Research Paper

Struct Multidisc Optim 29, 198–212 (2005)
DOI 10.1007/s00158-004-0467-z

Coordination specification in distributed optimal design of multilevel systems using the χ language

L.F.P. Etman, M. Kokkolaras, A.T. Hofkamp, P.Y. Papalambros and J.E. Rooda

Abstract Coordination plays a key role in solving decomposed optimal design problems. Several coordination strategies have been proposed in the multidisciplinary optimization (MDO) literature. They are usually presented as a sequence of statements. However, a precise description of the concurrency in the coordination is needed for large multilevel or non-hierarchic coordination architectures. This article proposes the use of communicating sequential processes (CSP) concepts from concurrency theory for specifying and implementing coordination strategies in distributed multilevel optimization rigorously. CSP enables the description of the coordination as a number of parallel processes that operate independently and communicate synchronously. For this purpose, we introduce elements of the language χ, a CSP-based language that contains advanced data modeling constructs. The associated software toolkit allows execution of the specified coordination. Coordination specification using χ is demonstrated for analytical target cascading (ATC), a methodology for design optimization of hierarchically decomposed multilevel systems. It is shown that the ATC coordination can be compactly specified for various coordination schemes. This illustrates the advantage of using a high-level concurrent language, such as χ, for specifying the coordination of distributed optimal design problems. Moreover, the χ software toolkit is useful in implementing alternative schemes rapidly, thus enabling the comparison of different MDO methods.

Key words multidisciplinary optimization, decomposition, coordination language, communicating sequential processes, hierarchical multilevel systems, analytical target cascading

Received: 10 December 2002
Revised manuscript received: 9 February 2004
Published online: 30 September 2004

L.F.P. Etman[1, ✉], M. Kokkolaras[2], A.T. Hofkamp[1], P.Y. Papalambros[2] and J.E. Rooda[1]

[1] Department of Mechanical Engineering, Eindhoven University of Technology, Eindhoven, PO Box 513, 5600 MB, Eindhoven, The Netherlands
e-mail: {l.f.p.etman, a.t.hofkamp,j.e.rooda}@tue.nl
[2] Department of Mechanical Engineering, University of Michigan, 2250 G.G. Brown Bldg., Ann Arbor, Michigan 48109-2125, USA
e-mail: {mk, pyp}@umich.edu

1 Introduction

Several multidisciplinary optimization (MDO) approaches have been proposed during the last two decades for solving optimal design problems of systems that involve more than one engineering discipline. In these approaches large optimal design problems are typically partitioned into collections of smaller and more tractable subproblems, each associated with a discipline. Generally speaking, a subproblem does not need to be associated with a discipline; it may represent a subsystem or component of the decomposed system. The terms discipline and MDO are used here in this broader context. An additional advantage of partitioning may be the ability to solve the subproblems concurrently. The challenge of solving such distributed optimal design problems lies in addressing the interactions among the individual disciplines. System constraints may depend on design variables that are present in more than one discipline, and discipline constraints may depend on responses from other disciplines. The solution of the subproblems has to be coordinated in a manner that ensures convergence to a solution that is consistent with the one that would have been obtained if the original problem could be solved without decomposition.

Development of theoretically sound and practical decomposition and coordination methods is still an open research topic. Decomposition methods exploit some structure in the coupling of the subsystems. Two

main types of decomposition methods can be identified: methods that originate in rigorous mathematical formulations, and methods that emerged from MDO approaches in engineering; the latter tend to be heuristic. Mathematical decomposition methods generally use the structure of a large set of analytical constraint equations to obtain a partition that can be coordinated efficiently. Engineering-based methods are either aspect-driven or component-driven. They deal with "black-box" discipline analysis models that cannot be represented by analytical equations in the system optimization problem. Motivated by design practice, engineering-based methods aim at disciplinary optimization autonomy by means of a multilevel decomposition. Alexandrov and Lewis (2000, 2002) showed that some of those formulations exhibit convergence difficulties. Theoretically convergent multilevel optimization methods are presented in Michelena *et al.* (2003) and Haftka and Watson (2004).

Precise and compact specification of the coordination becomes increasingly important as the number of levels and subproblems grows. Implementing coordination schemes for multilevel hierarchies or non-hierarchic problems becomes complicated and is prone to errors. Specifying the sequence of solving the optimization subproblems and managing the necessary data exchange using programming languages such as Fortran or C is not a trivial task. Parallel processing complicates matters even more. Having the ability to implement the coordination at a higher level of abstraction is therefore advantageous. A coordination language with a clear concept of concurrency is necessary. It is preferable for such a concurrent specification language to have a highly expressive syntax with formal semantics.

Available mathematical programming languages, e.g., AMPL (Fourer *et al.* 1993), are typically geared towards formulating numerical optimization problems. In computer science, several languages have been developed to describe the coordination of concurrent processes. These coordination languages can be classified into data-driven and control-driven (Papadopoulos and Arbab 1998). A data-driven language coordinates data by means of a shared data space, while a control-driven (or process-oriented) language treats processes as "black-boxes" that are coordinated through exchanging state values or broadcasting control messages. Aforementioned mathematical decomposition methods are suited to data-driven coordination, while "black-box" engineering-based methods fit the process-oriented coordination approach.

Communicating sequential processes (CSP) is a popular theoretical foundation of process-oriented coordination languages (Hoare 1985; Roscoe 1997). CSP is particularly attractive for modeling coupled disciplines and optimization subproblems in MDO, provided that suitable data language elements are available to deal with the numerical optimization setting. We propose to utilize CSP concepts for specifying coordination in distributed optimal design of multilevel systems. In particular, we adopt the CSP-based language χ (Hofkamp and Rooda 2002a,b; Vervoort and Rooda 2003), which includes data types required for numerical optimization. The χ language was developed originally for simulating discrete-event and hybrid (combined discrete-event and continuous-time) manufacturing systems (Van Beek *et al.* 2000). It is designed primarily for modeling purposes; hence, it is easy to understand and has only few language constructs. The discrete-event part of χ is used in this work.

The paper is organized as follows. First, MDO coordination architectures are reviewed and placed into a CSP perspective. Then, several basic χ language elements necessary to specify the parallel processes and their interactions are introduced. This paper focuses on hierarchically decomposed multilevel systems. In this regard, coordination specification by means of the χ language is demonstrated for analytical target cascading (ATC). An overview of the ATC formulation is given and different coordination strategies are discussed. A simple yet illustrative example with a three-level hierarchy is used to show the advantage of the approach and demonstrate that different ATC coordination strategies can be specified efficiently. Finally, the main findings are summarized and discussed.

2 Multidisciplinary optimization

In this section we review the general MDO problem formulation, summarize main solution strategies, and consider a new concept for MDO coordination.

2.1 Problem formulation

The general MDO problem can be stated as follows: Find the values of the discipline design variables such that a system objective is optimized subject to system and disciplinary design constraints, interdisciplinary design variable coupling constraints, and interdisciplinary response variable coupling constraints.

A set of design variables $\mathbf{x}_i$ is identified for each discipline $i = 1, 2, \ldots, m$, where m is the number of disciplines. Some of the design variables may be shared in several disciplines: this sharing is represented by interdisciplinary design variable coupling equations $\mathbf{k}(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m) = \mathbf{0}$. Discipline constraints $\mathbf{g}_i \leq \mathbf{0}$ depend only on discipline design variables $\mathbf{x}_i$ and discipline responses $\mathbf{r}_i$. System constraints $\mathbf{g}_0 \leq \mathbf{0}$ may depend on all design variables and responses. The same holds for the system objective f.

Interdisciplinary response variables coupling is represented by equations $\mathbf{l}(\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_m, \mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_m) = \mathbf{0}$. These equations relate responses $\mathbf{r}_j$ of discipline j that are inputs $\mathbf{c}_i$ to discipline i, $j \neq i$. Finally, discipline responses are functions of discipline design variables and, possibly,

responses of other disciplines, and are computed using analysis or simulation models: $\mathbf{r}_i = \mathbf{a}_i(\mathbf{x}_i, \mathbf{c}_i)$.

Based on the above definitions, the MDO problem can be formulated mathematically as

$$
\begin{aligned}
&\min \quad f(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m, \mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_m)\\
&\text{w.r.t.} \quad \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m, \mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_m, \mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_m\\
&\text{s.t.} \quad \mathbf{g}_0(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m, \mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_m) \le \mathbf{0}\\
&\mathbf{g}_i(\mathbf{x}_i, \mathbf{r}_i) \le \mathbf{0}\,, \qquad i = 1, 2, \ldots, m\\
&\mathbf{k}(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m) = \mathbf{0}\\
&\mathbf{l}(\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_m, \mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_m) = \mathbf{0}\\
&\mathbf{r}_i - \mathbf{a}_i(\mathbf{x}_i, \mathbf{c}_i) = \mathbf{0}\,, \qquad i = 1, 2, \ldots, m\\
&\mathbf{x}_i \in \mathcal{X}_i\,, \qquad i = 1, 2, \ldots, m\,. \qquad (1)
\end{aligned}
$$

2.2 Classification of MDO architectures

Several methods for MDO can be found in the literature. Classifications of the associated architectures are presented in Cramer *et al.* (1994), Balling and Sobieszczanski-Sobieski (1996), and Alexandrov and Lewis (1999). The key element in these classifications is the way feasibility of the constraints in Problem (1) is maintained.

Cramer *et al.* (1994) classify MDO methods into "all-at-once" (AAO), individual discipline feasible (IDF), and multidisciplinary feasible (MDF). The formulation of Problem (1) corresponds to the IDF strategy. The analysis equations appear *nested* (Balling and Sobieszczanski-Sobieski 1996) or *closed* (Alexandrov and Lewis 1999) with respect to the system optimization problem (*cf.* Fig. 1). Since response coupling is included in the system optimization, the system will be interdisciplinarily feasible (i.e., satisfy the response coupling constraints) only after convergence has been achieved.

To guarantee interdisciplinary feasibility at each iteration of the optimization, one has to treat response coupling as nested with respect to the system optimization, as shown in Fig. 2. Evaluating a system design $\mathbf{x} = [\mathbf{x}_1^t, \mathbf{x}_2^t, \ldots, \mathbf{x}_m^t]^t$ to obtain responses $\mathbf{r} = [\mathbf{r}_1^t, \mathbf{r}_2^t, \ldots, \mathbf{r}_m^t]^t$ automatically implies that coupled disciplinary responses are consistent. Cramer *et al.* (1994) refer to this strategy as MDF.

Balling and Sobieszczanski-Sobieski (1996) distinguish between single-level and multilevel architectures. Single-level refers to an architecture where only the system optimization problem determines the design variable values. In the multilevel case disciplinary optimizers are introduced to determine the independent discipline design variables, while the system optimizer determines the shared design variables. From the system optimizer's point of view the disciplinary constraints $\mathbf{g}_i \le \mathbf{0}$ will always be satisfied (i.e., they are closed design constraints

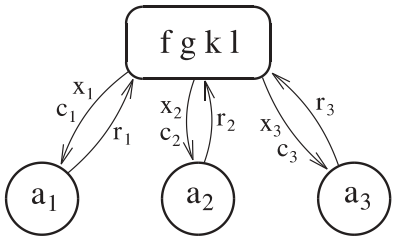

Fig. 1 Individual discipline feasible architecture

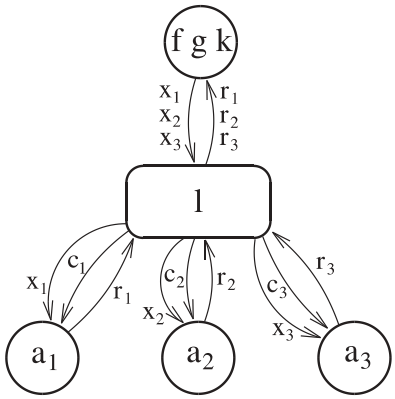

Fig. 2 Multidisciplinary feasible architecture

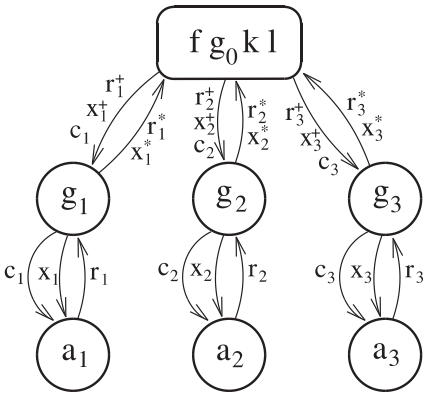

Fig. 3 Hierarchical multilevel architecture

according to the classification in Alexandrov and Lewis (1999)). An example of a hierarchical multilevel architecture is shown in Fig. 3, where $\mathbf{x}_i^+$ and $\mathbf{r}_i^+$ represent targets for design variables and responses provided by the system optimizer, respectively, while $\mathbf{x}_i^*$ and $\mathbf{r}_i^*$ represent the values that are returned from the discipline optimizers.

Appropriate coordination strategies must be specified for solving the subproblems. Coordination algorithms in the MDO literature are usually presented as step-wise sequential procedures, sometimes visualized by flow diagrams. However, a sequential description of the coordination in a distributed MDO architecture requires the description of the exact order of communications (data exchange) for all subsystems. For multilevel and nonhierarchic decompositions this is a tedious task that is prone to errors, especially when the problem size grows (many levels and/or subproblems). To avoid this, one

should be able to specify the (concurrency in the) coordination at a higher level of abstraction, i.e., by means of an appropriate concurrent language based on computer science.

2.3 Communicating sequential processes and MDO coordination

Several theoretical foundations are available to describe systems that exhibit concurrent behavior. A short overview is given in the introduction of Bos and Kleijn (2002). Communicating sequential processes (CSP) concepts (Hoare 1978, 1985; Roscoe 1997) are highly suited for specifying coordination in MDO. They model coordination as parallel processes that operate independently and communicate synchronously over predefined channels. The CSP concept matches engineering-based MDO methods that partition optimal system design problems into a collection of smaller subproblems. Each subproblem can be seen as an independent "black-box" process, while the necessary data exchange among subproblems can be viewed as a series of communications.

The MDO problem formulations shown in Figs. 1, 2, and 3 can be coordinated using CSP concepts. Each circle represents a process that is responsible for the closure of a specific set of constraints. Processes can be executed in parallel assuming that they are not waiting for data from other processes. Such input/output relations among processes are modeled as communication channels, visualized by arrows that illustrate the direction of information flow. Using CSP, communication sequences are defined locally for each process. Any order of communication among processes is allowed as long as the local process communication sequences are obeyed.

In our opinion, MDO coordination benefits greatly by using a specification language based on elements of concurrent programming, especially for multilevel hierarchies and non-hierarchic schemes. Such a language enables a formal and precise description of the MDO processes (typically related to analysis or optimization) and the communication among them. We demonstrate this using the CSP-based χ language developed by the Systems Engineering group at the Eindhoven University of Technology in the Netherlands.

3 Coordination specification using the χ language

The χ specification language has been developed originally for modeling manufacturing systems that exhibit complex concurrent behavior. It is a language designed primarily for modeling pure discrete-event concurrent systems or systems that combine discrete-event and continuous time behavior (Van Beek *et al.* 2000). We will show that the discrete-event part of χ is well suited for specifying coordination of distributed optimal system design problems in a formal and rigorous manner. A brief informal description of the χ syntax (denotation of language elements) and semantics (meaning of language elements) is presented in this section. A complete formal definition of the language can be found in Bos and Kleijn (2002). A tutorial introduction is given by Vervoort and Rooda (2003).

The χ language is highly expressive with only a small number of orthogonal language elements. It is easy to understand and combines a well-defined concept of concurrency with advanced data modeling constructs. The discrete-event part of χ is based on CSP (Hoare 1978) and Dijkstra's guarded command language (Dijkstra 1975). Parallel behavior is restricted to occur among processes (following Van de Mortel-Fronczak *et al.* (1995)). Individual processes are specified in an imperative way using a sequence of statements. For readability of the specification, systems can be introduced to represent collections of coupled processes, as well as functions to define calculations. Interactions among processes are modeled as synchronous communications over channels (Hoare 1985). Synchronous means that communication between two processes takes place only when both are willing to communicate, and that such a communication takes place instantly (no storage in the channel). The concepts of time and probability, which play a key role in modeling manufacturing systems, are also available in χ. In this work, these concepts do not play a role in MDO coordination, and are therefore omitted.

The main definitions for communicating sequential processes using χ can be summarized as follows:

- A *process* represents a sequentially behaving component in a larger concurrent system.
- A *system* is a collection of concurrent processes that cooperate by synchronous interaction. A system behaves like a process and can again be part of other systems.
- A *channel* represents a connection between two processes, and enables interaction between them.
- *Interaction* (communication) between two processes means instantaneous data exchange (through a point-to-point channel).

3.1 Data types

MDO coordination formulations usually generate considerable amounts of (numerical) data that have to be exchanged among the subproblems. Therefore, a coordination language for MDO has to provide language constructs to model this data flow compactly. The χ language satisfies this requirement. χ has several built-in data types, and distinguishes basic data types and container data types. The basic data types are: bool (boolean), nat (natural), int (integer), real, string, and void. The void type is the empty data type used in the declaration of

synchronization channels and ports. Container data types are: array, tuple, and list, among others. These three container data types are briefly explained below, where T denotes a data type that is either basic or container.

T^n is an array of fixed length n containing data elements of type T. As an example, $\langle 2.1, 4.8, -4.9\rangle$ is an array of type real^3. Arrays can be built from any basic or container data type provided that the elements are of identical type. This means that an $m \times n$ matrix of reals can be represented by $(\text{real}^n)^m$. The index operator $.i$ $(0 \leq i \leq n-1)$ allows to access the elements in the array, e.g., $\langle 2.1, 4.8, -4.9\rangle.1$ returns 4.8.

$T_0 \times T_1 \times T_2 \times \ldots \times T_m$ denotes a tuple that is a more general form of an array in the sense that the elements of a tuple need not be of the same type. A tuple is comparable to a record in Pascal. For example, we may have two-tuple containing arrays, like $\text{bool}^2 \times \text{real}^3$. Similar to the array, elements of a tuple may be either basic or container data types and can be accessed by the index operator.

T^* is a list containing an ordered sequence of elements that must all be of the same type T. An example of a list of type nat^* is $[1, 2, 3]$. The length of the list is variable, that is, elements can be added to or removed from the list. The empty list is $[\,]$. In addition to the concatenation (addition) and subtraction (removal) operators, a number of functions are available, e.g., for accessing the value of the first element of a list or querying the length of a list.

Disciplines responses and optimization results are typically generated in the form of arrays or tuples of arrays. Lists are suitable for storing data that are needed in later iterations.

3.2 Processes

The basic building block of a χ model is a process. The process definition has the following general format:

$$\mathsf{proc}\ N(V_p) = [\![V_l | S_p]\!]\ .$$

The process is identified by its name N and parameters V_p; the latter are represented by a comma-separated list of (formal) parameters of the form $v\colon$ *type*, where *type* can be a standard data type T, a send port $(v\colon !T)$ data type, or a receive port $(v\colon ?T)$ data type. It is also allowed to have arrays of ports $(v\colon (!T)^n$, and $v\colon (?T)^n)$. The body of the process is specified between the brackets $[\![$ and $]\!]$. Local variables V_l are declared first, followed by the sequence of statements S_p to be executed by the process.

Table 1 presents, using BNF format (Backus 1960), the syntax of a subset of χ process statements that is relevant for specifying an MDO coordination. The statements are explained below informally.

Table 1 Syntax of χ process statements

$S_p ::= \mathsf{skip}$	(skip)
$\mid x := e$	(assignment)
$\mid E$	(event)
$\mid S_p\,;\ S_p$	(sequential composition)
$\mid [GC]$	(guarded command)
$\mid *[GC]$	(repetitive guarded command)
$\mid [SW]$	(selective waiting)
$\mid *[SW]$	(repetitive selective waiting)
$E ::= p\,!\,e$	(send)
$\mid p\,?\,x$	(receive)
$\mid p\,!$	(synchronization send)
$\mid p\,?$	(synchronization receive)
$GC ::= e_b \longrightarrow S_p$	$SW ::= e_b\,;\ E \longrightarrow S_p$
$\mid R : e_b \longrightarrow S_p$	$\mid R\colon e_b\,;\ E \longrightarrow S_p$
$\mid GC \,[\!]\, GC$	$\mid SW \,[\!]\, SW$
$R ::= i\colon \text{nat} \leftarrow l..u$	(range including l, excluding u)
$\mid R, R$	(range list).

skip means do nothing. It is used in selection statements to express that nothing needs to be done when a certain guard evaluates to true.

$x := e$ denotes the assignment statement. The value that follows from the evaluation of expression e is assigned to variable x. The types of x and e have to be the same. Multi-assignment is also allowed, e.g., $\langle x, y\rangle := \langle e_1, e_2\rangle$

$S_{p1}\,;\ S_{p2}$ denote that statement S_{p2} is executed after the execution of statement S_{p1} has been completed, that is, process statements are executed sequentially. In the sequel, a statement denoted by S_p may also be the concatenation of multiple process statements.

E represents an event statement. This includes the send statement $(p\,!\,e)$, the receive statement $(p\,?\,x)$, the synchronization send statement $(p\,!)$, and the synchronization receive statement $(p\,?)$. The send statement $p\,!\,e$ tries to send the evaluation outcome of expression e over the channel connected to port p. This send statement succeeds if the other process connected to the same channel is willing to receive. Similarly, the receive statement $p\,?\,x$ waits until data through port p is received and assigns this data to variable x. The ports, variables, and expressions must have equal types. Synchronization is a communication statement without transferring data. It is used to exchange an acknowledgment. A synchronization statement succeeds when the process connected to the same channel is also willing to synchronize.

$[GC]$ stands for guarded command statement or selection statement. This statement offers a choice between several guarded alternatives. Each alternative is specified using the syntax $e_b \longrightarrow S_p$, where e_b is the boolean expression denoting the guard. The different alterna-

tives are separated by the symbol [] . Upon execution of the selection statement the guards of all alternatives are evaluated. If one of the guards evaluates to true, the corresponding process statement S_p is executed. If more than one guard happens to be true then one of the true alternatives is chosen non-deterministically, i.e., nothing can be said about which choice will be made. If no guard evaluates to true an error occurs.

$*[GC]$ is the repetitive guarded command or repetitive selection statement that allows one to carry out the selection statement GC repeatedly. The repetition is continued until all guards evaluate to false. When this happens, the repetition ends and the statement following the repetitive guarded command is executed.

$[SW]$ denotes selective waiting. The selective waiting statement is an extended version of the selection statement where the guard of an alternative is replaced by the pair of a guard (boolean expression) *and* an event statement: $e_b\,;\ E \longrightarrow S_p$. When the selective waiting statement is executed, all guards are evaluated once. For the guards that have evaluated to true, the construct waits until at least one of the event statements can be carried out. If the event statement of just one alternative is possible, this statement is executed followed by the corresponding process statements. If event statements of multiple alternatives happen to be possible, one event statement is chosen non-deterministically followed by the execution of the process statements of the corresponding alternative. If none of the guards evaluates to true an error occurs.

$*[SW]$ represents repetitive selective waiting and repeats the selective waiting statement until all guards evaluate to false. After the end of the repetition, the statement following the repetitive selective waiting statement is executed.

A range expression R can be used in the guarded command and selective waiting statement to enable compact notation. The range expression allows the definition of variables that are varied within certain lower and upper bounds.

The key statements to specify communication among concurrent processes are the send, receive, and synchronize statements, as well as the (repetitive) selective waiting statement. The latter is the most powerful statement of χ for the specification of the communication between concurrent processes. The communication of a process with other processes can be specified without the need to predefine some sequence of communication. Such a selective waiting construct is essential for the specification of complex coordination schemes.

3.3 Systems

Processes can be grouped together in a system by parallel composition. The processes in the system are coupled through channels. Such a system acts like a process and can be combined with other processes to form a new system. A χ system is defined as follows:

$$\mathsf{syst}\ N(V_s) = [\![V_c|S_s]\!]$$

A system is identified by its name N and parameters V_s. System parameters V_s have the same format as process parameters V_p explained in the previous section. The system body resides between [[and]] brackets, and starts with a declaration list of local channel variables V_c, followed by system statement S_s. A channel variable c of type T is declared using the syntax $c\colon -T$. Only values of type T can be communicated through this channel.

The processes and systems are instantiated in the system statements S_s with the appropriate channels and parameters. Instantiations are written as $N(e_1,\ e_2,\ \ldots,\ e_n)$, where N is the name of an existing process or system and e_i $(1 \leq i \leq n)$ is an expression resulting in a value of the appropriate data type. Processes and systems are instantiated in parallel using the parallel composition operator $S_s \parallel S_s$. The local channel variables are used to connect different process instantiations to each other. A single channel connects one send port to one receive port. The data types of the two connected ports and the channel must match. Bundles (arrays) of channels can be specified as well.

A closed system has to be instantiated at the top level. This closed system has no communication ports parameters. The environment

$$\mathsf{xper}\ = [\![N(e_1, e_2, \ldots, e_n)]\!]$$

instantiates top level system N with parameter values e_1 to e_n, where e_i can be a basic or container data type, but not a port or channel data type.

3.4 Functions

Functions can be used to define calculations that cannot be expressed in a single line or that appear at several different places in the specification. The calculation is performed each time the function is called by a process. A χ function is defined as

$$\mathsf{func}\ N(V_f) \to T_r = [\![V_l|S_f]\!]$$

and identified by its name N and a list of formal input parameters V_f of type $v\colon T$. The return type of the function is T_r. Both T and T_r are basic or container data types. Local variables $x\colon T$ may be introduced in V_l, followed by the sequence of statements S_f that defines the function.

The statements that may be used in a function are the guarded command statement, the repetitive guarded command statement, sequential composition, assignment, and the skip statement as defined earlier in Sect. 3.2. One new statement is the return statement

$\uparrow e$ that completes the execution of the function and returns the value of expression e to the process statement or function statement that called the function. Multiple return statements are allowed in one function. The χ semantics assumes that functions behave in a strictly mathematical sense. Event (e.g., send or receive) statements are not allowed in functions.

3.5 Python interface

An MDO coordination specified in χ is represented by processes that communicate through channels. The numerical computations performed by the individual processes are modeled as functions. Generally, these calculations require routines external to χ, which means that a χ process has to be able to call external software. This has been realized by allowing functions written in Python (Python 2004; Lutz *et al.* 1999) to be treated like functions written in native χ (Hofkamp 2001). Python can be linked readily to other software. The Python interface is supported by the χ compiler that generates the executable to run the coordination, as explained below.

3.6 Execution of the coordination

Hofkamp and Rooda (2002b) developed a compiler to translate χ specifications into C++ code. Compilation of the generated code yields an executable program. After successful compilation, one can run the χ program, i.e., in our case run the coordination to solve the decomposed MDO problem. Since χ is a CSP-based language, it uses interleaving semantics for its execution, which means that *one* statement in *one* process at a time is being executed (except for communication statements between two processes that are always processed synchronously). The scheduler of χ determines which statement in which process will be executed next. In this manner, the χ scheduler takes care of the sequence of execution of the (parallel) process statements. Therefore, the concurrent χ language enables a straightforward implementation of the coordination; the user does not need to program the actual sequence in which the subproblems have to be solved, or to coordinate the data transfer among them.

However, interleaving semantics implies that function calls to external numerical routines are carried out one at a time, even if they occur in parallel processes. To allow true parallel execution, each external function call has to be decomposed into a four step procedure: initiate, notify initiation, ask for clearance to proceed, and retrieve results. An additional synchronization process has to be introduced. Processes get clearance to proceed through a synchronization with the synchronization process when they have all started their jobs. This is illustrated in the example presented in Etman *et al.* (2002). We are investigating whether this approach can be replaced by a more elegant one. The series of jobs generated during execution of the coordination can be queued and distributed over the available processors.

4 Application to analytical target cascading

Analytical target cascading (ATC) is a design optimization methodology of hierarchically decomposed engineering systems (Kim 2001; Kim *et al.* 2003). The original system design problem is partitioned into a model-based, multilevel hierarchical set of subproblems associated with subsystems and components. System design targets are defined at the top level and "cascaded down" to lower levels by formulating optimization subproblems to match subsystem and component response values with cascaded specifications.

ATC is a rigorous methodology for multilevel system design and has been demonstrated to be convergent under standard convexity and smoothness assumptions (Michelena *et al.* 2003). The ATC process has been applied successfully in vehicle design case studies (Michelena *et al.* 2001; Kim *et al.* 2002) and has been extended to the design of product families (Kokkolaras *et al.* 2002).

The key to the success of the ATC process lies in coordinating the solution process of the subproblems. Several coordination strategies are discussed in Michelena *et al.* (2003). However, it is not known a priori which strategy is the most efficient. This may even be problem dependent. We demonstrate that the χ language can be used to specify the coordination of the ATC process efficiently and compactly, allowing thus rapid implementation and investigation of alternative coordination strategies.

4.1 Review of the ATC formulation

Analytical target cascading is presented using a general notation, from which the design problem for each element (i.e., system, subsystem, or component) can be recovered as a special case. The formulation allows for design specifications to be introduced not only at the top level for the overall product, but also "locally" to account for individual subsystem and component requirements.

A typical example of a hierarchically decomposed system is shown in Fig. 4. We define the set $\mathcal{E}_i$ to include the elements of level i. For each element j in the set $\mathcal{E}_i$, we define the set $\mathcal{C}_{ij}$ to include the "children" of this element. For example, we have $\mathcal{E}_1 = \{\mathrm{B}, \mathrm{C}\}$ and $\mathcal{C}_{1\mathrm{B}} = \{\mathrm{D}, \mathrm{E}\}$.

The responses $\mathbf{R}$ of each element are classified into two types: responses $\hat{\mathbf{R}}$ associated to "local" targets and responses $\tilde{\mathbf{R}}$ associated to "cascaded" targets; the latter link successive levels in the problem hierarchy. Similarly, two types of design variables can be distinguished for each subproblem: local design variables $\mathbf{x}_{ij}$ and shared design

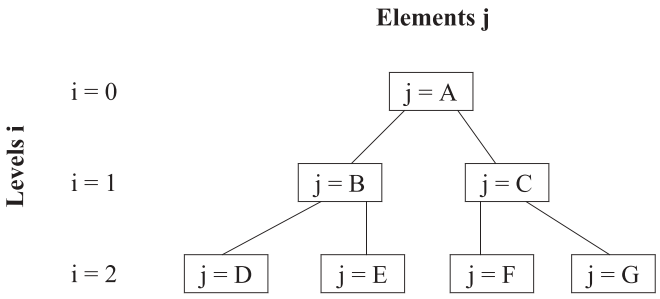

Fig. 4 Example of hierarchically partitioned design problem

variables $\mathbf{y}_{ij}$. The design problem P_{ij} corresponding to the jth element at the ith level is formulated as follows:

$$\min_{\bar{\mathbf{x}}_{ij}} \left\|\hat{\mathbf{R}}_{ij} - \mathbf{T}_{ij}\right\|_2^2 + \left\|\tilde{\mathbf{R}}_{ij} - \tilde{\mathbf{R}}_{ij}^U\right\|_2^2 + \|\mathbf{y}_{ij} - \mathbf{y}_{ij}^U\|_2^2 + \varepsilon_{ij}^R + \varepsilon_{ij}^y$$

$$\text{subject to} \sum_{k\in\mathcal{C}_{ij}} \left\|\tilde{\mathbf{R}}_{(i+1)k} - \tilde{\mathbf{R}}_{(i+1)k}^L\right\|_2^2 \le \varepsilon_{ij}^R$$

$$\sum_{k\in\mathcal{C}_{ij}} \|\mathbf{y}_{(i+1)k} - \mathbf{y}_{(i+1)k}^L\|_2^2 \le \varepsilon_{ij}^y$$

$$\mathbf{g}_{ij}(\mathbf{R}_{ij}, \mathbf{x}_{ij}, \mathbf{y}_{ij}) \le 0\,,$$

$$\mathbf{h}_{ij}(\mathbf{R}_{ij}, \mathbf{x}_{ij}, \mathbf{y}_{ij}) = 0\,, \qquad (2)$$

where $\mathbf{R}_{ij} = [\hat{\mathbf{R}}_{ij}^t, \tilde{\mathbf{R}}_{ij}^t]^t = \mathbf{r}_{ij}(\tilde{\mathbf{R}}_{(i+1)k_1}, \ldots, \tilde{\mathbf{R}}_{(i+1)k_{c_{ij}}}, \mathbf{x}_{ij}, \mathbf{y}_{ij})$, $\mathcal{C}_{ij} = \{k_1, \ldots, k_{c_{ij}}\}$, and c_{ij} is the number of children. Note that an element's response depends both on the element's design variables and its children's responses. In the above problem formulation,

- $\bar{\mathbf{x}}_{ij} = [\mathbf{x}_{ij}^t, \mathbf{y}_{ij}^t, \mathbf{y}_{(i+1)k_1}^t, \ldots, \mathbf{y}_{(i+1)k_{c_{ij}}}^t, \tilde{\mathbf{R}}_{(i+1)k_1}^t, \ldots, \tilde{\mathbf{R}}_{(i+1)k_{c_{ij}}}^t, \varepsilon_{ij}^R, \varepsilon_{ij}^y]^t$ is the vector of all optimization variables,
- $\mathbf{x}_{ij}$ is the vector of local design variables, that is, variables exclusively associated with the element,
- $\mathbf{y}_{ij}$ is the vector of shared design variables, that is, variables associated with two or more elements that share the same parent,
- ε_{ij}^R is the tolerance optimization variable for coordinating the responses of the element's children,
- ε_{ij}^y is the tolerance optimization variable for coordinating the shared design variables of the element's children,
- $\mathbf{T}_{ij}$ is the vector of local target values,
- $\tilde{\mathbf{R}}_{ij}^U$ is the vector of response values cascaded down to the element from its parent,
- $\mathbf{y}_{ij}^U$ is the vector of shared design variable values cascaded down to the element from its parent,
- $\tilde{\mathbf{R}}_{(i+1)k}^L$ is the vector of response values cascaded up to the element from its kth child,
- $\mathbf{y}_{(i+1)k}^L$ is the vector of shared design variable values cascaded up to the element from its kth child, and
- $\mathbf{g}_{ij}$ and $\mathbf{h}_{ij}$ are vector functions representing inequality and equality design constraints, respectively.

4.2 ATC coordination

Analytical target cascading requires the iterative solution of the optimization subproblems according to the hierarchical structure of the decomposed problem. Figure 5 illustrates the information flow between a subproblem and a) its parent and b) its children before and after the optimization subproblem has been solved.

One of the advantages of ATC is that subproblems at each level can be solved in parallel assuming that the necessary parameters have been updated. Therefore, the coordination strategy has to specify the sequence in which the levels will be visited. The number of possibilities increases in hierarchies with multiple levels. Figure 6 depicts coordination alternatives for a hierarchy of three levels. Note that (Michelena *et al.* 2003) showed theoretical convergence properties for Schemes III and IV only.

Next we explain how the ATC coordination can be specified as a coupled system of χ processes. A process will be instantiated for each element in the hierarchy. This process specifies the sequence of statements to exchange data with its parents and children and solve the optimization problem P_{ij}. The exact process specification for problem P_{ij} depends on the coordination scheme.

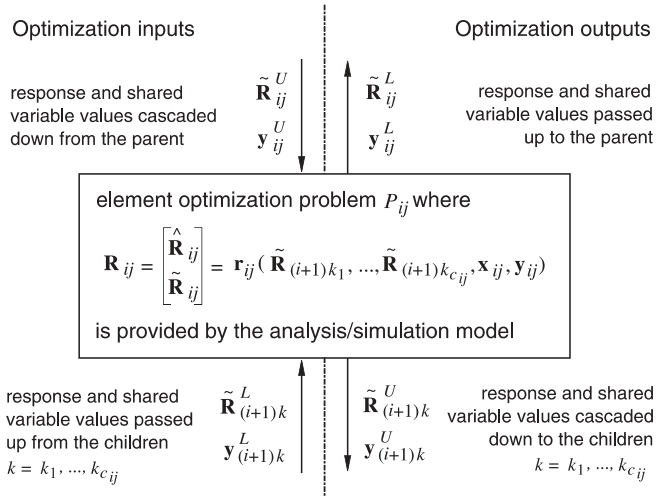

Fig. 5 Information flow for subproblem P_{ij}

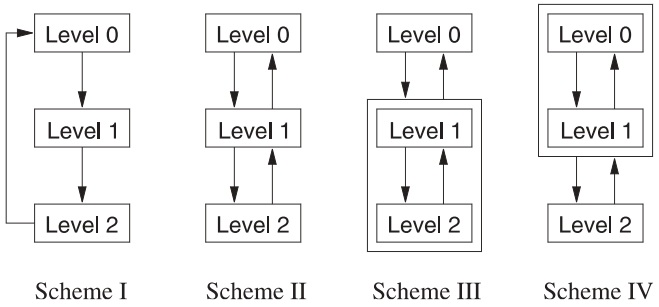

Fig. 6 Coordination alternatives for hierarchy of three levels

4.3 Loop-like coordination

Consider first a coordination scheme that visits all levels in the hierarchy in a loop-like sequence (Scheme I in Fig. 6): $i = 0, 1, 2, \ldots, N, 0, \ldots, N$, etc. where $i = 0$ and $i = N$ represent the top and bottom level, respectively. Note that Fig. 6 only illustrates the sequence in which subproblems within a level are solved; it does not depict data exchange or flow.

In the loop-like coordination, the top-level process C_{top} associated to problem P_0 (index j is dropped since there is only one element) starts by carrying out an optimization to determine response and shared variable target values for its children. During the first iteration, C_{top} uses some appropriate (possibly zero) initial values for the response and shared variables that are normally passed up from its children. After C_{top} has cascaded its targets, it waits until it receives updated values of children response and shared variables. It then carries out a new optimization, compares the new vector of optimization variables to the previous one, checks the convergence status of the children processes, and either stops (when optimization variable values have not changed significantly and all children process are converged) or reiterates.

A second type of process is needed for the intermediate-level subproblems P_{ij} that have both a parent and one or more children, i.e., $i \in 1, \ldots, N-1$. For the loop-like scheme, such an intermediate-level process C_{mid} carries out the following sequence of steps: Receive targets from the parent, carry out an optimization (during the first iteration use an initial guess for the children response and shared values), cascade determined targets to the children, wait until all the children have sent updated response and shared values, pass updated values to the parent.

Thirdly, we need bottom-level processes C_{bot} for subproblems P_{Nj}. A bottom-level process waits until it receives targets from its parent, carries out an optimization, and returns updated values.

The loop-like coordination scheme for a decomposed problem is specified by coupling instances of the top-level, intermediate-level, and bottom-level processes. No additional process is needed to control the overall coordination. Initially, all processes are waiting to receive target data from their parents, except for the top-level process. The top-level process carries out an optimization and cascades targets down to its children. These intermediate-level processes may carry out their optimizations in parallel; all the other processes are waiting to receive new data. The computed targets are cascaded down. This proceeds until the bottom-level processes have been reached and have carried out their optimizations. The bottom-level processes pass up updated response and shared variable target values to their parents, which update their parents without carrying out any new optimizations. In this manner, updated target values are passed up level-by-level until the top-level process is reached. If convergence has not occurred, a new ATC iteration starts.

Note that the tree of processes does not need to be symmetrical. The same coordination arises if one branch has more levels than another branch. If the ATC hierarchy consists of only two levels, the coordination specification contains only top-level and bottom-level process instances.

A repository process R is introduced to facilitate monitoring processes during the ATC iterations. The subproblem processes send updated values of their optimization variables to this process R. These optimization variable values are stored in R. After completion of the ATC process, the complete iteration history is available in R, and may be used for further analysis.

4.4 Definition of χ processes for loop-like coordination

The χ specifications of the processes in the loop-like coordination scheme are presented below. First, the variable types, as presented in Fig. 7, are introduced. Herein, types vx, vr, and vy denote vector arrays of reals of fixed (maximum) length mx, my, and mr, respectively; χ is a strong-typed language requiring that the dimensions of arrays are known at compilation time. Types vys and vrs are matrix arrays of sizes $ms \times mr$ and $ms \times my$, respectively. Types vxbtop, vxbmid, and vxbbot are tuples containing two or more data-elements of type vx, vy, vys, vrs, or real. These three tuple types match the optimization variables of the top-level, intermediate-level, and bottom-level design problems, respectively. The elements in the tuples can be accessed through their identifiers defined before the dot operator. Finally, type par is defined as a tuple of reals and nats to store the scaling values used in the optimization problem of the process, the number of children of the process, and the actual array dimensions used in the process.

Using these type definitions, the χ specification of the top-level process C_{top} is presented in Fig. 8. The first element f in the parameter list of C_{top} defines the external function that will carry out the optimization. Parameters b and c represent the send and receive port arrays, respectively, through which data are sent to and received from the children of C_{top}. The fourth parameter e

```
type vx     = real^mx
   , vr     = real^mr
   , vy     = real^my
   , vys    = vy^ms
   , vrs    = vr^ms
   , vxbtop = x.vx ×          ys.vys × rs.vrs × epsr.real × epsy.real
   , vxbmid = x.vx × y.vy × ys.vys × rs.vrs × epsr.real × epsy.real
   , vxbbot = x.vx × y.vy
   , par    = wr.real × wy.real × ns.nat × nx.nat × ny.nat
              × nys.nat × nrs.nat
```

Fig. 7 Data types needed for the processes

```
proc Ctop(f : (vxbtop, vr, vrs, vys, par) → vxbtop × vr
          ,b : (!vr × vy)^ms, c : (?vr × vy × bool)^ms, e :!vxbtop
          ,s :!void, T : vr, xb0 : vxbtop, p : par, tol : real
          ) =
[[ r : vr, rLs : vrs, yLs : vys, xb : vxbtop, cvrg : bool, cvrgs : bool^ms
, i, n : nat
| n := 0; cvrg := false; rLs := xb0.rs; yLs := xb0.ys
; ⟨xb, r⟩ := f(xb0, T, rLs, yLs, p); e!xb
; xb0 := xb
; *[¬cvrg ∧ n < maxiter
    ⟶ n := n + 1
      ; i := 0; *[i < p.ns ⟶ b.i!⟨(xb.rs).i, (xb.ys).i⟩; i := i + 1]
      ; i := 0; *[i < p.ns ⟶ c.i?⟨rLs.i, yLs.i, cvrgs.i⟩; i := i + 1]
      ; ⟨xb, r⟩ := f(xb0, T, rLs, yLs, p); e!xb
      ; cvrg := topnorm(xb0, xb, p) < tol
      ; cvrg := checkcvrg(cvrg, cvrgs, p.ns)
      ; xb0 := xb
  ]
; s!
]]
```

Fig. 8 Top-level coordination process C_{top}

is a send port to repository R for updating the coordination history; the fifth parameter s is a synchronization port to R for notifying completion of the ATC process. The last four parameters correspond to top-level response targets, initial values of the optimization variables, subproblem parameter values, and convergence tolerance, respectively.

The following local variables are introduced in C_{top}: a response vector r, matrices rLs and yLs of response and shared variable target values, respectively, that have been passed up from the children, a tuple of optimization variables xb, a boolean variable $cvrg$, a boolean array $cvrgs$, and two natural variables i and n. C_{top} starts by carrying out an initial optimization. The following sequence of tasks is then executed: cascade targets (i.e., send targets to children), receive updated target values (from children), carry out optimization, check convergence, and update optimization variable values.

C_{top} is declared locally converged if the square of the norm of the difference between the previous and the current iterates is smaller than some value tol, i.e., if $\|\bar{\mathbf{x}}^{(n)} - \bar{\mathbf{x}}^{(n-1)}\|_2^2 \leq tol$. Note that the vector of optimization variables $\bar{\mathbf{x}}$ has different instantiations for the top, intermediate, and bottom levels. C_{top} is declared globally converged if convergence has occurred for the "local" optimization problem as well as for all the children optimization problems. The function *checkcvrg* returns true if this is the case. To this end, the children pass up their convergence status in addition to the updated response and shared variable values. C_{top} stores the convergence status of its children in the boolean array $cvrgs$. The functions *topnorm* and *checkcvrg* are specified as χ functions. The ATC process is terminated when C_{top} has converged or some predefined maximum number of iterations has been reached. A synchronization is sent to repository process R to acknowledge this.

The intermediate-level process C_{mid} is specified in a similar way as shown in Fig. 9. C_{mid} has four communication ports to parent and children: a and d are receive and sent ports coupled to the parent; b and c are receive and sent port arrays (bundles), which are coupled to the children. After the initialization, C_{mid} repeats indefinitely the following sequence of statements: receive response and shared variable target values through port a; cascade targets to the children through ports $b.i$; receive updated response and shared variable values as well as the status of convergence of children-problems through ports $c.i$; send updated values and convergence status to the parent through port d. After C_{mid} has received the target values cascaded down from its parent, it carries out the local optimization defined by function f, using initial optimization variable values $xb0$ and subsystem parameters p. After the convergence status of the children has been received, the local and overall convergence of C_{mid} is determined as described for C_{top}, however, a modified norm function (*midnorm*) is used.

```
proc Cmid(f : (vxbmid, vr, vy, vrs, vys, par) → vxbmid × vr
          ,a :?vr × vy, b : (!vr × vy)^ms, c : (?vr × vy × bool)^ms
          ,d :!vr × vy × bool, e :!vxbmid
          ,xb0 : vxbmid, p : par, tol : real
          ) =
[[ r, rUS : vr, rLs : vrs, yUS : vy, yLs : vys, xb : vxbmid
, cvrg : bool, cvrgs : bool^ms, i : nat
| rLs := xb0.rs; yLs := xb0.ys
; *[true
    ⟶ a?⟨rUS, yUS⟩
      ; ⟨xb, r⟩ := f(xb0, rUS, yUS, rLs, yLs, p); e!xb
      ; i := 0; *[i < p.ns ⟶ b.i!⟨(xb.rs).i, (xb.ys).i⟩; i := i + 1]
      ; i := 0; *[i < p.ns ⟶ c.i?⟨rLs.i, yLs.i, cvrgs.i⟩; i := i + 1]
      ; cvrg := midnorm(xb0, xb, p) < tol
      ; cvrg := checkcvrg(cvrg, cvrgs, p.ns)
      ; xb0 := xb
      ; d!⟨r, xb.y, cvrg⟩
  ]
]]
```

Fig. 9 Intermediate-level coordination process C_{mid}

```
proc Cbot(f : (vxbbot, vr, vy, par) → vxbbot × vr, a :?vr × vy
          ,d :!vr × vy × bool, e :!vxbbot
          ,xb0 : vxbbot, p : par, tol : real
          ) =
[[ r, rUS : vr, yUS : vy, xb : vxbbot, cvrg : bool
| *[true
    ⟶ a?⟨rUS, yUS⟩
      ; ⟨xb, r⟩ := f(xb0, rUS, yUS, p); e!xb
      ; cvrg := botnorm(xb0, xb, p) < tol
      ; xb0 := xb
      ; d!⟨r, xb.y, cvrg⟩
  ]
]]
```

Fig. 10 Bottom-level coordination process C_{bot}

```
proc R(t :?vxbtop, m : (?vxbmid)^nm, b : (?vxbbot)^nb, s :?void
       , tol, sca : real
       ) =
[[ vt : vxbtop, td : vxbtop*, vm : vxbmid, md : (vxbmid*)^nm
 , vb : vxbbot, bd : (vxbbot*)^nb, p : bool
 | td := [ ]; md := ini_mid(); bd := ini_bot()
 ; *[                  true; t?vt      --> td := td ++ [vt]
   [] j : nat <- 0..nm : true; m.j?vm  --> md.j := md.j ++ [vm]
   [] j : nat <- 0..nb : true; b.j?vb  --> bd.j := bd.j ++ [vb]
   []                  true; s?        --> p := pp(td, md, bd, tol, sca)
   ]
]]
```

Fig. 11 Repository process R

The bottom-level process C_{bot} is specified in Fig. 10. C_{bot} receives targets through port a, carries out the optimization defined by f, and passes up updated values and convergence status through port d.

Finally, there is a repository process R that collects optimization results every time a subproblem is solved, and stores them in lists (see Fig. 11). Note that process R is not essential for the ATC coordination itself. Process R has ports to all subproblem processes. R has three different port parameters since the optimization variables tuples differ for the top-level, intermediate-level, and bottom-level processes, respectively. Additionally we have a synchronization port to the top-level process for the acknowledgment of the ATC finish. The key statement of R is a repetitive selective waiting statement. The repetitive selective waiting statement waits until a new iteration update is received from one of the subproblem processes or until a synchronization is received from the top-level process. R stores the iteration updates of the subproblems in separate lists designated to each of the processes. If a synchronization communication takes place, the lists of the subproblems are processed by function pp.

4.5 Alternative coordination schemes

Alternative coordination schemes can be obtained by modifying the specifications of the subproblem processes. For example, coordination Scheme II ($i = 0, 1, \ldots, N-1, N, N-1, \ldots, 1, 0$, etc.) is obtained by simply inserting the line

$$; \langle xb,\, r\rangle := f(xb0,\, rUS,\, yUS,\, rLs,\, yLs,\, p)$$

after the statement

$$; *[i < p.ns \longrightarrow c.i?\langle rLs.i,\, yLs.i,\, cvrgs.i\rangle; i := i + 1]$$

in the C_{mid} process specification of Fig. 9. By doing this, intermediate-level processes carry out an additional optimization every time updated values are passed up.

Michelena *et al.* (2003) considered nested coordination schemes (Schemes III and IV in Fig. 6). Scheme III, for instance, can be specified by inserting an iteration loop between the receive and send statements with respect to the parent-related communication. In this manner, communications with the children are nested with respect to communications with the parent. The repetition statement is inserted into the C_{mid} process specification of Fig. 9 as follows:

```
; *[true
    --> a?⟨rUS, yUS⟩
    ; *[¬cvrg ∧ n < maxiter
        --> n := n + 1
          ; ⟨xb, r⟩ := f(xb0, rUS, yUS, rLs, yLs, p)
          ; ...
          ; xb0 := xb
       ]
    ; d!⟨r, xb.y, cvrg⟩
  ] .
```

The ease by which these different coordination schemes can be specified is a clear advantage of a high-level coordination language such as χ. Note that processes C_{top}, C_{mid}, C_{bot}, and R have been specified for analytical target cascading in general. By using specific instances of these processes one can easily build the coordination architecture of any ATC application at hand. This is demonstrated for a hierarchy of three levels in the next section. Once the overall system coupling has been specified, the χ software allows one to run the ATC process using the Python interface.

5 Example

We will now demonstrate the specification of different coordination strategies using the χ language on a simple but illustrative analytical target cascading problem.

5.1 Mathematical formulation of example problem

Our example is based on the geometric programming problem

$$\min_{\mathbf{z}\geq\mathbf{0}} \quad z_1^2 + z_2^2$$

$$\text{s.t.} \quad \frac{z_3^{-2} + z_4^2}{z_5^2} - 1 \leq 0\,; \quad \frac{z_5^2 + z_6^{-2}}{z_7^2} - 1 \leq 0$$

$$\frac{z_8^2 + z_9^2}{z_{11}^2} - 1 \leq 0\,; \quad \frac{z_8^{-2} + z_{10}^2}{z_{11}^2} - 1 \leq 0$$

$$\frac{z_{11}^2 + z_{12}^{-2}}{z_{13}^2} - 1 \leq 0\,; \quad \frac{z_{11}^2 + z_{12}^2}{z_{14}^2} - 1 \leq 0$$

$$z_1^2 - z_3^2 - z_4^{-2} - z_5^2 = 0$$
$$z_2^2 - z_5^2 - z_6^2 - z_7^2 = 0$$
$$z_3^2 - z_8^2 - z_9^{-2} - z_{10}^{-2} - z_{11}^2 = 0$$
$$z_6^2 - z_{11}^2 - z_{12}^2 - z_{13}^2 - z_{14}^2 = 0\,. \tag{3}$$

Kim (2001) decomposed this problem using the equality constraints as responses within a bi-level hierarchical structure and demonstrated the application of the ATC process. Here, we decompose the original problem into three levels as shown in Fig. 12; z_5 is the shared variable that couples the subproblems of the intermediate level. Note that z_{11} is a shared variable coupling the two problems of the bottom level. The ATC formulation does not allow subproblems to share variables unless they are children of the same parent. Since the purpose of this example is to illustrate the χ implementation of alternative coordination strategies, we treat z_{11} as a parameter using its known optimal value.

The subproblems are formulated in the next subsections following the notation presented in Sect. 4.1. The index j is dropped at the top level since there is only one element.

Top-level problem

Problem P_0 is formulated as:

$$\min_{\mathbf{R}_{11},\mathbf{R}_{12},\mathbf{y}_0,\varepsilon_0^y,\varepsilon_0^R} \quad \|\mathbf{R}_0 - \mathbf{T}_0\| + \varepsilon_0^y + \varepsilon_0^R$$
$$\text{s.t.} \quad \left\|\mathbf{y}_0 - \mathbf{y}_{11}^L\right\| + \left\|\mathbf{y}_0 - \mathbf{y}_{12}^L\right\| \le \varepsilon_0^y$$
$$\left\|\mathbf{R}_{11} - \mathbf{R}_{11}^L\right\| + \left\|\mathbf{R}_{12} - \mathbf{R}_{12}^L\right\| \le \varepsilon_0^R\,, \tag{4}$$

where $\mathbf{R}_{11} := z_1, \mathbf{R}_{12} := z_2, \mathbf{y}_0 := z_5, \mathbf{R}_0 = r_0(\mathbf{R}_{11}, \mathbf{R}_{12}) = z_1^2 + z_2^2$, and $\mathbf{T}_0 = 0$. Note that z_1, z_2, and z_5, correspond to the formulation of the original problem, and that z_5 is a shared variable computed at the problems of the intermediate level and coordinated at the top level.

Intermediate-level problems

There are two problems at the intermediate level. Problem P_{11} is formulated as:

$$\min_{\mathbf{R}_{21},\mathbf{y}_{11},\mathbf{x}_{11},\varepsilon_{11}^R} \quad \left\|\mathbf{R}_{11} - \mathbf{R}_{11}^U\right\| + \left\|\mathbf{y}_{11} - \mathbf{y}_0^U\right\| + \varepsilon_{11}^R$$
$$\text{s.t.} \quad \left\|\mathbf{R}_{21} - \mathbf{R}_{21}^L\right\| \le \varepsilon_{11}^R$$
$$\mathbf{g}_{11}(\mathbf{R}_{21}, \mathbf{x}_{11}, \mathbf{y}_{11}) \le \mathbf{0}\,, \tag{5}$$

where $\mathbf{R}_{21} := z_3$, $\mathbf{x}_{11} := z_4$, $\mathbf{y}_{11} := z_5$, $\mathbf{R}_{11} = r_{11}(\mathbf{R}_{21}, \mathbf{x}_{11}, \mathbf{y}_{11}) = \sqrt{z_3^2 + z_4^{-2} + z_5^2}$, and $\mathbf{g}_{11}(\mathbf{R}_{21}, \mathbf{x}_{11}, \mathbf{y}_{11}) = (z_3^{-2} + z_4^2) z_5^{-2} - 1$.

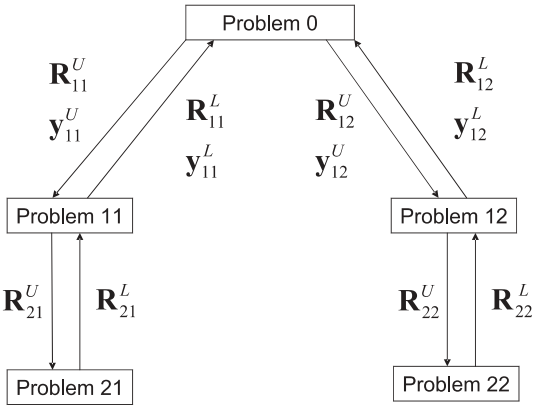

Fig. 12 Hierarchical structure of decomposed problem

The second intermediate-level subproblem P_{12} is stated as:

$$\min_{\mathbf{R}_{22},\mathbf{y}_{12},\mathbf{x}_{12},\varepsilon_{12}^R} \quad \left\|\mathbf{R}_{12} - \mathbf{R}_{12}^U\right\| + \left\|\mathbf{y}_{12} - \mathbf{y}_0^U\right\| + \varepsilon_{12}^R$$
$$\text{s.t.} \quad \left\|\mathbf{R}_{22} - \mathbf{R}_{22}^L\right\| \le \varepsilon_{12}^R\,,$$
$$\mathbf{g}_{12}(\mathbf{R}_{22}, \mathbf{x}_{12}, \mathbf{y}_{12}) \le \mathbf{0}\,, \tag{6}$$

where $\mathbf{R}_{22} := z_6$, $\mathbf{x}_{12} := z_7$, $\mathbf{y}_{12} := z_5$, $\mathbf{R}_{12} = r_{12}(\mathbf{R}_{22}, \mathbf{x}_{12}, \mathbf{y}_{12}) = \sqrt{z_5^2 + z_6^2 + z_7^2}$, and $\mathbf{g}_{12}(\mathbf{R}_{22}, \mathbf{x}_{12}, \mathbf{y}_{12}) = (z_5^2 + z_6^{-2}) z_7^{-2} - 1$.

Bottom-level problems

There are two problems at the bottom level. Problem P_{21} is given by:

$$\min_{\mathbf{x}_{21}} \quad \left\|\mathbf{R}_{21} - \mathbf{R}_{21}^U\right\|$$
$$\text{s.t.} \quad \mathbf{g}_{21}(\mathbf{x}_{21}) \le \mathbf{0}\,, \tag{7}$$

where $\mathbf{x}_{21} := [z_8, z_9, z_{10}]$, $p = 1.3 (= z_{11})$, $\mathbf{R}_{21} = r_{21}(\mathbf{x}_{21}) = \sqrt{z_8^2 + z_9^{-2} + z_{10}^{-2} + p^2}$, and

$$\mathbf{g}_{21}(\mathbf{x}_{21}) = \begin{bmatrix} \left(z_8^2 + z_9^2\right) p^{-2} - 1 \\ \left(z_8^{-2} + z_{10}^2\right) p^{-2} - 1 \end{bmatrix}.$$

The formulation of subproblem P_{22} is:

$$\min_{\mathbf{x}_{22}} \quad \left\|\mathbf{R}_{22} - \mathbf{R}_{22}^U\right\|$$
$$\text{s.t.} \quad \mathbf{g}_{22}(\mathbf{x}_{22}) \le \mathbf{0}\,, \tag{8}$$

where $\mathbf{x}_{22} := [z_{12}, z_{13}, z_{14}]$, $p = 1.3 (= z_{11})$, $\mathbf{R}_{22} = r_{22}(\mathbf{x}_{22}) = \sqrt{z_{12}^2 + z_{13}^2 + z_{14}^2 + p^2}$, and

$$\mathbf{g}_{22}(\mathbf{x}_{22}) = \begin{bmatrix} \left(p^2 + z_{12}^{-2}\right) z_{13}^{-2} - 1 \\ \left(p^2 + z_{12}^2\right) z_{14}^{-2} - 1 \end{bmatrix}.$$

5.2 Implementation of coordination using χ

Implementation of the ATC coordination using χ requires the instantiation of a number of processes such as those specified in Sect. 4. This is illustrated for Scheme I in our example problem. The following constant values are used: the maximum length of design variable arrays $mx = 3$ (there is a maximum of three local design variables in any of the subproblems), the maximum length of response variable arrays $mr = 1$, the maximum length of shared variable arrays $my = 1$, and the maximum number of children in any subproblem $ms = 2$. The actual number of intermediate-level processes is $nm = 2$ (P_{11} and P_{12}), and the actual number of bottom-level processes is $nb = 2$ (P_{21} and P_{22}). Moreover, we define the maximum number of iterations *maxiter* and a "not-a-number" value *nan* that is used for empty entries in the fixed-length arrays.

The system that couples the processes is instantiated according to Fig. 13. The top-level problem P_0 requires a process instantiation of C_{top} as specified in Fig. 8. Each of the two children of P_0 requires an instantiation of the intermediate-level process C_{mid} as shown in Fig. 9, representing problems P_{11} and P_{12}. The bottom-level subproblems P_{21} and P_{22} are represented by two process instantiations of C_{bot} as defined in Fig. 10. Finally, repository process R needs to be instantiated (see Fig. 11).

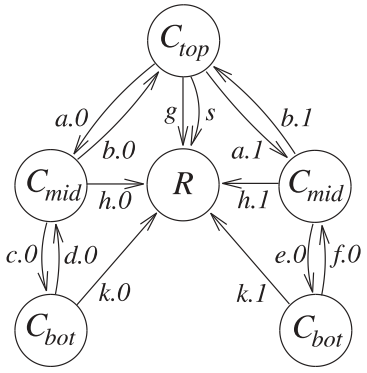

Fig. 13 Processes and channels in the three-level ATC example

These six processes are coupled to each other by channels of appropriate data types. The channel names are defined in Fig. 13. For example, C_{top} sends data of type $\text{vr} \times \text{vy}$ to each of its children via channel array a, and receives data of type $\text{vr} \times \text{vy} \times \text{bool}$ from its children via channel array b. C_{top} has two send channels to repository R: channel g to send iteration updates, and channel s to acknowledge the completion of the ATC coordination. The intermediate-level process C_{mid} representing design problem P_{11} receives data of type $\text{vr} \times \text{vy}$ from C_{top} via channel $a.0$ and sends data of type $\text{vr} \times \text{vy} \times \text{bool}$ via channel $b.0$. The C_{mid} process representing P_{11} also communicates through channels $c.0$ and $d.0$ with the C_{bot} process that represents its child P_{21}. Problems P_{12} and P_{22} are represented in a similar fashion. The two C_{mid} processes and the two C_{bot} processes send updates to repository R through channel arrays h and k, respectively.

Note that channels a to f are channel arrays of size $ns = 2$. However, P_{11} has only one child, therefore only the first elements of the channel arrays c and d are used, i.e., $c.0$ and $d.0$. The same holds for P_{12} and channel arrays e and f. The constant ns cannot be defined as a separate parameter in the process definition; χ is a strongly typed language and does not allow (channel) array type specifications of variable size. At present, χ also does not allow specification of the actual array sizes used in a process upon instantiation of the process in a system.

The complete coordination instantiation of the three-level geometric programming ATC example is included in Fig. 14. All processes in system S are instantiated with the appropriate channels and parameters. All optimization variables in the tuple $xb0$ of processes C_{top}, C_{mid}, and C_{bot} are given an initial value of one, except for the ε_{ij}^{R} and ε_{ij}^{y} variables, which are initialized to zero. Parameter array p in processes C_{top}, C_{mid}, and C_{bot} contains for each instantiation: the scaling parameter value *sca* for the ε terms in the objective function of problems P_0, P_{11}, and P_{12}; the number of children; the number of local design variables; the number of shared variables; and the num-

```
const mx : nat = 3, mr : nat = 1, my : nat = 1, ms : nat = 2
      , nm : nat = 2, nb : nat = 2
      , maxiter : nat = 1000, nan : real = −9.9e99

syst S(tol,sca : real) =
[[a,c,e : (−vr × vy)^ms, b,d,f : (−vr × vy × bool)^ms, g : −vxbtop, h : (−vxbmid)^nm, k : (−vxbbot)^nb, s : −void
 |Ctop(P0, a,b,g,s, ⟨0.0⟩, ⟨⟨nan,nan,nan⟩,⟨⟨1.0⟩,⟨1.0⟩⟩,⟨⟨1.0⟩,⟨1.0⟩⟩,0.0,0.0⟩, ⟨sca,sca,2,0,0,1,1⟩, tol)
||Cmid(P11, a.0,c,d,b.0,h.0, ⟨⟨1.0,nan,nan⟩,⟨1.0⟩,⟨⟨nan⟩,⟨nan⟩⟩,⟨⟨1.0⟩,⟨nan⟩⟩,0.0,0.0⟩, ⟨sca,sca,1,1,1,0,1⟩, tol)
||Cmid(P12, a.1,e,f,b.1,h.1, ⟨⟨1.0,nan,nan⟩,⟨1.0⟩,⟨⟨nan⟩,⟨nan⟩⟩,⟨⟨1.0⟩,⟨nan⟩⟩,0.0,0.0⟩, ⟨sca,sca,1,1,1,0,1⟩, tol)
||Cbot(P21, c.0,d.0,k.0, ⟨⟨1.0,1.0,1.0⟩,⟨nan⟩⟩, ⟨nan,nan,0,3,0,0,0⟩, tol)
||Cbot(P22, e.0,f.0,k.1, ⟨⟨1.0,1.0,1.0⟩,⟨nan⟩⟩, ⟨nan,nan,0,3,0,0,0⟩, tol)
||R(g,h,k,s, tol,sca)
]]

xper(tol,sca : real) =[[S(tol,sca)]]
```

Fig. 14 Coordination instantiation of the three-level ATC example

ber of response and shared variable target values of the children (assumed to be equal), respectively. The last line of the coordination instantiation in Fig. 14 denotes that after compilation a system execution of S can be carried with tolerance *tol* and scaling *sca* as input.

Schemes I, II, III, and IV of Fig. 6 have been specified and implemented using χ. Results are presented and discussed in Tzevelekos *et al.* (2003) and Hulshof (2003).

6 Summary and discussion

In our opinion, a high-level concurrent programming language for specifying MDO coordination rigorously can significantly improve implementation and facilitate testing of alternative strategies. Such a language is especially needed when the scale and complexity of the distributed optimal design problem architecture is large, e.g., in hierarchically decomposed multilevel systems. Moreover, concurrency requires precise treatment at a high level of abstraction to avoid detailed coding of the sequence of solving subproblems and exchange of data. For practical use, the high-level concurrent language should support the execution of the specified coordination. Finally, a specification language for engineering-based MDO approaches has to be able to deal with the "black-box" nature of the disciplines (or subsystems), as well as the large amount of numerical data that must be exchanged.

We have used the χ language, which meets both the "black-box" and the data handling requirements. It is a highly expressive CSP-based language that contains advanced data modeling constructs. Using χ, MDO coordination is specified as a number of parallel processes that operate independently and communicate synchronously over pre-defined channels. The advantage of χ is that it has been designed for modeling purposes: it is compact (has few language constructs), easy to understand, and offers a clear concept of concurrency. Furthermore, χ can execute function calls to external software by means of a Python interface.

We demonstrated how χ can be used to specify and implement alternative coordination strategies in analytical target cascading, a design optimization methodology of hierarchically decomposed multilevel systems. Top-level, intermediate-level, and bottom-level processes representing the optimization subproblems in ATC have been defined and composed into a system. We have shown that once a specific coordination strategy is implemented, additional schemes can be easily derived by modifying the specification. Moreover, larger problems can be treated readily by simply adding instantiations of the pre-defined processes.

An analytical example was used to illustrate that the χ language is well-suited for specifying and implementing the ATC coordination, and that it enables rapid investigation of alternative strategies. Response surface techniques are used frequently in MDO (Kodiyalam and Sobieszczanski-Sobieski 2000; Sobieski and Kroo 2000; Liu *et al.* 2004). In Etman *et al.* (2002) we illustrated that response surface modeling can also be specified as part of the coordination. We conclude that a coordination specification language such as χ may provide new opportunities in MDO implementations of large-scale and complex problems.

Acknowledgements We would like to thank Erjen Lefeber for providing the new χ-style file for LaTeX.

References

Alexandrov, N.M.; Lewis, R.M. 1999: Comparative properties of collaborative optimization and other approaches to MDO. In: *First ASMO UK/ISSMO Conference on Engineering Design Optimization*, 8–9July 1999, MCB press

Alexandrov, N.M.; Lewis, R.M. 2000: Analytical and computational properties of distributed approaches to MDO. In: *8th AIAA/USAF/MASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, 6–8 September 2000, Long Beach, paper AIAA-2000-4718

Alexandrov, N.M.; Lewis, R.M. 2002: Analytical and computational aspects of collaborative optimization for multidisciplinary design. *AIAA J.* **40**, 301–309

Backus, J. 1960: The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM conference. In: *Proceedings ICIP*, Unesco, 125–131

Balling, R.J.; Sobieszczanski-Sobieski, J. 1996: Optimization of coupled systems: a critical overview of approaches. *AIAA J.* **34**, 6–17

Bos, V.; Kleijn, J.J.T. 2002: *Formal specification and analysis of industrial systems*, Dissertation, Eindhoven University of Technology

Cramer, E.J.; Dennis, J.E. Jr.; Frank, P.D.; Lewis, R.M.; Shubin, G.R. 1994: Problem formulation for multidisciplinary optimization. *SIAM J. Optim.* **4**, 754–776

Dijkstra, E.W. 1975: Guarded commands, nondeterminacy, and formal derivation of programs. *Commun. ACM* **18**, 453–457

Etman, L.F.P.; Hofkamp, A.T.; Rooda, J.E.; Kokkolaras, M.; Papalambros, P.Y. 2002: Coordination specification for distributed optimal system design. *Proc. 9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Atlanta, GA, paper no. AIAA-2002-5410

Fourer, R.; Gay, D.M.; Kernighan, B.W. 1993: *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press

Haftka, R.T.; Watson, L.T. 2004: Multidisciplinary design optimization with quasiseparable subsystems. *Optim. Eng.*, in press

Hoare, C.A.R. 1978: Communicating sequential processes, *Commun. ACM*. **21**, 666–677

Hoare, C.A.R. 1985: *Communicating Sequential Processes*. Englewood Cliffs: Prentice-Hall

Hofkamp, A.T. 2001: *Python from* χ. Note, Systems Engineering group, Eindhoven University of Technology, http://se.wtb.tue.nl

Hofkamp, A.T.; Rooda, J.E. 2002a: χ *Reference manual*. http://se.wtb.tue.nl, Systems Engineering group, Eindhoven University of Technology

Hofkamp, A.T.; Rooda, J.E. 2002b: *Chi tool set reference manual*. http://se.wtb.tue.nl, Systems Engineering group, Eindhoven University of Technology

Hulshof, M.F. 2003: *Analytical target cascading: numerical convergence evaluation and manufacturing system application.* MSc thesis report SE-420338, Systems Engineering group, Eindhoven University of Technology

Kim, H.M. 2001: *Target Cascading in Optimal System Design*, Dissertation, The University of Michigan, Ann Arbor, Michigan

Kim, H.M.; Kokkolaras, M.; Louca, L.S.; Delagrammatikas, G.J.; Michelena, N.F.; Filipi, Z.S.; Papalambros, P.Y.; Stein, J.L.; Assanis, D.N. 2002: Target cascading in vehicle redesign: A class VI truck study. *Int. J. Veh. Des.* **29**, 1–27

Kim, H.M.; Michelena, N.F.; Papalambros, P.Y.; Jiang, T. 2003: Target cascading in optimal system design. *J. Mech. Des.* **125**, 474–480

Kodiyalam, S.; Sobieszczanski-Sobieski, J. 2000: Bilevel integrated system synthesis with response surfaces. *AIAA J.* **38**(8), 1479–1485

Kokkolaras, M.; Fellini, R.; Kim, H.M.; Michelena, N.F.; Papalambros, P.Y. 2002: Extension of the target cascading formulation to the design of product families. *Struct. Multidisc. Optim.* **24**, 293–301

Liu, B.; Haftka, R.T.; Watson, L.T. 2004: Global-local structural optimization using response surfaces of local optimization margins. *Struct. Multidisc. Optim.*, in press

Lutz, M.; Ascher, D. 1999: *Learning Python*. Cambridge: O'Reilly

Michelena, N.F.; Kokkolaras, M.; Louca, L.S.; Lin, C.C.; Jung, D.; Filipi, Z.S.; Assanis, D.N.; Papalambros, P.Y.; Peng, H.; Stein, J.L.; Feury, M. 2001: Design of an advanced heavy tactical truck: a target cascading case study. *Proc. SAE International Truck & Bus Meeting and Exhibition*, Chicago, IL, paper no. 2001-01-2793

Michelena, N.; Park, H.; Papalambros, P.Y. 2003: Convergence properties of analytical target cascading. *AIAA J.* **41**, 897–905

Papadopoulos, G.A.; Arbab, F. 1998: *Coordination Models and Languages*. CWI Software Engineering report SEN-R9834, National Research Institute for Mathematics and Computer Science, http://www.cwi.nl, Amsterdam, the Netherlands

Python 2004: http://www.python.org

Roscoe, A.W. 1997: *The Theory and Practice of Concurrency*. London: Prentice-Hall

Sobieski, I.P.; Kroo, I.M., 2000: Collaborative optimization using response surface estimation. *AIAA J.* **38**, 1931–1938

Tzevelekos, N.; Kokkolaras, M.; Papalambros, P.Y.; Hulshof, M.F.; Etman, L.F.P.; Rooda, J.E. 2003: An empirical local convergence study of alternative coordination schemes in analytical target cascading. *5th World Congress of Structural and Multidisciplinary Optimization*, Venice, 19–23 May 2003 (CD-ROM)

Van Beek, D.A.; Rooda, J.E. 2000: Languages and applications in hybrid modelling and simulation: the positioning of Chi. *Contr. Eng. Pract.* **8**, 81–91

Van de Mortel-Fronczak, J.M.; Rooda, J.E.; Van den Nieuwelaar, N.J.M. 1995: Specification of a flexible manufacturing system using concurrent programming. *Concurrent Eng.-Res. A.* **3**, 187–194

Vervoort, J.; Rooda. J.E. 2003: *Learning* χ. Systems Engineering Group, Eindhoven University of Technology, http://se.wtb.tue.nl