max-Planck-Institut für Informatik, Research Report MPI-I-95-2-003, 1995.

[13] M. Zeller, C. Prehofer, G. Weiss, D. Eilers, and R. Knorr, "Towards Self-Adaptation in Real-time, Networked Systems: Efficient Solving of System Constraints for Automotive Embedded Systems," in SASO 2011, 2011, pp. 79–88.

[14] F. Fleurey and A. Solberg, "A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems," in MODELS 2009, ser. LNCS, vol. 5795. Springer, 2009, pp. 606–621.

[15] D. Jackson, Software Abstractions – Logic, Language, and Analysis. MIT Press, 2012.

 ISBN: 978-1-61208-274-5