Struct Multidisc Optim (2010) 42:707–723
DOI 10.1007/s00158-010-0512-z

RESEARCH PAPER

A specification language for problem partitioning in decomposition-based design optimization

S. Tosserams · A. T. Hofkamp · L. F. P. Etman · J. E. Rooda

Received: 17 June 2009 / Revised: 2 April 2010 / Accepted: 3 April 2010 / Published online: 12 May 2010

Abstract Decomposition-based design optimization consists of two steps: partitioning of a system design problem into a number of subproblems, and coordination of the design of the decomposed system. Although several generic frameworks for coordination method implementation are available (the second step), generic approaches for specification of the partitioned problem (the first step) are rare. Available specification methods are often based on matrix or graph representations of the entire system. For larger systems these representations become intractable due to the large number of design variables and functions. This article presents a new linguistic approach for specification of partitioned problems in decomposition-based design optimization. With the elements of the proposed specification language, called Ψ (the Greek letter "Psi"), a designer can define subproblems, and assemble these into larger systems in a bottom-up fashion. The assembly process allows the system designer to control the complexity and tractability of the problem partitioning task. To facilitate coupling to generic coordination frameworks, a compiler has been developed for Ψ that generates an interchange file in the INI format. This INI-definition of the partitioned problem can easily be interpreted by programs written in other languages. The flexibility provided by the Ψ language and the automated generation of input files for computational frameworks is demonstrated on a vehicle chassis design problem. The developed tools, including user manuals and examples, are made publicly available.

Keywords Decomposition · Problem partitioning · Specification language · Distributed analysis · Distributed optimization · Implementation · Multidisciplinary design optimization · Ψ language

This work is funded by MicroNed, grant number 10005898.

S. Tosserams (✉) · A. T. Hofkamp · L. F. P. Etman · J. E. Rooda
Department of Mechanical Engineering, Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: s.tosserams@tue.nl

1 Introduction

Decomposition-based optimization approaches are attractive for addressing the challenges that arise in the optimal design of advanced engineering systems (see, e.g., Wagner and Papalambros 1993; Papalambros 1995; Sobieszczanski-Sobieski and Haftka 1997; Alexandrov 2005). The main motivation for the use of decomposition-based optimization is the organization of the design process itself. Since a single designer is not able to oversee each relevant aspect, the design process is distributed over a number of design teams. Each team is responsible for a part of the system, and typically uses specialized analysis and design tools to solve its design subproblems. Generally speaking, a subproblem can be associated with a design discipline, or represent a subsystem or component in the entire system.

Solving a system optimization problem with a decomposition-based approach entails three steps (Fig. 1):

1. Specifying the variables and functions of each discipline

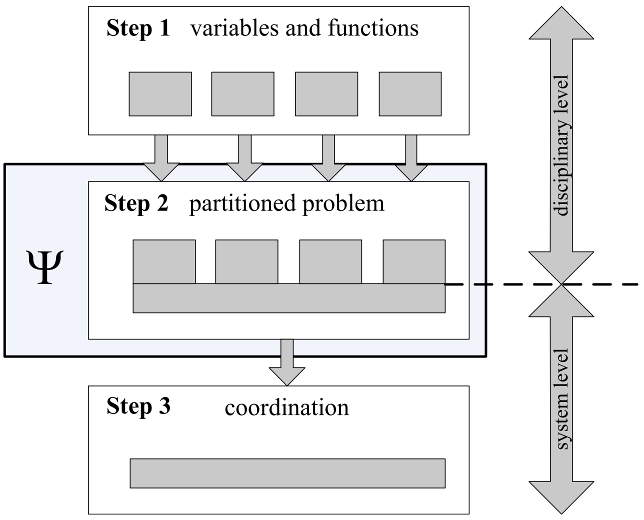


Fig. 1 The three steps involved in decomposition-based system optimization. The proposed Ψ language is developed for the specification of partitioned problems in Step 2

2. Specifying the partitioned problem (i.e. the distribution of variables and functions over subproblems and systems)
3. Coordinating the solution of the partitioned system

Variable and function specifications of Step 1 include information such as initial estimates and bounds for variables, how functions and their sensitivities are to be evaluated, and on which variables each function depends. These specifications are typically provided by each discipline separately, most likely with only partial knowledge of the interdisciplinary interactions.

The interactions are defined in the partitioned system specification of Step 2. A partitioned system specification defines how the variables and functions are distributed over design subproblems and which interactions are present between subproblems. Defining the partitioned system is a task performed partially by the disciplinary designers that define the subproblems, and a system designer that defines the interaction between the subproblems.

Once the partitioned problem is defined, a coordination algorithm needs to be implemented to solve the system design problem in Step 3. The coordination process requires the specifications of the first two steps as inputs. Which coordination algorithm is used needs to be defined as well (e.g. distributed analysis or distributed optimization). The coordination algorithm drives the design subproblems defined in Step 2 towards a system design that is consistent, feasible, and optimal. Here, consistency assures that quantities shared by multiple subproblems take equal values, feasibility refers to the satisfaction of all design constraints of all disciplines, and optimality reflects that the obtained design is optimal for the system as a whole.

Most research on decomposition-based optimal design has focussed on the final, most challenging step: coordination. Many coordination algorithms are available (see Balling and Sobieszczanski-Sobieski 1996; Sobieszczanski-Sobieski and Haftka 1997; Tosserams et al. 2009a, for overviews), as well as generic approaches for the implementation of these methods (e.g. Michelena et al. 1999; Etman et al. 2005; Huang et al. 2006; Moore et al. 2008; de Wit and van Keulen 2008; Martins et al. 2009). The first two steps have received far less attention.

Theoretical and numerical studies however show that both the choice of coordination algorithm and the way the system is partitioned have an effect on the efficiency and effectiveness of decomposition-based optimization. See Balling and Sobieszczanski-Sobieski (1996), Sobieszczanski-Sobieski and Haftka (1997), Perez et al. (2004), Tosserams et al. (2007), Allison et al. (2007), de Wit and van Keulen (2007), Yi et al. (2008), and Tosserams et al. (2009a) for examples of these observations. Experimentation with different system decompositions within the available generic coordination frameworks is not straightforward. Since each framework is developed for its appropriateness for implementation of coordination methods, it typically does not provide an intuitive environment for specifying partitioned problems. As systems become larger, their specification in such a non-intuitive environment becomes complicated and is prone to errors. Being able to specify partitioned problems in a more intuitive language is clearly preferable. Such a generic specification language provides a tool for the easy manipulation of the way a system is partitioned.

In this paper, we present a linguistic approach that allows an intuitive, compact, and flexible specification of partitioned problems. We adopt the name Ψ (the Greek letter "Psi"), an acronym for partitioning and specification. The proposed language Ψ is highly expressive and has only a small set of language elements, which is a clear advantage over more generic system modeling languages such as SysML (Friedenthal et al. 2008).

Ψ follows a composition paradigm that starts from the definition of individual components (subproblems) that are assembled into larger systems. Components are typically specified by disciplinary designers, while defining systems is the task of a system designer (see Fig. 1). The composition process is modular in that definitions of variables and functions are local to components and systems; disciplinary designers do not have to worry whether a variable or function they use locally is used by another designer elsewhere in the system. Instead, the user must *specify* interactions between components by defining systems that describe the interactions between the local definitions.

The *non-automated* composition process of Ψ provides specification autonomy to disciplines, it provides control over the composition process, and allows for the definition of multi-level systems. This in contrast to *automated* composition methods that assemble component definitions based on overlaps in variable and function names; a process that can become intractable for larger systems. An example of an automated system composition approach is given by Alexandrov and Lewis (2004a, b).

We would like to point out that a specification in Ψ is *independent* of the type of coordination method selected for Step 3. Subproblems become analysis subproblems if a single-level coordination method is selected, or they become optimization subproblems if a multi-level coordination algorithm is selected. Similarly, the language does not differentiate between hierarchical or non-hierarchical coordination methods.

As a second contribution, a compiler and two generators have been developed for Ψ. The compiler is required for processing specifications in Ψ, and the generators have been developed as examples of how input files for computational coordination frameworks can be automatically generated. The compiler and the two generators provide an easy transition from the specification of the partitioned problem in Step 2 to the solution of the partitioned problem in Step 3. The compiler checks the Ψ specification for correctness, and translates it to a normalized structure designed to simplify further automatic processing. The data is written to a file in the generic INI format. The INI format can easily be interpreted by programs in other languages such that framework-specific input files can easily be generated. As validation of the concept, we have implemented two additional generators that operate on this INI format. One generator derives the functional dependence table of the system, and another generator derives Matlab files that are used as inputs for a generic implementation of the augmented Lagrangian coordination algorithm (ALC, Tosserams et al. 2008, 2009c). This generator for ALC was the original motivation for the work presented in this article, and we expect that similar generators can be developed for other computational frameworks.

The paper is organized as follows. First, the general system design problem and its decomposition are discussed in Section 2. Second, we illustrate the elements of Ψ for a simple example in Section 3, and discuss the compiler-generated output formats in Section 4. The application of the language and the developed tools to a larger example is described in Section 5. Concluding remarks are offered in Section 6.

The developed tools, including user manuals and several examples, are available for download at http://se.wtb.tue.nl/sewiki/mdo.

2 Decomposition-based optimization for system design

Decomposition-based optimization approaches are used for the distributed design of large-scale and/or multidisciplinary systems. Decomposition methods consist of two main steps: partitioning the system and coordinating the partitioned system (Wagner and Papalambros 1993; Papalambros 1995). In partitioning, the optimal design problem is divided into a number of smaller subproblems, each typically associated with a discipline or component of the system. The task of coordination is then to drive these individual subproblems towards a design that is consistent, feasible, and optimal for the system as a whole. The main advantage of decomposition methods is that a degree of disciplinary autonomy is given to each subproblem, such that designers are free to select their own analysis and design tools.

In this section, we introduce the general system design problem followed by a description of the two main steps in decomposition: partitioning and coordination.

2.1 Optimal design problem in integrated form

The starting point of decomposition methods is the system design problem in integrated form:

$$\begin{aligned} \min_{\mathbf{z}} \quad & \mathbf{f}(\mathbf{z}, \mathbf{r}) \\ \text{subject to} \quad & \mathbf{g}(\mathbf{z}, \mathbf{r}) \leq \mathbf{0} \\ & \mathbf{h}(\mathbf{z}, \mathbf{r}) = \mathbf{0} \\ \text{where} \quad & \mathbf{r} = \mathbf{a}(\mathbf{z}, \mathbf{r}) \end{aligned} \tag{1}$$

where $\mathbf{z} = [z_1, \ldots, z_n]$ is the vector that contains the design variables of the entire system. Response variables $\mathbf{r} = [r_1, \ldots, r_{m_\mathrm{a}}]$ are intermediate quantities computed by analysis functions $\mathbf{a} = [a_1, \ldots, a_{m_\mathrm{a}}]$. These response variables are also known as coupling variables. With $\mathbf{r} = \mathbf{a}(\mathbf{z}, \mathbf{r})$, we mean to express that each analysis function a_i for response r_i may depend on the *other* responses $r_i | i \neq j$, i.e. $r_i = a_i(\mathbf{z}, r_j | j \neq i)$. The response r_i may not depend on itself. $\mathbf{f} = [f_1, \ldots, f_{m_\mathrm{f}}]$ is the vector of objective functions, and constraints $\mathbf{g} = [g_1, \ldots, g_{m_\mathrm{g}}]$ and $\mathbf{h} = [h_1, \ldots, h_{m_\mathrm{h}}]$ are the collections of inequality and equality constraints, respectively. Although the majority of the coordination methods do not allow multiple objectives, we do so here for the sake of generality. We refer to the above formulation as integrated since it includes the variables and functions of all disciplines in a single optimization problem.

2.2 Partitioning

The purpose of partitioning is to distribute the variables and functions of the integrated problem (1) over a number of subproblems. These subproblems are typically mathematical entities that perform (possibly coupled) analyses, evaluate objective and constraint values, or solve optimization problems. The subproblems may therefore (partially) differ from the original disciplines from which the integrated problem was synthesized.

Three partitioning strategies are often identified (Wagner and Papalambros 1993): aspect-based, object-based, and model-based partitioning. Aspect-based partitioning follows the human organization of disciplinary experts and analysis tools. Object-based partitioning is aligned with the subsystems and components that comprise the system. Model-based partitioning relies on mathematical techniques to obtain an appropriately balanced partition computationally. Model-based partitioning methods often rely on graph theory or matrix representations of problem structure (see, e.g., Krishnamachari and Papalambros 1997; Michelena and Papalambros 1997; Chen et al. 2005; Li and Chen 2006; Allison et al. 2007, for examples of model-based partitioning methods).

Partitioning problem (1) requires a distribution of all variables and functions over a number of subproblems. To this end, the variables $\mathbf{z}$ are partitioned into M sets of variables $\overline{\mathbf{x}}_j$ allocated to subproblems $j = 1, \ldots, M$, and a set of system-level variables $\overline{\mathbf{x}}_0$. Each set of subproblem variables $\overline{\mathbf{x}}_j = [\mathbf{y}_j, \mathbf{x}_j, \mathbf{r}_j, \mathbf{r}_{mj} | m \in \mathcal{N}_j]$ consists of a set of local design variables $\mathbf{x}_j$ associated exclusively to subproblem j, and a set of shared design variables $\mathbf{y}_j$ and response variables $\mathbf{r}_j$ and $\mathbf{r}_{mj}$, $m \in \mathcal{N}_j$. Here, $\mathcal{N}_j$ is the set of neighbors from which subproblem j requires analysis responses, and $\mathbf{r}_{mj}$ is an auxiliary variable introduced at subproblem j for the responses received from subproblem m. The set of system variables $\overline{\mathbf{x}}_0$ contains the system-level response variables $\mathbf{r}_0$.

Some of the shared variables $\mathbf{y}_j$ and all coupling responses $\mathbf{r}_{mj}$ are auxiliary variables introduced for decoupling the optimization subproblems. Interactions between the various shared and coupling variables are defined in a set of consistency constraints $\mathbf{c}(\overline{\mathbf{x}}_0, \overline{\mathbf{x}}_1, \ldots, \overline{\mathbf{x}}_M) = \mathbf{0}$, where it is understood that these constraints depend only on the shared variables and coupling responses. For further details on the use of consistency constraints in distributed optimization, the reader is referred to Cramer et al. (1994), Alexandrov and Lewis (1999), and Tosserams et al. (2009a).

Objective functions $\mathbf{f}$, constraints $\mathbf{g}$ and $\mathbf{h}$, and analyses $\mathbf{a}$ are partitioned into M sets of local functions $\mathbf{f}_j$, $\mathbf{g}_j$, $\mathbf{h}_j$, $\mathbf{a}_j$, $j = 1, \ldots, M$, and a set of system-wide functions $\mathbf{f}_0$, $\mathbf{g}_0$, $\mathbf{h}_0$, $\mathbf{a}_0$.

The partitioned problem can then be written as:

$$
\begin{array}{rll}
\min\limits_{\overline{\mathbf{x}}_0, \overline{\mathbf{x}}_1, \ldots, \overline{\mathbf{x}}_M} & [\mathbf{f}_0(\overline{\mathbf{x}}_0, \overline{\mathbf{x}}_1, \ldots, \overline{\mathbf{x}}_M), \mathbf{f}_1(\overline{\mathbf{x}}_1), \ldots, \mathbf{f}_M(\overline{\mathbf{x}}_M)] & \\
\text{subject to} & \mathbf{g}_0(\overline{\mathbf{x}}_0, \overline{\mathbf{x}}_1, \ldots, \overline{\mathbf{x}}_M) \leq \mathbf{0} & \\
& \mathbf{h}_0(\overline{\mathbf{x}}_0, \overline{\mathbf{x}}_1, \ldots, \overline{\mathbf{x}}_M) = \mathbf{0} & \\
& \mathbf{r}_0 = \mathbf{a}_0(\overline{\mathbf{x}}_0, \overline{\mathbf{x}}_1, \ldots, \overline{\mathbf{x}}_M) & \\
& \mathbf{g}_j(\overline{\mathbf{x}}_j) \leq \mathbf{0} & j = 1, \ldots, M \\
& \mathbf{h}_j(\overline{\mathbf{x}}_j) = \mathbf{0} & j = 1, \ldots, M \\
& \mathbf{r}_j = \mathbf{a}_j(\overline{\mathbf{x}}_j) & j = 1, \ldots, M \\
& \mathbf{c}(\overline{\mathbf{x}}_0, \overline{\mathbf{x}}_1, \ldots, \overline{\mathbf{x}}_M) = \mathbf{0} & \\
\text{where} & \overline{\mathbf{x}}_0 = [\mathbf{r}_0] & \\
& \overline{\mathbf{x}}_j = [\mathbf{y}_j, \mathbf{x}_j, \mathbf{r}_j, \mathbf{r}_{mj} | m \in \mathcal{N}_j] & j = 1, \ldots, M
\end{array}
\tag{2}
$$

Similar to the integrated formulation, with $\mathbf{r}_j = \mathbf{a}_j(\overline{\mathbf{x}}_j)$ we mean to express that a response in $\mathbf{r}_j$ may depend on *other* responses in $\mathbf{r}_j$, but a response cannot depend on itself.

Although the above formulation assumes that the integrated problem (1) possesses a certain sparsity structure (i.e. the presence of local variables and functions), it is general enough to encompass many practical engineering problems. Several coordination methods have been developed for a subclass of (2), so-called quasiseparable problems that do not have coupling objectives $\mathbf{f}_0$, but may have additively separable coupling constraints $\mathbf{g}_0$ and $\mathbf{h}_0$ and analysis functions $\mathbf{a}_0$ (see Tosserams et al. 2009a). For the remainder of this article, however, we consider partitioned problems of the form (2) with general coupling objectives $\mathbf{f}_0$, general coupling constraints $\mathbf{g}_0$, $\mathbf{h}_0$, and general analysis functions $\mathbf{a}_0$.

The artificially introduced variables in the above problem can be eliminated from the optimization variables through the consistency constraints $\mathbf{c}$ or the analysis equations. Whether or not these variables are eliminated is however a matter of *coordination*, and not a choice we want to make at the *partitioning* stage. Similarly, the coordination method determines how local and coupling variables and functions are treated. Hence, we will refer to problem (2) with all optimization variables included when we speak of the partitioned problem for the remainder of this article.

2.3 Coordination

After partitioning, a coordination strategy prescribes how the partitioned problem is to be solved. Single-level methods typically act directly on the partitioned problem (2), while the use of multi-level methods involves the formulation of optimization subproblems for $j = 1, \ldots, M$ and a method for coordinating the solution of these subproblems. Each coordination method is unique in its treatment of variables and functions, the way in which the partitioned problem is reformulated, and how the reformulated problem is solved. For reviews of coordination methods, the reader is referred to the works of Wagner and Papalambros (1993), Cramer et al. (1994), Balling and Sobieszczanski-Sobieski (1996), Alexandrov and Lewis (1999), and Tosserams et al. (2009a). Note that partitioned problem specifications in Ψ are *independent* of the choice of coordination method.

Numerical and analytical studies indicate that the choice of coordination method has a direct influence on the computational performance with which a problem can be solved (Perez et al. 2004; de Wit and van Keulen 2007; Yi et al. 2008). Computational frameworks have been developed to facilitate the implementation and testing of coordination methods (see again the introduction section for references). The execution of the coordination algorithms is typically automated for these frameworks, and the user is required to supply a problem specification. Such a problem specification has two ingredients (Steps 1 and 2 of Fig. 1):

1. Variable and function information
2. Partitioned problem structure

Variable specifications typically include a definition of properties such as its name, a description, its type (e.g. real/integer, scalar/vector), its size, and upper and lower bounds. Function definitions include similar properties together with additional information regarding function arguments and outputs, and how these output values are actually computed. This may for example be an explicit expression or a path to a script that should be executed. The second ingredient is the specification of the partitioned problem. The specification describes how variables and functions are allocated to subproblems, and how their couplings are defined.

For the computational frameworks listed in the introduction, the problem specification needs to be supplied in the programming language environment in which the framework is implemented. This programming language is selected for its appropriateness as a computational environment, and its language elements are relatively well-suited for defining variable and function specifications. The definition of the problem partitioning may however be less intuitive. The language limitations become more pronounced if a decomposition has non-hierarchical couplings, or has multiple levels. For such decompositions, specifying the problem becomes a tedious process that is prone to errors (Alexandrov and Lewis 2004a, b). A more intuitive specification process is clearly desired. In addition, having partitioned problem specifications in a unified format promotes their portability between computational frameworks. In the following section, several representation concepts are reviewed with respect to their appropriateness for specifying the partitioned problem.

2.4 Existing approaches for specification of the partitioned problem

Several representations have appeared in research focused on model-based partitioning (see, e.g., Kusiak and Larson 1995; Krishnamachari and Papalambros 1997; Michelena and Papalambros 1997; Chen et al. 2005). Model-based partitioning methods typically rely on matrix or graph abstractions of the couplings between variables and functions to define a sparsity structure of the integrated problem (1). Examples of such representations are the functional dependence table and the adjacency matrix. The use of matrices and graphs to specify the partitioning becomes prohibitive for larger systems due to the large number of variables and functions.

Alternative matrix and graph representations have appeared in research on the decomposition of the system design process into individual design tasks (see, e.g., Steward 1981; Eppinger et al. 1994; Kusiak and Larson 1995; Browning 2001). The transfer of information between engineers defines precedence relations between the individual tasks that can be captured in matrices or graphs. For example, element i, j of the so-called design structure matrix is non-zero if task j requires information from task i, and zero otherwise. In a graph format, vertices can be defined for each task, and precedence relations between two task can be represented by directed edges between the associated vertices. Partitioning methods for process decomposition aim at obtaining a sequence of tasks that minimizes the amount of feedback coupling between tasks or maximizes concurrency of tasks. The main difference between process decomposition and optimization problem decomposition is that the amount of detail in process decomposition is much smaller than for optimization. The number of tasks in design processes is typically one or more orders of magnitude smaller than the number of variables and functions in system optimization.

2.5 Linguistic approach to partitioned problem specification

To our opinion, the decomposition-based design community would benefit from an approach that allows the intuitive specification of partitioned problems from which matrix or graphs representations can be automatically generated, instead of working directly with matrices or graphs.

We propose to use a linguistic approach to specifying partitioned problems. The developed language is similar to the reconfigurable multidisciplinary synthesis approach

(REMS) proposed by Alexandrov and Lewis (2004a, b). REMS is a linguistic approach to problem description, formulation, and solution that follows a bottom-up assembly process. The method starts from the definition of individual subproblems that are automatically assembled into a complete system optimization problem.

The language we propose in the next section follows a similar bottom-up process, but does not automate the assembly process. Instead, disciplines have purely local definitions of variables and functions, and it is the system designer that assembles the subproblem definitions into subsystems and systems. The advantage of having a multi-level architecture composed of subproblems, lower-level systems and higher-level systems is that a designer no longer has to work on the entire system. Instead, the designer can divide the assembly into multiple levels, where each level is associated to a level of abstraction in the system. The designer can control the complexity of the specification tasks and does not have to oversee the entire system. We expect that this controllability of the complexity improves a designer's overview of the system, and provides control over the interactions between disciplines. This control over the assembly process is not available in the REMS approach.

An additional advantage is that variable and function definitions are local to subproblems. One designer does not have to worry about whether a variable defined locally also exists in another context somewhere else in the system. Instead, subproblems are free to use local nomenclatures, and the interactions between subproblem nomenclatures have to be defined explicitly at the system level. Such a decoupling of definitions appears to be appropriate in a distributed design environment.

The proposed specification approach is similar to the python-based format used for the pyMDO framework of Martins et al. (2009). However, the pyMDO approach does not have local subproblem nomenclatures, nor does it allow the definition of multi-level systems.

The multi-level assembly process has another advantage. Since a different coordination method can be assigned to each system, a multi-level nested coordination process can be formulated. For example, one can nest a lower-level system that is coordinated with a multidisciplinary feasible formulation within a higher-level system that is coordinated with collaborative optimization. Note that it is the choice of the designer how to assign coordination tasks to lower-level systems.

3 The Ψ language

The Ψ language is a linguistic approach for the intuitive specification of partitioned problems. Before describing Ψ in more detail,[1] we first introduce the following main definitions:

- A *variable* is an optimization variable of the system design problem (2), and can be an actual design variable or a response variable computed as the output of an analysis.
- A *function* represents an analysis that takes variables as arguments, and computes responses based on the values of the variables.
- A *component* represents a computational subproblem in a partitioned problem, which contains a number of variables and functions.
- A *system* contains a collection of coupled subcomponents whose coupled solution is guided by a coordination method.
- A *sub-component* is a component or system that is a direct child of another system.

The Ψ language specifies a partitioned problem by defining how variables and functions are distributed over the components, and how these components are combined into larger subsystems and systems. The building blocks of a specification in Ψ are therefore components and systems. The specification of detailed information of variables and functions (Step 1 in Fig. 1) is beyond its purpose. It is assumed that such additional information is supplied in conjunction with the specification of the partitioned problem and that the variables and functions defined in Ψ specification are pointers to this externally supplied information.

Note that Ψ is not a model-based partitioning method that automatically derives problem decompositions that can be efficiently coordinated. Ψ is dedicated to *specification* of partitioned problem, no claim is made regarding the "optimality" of the partitioning. Model-based partitioning techniques are beyond the scope of this article, and the interested reader is referred to Krishnamachari and Papalambros (1997), Michelena and Papalambros (1997), Chen et al. (2005), Li and Chen (2006), and Allison et al. (2007, 2009) for examples of such approaches. Ψ can of course be used to generate input for a model-based partitioning software tool with the aim to optimize the partitioning structure.

3.1 Components

The main building blocks of a partitioned problem definition in Ψ are components. These components are typically associated with analysis disciplines in aspect-based

[1]For a formal definition of the Ψ language, the reader is referred to the Ψ reference manual (Tosserams et al. 2009b).

decompositions or with components in object-based partitions, but may also be purely computational subproblems that have no direct relation to the physical system. At the partitioning stage, we are not concerned with the assignment of analysis and/or optimization authorities to components. This is a choice that is made at the coordination stage, and does therefore not appear in the component definitions.

To illustrate the use of Ψ, consider the following optimization problem:

$$\begin{aligned} \min_{x_1,x_2,x_3,x_4,y,r,u} \quad & [f_0(x_2, x_3),\ f_1(x_1, x_2, y, r),\ f_2(x_3, x_4, y)] \\ \text{subject to} \quad & g_1(x_1, y) \le 0 \\ & g_2(x_3, x_4, y, u) \le 0 \\ & r = a_1(x_2, y) \\ & u = a_2(x_3, x_4, r) \end{aligned} \tag{3}$$

This problem may be partitioned into two subproblems. The first subproblem has local variables $\{x_1, x_2\}$ and local functions $\{f_1, g_1, a_1\}$. The second subproblem has local variables $\{x_3, x_4, u\}$ and local functions $\{f_2, g_2, a_2\}$. The two components are coupled through the variables $\{y, r\}$ and the system-wide objective $\{f_0\}$ that depends on variable x_2 of the first subproblem and on x_3 of the second. The partitioning structure is depicted in Fig. 2.

For this partitioned problem, the specification of the two components is given below.

```
comp First =
|[ extvar x2, y, r
   intvar x1
   objfunc f1(x1, x2, y, r)
   confunc g1(x1, y)
   resfunc r = a1(x2, y)
]|

comp Second =
|[ extvar x3, y, r
   intvar x4, u
   objfunc f2(x3, x4, y)
   confunc g2(x3, x4, y, u)
   resfunc u = a2(x3, x4, r)
]|
```

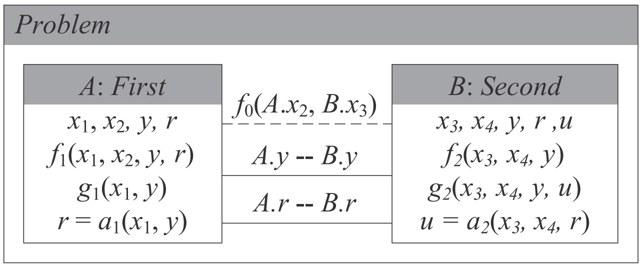


Fig. 2 Illustration of the specified partition for problem (3)

The first component has name *First* and has four variables x_1, x_2, y, r and three functions f_1, g_1, a_1. The language distinguishes between two types of variables: external variables defined after the keyword `extvar`, and internal variables defined after the keyword `intvar`. External variables can be accessed by the system the component is part of. External variables can be shared variables or coupling variables that are communicated between components, or local variables on which system-wide functions depend. Variables y, r fall in the former category and x_2 falls in the latter since it is an argument of the system-wide objective f_0. Internal variables are only accessible within the component.

The reason for taking a division of variables different from the traditional local and coupling/shared variables is that from a system designer's viewpoint it is relevant to know which variables have an influence beyond the component in which they are defined. From this perspective, external variables are those variables that affect other components and systems and therefore also include local variables that are arguments of system-wide functions.

Three groups of functions are available in the Ψ language: objective functions, constraint functions, and response functions. In component *First* of the example, function $f_1(x_1, x_2, y, r)$ is a local objective with four arguments x_1, x_2, y, r, and function $g_1(x_1, y)$ is a local constraint with two arguments x_1, y. Function $a_1(x_2, y)$ is a response function that determines the values of variable r. A response function may have multiple variables as outputs. It is possible to apply the same function multiple times with different arguments.

Definitions of variables and functions in components (and systems) have a local scope. Variables and functions defined in one component may have the same name as other variables and functions of another component without being automatically coupled. Instead, interactions between components have to be specified in systems.

It is important to realize that a component definition is *independent* on the choice of coordination method. At the coordination stage, the system designer can use a multi-level coordination method and formulate an optimization problem for each component. Alternatively, using a single-level coordination method only assigns analysis capabilities to components, and decision-making is centralized in a single optimization problem. Hence, defining design variables and objective and constraint functions in a component does not necessarily imply that an optimization problem is actually formulated for this component. It simply indicates where the variables and functions originate from.

3.2 Systems

Once the components of a partitioned problem are defined, they can be assembled into systems. A system definition includes two or more subcomponents and describes the couplings between them. The subcomponents of a system can be components or other systems. In the latter case, a multi-level system is obtained.

The system definition for the example partitioning of problem (3) is given by

```
syst Problem =
|[ sub A: First, B: Second
   link A.y -- B.y, A.r -- B.r
   objfunc f0(A.x2, B.x3)
]|
```

The system is named *Problem* and has two subcomponents: A of type *First*, and B of type *Second*. Multiple subcomponents of the same type can be instantiated in a system. The expression A_1, A_2: *First* instantiates two subcomponents A_1 and A_2 of the same type *First*. These multiple instantiations are useful for systems that have many identical components, such as structural systems consisting of many similar elements.

The consistency constraints between the two components are given by the `link` statement that connects variables y and r of component A to variables y and r of component B, respectively (note that the linked variables need not have the same local name). In systems, the dot notation $A.y$ denotes variable y from component A.

The specification of the system is completed by the definition of the system-wide objective function f_0 that depends on variable x_2 of A and x_3 of subcomponent B. Systems can also have system-wide constraint functions or response functions.

In contrast to components, a system does not have *design* variables of its own. However, *response* variables associated with the coupling analysis functions have to be included as variables of the system definition. Similar to components, the keywords `extvar` and `intvar` are used to define which response variables are external and which are internal.

The systems used in Ψ are different from the traditional notion of systems in the MDO context. Here, a system is simply a collection of components that are coordinated jointly. Systems in the MDO context typically also include design aspects and typically have design variables of their own (so-called global variables or system variables). The task of these MDO systems is to solve the system-level design problem *while at the same time* coordinating the solution of the subproblems. These are actually two separate tasks that should be considered as such. In Ψ, this distinction is made explicit since a user needs to define the design part of the MDO system in a component, while the couplings associated with the coordination part are specified in a system definition.

The final ingredient of the partitioned problem specification for the example partitioning of problem (3) is the statement

```
topsyst Problem
```

which instantiates the partitioned problem by defining that the highest system in the hierarchy is *Problem*. The definitions for components *First* and *Second*, system *Problem*, and the `topsyst` statement comprise the specification of the partitioned problem for our example problem.

4 Automatic processing and generation of input files

A compiler and two generators have been developed to automatically derive input files for a coordination framework and a matrix representation of the problem structure. The two generators presented in this article should be seen as examples of how framework-specific input files can be automatically derived. The development of additional generators for other frameworks are expected to be easy to add due to the use of the generic INI format, and the information created by the compiler. The compiler and the two generators have been coded in Python (Lutz 2006).

The compiler-based approach proposed in this article offers developers of coordination methods the freedom to focus on input files that integrate easily with the computational routines they are designing. The Ψ language and the associated compiler and generators should therefore be seen as powerful, generic pre-processors that provide these computational frameworks with easy-to-process input specifications while allowing users to specify the partitioned problem in an intuitive and easy way.

4.1 Partitioned problem normalized format

The compiler checks a partitioned problem specification in Ψ for errors, and translates it into a specification in INI format. The compiler checks for around 50 semantic requirements such as

- Uniqueness of variable/component/system names,
- Whether arguments and outputs of functions are defined as variables in components/systems,
- Whether sub-components of a system refer to existing component or system definitions,

- Whether variables used in systems exist in the associated sub-component,
- Etc.

Informative error messages are generated to assist the user in debugging incorrect specifications. Checking partitions in this early stage assures that further automated processing at later stages does not require to do so and can rely on correctly specified partitions. The reader is referred to the user manual (Tosserams et al. 2009b) for a complete list of the semantic requirements that are checked for.

After a specification is checked for errors, an INI-specification is generated. The generated INI-specifications are less compact and harder to read than specifications in Ψ, but have the advantage that they can be easily interpreted by programs in other languages (Cloanto 2009). The INI-format serves as a normalized format between Ψ and coordination frameworks. Figure 3 illustrates the relations between the different files and the associated compiler and generators.

The specification of the partitioned problem in INI format is defined by a number of sections. Each section contains a section header [*section*] and a number of key/value pairs of the form *keyname* = *value*. Separate sections are introduced for each variable, each function, each component, each coupling link, each system, and one for the top-level system. The collection of sections contains the necessary information to uniquely represent the partitioned problem.

The contents of the normalized file generated from the Ψ-specification for the example partitioning of problem (3) are given in Fig. 4. The order of sections and keys in this file may appear unconventional, but this is not an issue since the file is intended for further automatic processing rather than for human understanding. For variable and function sections, the key-value pairs define, respectively, the variable's/function's name in the Ψ-specification (`name`), the component or system definition in which it is specified (`defined_in`), and the instantiation path of this definition (`path`). Function sections also include the keys `argvars` and `resvars` that define the arguments and responses of a function, respectively (where only analysis functions have responses).

Component and system sections include keys for its definition name (`type`), the name of its instantiation (`name`) and the associated instantiation path (`path`),

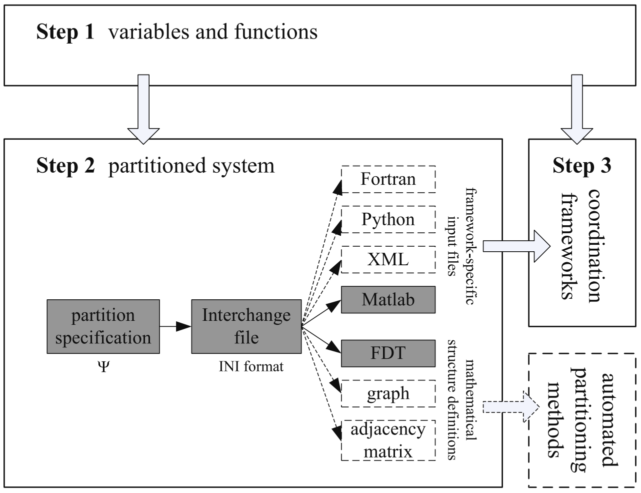

Step 1 variables and functions
Step 2 partitioned system
Step 3 coordination frameworks
partition specification
Ψ
Interchange file
INI format
Fortran
Python
XML
Matlab
FDT
graph
adjacency matrix
framework-specific input files
mathematical structure definitions
automated partitioning methods

Fig. 3 Relations between the available specification formats for Step 2. Within Step 2, *boxes* represent partition formats and *arrows* are associated with compilers and generators. *Shaded boxes* and *solid arrows* represent the currently implemented format, compiler, and generators

```
[func_5]
path = Problem.B
argvars = var_7, var_5, var_8, var_6
name = g2
defined_in = Second

[func_4]
path = Problem.B
argvars = var_7, var_5, var_8
name = f2
defined_in = Second

[func_7]
path = Problem
argvars = var_2, var_7
name = f0
defined_in = Problem

[func_6]
path = Problem.B
resvars = var_6
argvars = var_7, var_5, var_9
name = a2
defined_in = Second

[func_1]
path = Problem.A
argvars = var_1, var_2, var_3
name = f1
defined_in = First

[func_3]
path = Problem.A
resvars = var_4
argvars = var_2, var_3
name = a1
defined_in = First

[func_2]
path = Problem.A
argvars = var_1, var_3
name = g1
defined_in = First

[top]
system = syst_1

[var_9]
path = Problem.B
name = r
defined_in = Second

[var_8]
path = Problem.B
name = y
defined_in = Second

[var_7]
path = Problem.B
name = x3
defined_in = Second

[var_6]
path = Problem.B
name = u
defined_in = Second

[var_5]
path = Problem.B
name = x4
defined_in = Second

[var_4]
path = Problem.A
name = r
defined_in = First

[var_3]
path = Problem.A
name = y
defined_in = First

[var_2]
path = Problem.A
name = x2
defined_in = First

[var_1]
path = Problem.A
name = x1
defined_in = First

[syst_1]
name = Problem
links = link_1, link_2
objfuncs = func_7
sub_comps = comp_1, comp_2
path = Problem
type = Problem

[comp_2]
resfuncs = func_6
name = B
local_resvars = var_6
objfuncs = func_4
local_vars = var_7, var_5
coupling_vars = var_9, var_8
confuncs = func_5
path = Problem.B
type = Second

[comp_1]
resfuncs = func_3
name = A
objfuncs = func_1
coupling_resvars = var_4
local_vars = var_2, var_1
coupling_vars = var_3
confuncs = func_2
path = Problem.A
type = First

[link_2]
defined_in = syst_1
coupling = var_4, var_9
path = Problem

[link_1]
defined_in = syst_1
coupling = var_3, var_8
path = Problem
```

Fig. 4 Contents of normalized file generated from Ψ-specification of the example partitioning of problem (3)

its shared and local variables (`coupling_vars`[2] and `local_vars`, only for components), its coupling and local responses (`coupling_resvars` and `local_resvars`), and its objective, constraint, and response functions (`objfuncs`, `confuncs`, `resfuncs`). System sections also include a list of sub-components (`sub_comps`) and links (`links`). Note that the local and shared variables correspond to the definitions of $\mathbf{x}_j$ and $\mathbf{y}_j$ in the partitioned problem (2). Response functions $\mathbf{r}_j$ are split into local (i.e. disciplinary) responses and coupling responses similarly.

A coupling section (`link_`) includes the variables that it couples (`coupling`), the name of the system definition in which it is defined (`defined_in`), and the instantiation path of this system (`path`). A coupling can be defined between two shared design variables or between two coupling response variables. Finally, the top section includes the key `system` whose value denotes the name of the top-level system.

4.2 Matlab input file for ALC toolbox

The first generator translates the INI output into Matlab problem specification files that can be used as input for our Matlab implementation of the augmented Lagrangian coordination algorithm (ALC, Tosserams et al. 2008, 2009c). This ALC-generator was the original motivation for the work presented in this article. It is beyond the scope of this article to discuss the ALC method or the details of the input files in greater detail. Our intention is to present the generated ALC files to demonstrate the possibilities that the compiler-based approach offers. The interested reader is referred to the references given above for further details on the ALC method.

The ALC input files make use of matrices, vectors, and similar data types, which are easily processable with standard Matlab commands. Although the ALC format is very different from Ψ or normalized specifications, the generator can automatically generate ALC files from the partitioned problem specification in INI format. The contents of the generated Matlab file for the example partitioning of problem (3) is given in Fig. 5. The ALC toolbox does not allow response functions or response variables, and the response functions $r = a_1(x_2, y)$ and $u = a_2(x_3, x_4, r)$ of the example have been included as constraint functions $h_1(r, x_2, y) = r - a_1(x_2, y) = 0$ and $h_2(u, x_3, x_4, r) = u - a_2(x_3, x_4, r) = 0$ for this purpose. The ALC-generator automatically checks whether a Ψ specification has response functions or not. The reason for including these checks in the ALC-generator (and not in the compiler) is that Ψ is generic, i.e. independent of the coordination method. The difference between the Matlab and Ψ specifications is obvious, as well as the difference in readability between the two. Specification of the partitioned problem using Ψ is clearly more intuitive than specifying them using the ALC format in Matlab.

```
%
% Generated by np2alc
%
function PR = example1alc(PR)

PR.main.z_id = [1, 2, 3, 4, 5, 6, 7, 8, 9];
PR.main.z_names = {'x1', 'x2', 'y', 'r', 'x4', 'u', 'x3', 'y', 'r'};
PR.main.z_comps = {'First', 'First', 'First', 'First',...
 'Second', 'Second', 'Second', 'Second', 'Second'};

PR.main.m = 2;
PR.main.comp_id = [1, 2];
PR.main.comp_names = {'A', 'B'};

PR.main.obj(1).name = 'f0';
PR.main.obj(1).args = [2, 7];

PR.main.con(1).name = 'none';
PR.main.con(1).args = [];

PR.sub(1).x_id = [1, 2];
PR.sub(1).y_id = [3, 4];

PR.sub(1).obj(1).name = 'f1';
PR.sub(1).obj(1).args = [1, 2, 3];

PR.sub(1).con(1).name = 'g1';
PR.sub(1).con(1).args = [1, 3];
PR.sub(1).con(2).name = 'h1';
PR.sub(1).con(2).args = [4, 2, 3];

PR.sub(1).to(2).Sij = [0,1; 1,0];

PR.sub(2).x_id = [5, 6, 7];
PR.sub(2).y_id = [8, 9];

PR.sub(2).obj(1).name = 'f2';
PR.sub(2).obj(1).args = [7, 5, 8];

PR.sub(2).con(1).name = 'g2';
PR.sub(2).con(1).args = [7, 5, 8, 6];
PR.sub(2).con(2).name = 'h2';
PR.sub(2).con(2).args = [6, 7, 5, 9];

PR.sub(2).to(1).Sij = [0,1; 1,0];
```

Fig. 5 Contents of ALC input file generated from normalized specification of the example partitioning of problem (3)

4.3 Function dependence table file

A second generator creates a file that contains the functional dependence table (FDT) of the specified problem. The FDT is a matrix whose rows and columns are associated with the functions and variables of the problem, respectively. The (i, j)-th entry of the matrix is 1 if the function of row i depends on the variable of column j. The FDT and related mathematical representations are typical inputs to model-based partitioning methods such as those proposed by Krishnamachari and Papalambros (1997), Michelena and Papalambros (1997), Chen et al. (2005), Li and Chen (2006), and Allison et al. (2007).

[2]We use the term `coupling_vars` for shared variables, and `coupling_resvars` for coupling responses.

```
Columns:
 1: Problem.A.x2
 2: Problem.B.x3
 3: Problem.A.y
 4: Problem.B.y
 5: Problem.A.r
 6: Problem.B.r
 7: Problem.A.x1
 8: Problem.B.x4
 9: Problem.B.u

Rows:
 1: Problem.f0
 2: Problem.A.y -- Problem.B.y
 3: Problem.A.r -- Problem.B.r
 4: Problem.A.f1
 5: Problem.A.g1
 6: Problem.A.a1
 7: Problem.B.f2
 8: Problem.B.g2
 9: Problem.B.a2

FDT:
1, 1, 0, 0, 0, 0, 0, 0, 0
0, 0, 1, 1, 0, 0, 0, 0, 0
0, 0, 0, 0, 1, 1, 0, 0, 0
1, 0, 1, 0, 0, 0, 1, 0, 0
0, 0, 1, 0, 0, 0, 1, 0, 0
1, 0, 1, 0, 1, 0, 0, 0, 0
0, 1, 0, 1, 0, 0, 0, 1, 0
0, 1, 0, 1, 0, 0, 0, 1, 1
0, 1, 0, 0, 0, 1, 0, 1, 1
```

Fig. 6 Contents of FDT file generated from the normalized specification of example (3)

The generated functional dependence table file for the example partitioning of problem (3) is given in Fig. 6. Having to specify a problem's structure in a functional dependence table is clearly a tedious process that is prone to errors and becomes increasingly prohibitive as systems become larger. Specifying problem structures using Ψ provides a much more intuitive environment for this purpose. The FDT can be automatically generated from the Ψ specification, using the INI normalized format.

Being able to generate different types of input files from the same Ψ specification not only saves time, but also leads to consistent definitions of the partitioned problem. These advantages make comparing results from different computational frameworks easier.

5 Chassis design example

In this section, we demonstrate the use and advantages of Ψ on a larger example. Two variants of partitioning the problem are demonstrated using the Ψ language. For one variant, the INI output files, as well as the Matlab ALC files and the FDT table are generated, clearly showing the compactness and intuitiveness of the specification in Ψ.

The example is a vehicle chassis design problem taken from Kim et al. (2003) that aims at optimizing five handling and ride quality metrics while considering the design of front and rear suspensions, and vertical and cornering stiffness models. A detailed description of the problem can be found in Kim et al. (2003). The reader is referred to Table 1 for a brief description of the optimization variables.

Table 1 Description of the optimization variables for the vehicle chassis problem

Design variables		Response variables	
a	Tire position	ω_{sf}	Spring nat. freq.
b	Tire position	ω_{sr}	Spring nat. freq.
P_{if}	Tire pressure	ω_{tf}	Tire nat. freq.
P_{ir}	Tire pressure	ω_{tr}	Tire nat. freq.
D_{f}	Coil diameter	k_{us}	Understeer gradient
D_{r}	Coil diameter	K_{sf}	Spring stiffness
d_{f}	Wire diameter	K_{sr}	Spring stiffness
d_{r}	Wire diameter	K_{tf}	Tire stiffness
p_{f}	Pitch	K_{tr}	Tire stiffness
p_{r}	Pitch	$C_{\alpha\mathrm{f}}$	Cornering stiffness
Z_{sf}	Suspension deflection	$C_{\alpha\mathrm{r}}$	Cornering stiffness
Z_{sr}	Suspension deflection	K_{Lf}	Linear stiffness
		K_{Lr}	Linear stiffness
		K_{Bf}	Bending stiffness
		K_{Br}	Bending stiffness
		$L_{0\mathrm{f}}$	Free length
		$L_{0\mathrm{r}}$	Free length

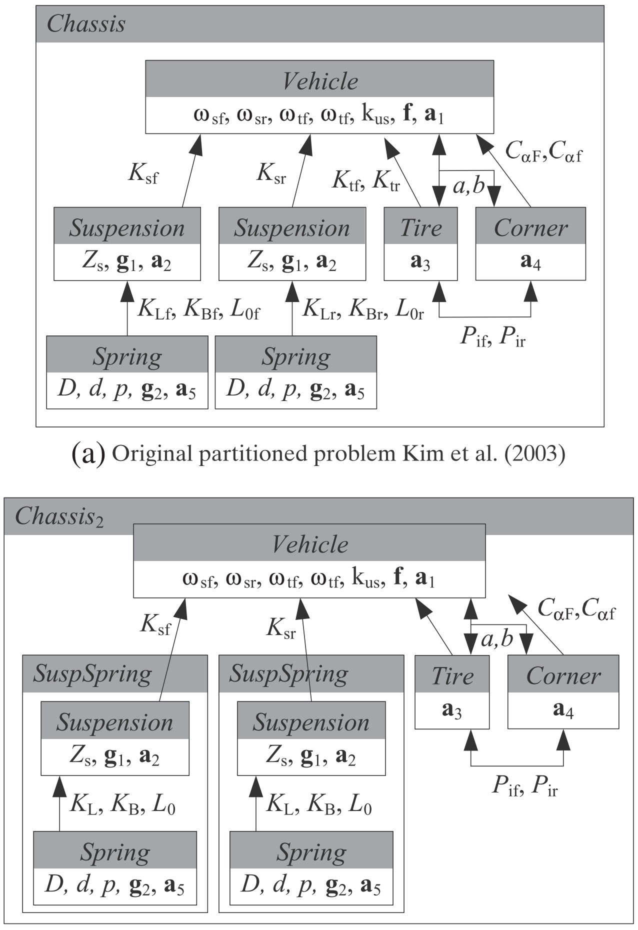
Indices "*f*" refer to front and "*r*" to rear

The chassis design optimization problem is given by

$$\begin{aligned}
\text{find}\quad & a, b, \omega_{\mathrm{sf}}, \omega_{\mathrm{sr}}, \omega_{\mathrm{tf}}, \omega_{\mathrm{tr}}, k_{\mathrm{us}}, K_{\mathrm{sf}}, K_{\mathrm{sr}}, K_{\mathrm{tf}}, K_{\mathrm{tr}},\\
& C_{\alpha\mathrm{f}}, C_{\alpha\mathrm{r}}, Z_{\mathrm{sf}}, Z_{\mathrm{sr}}, K_{\mathrm{Lf}}, K_{\mathrm{Lr}}, K_{\mathrm{Bf}}, K_{\mathrm{Br}}, L_{0\mathrm{f}},\\
& L_{0\mathrm{r}}, P_{\mathrm{if}}, P_{\mathrm{ir}}, D_{\mathrm{f}}, D_{\mathrm{r}}, d_{\mathrm{f}}, d_{\mathrm{r}}, p_{\mathrm{f}}, p_{\mathrm{r}}\\
\text{min}\quad & \mathbf{f}(\omega_{\mathrm{sf}}, \omega_{\mathrm{sr}}, \omega_{\mathrm{tf}}, \omega_{\mathrm{tr}}, k_{\mathrm{us}})\\
\text{subject to}\quad & \mathbf{g}_1(Z_{\mathrm{sf}}, K_{\mathrm{Lf}}, K_{\mathrm{Bf}}, L_{0\mathrm{f}}) \le \mathbf{0}\\
& \mathbf{g}_1(Z_{\mathrm{sr}}, K_{\mathrm{Lr}}, K_{\mathrm{Br}}, L_{0\mathrm{r}}) \le \mathbf{0}\\
& \mathbf{g}_2(D_{\mathrm{f}}, d_{\mathrm{f}}, p_{\mathrm{f}}) \le \mathbf{0}\\
& \mathbf{g}_2(D_{\mathrm{r}}, d_{\mathrm{r}}, p_{\mathrm{r}}) \le \mathbf{0}\\
& (\omega_{\mathrm{sf}}, \omega_{\mathrm{sr}}, \omega_{\mathrm{tf}}, \omega_{\mathrm{tr}}, k_{\mathrm{us}})\\
& \quad = \mathbf{a}_1(a, b, K_{\mathrm{sf}}, K_{\mathrm{sr}}, K_{\mathrm{tf}}, K_{\mathrm{tr}}, C_{\alpha\mathrm{f}}, C_{\alpha\mathrm{r}})\\
& K_{\mathrm{sf}} = \mathbf{a}_2(Z_{\mathrm{sf}}, K_{\mathrm{Lf}}, K_{\mathrm{Bf}}, L_{0\mathrm{f}})\\
& K_{\mathrm{sr}} = \mathbf{a}_2(Z_{\mathrm{sr}}, K_{\mathrm{Lr}}, K_{\mathrm{Br}}, L_{0\mathrm{r}})\\
& (K_{\mathrm{tf}}, K_{\mathrm{tr}}) = \mathbf{a}_3(P_{\mathrm{if}}, P_{\mathrm{ir}}, a, b)\\
& (C_{\alpha\mathrm{f}}, C_{\alpha\mathrm{r}}) = \mathbf{a}_4(P_{\mathrm{if}}, P_{\mathrm{ir}}, a, b)\\
& (K_{\mathrm{Lf}}, K_{\mathrm{Bf}}) = \mathbf{a}_5(D_{\mathrm{f}}, d_{\mathrm{f}}, p_{\mathrm{f}}, L_{0\mathrm{f}})\\
& (K_{\mathrm{Lr}}, K_{\mathrm{Br}}) = \mathbf{a}_5(D_{\mathrm{r}}, d_{\mathrm{r}}, p_{\mathrm{r}}, L_{0\mathrm{r}})
\end{aligned} \tag{4}$$

5.1 Specification of the partitioned problem

The partitioned problem given in Kim et al. (2003) is specified in Ψ below, and is illustrated in Fig. 7a. The system *Chassis* has seven sub-components: *Vehicle*, *Tire*, *Corner*, two of type *Suspension*, and two of type *Spring*. Each sub-component includes its relevant set of optimization variables and functions. The similarity of the front



(a) Original partitioned problem Kim et al. (2003)

(b) Alternative partitioned problem with suspension-spring systems

Fig. 7 Two problem partitions for the chassis design example

and rear suspensions and springs is exploited by defining a single suspension and a single spring component. By instantiating these components twice in system *Chassis*, two independent subproblems are defined, each with a separate set of design variables.

```
comp Vehicle =
|[ extvar $a, b, K_{sf}, K_{sr}, K_{tf}, K_{tr}, C_{\alpha f}, C_{\alpha r}$
   intvar $\omega_{sf}, \omega_{sr}, \omega_{tf}, \omega_{tr}, k_{us}$
   objfunc $\mathbf{f}(\omega_{sf}, \omega_{sr}, \omega_{tf}, \omega_{tr}, k_{us})$
   resfunc $(\omega_{sf}, \omega_{sr}, \omega_{tf}, \omega_{tr}, k_{us}) =$
              $\mathbf{a}_1(a, b, K_{sf}, K_{sr}, K_{tf}, K_{tr}, C_{\alpha f}, C_{\alpha r})$
]|

comp Tire =
|[ extvar $a, b, K_{tf}, K_{tr}, P_{if}, P_{ir}$
   resfunc $(K_{tf}, K_{tr}) = \mathbf{a}_3(P_{if}, P_{ir}, a, b)$
]|

comp Corner =
|[ extvar $a, b, C_{\alpha f}, C_{\alpha r}, P_{if}, P_{ir}$
   resfunc $(C_{\alpha f}, C_{\alpha r}) = \mathbf{a}_4(P_{if}, P_{ir}, a, b)$
]|

comp Suspension =
|[ extvar $K_s, K_L, K_B, L_0$
   intvar $Z_s$
   confunc $\mathbf{g}_1(Z_s, K_L, K_B, L_0)$
   resfunc $K_s = \mathbf{a}_2(Z_s, K_L, K_B, L_0)$
]|

comp Spring =
|[ extvar $K_L, K_B, L_0$
   intvar $D, d, p$
   confunc $\mathbf{g}_2(D, d, p)$
   resfunc $(K_L, K_B) = \mathbf{a}_5(D, d, p, L_0)$
]|

syst Chassis =
 |[ sub   $V$: Vehicle, $T$: Tire, $C$: Corner
        , $S_f, S_r$: Suspension, $Sp_f, Sp_r$: Spring
    link  $V.a$ -- $\{T.a, C.a\}$,    $T.P_{if}$ -- $C.P_{if}$
        , $V.b$ -- $\{T.b, C.b\}$,    $T.P_{ir}$ -- $C.P_{ir}$
        , $V.K_{tf}$ -- $T.K_{tf}$,   $V.C_{\alpha f}$ -- $C.C_{\alpha f}$
        , $V.K_{tr}$ -- $T.K_{tr}$,   $V.C_{\alpha r}$ -- $C.C_{\alpha r}$
        , $V.K_{sf}$ -- $S_f.K_s$,    $V.K_{sr}$ -- $S_r.K_s$
        , $S_f.K_L$ -- $Sp_f.K_L$,    $S_f.L_0$ -- $Sp_f.L_0$
        , $S_r.K_L$ -- $Sp_r.K_L$,    $S_r.L_0$ -- $Sp_r.L_0$
        , $S_f.K_B$ -- $Sp_f.K_B$
        , $S_r.K_B$ -- $Sp_r.K_B$
]|

topsyst Chassis
```

A second partitioning of the problem as shown in Fig. 7 is used to demonstrate how multi-level coordination can be facilitated by including systems as sub-components of other systems. This partition has a subsystem *SuspSpring* that includes a *Suspension* and a *Spring* component. Two instantiations of this lower-level system are included in a system *Chassis*$_2$ that also includes the *Vehicle*, *Tire*, and *Corner* components of the first definition above. The differences between the two partitioned problems are illustrated in Fig. 7. The specification of the systems *SuspSpring* and *Chassis*$_2$ for second problem partitioning is given below.

```
syst SuspSpring =
|[ sub $S$: Suspension, $Sp$: Spring
   link $S.K_L$ -- $Sp.K_L$, $S.L_0$ -- $Sp.L_0$, $S.K_B$ -- $Sp.K_B$
   alias $K_s = S.K_s$
]|
[var_44]
path = Chassis.Spr
name = p
defined_in = Spring

[var_45]
path = Chassis.Spr
name = KL
defined_in = Spring

[var_46]
path = Chassis.Spr
name = KB
defined_in = Spring

[var_47]
path = Chassis.Spr
name = L0
defined_in = Spring

[var_40]
path = Chassis.Spf
name = KB
defined_in = Spring

[var_41]
path = Chassis.Spf
name = L0
defined_in = Spring

[var_42]
path = Chassis.Spr
name = D
defined_in = Spring

[var_43]
path = Chassis.Spr
name = d
defined_in = Spring

[func_11]
path = Chassis.Spr
argvars = var_42, var_43, var_44
name = g2
defined_in = Spring

[func_10]
path = Chassis.Spf
resvars = var_39, var_40
argvars = var_36, var_37, var_38,
    var_41
name = a5
defined_in = Spring

[func_12]
path = Chassis.Spr
resvars = var_45, var_46
argvars = var_42, var_43, var_44,
    var_47
name = a5
defined_in = Spring

[var_9]
path = Chassis.V
name = Ksr
defined_in = Vehicle

[var_8]
path = Chassis.V
name = Ksf
defined_in = Vehicle

[var_7]
path = Chassis.V
name = b
defined_in = Vehicle

[var_6]
path = Chassis.V
name = a
defined_in = Vehicle

[var_5]
path = Chassis.V
name = kus
defined_in = Vehicle

[var_4]
path = Chassis.V
name = wtr
defined_in = Vehicle

[var_3]
path = Chassis.V
name = wtf
defined_in = Vehicle

[var_2]
path = Chassis.V
name = wsr
defined_in = Vehicle

[var_1]
path = Chassis.V
name = wsf
defined_in = Vehicle

[func_9]
path = Chassis.Spf
argvars = var_36, var_37, var_38
name = g2
defined_in = Spring

[func_8]
path = Chassis.Sr
resvars = var_32
argvars = var_31, var_33, var_34,
    var_35
name = a2
defined_in = Suspension

[func_5]
path = Chassis.Sf
argvars = var_26, var_28, var_29,
    var_30
name = g1
defined_in = Suspension

[func_4]
path = Chassis.C
resvars = var_22, var_23
argvars = var_24, var_25, var_20,
    var_21
name = a4
defined_in = Corner

[func_7]
path = Chassis.Sr
argvars = var_31, var_33, var_34,
    var_35
name = g1
defined_in = Suspension

[func_6]
path = Chassis.Sf
resvars = var_27
argvars = var_26, var_28, var_29,
    var_30
name = a2
defined_in = Suspension

[func_1]
path = Chassis.V
argvars = var_1, var_2, var_3,
    var_4, var_5
name = f
defined_in = Vehicle

[func_3]
path = Chassis.T
resvars = var_16, var_17
argvars = var_18, var_19, var_14,
    var_15
name = a3
defined_in = Tire

[func_2]
path = Chassis.V
resvars = var_1, var_2, var_3,
    var_4, var_5
argvars = var_6, var_7, var_8,
    var_9, var_10, var_11, var_12,
    var_13
name = a1
defined_in = Vehicle

[var_28]
path = Chassis.Sf
name = KL
defined_in = Suspension

[var_29]
path = Chassis.Sf
name = KB
defined_in = Suspension

[var_26]
path = Chassis.Sf
name = Zs
defined_in = Suspension

[var_27]
path = Chassis.Sf
name = Ks
defined_in = Suspension

[var_24]
path = Chassis.C
name = Pif
defined_in = Corner

[var_25]
path = Chassis.C
name = Pir
defined_in = Corner

[var_22]
path = Chassis.C
name = Caf
defined_in = Corner

[var_23]
path = Chassis.C
name = Car
defined_in = Corner

[var_20]
path = Chassis.C
name = a
defined_in = Corner

[var_21]
path = Chassis.C
name = b
defined_in = Corner

[var_39]
path = Chassis.Spf
name = KL
defined_in = Spring

[var_38]
path = Chassis.Spf
name = p
defined_in = Spring

[var_35]
path = Chassis.Sr
name = L0
defined_in = Suspension

[var_34]
path = Chassis.Sr
name = KB
defined_in = Suspension

[var_37]
path = Chassis.Spf
name = d
defined_in = Spring

[var_36]
path = Chassis.Spf
name = D
defined_in = Spring

[var_31]
path = Chassis.Sr
name = Zs
defined_in = Suspension

[var_30]
path = Chassis.Sf
name = L0
defined_in = Suspension

[var_33]
path = Chassis.Sr
name = KL
defined_in = Suspension

[var_32]
path = Chassis.Sr
name = Ks
defined_in = Suspension

[link_9]
defined_in = syst_1
coupling = var_11, var_17
path = Chassis

[link_8]
defined_in = syst_1
coupling = var_12, var_22
path = Chassis

[link_3]
defined_in = syst_1
coupling = var_18, var_24
path = Chassis

[link_2]
defined_in = syst_1
coupling = var_6, var_14
path = Chassis

[link_1]
defined_in = syst_1
coupling = var_6, var_20
path = Chassis

[link_7]
defined_in = syst_1
coupling = var_10, var_16
path = Chassis

[link_6]
defined_in = syst_1
coupling = var_19, var_25
path = Chassis

[link_5]
defined_in = syst_1
coupling = var_7, var_15
path = Chassis

[link_4]
defined_in = syst_1
coupling = var_7, var_21
path = Chassis

[topsyst]
system = syst_1

[var_17]
path = Chassis.T
name = Ktr
defined_in = Tire

[var_16]
path = Chassis.T
name = Ktf
defined_in = Tire

[var_15]
path = Chassis.T
name = b
defined_in = Tire

[var_14]
path = Chassis.T
name = a
defined_in = Tire

[var_13]
path = Chassis.V
name = Car
defined_in = Vehicle

[var_12]
path = Chassis.V
name = Caf
defined_in = Vehicle

[var_11]
path = Chassis.V
name = Ktr
defined_in = Vehicle

[var_10]
path = Chassis.V
name = Ktf
defined_in = Vehicle

[var_19]
path = Chassis.T
name = Pir
defined_in = Tire

[var_18]
path = Chassis.T
name = Pif
defined_in = Tire

[syst_1]
path = Chassis
type = Chassis
name = Chassis
links = link_1, link_2, link_3,
    link_4, link_5, link_6, link_7,
    link_8, link_9, link_10,
    link_11, link_12, link_13,
    link_14, link_15, link_16,
    link_17, link_18
subs = comp_1, comp_2, comp_3,
    comp_4, comp_5, comp_6, comp_7

[link_13]
defined_in = syst_1
coupling = var_30, var_41
path = Chassis

[link_12]
defined_in = syst_1
coupling = var_9, var_32
path = Chassis

[link_11]
defined_in = syst_1
coupling = var_8, var_27
path = Chassis

[link_10]
defined_in = syst_1
coupling = var_13, var_23
path = Chassis

[link_17]
defined_in = syst_1
coupling = var_29, var_40
path = Chassis

[link_16]
defined_in = syst_1
coupling = var_33, var_45
path = Chassis

[link_15]
defined_in = syst_1
coupling = var_28, var_39
path = Chassis

[link_14]
defined_in = syst_1
coupling = var_35, var_47
path = Chassis

[link_18]
defined_in = syst_1
coupling = var_34, var_46
path = Chassis

[comp_6]
resfuncs = func_10
name = Spf
coupling_resvars = var_39,
    var_40
local_vars = var_37, var_36,
    var_38
coupling_vars = var_41
confuncs = func_9
path = Chassis.Spf
type = Spring

[comp_7]
resfuncs = func_12
name = Spr
coupling_resvars = var_45,
    var_46
local_vars = var_44, var_42, var_43
coupling_vars = var_47
confuncs = func_11
path = Chassis.Spr
type = Spring

[comp_4]
resfuncs = func_6
name = Sf
coupling_resvars = var_27
local_vars = var_26
coupling_vars = var_30, var_28, var_29
confuncs = func_5
path = Chassis.Sf
type = Suspension

[comp_5]
resfuncs = func_8
name = Sr
coupling_resvars = var_32
local_vars = var_31
coupling_vars = var_35, var_34, var_33
confuncs = func_7
path = Chassis.Sr
type = Suspension

[comp_2]
resfuncs = func_3
name = T
coupling_resvars = var_17, var_16
coupling_vars = var_15, var_14,
    var_19, var_18
path = Chassis.T
type = Tire

[comp_3]
resfuncs = func_4
name = C
coupling_resvars = var_22, var_23
coupling_vars = var_24, var_25,
    var_20, var_21
path = Chassis.C
type = Corner

[comp_1]
resfuncs = func_2
name = V
local_resvars = var_5, var_4, var_3,
    var_2, var_1
objfuncs = func_1
coupling_vars = var_13, var_12,
    var_11, var_10, var_9, var_8,
    var_7, var_6
path = Chassis.V
type = Vehicle
```

Fig. 8 Contents of normalized file generated from Ψ-specification of chassis example—partition 1

```
syst $Chassis_2$ =
 |[ sub   $V$: $Vehicle$, $T$: $Tire$, $C$: $Corner$
        , $S_f$, $S_r$: $SuspSpring$
    link  $V.a$ -- $\{T.a, C.a\}$,     $T.P_{if}$ -- $C.P_{if}$
        , $V.b$ -- $\{T.b, C.b\}$,     $T.P_{ir}$ -- $C.P_{ir}$
        , $V.K_{tf}$ -- $T.K_{tf}$,      $V.C_{\alpha f}$ -- $C.C_{\alpha f}$
        , $V.K_{tr}$ -- $T.K_{tr}$,      $V.C_{\alpha r}$ -- $C.C_{\alpha r}$
        , $V.K_{sf}$ -- $S_f.K_s$,       $V.K_{sr}$ -- $S_r.K_s$
 ]|

topsyst $Chassis_2$
```

The couplings between the variables of *Suspension* and *Spring* are included in the system *SuspSpring*. Two systems *SuspSpring* are instantiated in system *Chassis*$_2$, and links between the different sub-components are defined accordingly. With this second partitioning, the coordination of the *SuspSpring* lower-level systems can be performed nested within the coordination of the top-level system *Chassis*$_2$.

System *SuspSpring* includes the definition of an *alias* (K_s), which is introduced to make this variable of component *Suspension* accessible by system *Chassis*$_2$. In general, aliases are used in systems that are themselves part of

```
%
% Generated by np2alc
%
function PR = chassis1alc(PR)

PR.main.z_id = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,...
    22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,...
    44, 45, 46, 47];
PR.main.z_names = {'wsf', 'wsr', 'wtf', 'wtr', 'kus', 'a', 'b', 'Ksf', 'Ksr', 'Ktf', 'Ktr',...
    'Caf', 'Car', 'a', 'b', 'Ktf', 'Ktr', 'Pif', 'Pir', 'a', 'b', 'Caf', 'Car', 'Pif',...
    'Pir', 'Zs', 'Ks', 'KL', 'KB', 'L0', 'Zs', 'Ks', 'KL', 'KB', 'L0', 'D', 'd', 'p', 'KL',...
    'KB', 'L0', 'D', 'd', 'p', 'KL', 'KB', 'L0'};
PR.main.z_comps = {'Vehicle', 'Vehicle', 'Vehicle', 'Vehicle', 'Vehicle', 'Vehicle',...
    'Vehicle', 'Vehicle', 'Vehicle', 'Vehicle', 'Vehicle', 'Vehicle', 'Vehicle', 'Tire',...
    'Tire', 'Tire', 'Tire', 'Tire', 'Tire', 'Corner', 'Corner', 'Corner', 'Corner',...
    'Corner', 'Corner', 'Suspension', 'Suspension', 'Suspension', 'Suspension',...
    'Suspension', 'Suspension', 'Suspension', 'Suspension', 'Suspension', 'Suspension',...
    'Spring', 'Spring', 'Spring', 'Spring', 'Spring', 'Spring', 'Spring', 'Spring',...
    'Spring', 'Spring', 'Spring', 'Spring'};

PR.main.m = 7;
PR.main.comp_id = [1, 2, 3, 4, 5, 6, 7];
PR.main.comp_names = {'V', 'T', 'C', 'Sf', 'Sr', 'Spf', 'Spr'};

PR.main.obj(1).name = 'none';
PR.main.obj(1).args = [];

PR.main.con(1).name = 'none';
PR.main.con(1).args = [];

PR.sub(1).x_id = [1, 2, 3, 4, 5];
PR.sub(1).y_id = [6, 7, 8, 9, 10, 11, 12, 13];

PR.sub(1).obj(1).name = 'f';
PR.sub(1).obj(1).args = [1, 2, 3, 4, 5];

PR.sub(1).con(1).name = 'h1';
PR.sub(1).con(1).args = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13];

PR.sub(1).to(2).Sij = [0,0,0,0,0,1,0,0; 1,0,0,0,0,0,0,0; 0,0,0,0,1,0,0,0; 0,1,0,0,0,0,0,0];
PR.sub(1).to(3).Sij = [0,0,0,0,0,0,0,1; 0,0,0,0,0,0,1,0; 1,0,0,0,0,0,0,0; 0,1,0,0,0,0,0,0];
PR.sub(1).to(4).Sij = [0,0,1,0,0,0,0,0];
PR.sub(1).to(5).Sij = [0,0,0,1,0,0,0,0];
PR.sub(1).to(6).Sij = zeros(0, 8);
PR.sub(1).to(7).Sij = zeros(0, 8);

PR.sub(2).x_id = [];
PR.sub(2).y_id = [14, 15, 16, 17, 18, 19];

PR.sub(2).obj(1).name = 'none';
PR.sub(2).obj(1).args = [];

PR.sub(2).con(1).name = 'h3';
PR.sub(2).con(1).args = [16, 17, 18, 19, 14, 15];

PR.sub(2).to(1).Sij = [0,0,0,1,0,0; 1,0,0,0,0,0; 0,0,1,0,0,0; 0,1,0,0,0,0];
PR.sub(2).to(3).Sij = [0,0,0,0,1,0; 0,0,0,0,0,1];
PR.sub(2).to(4).Sij = zeros(0, 6);
PR.sub(2).to(5).Sij = zeros(0, 6);
PR.sub(2).to(6).Sij = zeros(0, 6);
PR.sub(2).to(7).Sij = zeros(0, 6);

PR.sub(3).x_id = [];
PR.sub(3).y_id = [20, 21, 22, 23, 24, 25];

PR.sub(3).obj(1).name = 'none';
PR.sub(3).obj(1).args = [];

PR.sub(3).con(1).name = 'h4';
PR.sub(3).con(1).args = [22, 23, 24, 25, 20, 21];

PR.sub(3).to(1).Sij = [0,0,0,1,0,0; 0,0,1,0,0,0; 1,0,0,0,0,0; 0,1,0,0,0,0];
PR.sub(3).to(2).Sij = [0,0,0,0,1,0; 0,0,0,0,0,1];
PR.sub(3).to(4).Sij = zeros(0, 6);
PR.sub(3).to(5).Sij = zeros(0, 6);
PR.sub(3).to(6).Sij = zeros(0, 6);
PR.sub(3).to(7).Sij = zeros(0, 6);

PR.sub(4).x_id = [26];
PR.sub(4).y_id = [27, 28, 29, 30];

PR.sub(4).obj(1).name = 'none';
PR.sub(4).obj(1).args = [];

PR.sub(4).con(1).name = 'g1';
PR.sub(4).con(1).args = [26, 28, 29, 30];
PR.sub(4).con(2).name = 'h2';
PR.sub(4).con(2).args = [27, 26, 28, 29, 30];

PR.sub(4).to(1).Sij = [1,0,0,0];
PR.sub(4).to(2).Sij = zeros(0, 4);
PR.sub(4).to(3).Sij = zeros(0, 4);
PR.sub(4).to(5).Sij = zeros(0, 4);
PR.sub(4).to(6).Sij = [0,0,0,1; 0,0,1,0; 0,1,0,0];
PR.sub(4).to(7).Sij = zeros(0, 4);

PR.sub(5).x_id = [31];
PR.sub(5).y_id = [32, 33, 34, 35];

PR.sub(5).obj(1).name = 'none';
PR.sub(5).obj(1).args = [];

PR.sub(5).con(1).name = 'g1';
PR.sub(5).con(1).args = [31, 33, 34, 35];
PR.sub(5).con(2).name = 'h2';
PR.sub(5).con(2).args = [32, 31, 33, 34, 35];

PR.sub(5).to(1).Sij = [1,0,0,0];
PR.sub(5).to(2).Sij = zeros(0, 4);
PR.sub(5).to(3).Sij = zeros(0, 4);
PR.sub(5).to(4).Sij = zeros(0, 4);
PR.sub(5).to(6).Sij = zeros(0, 4);
PR.sub(5).to(7).Sij = [0,1,0,0; 0,0,0,1; 0,0,1,0];

PR.sub(6).x_id = [36, 37, 38];
PR.sub(6).y_id = [39, 40, 41];

PR.sub(6).obj(1).name = 'none';
PR.sub(6).obj(1).args = [];

PR.sub(6).con(1).name = 'g2';
PR.sub(6).con(1).args = [36, 37, 38];
PR.sub(6).con(2).name = 'h5';
PR.sub(6).con(2).args = [39, 40, 36, 37, 38, 41];

PR.sub(6).to(1).Sij = zeros(0, 3);
PR.sub(6).to(2).Sij = zeros(0, 3);
PR.sub(6).to(3).Sij = zeros(0, 3);
PR.sub(6).to(4).Sij = [0,0,1; 0,1,0; 1,0,0];
PR.sub(6).to(5).Sij = zeros(0, 3);
PR.sub(6).to(7).Sij = zeros(0, 3);

PR.sub(7).x_id = [42, 43, 44];
PR.sub(7).y_id = [45, 46, 47];

PR.sub(7).obj(1).name = 'none';
PR.sub(7).obj(1).args = [];

PR.sub(7).con(1).name = 'g2';
PR.sub(7).con(1).args = [42, 43, 44];
PR.sub(7).con(2).name = 'h5';
PR.sub(7).con(2).args = [45, 46, 42, 43, 44, 47];

PR.sub(7).to(1).Sij = zeros(0, 3);
PR.sub(7).to(2).Sij = zeros(0, 3);
PR.sub(7).to(3).Sij = zeros(0, 3);
PR.sub(7).to(4).Sij = zeros(0, 3);
PR.sub(7).to(5).Sij = [1,0,0; 0,0,1; 0,1,0];
PR.sub(7).to(6).Sij = zeros(0, 3);
```

Fig. 9 Contents of ALC input file generated from normalized file of chassis example—partition 1

another system, and are included to make a variable of a sub-component accessible by a higher level system. An advantage of using aliases instead of an identifier such as *N.S.v* is that the higher-level systems do not need to have detailed knowledge of the structure of its subsystems. Additionally, the definition of the higher level system does not need to be changed if the structure of the subsystem is modified. Observe that an alias definition does *not* define a consistency constraint; aliases are simply used to forward variable values of lower to higher levels in the problem hierarchy.

5.2 Generated input files

For the first partitioned problem, the compiler and both generators are used to automatically generate the three input files from the Ψ-specification. Note that for the purpose of the ALC input file, the response functions have been included as constraint functions, similar to Section 4.2.

The generated normalized file, the ALC input file, and the FDT file are given in Figs. 8, 9, and 10, respectively. The advantages of being able to generate the ALC and FDT formats automatically are obvious since neither of these two formats is attractive for specification of the partitioned problem structure. Valuable time and effort can be saved by specifying partitioned problems using the intuitive and compact Ψ language.

6 Summary and discussion

Decomposition-based design of engineering systems requires two main ingredients: a problem specification that defines the structure of the system to be optimized, and a computational framework that performs the numerical operations associated with coordination and solution of the partitioned problem. Several generic computational frameworks have been developed over the past decade, but generic and intuitive approaches to partitioned problem specification are rare.

This article proposes a linguistic approach to partitioned problem specification that is generic, compact, and easy to use. The proposed language Ψ allows a designer to intuitively define partitioned optimization problems using only

```
Columns:
 1: Chassis.V.a
 2: Chassis.C.a
 3: Chassis.T.a
 4: Chassis.T.Pif
 5: Chassis.C.Pif
 6: Chassis.V.b
 7: Chassis.C.b
 8: Chassis.T.b
 9: Chassis.T.Pir
10: Chassis.C.Pir
11: Chassis.V.Ktf
12: Chassis.T.Ktf
13: Chassis.V.Caf
14: Chassis.C.Caf
15: Chassis.V.Ktr
16: Chassis.T.Ktr
17: Chassis.V.Car
18: Chassis.C.Car
19: Chassis.V.Ksf
20: Chassis.Sf.Ks
21: Chassis.V.Ksr
22: Chassis.Sr.Ks
23: Chassis.Sf.L0
24: Chassis.Spf.L0
25: Chassis.Sr.L0
26: Chassis.Spr.L0
27: Chassis.Sf.KL
28: Chassis.Spf.KL
29: Chassis.Sr.KL
30: Chassis.Spr.KL
31: Chassis.Sf.KB
32: Chassis.Spf.KB
33: Chassis.Sr.KB
34: Chassis.Spr.KB
35: Chassis.V.wsf
36: Chassis.V.wsr
37: Chassis.V.wtf
38: Chassis.V.wtr
39: Chassis.V.kus
40: Chassis.Sf.Zs
41: Chassis.Sr.Zs
42: Chassis.Spf.D
43: Chassis.Spf.d
44: Chassis.Spf.p
45: Chassis.Spr.D
46: Chassis.Spr.d
47: Chassis.Spr.p

Rows:
 1: Chassis.V.a -- Chassis.C.a
 2: Chassis.V.a -- Chassis.T.a
 3: Chassis.T.Pif -- Chassis.C.Pif
 4: Chassis.V.b -- Chassis.C.b
 5: Chassis.V.b -- Chassis.T.b
 6: Chassis.T.Pir -- Chassis.C.Pir
 7: Chassis.V.Ktf -- Chassis.T.Ktf
 8: Chassis.V.Caf -- Chassis.C.Caf
 9: Chassis.V.Ktr -- Chassis.T.Ktr
10: Chassis.V.Car -- Chassis.C.Car
11: Chassis.V.Ksf -- Chassis.Sf.Ks
12: Chassis.V.Ksr -- Chassis.Sr.Ks
13: Chassis.Sf.L0 -- Chassis.Spf.L0
14: Chassis.Sr.L0 -- Chassis.Spr.L0
15: Chassis.Sf.KL -- Chassis.Spf.KL
16: Chassis.Sr.KL -- Chassis.Spr.KL
17: Chassis.Sf.KB -- Chassis.Spf.KB
18: Chassis.Sr.KB -- Chassis.Spr.KB
19: Chassis.V.f
20: Chassis.V.a1
21: Chassis.T.a3
22: Chassis.C.a4
23: Chassis.Sf.g1
24: Chassis.Sf.a2
25: Chassis.Sr.g1
26: Chassis.Sr.a2
27: Chassis.Spf.g2
28: Chassis.Spf.a5
29: Chassis.Spr.g2
30: Chassis.Spr.a5

FDT:
1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0
1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1
```

Fig. 10 Contents of FDT file generated from normalized file of chassis example—partition 1

a small set of language elements. The developed tools, including user manuals and several examples, are available for download at http://se.wtb.tue.nl/sewiki/mdo.

So-called components are the building blocks of a specification in Ψ. A component definition includes a number of variables and objective, constraint, and response functions. Components are assembled into systems in which variable couplings between components are defined as well as coupling functions. These systems can themselves be part of another system, allowing an incremental multi-level assembly of the partitioned problem. This incremental assembly process allows the designer to control the complexity of the individual assembly tasks, and improves the overview of the system.

A generic compiler has been developed that produces an easy to process normalized format. Two generators automatically derive input files for computational coordination frameworks. The compiler-based approach proposed in this article offers developers of coordination methods the freedom to focus on input files that integrate easily with the computational routines they are designing. The Ψ language and the associated compiler should therefore be seen as powerful generic pre-processor that provides these computational frameworks with easy-to-process input specifications while allowing users to focus on partitioning the problem in an intuitive and easy way rather than handling the details needed by the coordination frameworks.

Users that want to use the Ψ language for their computational framework need to develop a generator. This generator is similar to the examples presented in this paper, and should automatically translate the partition specification in the INI format to an input file appropriate for the computational framework. It is recommended that this generator also checks for framework-specific requirements that are not covered by the generic Ψ-compiler. Examples of such requirements are not allowing system-wide functions or not allowing response functions.

The flexibility of Ψ can be used to experiment with different partitions of the same problem. By solving different decompositions of the same problem, insights can be gained with respect to the notion of coupling strength present in a partitioned problem. These insights can be used to further refine model-based partitioning methods as those proposed by Krishnamachari and Papalambros (1997), Li and Chen (2006), and Allison et al. (2007). In turn, the problem partitions derived using model-based methods can be stored in Ψ or INI format.

Finally, we note that in the development of Ψ we have not made *a priori* assumptions about the class of optimization problems that can be treated, nor about the coordination method that will be used to solve the problem. The language seems applicable to linear as well as non-linear problems, continuous or discrete variables, single- and multi-objective problems, deterministic or probabilistic optimization problems, and is suitable for both single-level as well as multi-level coordination methods.

Acknowledgments The authors are grateful for the comments made by the anonymous referees which helped to improve the presentation of the paper.

References

Alexandrov NM (2005) Editorial—multidisciplinary design optimization. Optim Eng 6(1):5–7

Alexandrov NM, Lewis RM (1999) Comparative properties of collaborative optimization and other approaches to MDO. In: ASMO UK/ISSMO conference on engineering design optimization. MCB University Press, Bradford, pp 39–46

Alexandrov NM, Lewis RM (2004a) Reconfigurability in MDO problem synthesis, part 1. In: Proceedings of the 10th AIAA/ISSMO multidisciplinary analysis and optimization conference, Albany, NY, AIAA paper 2004-4307

Alexandrov NM, Lewis RM (2004b) Reconfigurability in MDO problem synthesis, part 2. In: Proceedings of the 10th AIAA/ISSMO multidisciplinary analysis and optimization conference, Albany, NY, AIAA paper 2004-4308

Allison JT, Kokkolaras M, Papalambros PY (2007) Optimal partitioning and coordination decisions in system design using an evolutionary algorithm. In: Proceedings of the 7th world congress on structural and multidisciplinary optimization. Seoul, South Korea

Allison JT, Kokkolaras M, Papalambros PY (2009) Optimal partitioning and coordination decisions in decomposition-based design optimization. ASME J Mech Des 131(8):1–8

Balling RJ, Sobieszczanski-Sobieski J (1996) Optimization of coupled systems: a critical overview of approaches. AIAA J 34(1):6–17

Browning TR (2001) Applying the design structure matrix to system decomposition and integration problems: a review and new directions. IEEE Trans Eng Manage 48(3):292–306

Chen L, Ding Z, Li S (2005) A formal two-phase method for decomposition of complex design problems. ASME J Mech Des 127:184–195

Cloanto (2009) Cloanto implementation of INI file format. http://www.cloanto.com/specs/ini.html. Date accessed: 30 January 2009

Cramer EJ, Dennis JE, Frank PD, Lewis RM, Shubin GR (1994) Problem formulation for multidisciplinary optimization. SIAM J Optim 4(4):754–776

de Wit AJ, van Keulen F (2007) Numerical comparison of multi-level optimization techniques. In: Proceedings of the 3rd AIAA multi-disciplinary design optimization specialist conference, Honolulu, HI

de Wit AJ, van Keulen F (2008) Framework for multilevel optimization. In: Proceedings of the 5th China-Japan-Korea joint symposium on optimization of structural and mechanical systems, Seoul, South Korea

Eppinger SD, Whitney DE, Smith RP, Gebala DA (1994) A model-based method for organizing tasks in product development. Res Eng Des 6:1–17

Etman LFP, Kokkolaras M, Hofkamp AT, Papalambros PY, Rooda JE (2005) Coordination specification in distributed optimal design of multilevel systems using the χ language. Struct Multidisc Optim 29(3):198–212

Friedenthal S, Moore A, Steiner R (2008) A practical guide to SysML: the systems modeling language. Morgan Kaufmann, San Francisco

Huang GQ, Qu T, Cheung WL (2006) Extensible multi-agent system for optimal design of complex systems using analytical target cascading. Int J Adv Manuf Technol 30:917–926

Kim HM, Michelena NF, Papalambros PY, Jiang T (2003) Target cascading in optimal system design. ASME J Mech Des 125(3):474–480

Krishnamachari RS, Papalambros PY (1997) Optimal hierarchical decomposition synthesis using integer programming. ASME J Mech Des 119:440–447

Kusiak A, Larson N (1995) Decomposition and representation methods in mechanical design. ASME J Mech Des 117(3):17–24

Li S, Chen L (2006) Model-based decomposition using non-binary dependency analysis and heuristic partitioning analysis. In: Proceedings of the ASME design engineering technical conferences, Philadelphia, PY

Lutz M (2006) Programming python, 3rd edn. O'Reilly Media, Inc, Sebastopol

Martins JRRA, Marriage C, Tedford N (2009) pyMDO: an object-oriented framework for multidisciplinary design optimization. ACM Trans Math Softw 36(4):1–25

Michelena NF, Papalambros PY (1997) A hypergraph framework for optimal model-based decomposition of design problems. Comput Optim Appl 8(2):173–196

Michelena NF, Scheffer C, Fellini R, Papalambros PY (1999) CORBA-based object-oriented framework for distributed system design. Mech Des Struct Mach 27(4):365–392

Moore KT, Naylor BA, Gray JS (2008) The development of an open source framework for multidisciplinary analysis and optimization. In: Proceedings of the 12th AIAA/ISSMO multidisciplinary analysis and optimization conference, Victoria, BC, Canada, AIAA paper 2008-6069

Papalambros PY (1995) Optimal design of mechanical engineering systems. ASME J Mech Des 117(B):55–62

Perez RE, Liu HHT, Behdinan K (2004) Evaluation of multidisciplinary optimization approaches for aircraft conceptual design. In: Proceedings of the 10th AIAA/ISSMO multidisciplinary analysis and optimization conference, Albany, NY, AIAA paper 2004-4537

Sobieszczanski-Sobieski J, Haftka RT (1997) Multidisciplinary aerospace design optimization: survey of recent developments. Struct Optim 14(1):1–23

Steward DV (1981) The design structure system: a method for managing the design of complex systems. IEEE Trans Eng Manage EM-28(3):71–74

Tosserams S, Etman LFP, Rooda JE (2007) An augmented Lagrangian decomposition method for quasi-separable problems in MDO. Struct Multidisc Optim 34(3):211–227

Tosserams S, Etman LFP, Rooda JE (2008) Augmented Lagrangian coordination for distributed optimal design in MDO. Int J Numer Methods Eng 73(13):1885–1910

Tosserams S, Etman LFP, Rooda JE (2009a) A classification of methods for distributed system optimization based on formulation structure. Struct Multidisc Optim 39:503–517 doi:10.1007/s00158-008-0347-z

Tosserams S, Hofkamp AT, Etman LFP, Rooda JE (2009b) Ψ reference manual. SE-report 2009-04, Eindhoven University of Technology. http://se.wtb.tue.nl

Tosserams S, Hofkamp AT, Etman LFP, Rooda JE (2009c) Using the alc matlab toolbox with input files generated from Ψ specifications. SE-report 2009-05, Eindhoven University of Technology. http://se.wtb.tue.nl

Wagner TC, Papalambros PY (1993) General framework for decomposition analysis in optimal design. In: Gilmore B, Hoeltzel D, Azarm S, Eschenauer H (eds) Advances in design automation, Albuquerque, NM, pp 315–325

Yi SI, Shin JK, Park GJ (2008) Comparison of MDO methods with mathematical examples. Struct Multidisc Optim 35(5):391–402