9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization
4-6 September 2002, Atlanta, Georgia

AIAA-2002-5410
AIAA 2002-5410

COORDINATION SPECIFICATION FOR DISTRIBUTED OPTIMAL SYSTEM DESIGN USING THE χ LANGUAGE

L.F.P. Etman, A.T. Hofkamp, and J.E. Rooda
Eindhoven University of Technology, Eindhoven, The Netherlands
and
M. Kokkolaras and P.Y. Papalambros
University of Michigan, Ann Arbor, Michigan

ABSTRACT

Coordination plays a key role in solving decomposed optimal system design problems. Several coordination strategies have been proposed in the multidisciplinary optimization (MDO) literature. They are usually presented as a sequence of statements: the parallel nature of the multidisciplinary subproblems is often either not addressed or only briefly mentioned. However, a more formal description of the concurrency in the coordination is essential, in particular for large and non-hierarchic coordination architectures. This paper proposes to use concepts from communicating sequential processes (CSP) developed in concurrency theory. CSP allows the description of the MDO coordination as a number of parallel processes that operate independently and communicate with each other synchronously over pre-defined channels. For this purpose, we introduce elements of the language χ, a CSP-based language that contains data types such as reals, arrays, lists and tuples. The accompanying software tool set that enables the execution of a χ specification has been extended with a Python interface so that function calls to external software can be carried out. Through this interface, χ has been coupled with Matlab to run coordination specifications of distributed optimal system design problems on single or on multiple parallel computers. An optimal design example is used to illustrate this. It can be concluded that the use of a CSP-based language such as χ for coordinating the solution of MDO problems is quite promising.

1 INTRODUCTION

During the last two decades several multidisciplinary optimization (MDO) approaches have been developed to deal with design optimization problems that involve more than one engineering discipline. Typical applications are large-scale and complex engineering systems such as aerospace and automotive vehicles. Decomposition methods have been developed for partitioning large optimal design problems into collections of smaller and more tractable subproblems, each associated with a, possibly different, discipline. Generally speaking, a discipline does not need to be a "true" discipline, but can be any 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 of solving the subproblems concurrently. The difficulty in such distributed optimal design problems lies in the coupling between the individual disciplines (components). System constraints may depend on design variables 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 to ensure convergence to a solution that is consistent with respect to the system design problem.

Finding the best coordination strategy for a given decomposed system is an open research topic. Coordination strategies have been proposed based on some simplified structure in the coupling of the subsystems. Two types of coordination methods can be identified: methods that originate from rigorous mathematical formulations, and methods that originate from MDO approaches in the engineering sciences which are often more heuristic. Mathematical decomposition methods generally start from one large set of analytical constraint equations, and exploit the structure in this set of equations to obtain a system partitioning that can be well and efficiently coordinated. Engineering-based MDO methods typically start from discipline analysis models that cannot be represented by analytical equations in the system optimization problem. The analysis models are assumed to be "black-boxes". This means that the partitioning structure in engineering-based MDO methods is imposed by aspect-driven or component-driven decomposition.

The coordination strategy has to be defined such that it works best for the given system partition. In this regard, having a compact and unambiguous language to specify and formulate the distributed optimal design

problem and a straightforward way to solve it, e.g., on a cluster of parallel computers, would be quite helpful. A suitable programming language for distributed optimization problems is necessary for this purpose. The mathematical programming languages that are available (e.g. AMPL [1]), are typically geared towards the purpose of formulating numerical optimization problems. In computer science, several languages have evolved to describe the coordination between concurrent (computational) processes. These coordination languages can be classified into data-driven and control-driven languages [2]. A data-driven language coordinates data through a shared dataspace, while a control-driven or process-oriented language treats processes as "black-boxes" that are coordinated through exchange of state values or through broadcast of control messages. In MDO, a process-oriented coordination language is needed to deal with the characteristic "black-box" behavior of most engineering-based MDO approaches.

Communicating Sequential Processes (CSP) is a commonly used theoretical foundation of process-oriented coordination languages [3, 4]. CSP is particularly attractive for modeling the concurrent but coupled disciplines in MDO, provided that suitable data language elements are available to deal with the simulation-based numerical optimization setting. This paper aims to utilize CSP concepts for coordination specification in distributed optimal system design. We adopt the CSP-based language χ [5, 6], which includes data types required for numerical optimization, such as reals and arrays. The χ language was developed originally to model and simulate discrete-event and hybrid manufacturing systems (which combine discrete-event and continuous-time behavior) [7]. It is designed primarily for modeling purposes: it is easy to understand, and has only few language constructs. The discrete-event part of χ is used in this work. The χ software has been extended with a Python interface [8, 9, 10] that allows individual processes to carry out function calls to external software packages. This functionality enables to use χ for MDO problems, and is used here to couple χ with Matlab [11].

The paper is organized as follows. An overview of MDO coordination architectures is given and placed in a CSP-perspective. Several basic χ language elements are introduced to specify the parallel processes and their interactions in the context of distributed optimal system design. A simple analytical design optimization example is presented for illustration purposes. Based on the same example, we also demonstrate how surrogate modeling may be included for simulation-based MDO problems. Finally, the main findings are reviewed and discussed.

2 MDO COORDINATION

2.1 Optimization problem formulation

The general MDO problem can be stated as follows: Find the discipline design variables $\mathbf{x} = [\mathbf{x}_1, \ldots, \mathbf{x}_m]$ such that system objective f is minimized subject to system and disciplinary design constraints $\mathbf{g} = [\mathbf{g}_0, \mathbf{g}_1, \ldots, \mathbf{g}_m]$, interdisciplinary design variable coupling constraints $\mathbf{k}$, interdisciplinary response variable coupling constraints $\mathbf{l}$, and analysis equations $\mathbf{a} = [\mathbf{a}_1, \ldots, \mathbf{a}_m]$, where m is the number of disciplines.

A set of design variables $\mathbf{x}_i$ is identified for each discipline i. Some of the discipline design variables may be shared in several disciplines, which is represented by interdisciplinary design variable coupling equations $\mathbf{k}$. Discipline constraints $\mathbf{g}_i$ only depend on discipline design variables $\mathbf{x}_i$ and discipline responses $\mathbf{r}_i$. System constraints $\mathbf{g}_0$ depend on the design variables and analysis responses of the disciplines. The same holds for system objective f.

The interdisciplinary response variables coupling is represented by equations $\mathbf{l}$. These equations relate coupled input $\mathbf{c}_i$ of discipline i to response output $\mathbf{r}_j$ of discipline j, $j \neq i$. The discipline responses $\mathbf{r} = [\mathbf{r}_1, \ldots, \mathbf{r}_m]$ are computed from analysis equations $\mathbf{a}$. We assume in this paper that the analysis equations are not explicit; they are treated as "black-boxes" or external routines, implying that $\mathbf{r}_i$ of discipline i is computed as function of $\mathbf{x}_i$ and $\mathbf{c}_i$.

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

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

where $\mathcal{X}_i$ denote the discipline set constraints [12].

2.2 Classification of MDO architectures

In literature various coordination strategies for engineering-based MDO problems can be found. Classifications of these strategies are presented in the review papers of Cramer *et al.* [13], Balling and Sobieszczanski-Sobieski [14], and Alexandrov and Lewis [15]. The key element of these classifications is the way feasibility of the different sets of equations in problem (1) is maintained.

Cramer *et al.* [13] distinguish between multidisciplinary feasible and individual discipline feasible decomposition. Multidisciplinary feasible decomposition requires that the interdisciplinary response vari-

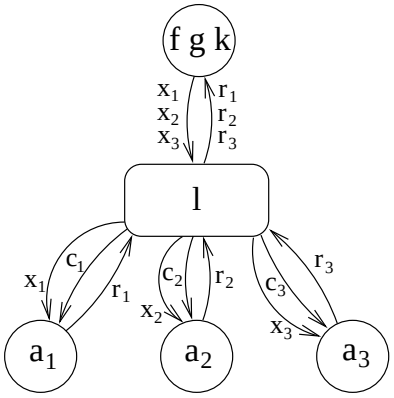

Figure 1: Multidisciplinary feasible decomposition

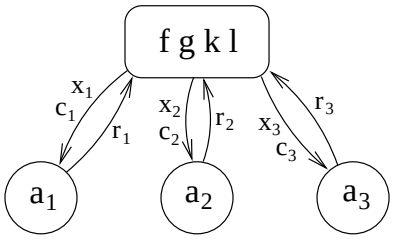

Figure 2: Individual discipline feasible decomposition

able coupling equations **l** are always satisfied at each iteration of the optimization. This means that an evaluation of a system design **x** to obtain responses **r** automatically implies that the coupled input and output of the disciplinary black-box analyses $\mathbf{a}_i$, $i = 1, \ldots, m$, match. The response variable coupling equations are called to appear *nested* [14] or *closed* [15] with respect to the system optimization problem. The black-box numerical analyses again appear nested with respect to the response variable linking. This is visualized in Figure 1 . In the individual discipline feasible approach, the response variable coupling equations do not appear nested and are included in the system optimization problem instead. As a consequence, the system will be interdisciplinary feasible (i.e. satisfy the coupling constraints) only after convergence has been achieved. The individual discipline feasible approach is depicted in Figure 2 .

In addition, Balling and Sobieszczanski-Sobieski [14] distinguish between single-level and multi-level decomposition architectures. Single-level refers to an architecture where only the system optimization problem determines the design variable values. In the multi-level case disciplinary optimizers are introduced to determine the independent discipline design variables, and a system optimizer to determine the shared design variables (coupling equations **l** are used for this partitioning of the design variables). From the system optimizer point of view the disciplinary constraints will always be satisfied (closed design constraints $\mathbf{g}_i (i = 1, \ldots, m)$ [15]). A popular hierarchical architecture from this category is shown in Figure 3, where variables $\mathbf{x}_i^t$ and $\mathbf{r}_i^t$ represent targets for design variables and responses provided by the system optimizer, respectively, while $\mathbf{x}_i^r$ and $\mathbf{r}_i^r$ represent the values that are returned by the discipline optimizers.

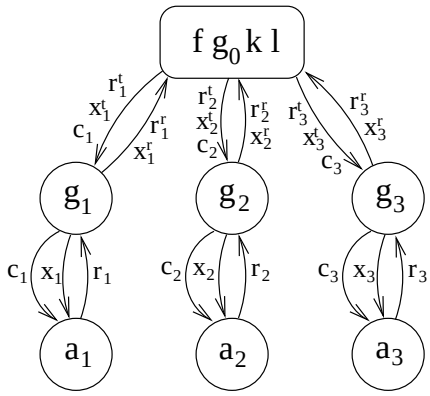

Figure 3: Multi-level hierarchical decomposition

An appropriate coordination strategy has to be specified for a defined decomposition architecture. This strategy has to specify *how* the coordination of the different subsystems in the decomposition will be carried out. Coordination algorithms are usually presented in the MDO literature as some step-wise procedure, sometimes visualized by a flow diagram. However, there is a discrepancy between the step-wise (sequential) description and the concurrency that is inherently present in the coordination of the decomposed system. Concurrency that is not precisely specified may cause serious confusion about how it should be actually implemented, especially when the coordination architecture grows in size (more levels, disciplines, components) or becomes more complex, e.g., non-hierarchical. Serious flaws such as deadlock may arise.

2.3 A CSP view on coordination in MDO

Several theoretical foundations are available to describe systems that exhibit concurrent behavior. A short overview is given in the introduction of [16]. Communicating Sequential Processes (CSP) concepts [17, 3, 4] are highly suited to specify coordination in MDO; they describe the coordination as a number of parallel processes that operate independently and communicate synchronously over predefined channels. The CSP view matches well with the engineering-based MDO methods to partition optimal system design problems into multiple smaller independent subproblems. Each subproblem can be seen as a "black-box" process, while the data exchange between the subproblems to arrive at a consistent coupling can be seen as communications between the processes.

Figures 1, 2, and 3 can be explained in terms of CSP. Each circle represents a process that is responsible for the closure of a specific set of constraints. Such a process may perform its operations in parallel to the other processes unless it needs some input from another process before it can proceed. Such a communication relation is modeled as a channel between the two processes, visualized by an arrow showing the direction of the communication. For example, the individual feasible decomposition is represented by four processes in Figure 2: three analysis models $\mathbf{a}_1$, $\mathbf{a}_2$, $\mathbf{a}_3$, and the system optimization process involving f, **g**, **k**, and **l**. The communications between these processes are that the system optimization process f**gkl** requires analysis processes $\mathbf{a}_1$, $\mathbf{a}_2$, and $\mathbf{a}_3$ to run the simulation models for design vectors $(\mathbf{x}_i, \mathbf{c}_i)$ and return the corresponding response values $\mathbf{r}_i$.

We believe that coordination in MDO would benefit from the use of a specification language based on elements of concurrent programming. Such a coordination language has to be able to define precisely both the contents of the separate MDO processes (typically related to numerical analysis or optimization), as well as the communication behavior of the parallel processes to solve the entire system problem. Preferably, the language should allow formal analysis to prove that the concurrency in the coordination is correctly specified and does not show failures such as deadlock. A CSP-based language fits the engineering-based MDO methods. This is demonstrated using the χ language developed by the Systems Engineering group at Eindhoven University of Technology.

3 Coordination specification using χ

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 [7]. We will show that the discrete-event part of χ is well suited to formalize and enhance coordination specification in distributed optimal system design. 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 [16].

3.1 The χ language and compiler

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 the language is based on CSP [17], and uses Dijkstra's guarded commands [18]. For readability of the specification, parallel behavior is restricted to occur between processes and between processes and systems (following [19]). Systems may contain several coupled processes that operate in parallel. Individual processes are specified in an imperative way using a sequence of statements. Functions can be defined to obtain compact and clear process specifications. Interactions between processes (systems) are modeled as synchronous communication over channels [3]. Synchronous means that communication between two processes takes place only when both are willing to communicate and then the communication takes place instantly (no storage in the channel). The concept of time and probability functions, which play a key role in modeling manufacturing systems, are contained in χ. At present, these do not play a role in the *coordination* of MDO formulations, and are therefore left out of further consideration.

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 *channel* represents a connection between two processes enabling interaction.
- *Interaction* (communication) between two processes means data exchange (through a point-to-point channel).

A compiler has been built to translate χ specifications into C++ code. χ is a so-called strong type language, implying that the type of a variable is known at compile time of the C++ code. Compilation of the generated code yields an executable program that can be run on a computer. In the MDO context, this means that one can run the coordination, provided that χ is able to carry out the necessary numerical analysis and optimization calculations. This is realized by means of a Python function interface as explained in Section 3.6.

3.2 Data types

MDO coordination formulations usually generate considerable amounts of (numerical) data that have to be passed around among the different subsystems. Therefore, a coordination language for MDO has to provide language constructs to model this data flow compactly. The χ language satisfies this data modeling requirement; it has several built-in data types, both basic and generic. 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. The generic data

types are: array, tuple, list, and set. These four generic data types are briefly explained below, where T denotes a data type that is either basic or generic. Details can be found in [5, 16].

T^n is an array of fixed length n containing data elements of type T. An example of an array of type real^3 is $\langle 2.1, 4.8, -4.9 \rangle$. Arrays can be built from any basic or generic 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 which 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 a 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 generic data types and can be accessed by the index.

T^* is a list containing an ordered sequence of elements which must 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.

T^+ denotes a set. The difference between a set and a list is that the ordering of the elements is not relevant in a set. Similar to the list, all elements of a set need to be of the same type (basic or generic). Each element can occur only once: the set $\{6,7\}$ equals the set $\{7,6\}$. An empty set is $\{\}$. Operators are available to add elements, to remove elements from the set, or to test whether a given element is present in the set.

Numerical analysis and optimization output of disciplines will usually be generated in the form of arrays or tuples of arrays. Lists and sets are suitable to temporarily store data that are needed in later iterations.

3.3 Processes

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

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

The process is identified by its name N and parameters V_p, the latter represented by a comma-separated list of (formal) parameters of the form v: *type*, where *type* can be a standard data type T, a send port (v: $!T$) data type, a receive port (v: $?T$) data type, or a synchronization port (v: $\sim$void) data type. The body of the process is specified between the brackets $[\![$ and $]\!]$. Local programming variables V_l are declared first, followed by the (sequence of) statements S_p that are executed by the process.

Table 1: Syntax of χ process statements.

$$
\begin{array}{lll}
S_p ::= & \mathsf{skip} & (\mathsf{skip}) \\
\mid & x := e & (\text{assignment}) \\
\mid & E & (\text{event}) \\
\mid & S_p\,;\, S_p & (\text{sequential composition}) \\
\mid & [\ GC\ ] & (\text{guarded command}) \\
\mid & *[\ GC\ ] & (\text{repetitive guarded command}) \\
\mid & [\ SW\ ] & (\text{selective waiting}) \\
\mid & *[\ SW\ ] & (\text{repetitive selective waiting}) \\
E ::= & p\,!\,e & (\text{send}) \\
\mid & p\,?\,x & (\text{receive}) \\
\mid & p\,! & (\text{synchronization send}) \\
\mid & p\,? & (\text{synchronization receive}) \\
\mid & p^{\sim} & (\text{directionless synchronization}),
\end{array}
$$

$$
\begin{array}{lll}
GC ::= e_b \longrightarrow S_p & & SW ::= e_b\,;\, E \longrightarrow S_p \\
\mid R\colon e_b \longrightarrow S_p & & \mid R\colon e_b\,;\, E \longrightarrow S_p \\
\mid GC \,[\!]\, GC & & \mid SW \,[\!]\, SW
\end{array}
$$

$$
\begin{array}{lll}
R ::= i\colon \text{nat} \leftarrow l..u & & (\text{range including } l, \text{excluding } u) \\
\mid R, R & & (\text{range list}).
\end{array}
$$

Table 1 presents in BNF format [20] the syntax of a subset of χ process statements that is relevant for specifying the coordination in MDO. The statements are explained informally below. Further details can be found in [5, 16].

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. This requires the types of x and e to be the same.

$S_{p_1}\,;\, S_{p_2}$ denotes that statement S_{p_2} is executed after the execution of statement S_{p_1} 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 ($p\,!$),

the synchronization receive statement ($p?$), and the directionless synchronization statement ($p^{\sim}$). 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 compatible types. Synchronization is a communication statement without transferring data, which may or may not have direction. It is commonly 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 alternatives 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 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 to define iterator variables that are varied within certain lower and upper bounds.

The key statements to specify the communication of a process with other concurrent processes in the coordination 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 more complex, e.g., non-hierarchical, coordination schemes.

3.4 Systems

Processes can be grouped together in a system by means of parallel composition. The processes in the system are coupled through channels and executed concurrently. Such a system can again act as a process and can be combined with other processes to form a new system. A χ system has the following form:

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

A system is identified by its name N and parameters V_s. Similar to the process definition, V_s is a comma separated list of (formal) parameters of the form v: *type*, which is either a standard basic or generic data type T, a send port (v: $!\,T$) data type, a receive port (v: $?\,T$) data type, or a synchronization port (v: $^{\sim}$void) data type. The system body resides between brackets $[\![$ and $]\!]$, 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: $-\,T$. 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 $S_s \,||\, S_s$. The local channel variables are used to connect different process instantiations to each other. A single channel connects either one send port to one receive port or two directionless synchronization ports to each other. The data types of the ports and the channel must match.

A closed system has to be instantiated at the top level. This closed system has no communication ports parameters. Only some standard data typed parameters may still remain in the parameter list. The environment

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

instantiates the top level system N with parameter values e_1 to e_n.

3.5 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 carried out each time the function is called by name in a process statement. A χ function is defined as

$$\mathsf{func}\ N(V_f) \longrightarrow T_r = [\![\, V_l \mid S_f \,]\!],$$

and identified by its name N and a list of formal input parameters V_f of type (v: T). The return type of the function is T_r. Both T and T_r are basic or generic data types as defined in Section 3.2. The body of the function is defined after the equality sign. Local programming variables (x: 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 Section 3.3. One new statement is the return statement $\uparrow e$ that ends 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. Communication statements are therefore not allowed in functions.

3.6 Application to MDO

An MDO coordination specified in χ is represented by processes that communicate through channels. The numerical computations carried out by the individual processes are modeled as functions. Generally, these calculations require routines external to χ, which means that a χ function has to be able to call external software. This has been realized by allowing functions written in Python [8] to be treated like functions written in native χ. 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.

With the concurrency formally specified, it would be highly advantageous if the coordination could be implemented on parallel computers. The difficulty is that χ 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 dealt with together). The scheduler of χ determines which statement in which process will be executed next. This implies that function calls to external numerical routines will be carried out one at a time even if they occur in parallel processes. To allow parallel execution, each external function call has to be split into a four step procedure: start, notification of this start, clearance to proceed , and retrieval of results. Processes get clearance to proceed through a synchronization with a synchronizer process when they have all started their jobs. This is illustrated in Section 4.3. We are currently investigating whether this approach can be replaced by a more elegant one. The generated series of jobs are queued and distributed over the available parallel computer resources.

4 Geometric programming problem

4.1 Optimization problem formulation

Consider test problem 104 of [21]:

$$\begin{aligned}
&\text{Find} \mathbf{x} = x_1, \ldots, x_8 \\
&\text{Min } f = f_1 + f_2 + f_3 + f_4 \\
&\quad f_1 = 0.4 x_1^{0.67} x_7^{-0.67} \\
&\quad f_2 = 0.4 x_2^{0.67} x_8^{-0.67} \\
&\quad f_3 = 10 - x_1 \\
&\quad f_4 = x_2 \\
&\text{s.t. } g_1 = 0.1 x_1 + 0.0588 x_5 x_7 - 1.0 \le 0 \\
&\quad g_2 = 0.1 x_1 + 0.1 x_2 + 0.0588 x_6 x_8 - 1.0 \le 0 \\
&\quad g_3 = 4 x_3 x_5^{-1} + 2 x_3^{-0.71} x_5^{-1} + 0.0588 x_3^{-1.3} x_7 \\
&\qquad\qquad - 1.0 \le 0 \\
&\quad g_4 = 4 x_4 x_6^{-1} + 2 x_4^{-0.71} x_6^{-1} + 0.0588 x_4^{-1.3} x_8 \\
&\qquad\qquad - 1.0 \le 0 \\
&\quad g_{i+4} = 0.1 - x_i \le 0 \qquad i = 1, \ldots, 8 \\
&\quad g_{i+12} = x_i - 10.0 \le 0 \qquad i = 1, \ldots, 8
\end{aligned} \tag{2}$$

Wagner [22] partitioned this single optimization problem into two subproblems $\mathrm{SP_a}$ and $\mathrm{SP_b}$ with one linking variable $y = x_1$ using a hyper-graph based partitioning technique. He used Wismer and Chattergy's relaxation strategy for the coordination of the two sub-

problems. The first subproblem of Wagner's partitioning is

$$
\begin{aligned}
&\text{Problem SP}_\text{a}\\
&\text{Receive } y\\
&\text{Find} \quad \mathbf{x}_\text{a} = [x_2, x_4, x_6, x_8]\\
&\text{Min} \quad f_\text{a} = f_2 + f_4\\
&\qquad f_2 = 0.4 x_2^{0.67} x_8^{-0.67}\\
&\qquad f_4 = x_2\\
&\text{s.t.} \quad g_2 = 0.1y + 0.1x_2 + 0.0588 x_6 x_8 - 1.0 \le 0\\
&\qquad g_4 = 4x_4 x_6^{-1} + 2x_4^{-0.71} x_6^{-1} + 0.0588 x_4^{-1.3} x_8 - 1.0 \le 0\\
&\qquad g_{i+4} = 0.1 - x_i \le 0 \qquad i = 2,4,6,8\\
&\qquad g_{i+12} = x_i - 10.0 \le 0 \qquad i = 2,4,6,8\\
&\text{Return} \quad \mathbf{x}_\text{a}^*, \mu_2
\end{aligned}
\tag{3}
$$

Solution of subproblem SP_a yields, for a given (fixed) linking value y, the optimum design variable values $\mathbf{x}_\text{a}^*$ and the Lagrange multiplier values of the (active) constraints at $\mathbf{x}_\text{a}^*$. The coordination scheme requires the subproblem optimum values $\mathbf{x}_\text{a}^*$ and Lagrange multiplier value μ_2 of constraint g_2 to be returned. The second subproblem is

$$
\begin{aligned}
&\text{Problem SP}_\text{b}\\
&\text{Receive } y, \lambda\\
&\text{Find} \quad \mathbf{x}_\text{b} = [x_1, x_3, x_5, x_7]\\
&\text{Min} \quad f_\text{b} = f_1 + f_3 + \lambda(x_1 - y)\\
&\qquad f_1 = 0.4 x_1^{0.67} x_7^{-0.67}\\
&\qquad f_3 = 10 - x_1\\
&\text{S.t.} \quad g_1 = 0.1x_1 + 0.0588 x_5 x_7 - 1.0 \le 0\\
&\qquad g_3 = 4x_3 x_5^{-1} + 2x_3^{-0.71} x_5^{-1} + 0.0588 x_3^{-1.3} x_7 - 1.0 \le 0\\
&\qquad g_{i+4} = 0.1 - x_i \le 0 \qquad i = 1,3,5,7\\
&\qquad g_{i+12} = x_i - 10.0 \le 0 \qquad i = 1,3,5,7\\
&\text{Return} \quad \mathbf{x}_\text{b}^*
\end{aligned}
\tag{4}
$$

Parameter λ is related herein to the Lagrange multiplier μ_2 of subproblem SP_a, and its value is provided by the coordination strategy. Subproblem optimum $\mathbf{x}_\text{b}^*$ is returned to the coordination scheme.

Wagner uses the following coordination strategy to solve this decomposed optimal design problem:

1. Initialize:
   (a) $k := 0$
   (b) $\mathbf{x}^{(k)} := [0.1, 0.1, 0.2, 0.2, 10.0, 10.0, 0.1, 0.1]$
   (c) $y^k := x_1^{(k)}$
   (d) $\lambda^{(k)} := 0.0$
2. Solve SP_a and SP_b to obtain $\mathbf{x}^{*(k)}$ and $\mu_2^{(k)}$
3. Update master problem:
   (a) $k := k + 1$
   (b) $\mathbf{x}^{(k)} := \mathbf{x}^{*(k-1)}$
   (c) $y^{(k)} := x_1^{*(k-1)}$
   (d) $\lambda^{(k)} := \mu_2^{(k-1)} \left[\frac{\partial g_2}{\partial y}\right]_{\mathbf{x}=\mathbf{x}^{*(k-1)}}$
4. Check convergence: if $\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\| < \varepsilon$, stop; otherwise goto 2.

The coordination strategy feeds the subproblem optimizations with updated values of y and λ at each iteration. Since $\left[\frac{\partial g_2}{\partial y}\right]_{\mathbf{x}=\mathbf{x}^{*(k-1)}}$ is a constant (equal to 0.1), it does not need to be provided by subproblem SP_a. The iterative coordination process stops if the norm of the difference between the current design variable vector and the vector of the previous iteration is smaller than some prescribed value ε.

4.2 Coordination specification using χ

Three processes are defined: processes SP_a and SP_b are related to the subproblem optimizations; process C defines the coordination strategy. They are coupled as shown in Figure 4. Process C sends through channels

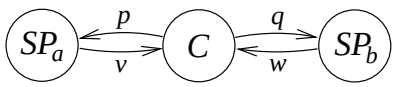

Figure 4: Processes and channels in decomposed geometric programming problem.

p and q updated values of design variables and linking variables to subproblem processes SP_a and SP_b. Once SP_a and SP_b receive new values, both optimization subproblems are solved, and the results are sent back to process C through channels v and w. The master problem is then updated, and, if convergence has not been achieved, new values are sent to SP_a and SP_b.

The χ specification is explained next. Two variable types are defined for shorthand notation in the following specification:

$$
\begin{aligned}
&\text{type vecx} = \text{real}^4\\
&\text{,} \quad \text{vecg} = \text{real}^{10},
\end{aligned}
$$

where real^4 denotes a vector array of 4 reals. The types vecx and vecg suffice since each subproblem has 4 design variables and 10 constraints.

Functions used in the process specifications are defined next. Three functions can be identified: *norm*, *optspa*, and *optspb*. The function *norm* takes as input

four vectors of type vecx and computes the l_2-norm of the difference between initial and optimum design:

```
func norm(x_a^o, x_b^o, x_a^*, x_b^*: vecx) ⟶ real =
[[ i: nat, s: real
| i := 0; s := 0.0
; *[ i < 4
     ⟶ s := s + (x_a^*.i − x_a^o.i)^2 + (x_b^*.i − x_b^o.i)^2
       ; i := i + 1
   ]
; ↑√s
]] .
```

The local variable i is a counter in the repetitive guarded command statement $*[\ i < 4 \longrightarrow \ldots\ ]$. The statements on the right hand side of the arrow are repeated as long as $i < 4$. If i becomes 4, the repetition statement is finished, and as final statement, the square root of s is returned.

The functions *optspa* and *optspb* have to carry out an optimization run for subproblems $\mathrm{SP_a}$ and $\mathrm{SP_b}$, respectively. Given initial design $\mathbf{x}^{\mathrm{o}}$ and linking variable value c, *optspa* solves the optimization problem $\mathrm{SP_a}$ and returns optimal design variable and Lagrange multiplier values. The type of the result is vecx × vecg, i.e., a tuple of a vector array of length 4 and a vector array of length 10. Function *optspb* is defined similarly:

```
func optspa(x^o: vecx, c: real) ⟶ vecx × vecg
func optspb(x^o: vecx, c1, c2: real) ⟶ vecx.
```

Both functions call the Matlab optimization toolbox routine *fmincon* [11], which is an implementation of the sequential quadratic programming optimization algorithm, through the Python-Matlab interface [8].

Process SP_a represents the optimization of subproblem $\mathrm{SP_a}$, and is specified as

```
proc SP_a(a: ?vecx × real, b: !vecx × real) =
[[ x_a^o, x_a^*: vecx, μ_a: vecg, y: real
| *[ true
     ⟶ a?⟨x_a^o, y⟩
       ; ⟨x_a^*, μ_a⟩ := optspa(x_a^o, y)
       ; b!⟨x_a^*, μ_a.0⟩
   ]
]] .
```

Process SP_a receives and sends information in the form of a tuple vecx × real via ports a and b (*formal* port names). It uses local variables $\mathbf{x}_{\mathrm{a}}^{\mathrm{o}}$, $\mathbf{x}_{\mathrm{a}}^{*}$ of type vecx, μ_{a} of type vecg, and y of type real. Note that $\mu_{\mathrm{a}}.0$ is the first element of array μ_{a} which corresponds with μ_2 in Equation (3). The process specification of SP_a is defined to repeat the following statements indefinitely:

- Wait until data is received via port a and assign the received values to $\mathbf{x}_{\mathrm{a}}^{\mathrm{o}}$ and y;
- Carry out the subproblem optimization by calling *optspa* using $\mathbf{x}_{\mathrm{a}}^{\mathrm{o}}$ as initial design and y as linking parameter value, and return the optimal subsystem design together with Lagrange multiplier values assigned to $\mathbf{x}_{\mathrm{a}}^{*}$ and μ_{a}, respectively;
- Try to send the optimal subsystem design and the Lagrange multiplier value of constraint g_2 (statement is finished if sending action has succeeded).

Process SP_b has a similar structure:

```
proc SP_b(a: ?vecx × real × real, b: !vecx) =
[[ x_b^o, x_b^*: vecx, y, λ: real
| *[ true
     ⟶ a?⟨x_b^o, y, λ⟩
       ; x_b^* := optspb(x_b^o, y, λ)
       ; b!x_b^*
   ]
]] .
```

The difference between SP_a and SP_b lies in the format of data that is received and sent and in the function call of *optspb* that represents the optimization of subproblem $\mathrm{SP_b}$.

Process C represents the coordination strategy and controls the iterative subproblem optimizations. The χ specification of process C is

```
proc C(a: !vecx × real, b: !vecx × real × real
       , c: ?vecx × real, d: ?vecx) =
[[ x_a^o, x_a, x_b^o, x_b: vecx, y, λ, μ_2: real
, k: nat, end: bool
| x_a := ⟨0.1, 0.2, 10.0, 0.1⟩
; x_b := ⟨0.1, 0.2, 10.0, 0.1⟩
; y := x_a.0; λ := 0.0; k := 0; end := false
; *[¬ end
     ⟶ k := k + 1; x_a^o := x_a; x_b^o := x_b
       ; a!⟨x_a^o, y⟩; b!⟨x_b^o, y, λ⟩
       ; [ true; c?⟨x_a, μ_2⟩ ⟶ d?x_b
         [] true; d?x_b       ⟶ c?⟨x_a, μ_2⟩
         ]
       ; y := x_a.0; λ := 0.1μ_2
       ; end := norm(x_a^o, x_b^o, x_a, x_b) < 10^−3
   ]
]] .
```

Process C sends data to processes SP_a and SP_b through ports a and b, and receives data through ports c and d. Once the local variables are declared, the first lines of the specification represent the coordination initialization. A number of statements is then repeated until *end* becomes true ($\neg$ denotes "not"). Process C sends the required data to SP_a and SP_b to carry out their respective subproblem optimizations, and waits for their response. A selective waiting statement is used to express that C does not know beforehand which of the two processes will reply first, SP_a or SP_b. If C first receives data via port c, then it waits until the data of the other subproblem is received via port d, and proceeds with the next statement. If communication is

first established via port d, then C waits for communication via c, and proceeds. Note that for this specific example the selective waiting statement can be replaced by simply a sequence of two receive statements: $c?\langle \mathbf{x}_a, \mu_2 \rangle$; $d?\mathbf{x}_b$. Even if the communication via d would be possible before c, the total execution time is the same. Selective waiting and repetitive selective waiting are very powerful language constructs to model communications between several parallel processes of more complicated coordination schemes.

Finally, processes SP_a, SP_b, and C have to be coupled at the system level, as shown in Figure 4. Process SP_a receives data of type vecx $\times$ real via channel p from C, and sends data of the same type back via channel v. Process SP_b receives data of type vecx $\times$ real $\times$ real via channel q from C, and sends data of type vecx back again via channel w. Accordingly, process C sends data of type vecx $\times$ real through channel p to SP_a and data of type vecx $\times$ real $\times$ real through channel q to SP_b, and receives data of type vecx $\times$ real via channel v from SP_a and data of type vecx via channel w from SP_b. This is specified as

$$
\begin{array}{l}
\mathsf{syst}\ S() = \\
[\![\ p,\ v\colon -\mathrm{vecx} \times \mathrm{real},\ q\colon -\mathrm{vecx} \times \mathrm{real} \times \mathrm{real} \\
,\ w\colon -\mathrm{vecx} \\
\mid C(p,\ q,\ v,\ w) \parallel SP_b(q,\ w) \parallel SP_a(p,\ v) \\
]\!]\,.
\end{array}
$$

Declaration $p\colon -\mathrm{vecx} \times \mathrm{real}$ means that p is a channel of type vecx $\times$ real. The χ specification ends with

$$\mathsf{xper} = [\![\ S()\ ]\!],$$

denoting that a system execution can be carried out. The optimum $\mathbf{x}^{*(k)} = [x_1, \ldots, x_8] = [6.46\ 2.23\ 0.667\ 0.596\ 5.93\ 5.53\ 1.01\ 0.401]$ was obtained after $k = 26$ system iterations by implementing the χ coordination as described. This solution corresponds with the one reported in [22].

4.3 Inclusion of RSM metamodels

Simulation-based MDO applications frequently require a lot of expensive computations. MDO methodologies that employ response surface approximations have been proposed to address this challenging issue. It should be emphasized that here we do not refer to using metamodels for evaluating expensive functions, but to representing response surfaces that approximate the solutions of simulation-based MDO problems, as in, e.g., [23, 24]. The χ language can also be used to specify the coordination of MDO problems solved by means of response surface (RSM) approximations. This is demonstrated by implementing such a methodology in the coordination of the geometric programming example. We realize that the example in itself is too simple to justify the use of response surfaces. It is used to illustrate how RSM metamodel building can be included in the χ specification of the coordination.

The following strategy is implemented to coordinate the solution of the geometric programming problem by means of response surface approximations:

1. Initialize:
   (a) $k := 0$, $p := 0.5$, $N := 6$
   (b) $\mathbf{x}^{(k)} := [0.1, 0.1, 0.2, 0.2, 10.0, 10.0, 0.1, 0.1]$
   (c) $y^{(k)} := 5.0$, $y_{\mathrm{lb}} := 0.0$, $y_{\mathrm{ub}} := 7.0$
   (d) $\lambda^{(k)} := 0.5$, $\lambda_{\mathrm{lb}} := 0.0$, $\lambda_{\mathrm{ub}} := 0.68$
2. Define the search subregion of linking variables y and λ, and plan for each variable a design of experiments (DOE) of N equally spaced points. Set the subregion lower bounds to $y_{\mathrm{l}} = (1-p)y^{(k)}$ and $\lambda_{\mathrm{l}} = (1-p)\lambda^{(k)}$, and the upper bounds to $y_{\mathrm{u}} = (1+p)y^{(k)}$ and $\lambda_{\mathrm{u}} = (1+p)\lambda^{(k)}$, respectively, while satisfying $y_{\mathrm{lb}} \le y_{\mathrm{l}} \le y_{\mathrm{u}} \le y_{\mathrm{ub}}$ and $\lambda_{\mathrm{lb}} \le \lambda_{\mathrm{l}} \le \lambda_{\mathrm{u}} \le \lambda_{\mathrm{ub}}$.
3. Solve SP_a for the DOE of y, and SP_b for the DOE of λ. The minimizer of SP_b is independent of y, so any y value may be selected for SP_b.
4. Build quadratic response surface approximations for μ_2 as function of y and x_1 as function of λ.
5. Update the master problem:
   (a) $k := k+1$
   (b) Compute $y^{(k)}$ and $\lambda^{(k)}$ by solving the nonlinear set of equations:
   $$
   \begin{aligned}
   y &= x_1 \\
   \lambda &= 0.1\mu_2 \\
   \mu_2 &= a_0 + a_1 y + a_2 y^2 \\
   x_1 &= b_0 + b_1\lambda + b_2\lambda^2,
   \end{aligned}
   $$
   where a_i and b_i have been determined in step 4.
   (c) Solve SP_a and SP_b using the updated $y^{(k)}$ and $\lambda^{(k)}$ to obtain $\mathbf{x}^{(k)}$.
6. Check convergence: if $\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\| < \varepsilon$, , then stop; otherwise goto 2.

Applying this methodology on our example, the values of the linking variables y and λ converged after four iterations; the obtained results matched the ones reported in Section 4.2 and [22]. We describe the χ implementation of the approach in the next paragraphs. Before we do that, we would like to note that the search subregions need to be selected carefully. For example, possible discontinuities in the solutions of the optimization problems for some values of the linking variables may result to meaningless response surface approximations. This is why λ is bounded above by the value 0.68 in our example.

The main change compared to the previous χ implementation of Section 4.2 is that N parallel processes SP_a and N parallel processes SP_b can be instantiated to carry out the optimization calculations. The process specifications of SP_a and SP_b need not be changed. The χ specification of process C becomes

```
const N: nat = 6

proc C(p: (!vecx × real)^N, q: (!vecx × real × real)^N
       , v: (?vecx × real)^N, w: (?vecx)^N) =
[[ x_a^o, x_a, x_b^o, x_b: vecx
, y_lb, y, y_ub, λ_lb, λ, λ_ub, μ_2, p: real
, y^doe, μ_2^doe, λ^doe: real^N, x_a^doe, x_b^doe: vecx^N
, k: nat, end: bool
| x_a := ⟨0.1, 0.2, 10.0, 0.1⟩
; x_b := ⟨0.1, 0.2, 10.0, 0.1⟩
; y_lb := 0.0; y_ub := 7.0; λ_lb := 0.0; λ_ub := 0.68
; y := 5.0; λ := 0.5; p := 0.5; k := 0; end := false
; *[¬ end
    --> k := k+1; x_a^o := x_a; x_b^o := x_b
      ; y^doe := doe(y, y_lb, y_ub, p)
      ; λ^doe := doe(λ, λ_lb, λ_ub, p)
      ; ba := bval(true); bb := bval(true)
      ; *[ i: nat <- 0..N: ba.i; p.i!⟨x_a^o, y^doe.i⟩
             --> ba.i := false
         [] i: nat <- 0..N: bb.i; q.i!⟨x_b^o, y^doe.i, λ^doe.i⟩
             --> bb.i := false
         ]
      ; ba := bval(true); bb := bval(true)
      ; *[ i: nat <- 0..N: ba.i; v.i?⟨x_a^doe.i, μ_2^doe.i⟩
             --> ba.i := false
         [] i: nat <- 0..N: bb.i; w.i?x_b^doe.i
             --> bb.i := false
         ]
      ; ⟨y, λ⟩ := update(y^doe, μ_2^doe, λ^doe, x_b^doe)
      ; p.0!⟨x_a^o, y⟩; q.0!⟨x_b^o, y, λ⟩
      ; v.0?⟨x_a, μ_2⟩; w.0?x_b
      ; end := norm(x_a^o, x_b^o, x_a, x_b) < 10^-3
  ]
]].
```

At first, process C generates the two designs of experiments for linking variables y and λ using a χ function *doe*:

```
func doe(q, q_lb, q_ub, p: real) --> real^N =
[[ q_l, q_u: real, d: real^N, i: nat
| q_l := (1-p)q; q_u := (1+p)q
; q_l := min(q_l, q_lb); q_u := max(q_u, q_ub)
; i := 0
; *[ i < N --> d.i := q_l + (q_u - q_l) i/(N-1); i := i+1 ]
; ↑d
]].
```

In the next step, process C sends the individual DOE and initial points to each of the optimization processes SP_a and SP_b. This is specified using a repetitive selective waiting statement with a range expression, which can be explained as: for all ports $p.i$, if no job has been sent, send the job and update the corresponding boolean array. The boolean arrays *ba* and *bb* administrate to which processes optimization jobs still have to be sent. Initialization of *ba* and *bb* to arrays full of true values is done through the function *bval*:

```
func bval(v: bool) --> bool^N =
[[ b: bool^N, i: nat
| i := 0; *[ i < N --> b.i := v; i := i+1 ]
; ↑b
]].
```

When all processes SP_a and SP_b have received their respective jobs, the first repetitive selective waiting statement is finished. A second repetitive selective waiting statement follows to receive all optimization results. The results are stored in the arrays $\mathbf{x}_a^{doe}$, μ_2^{doe} and $\mathbf{x}_b^{doe}$. When all results have been received, the response surfaces are generated and the nonlinear set of equations is solved by an external Matlab function *update*. The new iterates for y and λ are returned to χ. One SP_a and one SP_b optimization run are carried out to obtain the corresponding optimal system design update $\mathbf{x}^{(k)}$. Finally, a check is performed to determine whether convergence has been achieved .

System S is changed accordingly; instead of one send and one receive channel to a single process SP_a, now an array of channels (*bundle*) is needed to couple C with all N instantiations of SP_a. The same holds for the channels from C to processes SP_b. The new system specification becomes

```
syst S() =
[[ p, v: (-vecx × real)^N, q: (-vecx × real × real)^N
, w: (-vecx)^N
| C(p, q, v, w)
|| i: nat <- 0..N: SP_a(p.i, v.i)
|| i: nat <- 0..N: SP_b(q.i, w.i)
]].
```

As explained in Section 3.6, the external function calls *optspa* and *optspb* have to be split to be able to actually *run* the optimization calculations in parallel using the current version of the χ compiler. Each external function call is split into a startup function call, a synchronization send and a synchronization receive to a synchronizer process, and a call to a function that will retrieve the results when the job is finished. The synchronizer registers which processes have started their external function calls. When all processes have started their jobs, the synchronizer gives each process clearance to call the retrieval function. The new coordination architecture is visualized in Figure 5, where process *Sy* is the synchronizer process. Process *Sy* needs to know how many processes SP_a and how many processes SP_b have received a request for an optimiza-

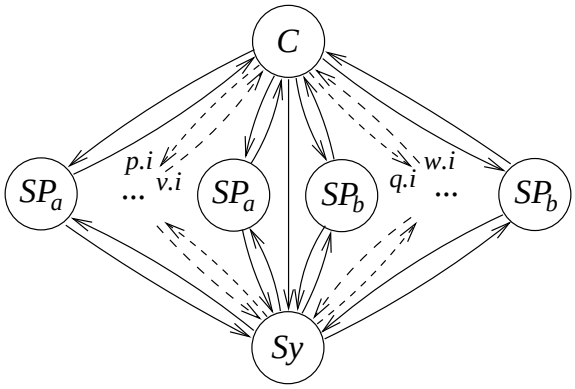

Figure 5: RSM-based coordination of geometric programming problem including synchronizer process for parallel execution.

tion run. Process *C* has to send these numbers to *Sy*.

The necessary modifications to the χ specification are as follows. Process *C* needs an additional send port through which the numbers of optimization jobs (SP_a and SP_b, respectively) can be sent to *Sy*. Processes SP_a and SP_b need two additional ports: one synchronization send port and one synchronization receive port. The split of the function call has to be implemented also. Process SP_a becomes

```
proc SP_a(a: ?vecx × real, b: !vecx × real
        , as: !void, sa: ?void) =
[[ x_a^o, x_a^*: vecx, μ_a: vecg, y: real, h: nat
| *[ true
     ⟶ a?⟨x_a^o, y⟩; h := start_optspa(x_a^o, y)
       ; as!; sa?
       ; ⟨x_a^*, μ_a⟩ := get_optspa(h); b!⟨x_a^*, μ_a.0⟩
   ]
]] .
```

Process SP_b has to be changed similarly. The synchronizer process *Sy* has to be added:

```
proc Sy(cs: ?nat × nat
      , as, bs: (?void)^N, sa, sb: (!void)^N) =
[[ ca, cb: nat, ba, bb: bool^N
| ba := bval(true); bb := bval(true)
; *[ true
     ⟶ cs?⟨ca, cb⟩
       ; *[ i: nat ← 0..N: ca > 0 and ba.i; as.i?
              ⟶ ba.i := false; ca := ca − 1
          [] i: nat ← 0..N: cb > 0 and bb.i; bs.i?
              ⟶ bb.i := false; cb := cb − 1
          ]
       ; *[ i: nat ← 0..N: ¬ba.i; sa.i! ⟶ ba.i := true
          [] i: nat ← 0..N: ¬bb.i; sb.i! ⟶ bb.i := true
          ]
   ]
]] .
```

Process *Sy* first receives the number of processes SP_a and the number of processes SP_b optimization jobs have been sent to. The repetitive selective waiting statement that follows implements the receive of the start notifications of the external function calls by processes SP_a and SP_b. The second repetitive selective waiting sends the clearance to proceed back. Finally, system *S* has to be extended to include *Sy* and the additional channels declarations.

5 Conclusions

In our opinion a concurrent programming language for distributed optimal system design can improve the implementation and testing of MDO coordination strategies significantly. Such a language is especially needed when the scale and complexity of coordination architectures requires a more precise treatment of the concurrency in the coordination. A language with a clear foundation in concurrency theory would furthermore provide a means to do a formal analysis of a specified coordination strategy regarding the existence of failures such as deadlock. A specification language for engineering-based MDO approaches has to be able to deal with the "black-box" nature of the disciplines and subsystems, as well as the large amounts of numerical data that may need to be passed around between them. A language based on Communicating Sequential Processes meets the first requirement. In addition, the language has to include suitable language constructs for data handling and data storage to meet the second requirement.

We propose to use the χ language, which meets both the "black-box" and the data handling requirement. It is a highly expressive CSP-based language that contains advanced data modeling constructs. Using χ, the MDO coordination is specified in a standard fashion as a number of parallel processes that operate independently and communicate with each other synchronously over pre-defined channels. The advantage of χ for application in MDO is that it has been designed for modeling purposes: it is compact (few language constructs), easy to understand, and offers a clear concept to define parallelism. Furthermore, χ can perform function calls to external software by means of a Python interface. In the paper the syntax and semantics of the main language elements that are needed for the MDO application are informally explained.

A simple analytical example has been used to demonstrate the χ implementation of a coordination strategy for distributed optimal system design. In addition, this example has been extended to show that χ can also be used for implementing the coordination of simulation-based MDO using response surface approximations. In both cases the example demonstrates how the concurrency in the coordination is specified. In future work, we will demonstrate the capabilities of

χ for specifying coordination in larger and more complicated MDO problems. At this point, we draw the conclusion that the use of a concurrent programming language such as χ to specify and implement MDO coordination strategies is quite promising. It may also provide new opportunities to scale up the size of MDO implementations.

Current work focuses on specifying coordination strategies for analytical target cascading using the χ language [25]. Convergence properties of analytical target cascading have been proven for a number of possible coordination strategies [26]. However, convergence rate of these strategies has not been studied yet; χ will provide a means to test and evaluate the performance of the latter in different problems.

REFERENCES

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

[2] Papadopoulos, G.A. and Arbab, F., 1998: "Coordination Models and Languages", CWI Software Engineering report SEN-R9834, National Research Institute for Mathematics and Computer Science, Amsterdam, the Netherlands.

[3] Hoare, C.A.R., 1985: *Communicating Sequential Processes*, Prentice-Hall.

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

[5] Kleijn, J.J.T., and Rooda. J.E., 2001: "χ Manual", Systems Engineering Group, Eindhoven University of Technology, http://se.wtb.tue.nl.

[6] Rooda. J.E. 2000: "Modelling Industrial Systems", lecture notes, Systems Engineering Group, Eindhoven University of Technology, http://se.wtb.tue.nl.

[7] Van Beek, D.A. and Rooda, J.E., 2000: "Languages and applications in hybrid modelling and simulation: the positioning of Chi", *Control Engineering Practice* **8**, 81–91.

[8] Python, http://www.python.org.

[9] Lutz, M. and Ascher, D., 1999: *Learning Python*, O'Reilly.

[10] Hofkamp, A.T. 2001: "Python from χ", note, Systems Engineering Group, Eindhoven University of Technology, http://se.wtb.tue.nl.

[11] Matlab 5.3, Matlab, The Mathworks, http://www.mathworks.com.

[12] Papalambros, P.Y. and Wilde, D.J., 2000: *Principles of Optimal Design*, Cambridge University Press, 2nd edition.

[13] Cramer, E.J., Dennis, J.E. Jr., Frank, P.D., Lewis, R.M., and Shubin, G.R., 1994: "Problem formulation for multidisciplinary optimization", *SIAM Journal on Optimization* **4**, 754–776.

[14] Balling, R.J. and Sobieszczanski-Sobieski, J., 1996: "Optimization of coupled systems: a critical overview of approaches", *AIAA Journal* **34**, 6–17.

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

[16] Bos, V. and Kleijn, J.J.T., 2002: *Formal specification and analysis of industrial systems*, Doctoral dissertation, Eindhoven University of Technology, Eindhoven, The Netherlands.

[17] Hoare, C.A.R., 1978: "Communicating sequential processes", *Communications of the ACM* **21**, 666–677.

[18] Dijkstra, E.W., 1975: "Guarded commands, nondeterminacy, and formal derivation of programs", *Communications of the ACM* **18**, 453–457.

[19] Van de Mortel-Franczak, J.M., Rooda, J.E., and Van den Nieuwelaar, N.J.M., 1995: "Specification of a flexible manufacturing system using concurrent programming", *Concurrent Engineering: Research and Applications* **3**, 187–194.

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

[21] Hock, W. and Schittkowski, K., 1981: *Test Examples for Nonlinear Programming Codes*, Springer Verlag.

[22] Wagner, T.C., 1993: *A General Decomposition Methodology for Optimal System Design*, Doctoral dissertation, The University of Michigan", Ann Arbor, Michigan.

[23] Kodiyalam, S. and Sobieszczanski-Sobieski, J., 2000: "Bilevel integrated system synthesis with response surfaces", *AIAA Journal* **38**(8), 1479–1485.

[24] Sobieski, I.P. and Kroo, I.M., 2000: "Collaborative optimization using response surface estimation", *AIAA Journal* **38**(10), 1931–1938.

[25] Etman, L.F.P., Kokkolaras, M., Papalambros P.Y., Hofkamp, A.T., and Rooda, J.E., 2002: "Coordination specification of the analytical target cascading process using the χ language", *9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Atlanta, GA, work in progress paper no. AIAA-2002-5637.

[26] Michelena, N., Park, H., and Papalambros, P.Y, 2002: "Convergence properties of analytical target cascading", *9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Atlanta, GA, paper no. AIAA-2002-5506.