On Scalable Reconfigurable Component Models for High-Performance Computing

Vincent Lanore

THÈSE

en vue de l'obtention du grade de

Docteur de l'Université de Lyon,
délivré par l'École Normale Supérieure de Lyon

discipline

Informatique

laboratoire

Laboratoire de l'Informatique du Parallélisme

école doctorale

École doctorale en Informatique et Mathématiques de Lyon

présentée et soutenue publiquement le 10 décembre 2015
par Monsieur Vincent LANORE

On Scalable Reconfigurable Component Models for High-Performance Computing

---

directeur de thèse

M. Christian Pérez

devant la commission d'examen formée de

Raymond	Namyst	Professeur, Univ. de Bordeaux	Président
Marco	Danelutto	Professeur associé, Univ. de Pise	Rapporteur
Laurence	Duchien	Professeure, Univ. de Lille 1	Examinatrice
Laxmikant V.	Kale	Professeur, Univ. de l'Illinois	Examinateur
Christian	Pérez	Directeur de recherche, Inria	Directeur
Jean-Bernard	Stefani	Directeur de recherche, Inria	Rapporteur

Acknowledgements / remerciements

(English part) First, I want to thank all the members of the jury: Mr Raymond Namyst and Ms Laurence Duchien for their presence at the defense; Mr Laxmikant V. Kale for his remote participation despite the time difference; Mr Marco Danelutto for reviewing the manuscript; and, especially, Mr Jean-Bernard Stefani for his presence at the defense, his review of the manuscript despite a very tight schedule, and his precious insight regarding the contents of the manuscript.

(French part) Je tiens à remercier tous ceux qui ont dû me supporter pendant la thèse : l'inénarrable professeur Simon ; François, pour son fascisme sans égal ; Mathias, mais pas pour ses pâtes carbo ; Fred, pour le beurre ; Julien, dont c'est clairement la faute ; Maxime, informaticien malgré lui ; Samantha, la reine des mouches ; sans oublier François (mais pas le même), Vincent, Clément et le reste des ambreux, pour m'avoir irrémédiablement condamné à ne pas avoir assez de temps libre. Je tiens également à remercier les autres lyonnais, ceux qui sont partis avant que je ne commence la thèse (coucou Hugo), et ceux que j'ai sans doute oublié. À cause de vous tous, je vais vraiment avoir du mal à quitter Lyon ; j'espère que vous êtes fiers de vous.

Un grand merci à toute ma famille qui m'a toujours soutenu dans mes études, et qui a toujours supposé que je savais ce que je faisais avec mes choix d'orientation.

Merci également à tout ceux qui m'ont poussé vers l'informatique et la recherche, sans qui je n'aurais jamais fait cette thèse. Je pense notamment aux enseignants qui ont participé à me donner le goût des maths, en particulier à M. Cozar qui m'a poussé vers l'ENS alors que je ne me donnais pas une chance.

Un grand merci bien sûr à toute l'équipe Avalon avec qui ça a été un plaisir de travailler pendant ces trois ans, et plus largement à tous ceux du LIP qui m'ont aidé au cours de la thèse. Merci en particulier à la team composants (Jérôme et Hélène), aux irremplaçables trolls du coin café (notamment Arnaud, Anthony, Laurent et Daniel), à Eddy avec qui j'ai eu beaucoup de plaisir à enseigner, et à tous ceux qui ont dû partager un bureau avec moi (Jonathan, Sylvain, Hélène et Jérôme). Merci à Cristian, avec qui j'ai eu beaucoup de plaisir à travailler lors de mon stage de master, et à Maverick, qui a été un stagiaire exemplaire. Je tiens à remercier tout particulièrement ceux qui m'ont aidé dans l'organisation de la soutenance, qui a été un moment particulièrement stressant pour moi : Violaine, Hélène, Jérôme, Jean-Christophe et Serge.

Enfin, et surtout, merci à Christian pour son encadrement tout au long de la thèse. Ça a été un grand plaisir de travailler avec lui pendant ces trois ans.

Contents

Chapter 1

Introduction

Since the early days of computer programming, code reuse has been the focus of research. Software engineering research has developed over the years a variety of concepts to help code reuse, such as procedures or objects. Code reuse is important to avoid having to reinvent the wheel for each new project, when third-party code could be used instead. Moreover, even within a single application, code quickly becomes redundant. Proper reuse and/or factorization techniques can tremendously ease the development and maintenance of applications.

Component-based software engineering (CBSE) is a software engineering paradigm, introduced in the 1960s [74], which proposes to reuse code through composition. Code to be reused is encapsulated in entities called components which are meant to be composed. Let us say that some component A needs to reuse the code from some other component B, then A can be composed with B. This approach is different from object-oriented software engineering which uses inheritance as its main means of code reuse.

Component-based programming (CBP) is a direct application of this paradigm and consists in writing applications through component composition only. This approach assumes that there exist components, written by third-parties, which are available "on a shelf". These components can be composed using a dedicated language to form a full application. This process is called "component assembly". Component-based programming requires the use of a dedicated model, called a component model, which defines components and composition. Component models have been the focus of a lot of academic and industrial efforts in the last decades. Many component models, with various features and properties, have been proposed and implemented over the years.

Compared to other software engineering paradigms, component-based approaches have two main advantages: easy separation of concerns, and high-level view of application architecture. Separation of concerns (SoC) is the process of dividing an applications into parts (in our case: components) so that each part deals with a small number of specific "concerns". SoC is an important software engineering property which ensures applications parts are easy to write (less concerns means less complexity and skills involved), and easy to reuse (SoC means less dependencies). CBSE proposes both separation of concerns between components (each component deals with a specific task), and between the component level (low-level concerns) and the assembly level (high-level concerns, application structure). Moreover, a component assembly provides a convenient high-level view of an application structure. Indeed, low-level concerns are hidden within components and composition can be used to express meaningful interactions between parts of the application. Also, this high-level view provides convenient high-level mechanisms to tweak applications, e.g., to adapt them to specific use cases.

CBSE has been successfully applied to High-Performance Computing (HPC) applications. HPC applications are applications whose sequential execution time is ludicrously high (e.g.,

years or centuries) and which, consequently, target highly-parallel hardware architectures such as supercomputers or computer clusters. HPC applications include, for example, large-scale simulations in scientific domains such as climatology, chemistry or astrophysics. HPC applications tend to be large (e.g., hundreds of thousands of lines of codes), complex (e.g., cutting-edge physics/maths, fine-grained multithreaded synchronization) and to have drastic scalability requirements (e.g., hundreds of thousands of cores). Moreover, HPC applications require frequent adaptation and tweaking, to fit different use cases and target hardware architectures, over the course of their long lives (typically tens of years). CBSE has been used in this case to both ease the expression of parallelism (thanks to the assembly-level view offered by CBP) and ease adaptation to different use cases and hardware (by tweaking the assembly).

Unfortunately CBP has a hard time dealing with applications whose code, communication topology and/or data topology change during execution. One example of such an application, in HPC, is Adaptive Mesh Refinement (AMR), a computing technique which involves dynamic refinement of a mesh of data. From a CBP perspective, this kind of application is difficult to implement because the application structure (which highly influences the component assembly) changes dynamically during execution. Most existing component models do not provide the same benefits (e.g., assembly view, third-party reuse) during execution, as they do at assembly time. While it is possible to circumvent the problem by using large components which encapsulate the dynamic parts, this defeats the purpose of using components in the first place. Moreover, it would be interesting to express structure changes as assembly changes, in order to benefit from the high-level view provided by CBP. In order to do that, one must use a reconfigurable component model, i.e., a component model which supports assembly modification (reconfiguration) at runtime.

Reconfigurable component models exist in the literature but none of them is compatible with the scalability and performance requirements of HPC applications. Indeed, these models impose synchronization constraints (e.g., global synchronization before any reconfiguration) which limit scalability in the general case (e.g., global synchronization is known not to be a scalable operation). HPC applications are a particularly challenging case because they often require (in order to scale) fine-grain custom synchronization (e.g., point-to-point message-passing synchronization across millions of processes).

Problem

The goal of the present thesis is to propose a reconfigurable component model compatible with HPC performance and scalability requirements. Such a model is important to help design, maintain and adapt HPC applications with dynamic structure, which are among the most complex HPC applications. This problem is not simple though, and presents several sub-problems.

The first sub-problem that must be addressed is the synchronization of concurrent reconfigurations. When several concurrent reconfigurations occur at once in an assembly, consistency and synchronization problems arise. This problem can be further decomposed into ensuring safety (i.e., consistency of application state during execution) and performance (i.e., minimizing reconfiguration time for HPC purposes).

The second sub-problem to be addressed is programmability of reconfigurable applications. Reconfigurable applications, and reconfigurable HPC applications in particular, typically are large and complex applications, which are difficult to develop. In addition, HPC applications must be easily adaptable to various hardware architectures. While high-level programmability-oriented features exist in the component literature (e.g., hierarchy, genericity), their use with reconfigurable assemblies is problematic (notably hierarchy).

In addition, solutions for these various problems can easily conflict and/or be difficult to reconcile. For example, performance and programmability are goals which typically hamper

one another. Also note that the proposed solutions must comply with the core principles of component-based programming, otherwise the software engineering benefits this approach provides may not longer hold.

Contributions

For this thesis, we have decided not to focus on one particular sub-problem but, instead, to adopt a "vertical slice" approach. We have endeavoured to tackle both software-engineering-oriented and HPC-oriented problems in order to provide a full end-to-end solution and overview of the larger problem.

DirectMOD The main contribution of the thesis, and the cornerstone of our approach, is the DirectMOD formal component model. This model aims to:

- provide an assembly-level language for assembly transformations;
- ease the reuse of transformations;
- separate the synchronization and locking concerns from the rest of the application.

In order to achieve these goals, two novel concepts are introduced: *domains* and *transformation adapters*. Domains are special components which are responsible for safely executing transformations. Transformations adapters are special ports which explicitly connect a transformation to its target subassembly. We provide full syntax and semantics for DirectMOD in a formal fashion. This first contribution is evaluated both through a model-level analysis, and using DirectL2C, an implementation we developed. Evaluation deals with both the programmability capabilities of DirectMOD and its performance.

Efficient Locking Our second contribution is a set of models and tools to help with stopping running components in an efficient and reliable fashion. We present a formal model for a stopping paradigm called *mutex-based locking*, which is efficient but difficult to use. We then propose an algorithm which helps to implement mutex-based locking while relying on very little metadata. We evaluate this approach on a series of AMR-based benchmarks, which we compare in terms of code metrics and performance. While this approach does not work in the general case, it works well for a certain class of applications which include useful use cases such as stencil codes.

SpecMOD Our third contribution is SpecMOD, a formal calculus for specialization of hierarchical component assemblies. SpecMOD models the element-by-element specialization of component assemblies and provides a formal framework to express specialization algorithms. We show how SpecMOD can support hierarchy, genericity and—more importantly in the context of this thesis—assembly transformations. Full syntax and semantics of SpecMOD are provided. The generality of the model is extensively discussed through a series of examples and extensions of the model. Preliminary work to apply SpecMOD to DirectMOD, in order to improve the transformation and assembly description, is also presented and evaluated.

Structure of this Document

The present thesis is divided into two parts. First, Part I presents the context of the thesis and relevant related works. This part is divided into Chapter 2, which gives the context on HPC and component models, and Chapter 3, which presents related works on reconfiguration in general. Second, Part II presents our contributions, one per chapter. This second part is divided into three chapters. Chapter 4 presents DirectMOD, which is a formal component model which

allows concurrent reconfiguration while preserving separation of concerns. Chapter 5 presents a series of models and tools aimed at stencil-style applications to help with efficient deadlock-free locking of component assemblies while requiring minimal metadata. Chapter 6 presents SpecMOD, a formal calculus for component assembly specialization which support genericity, hierarchy and DirectMOD-style transformations; usage with DirectMOD is also discussed. Finally, Chapter 7 concludes and present perspectives.

Part I

Context and Related Works

Chapter 2

Context

Contents

This chapter presents High-Performance Computing and Component Models, the two domains at the intersection of which lie the subject matter of the thesis. For each of those two domains, a presentation of the important concepts relevant to the understanding of the thesis is given. Section 2.1 presents High-Performance Computing (HPC), while Section 2.2 deals with component models.

2.1 High-Performance Computing

High-Performance Computing, abbreviated as HPC, is the domain of computer science that deals with applications with very large space and/or time requirements. Typically, a HPC application executing on a traditional desktop computer would take years or centuries to terminate. Alternatively, some HPC applications might require inordinate amounts of memory. HPC as a scientific domain deals with the development of new hardware and software techniques so as to be able to run more and more demanding applications.

HPC today deals with both unusual hardware architectures and unusual software, and a few notions about both are required to fully understand the context for the present thesis. Section 2.1.1 presents modern HPC hardware and afferent challenges, while Section 2.1.2 deals with HPC application characteristics, and finally Section 2.1.3 presents a specific HPC application (Adaptive Mesh Refinement) as a motivating example for this thesis.

2.1.1 Hardware Architectures Today

Over the years, a variety of different hardware architectures have been developed for HPC applications. Not all those architectures target the same kind of application, and their characteristics vary depending of their specific goals.

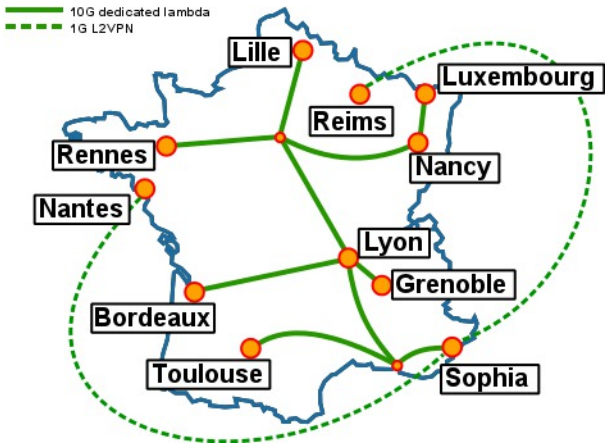


Figure 2.1: Grid'5000, an example of a computing grid.

High-performance hardware is a complex subject whose intricacies could easily fill a whole thesis, but this section gives only a brief overview. Indeed, the subject matter of the thesis requires no more than a coarse-grain understanding of the main characteristics of HPC hardware and underlying issues.

Brief taxonomy Most modern HPC architectures are collections of computing hardware, structured into large architectures. Different paradigms can be used to build such architectures. The most common of those paradigms are:

- *Supercomputers* are large, tightly-connected high-performance architectures built as a single machine. Supercomputers are different from the other approaches below in that they are designed from scratch as a single machine, instead of being a collection of smaller machines.
- *Clusters* are collections of desktop-style or server-style computers, assembled together using high-performance networks. The machines composing a cluster are typically called *computing nodes* (or just *nodes*). On many clusters (depending on the program used for resource reservations), users know the structure and can reserve specific subsets of the nodes for a given computing task.
- *Computing grids* are collections of clusters and supercomputers, often geographically distant from one another, connected to form a single infrastructure. While grids can be used to run applications on a single cluster or supercomputer, they also allow to run grid-wide applications. One example of a computing grid is Grid'5000 [10], an experimental platform regrouping clusters from sites all across France.
- *Clouds* are computing resources abstracted as on-demand services. Clouds typically try to hide away architectural information such as network topology or node location. While clouds are mostly known for non-HPC applications, there have been initiatives that try to use them for HPC (early examples include [92, 93]).

Size and parallelism HPC infrastructure are typically quite large and very parallel. For example, at the time of writing, the largest supercomputer is the Tianhe-2, developed at China's National University of Defense Technology [1], which sports 3,120,000 computing cores.

The point of those very large architectures is to push the boundaries of what a single application can compute in a reasonable time (up to months or years). Thus, such architecture are

not meant to be used by a lot of concurrent applications but, as much as possible, by large, very parallel applications.

There is a constant push towards larger architectures. At the time of writing, the largest supercomputers are capable of tens of quadrillions of floating point operations per second (i.e., tens of Pflop/s) [1]. This scale of architecture is called *petascale computing*, and a lot of initiatives are already working towards the next step, *exascale computing* (i.e., thousands of Pflop/s); see for example [18, 89, 46].

This race towards larger architectures means that architectures are more and more parallel. Indeed, as the frequency of circuits has pretty much stopped, increasing the power of chips was done by cramming more and more computing units in them, using the extra transistors predicted by Moore's law [76].

Heterogeneity In addition to being large and parallel, HPC architectures tend to be heterogeneous, in the sense that hardware inside a single architecture can be heterogeneous.

First, there might be heterogeneity inside computing nodes. Typically, a computing node can feature a hardware accelerator in addition to its CPU, such as a GPU, a FPGA, or an Intel MIC coprocessor. Also, HPC computing nodes may have non-uniform memory or processor architectures.

Second, there might be inter-node heterogeneity, either in the form of non-uniform access (due to network concerns), or simply different nodes hardware-wise. A typical example of inter-node heterogeneity is provided by computing grids.

Concluding remarks HPC architectures are very large, very parallel (up to millions of cores and growing), very complex, and can differ greatly from one another. This poses a challenge to programmers, particularly in a HPC context where making the most of every piece of hardware is crucial for performance.

2.1.2 HPC Applications

HPC applications are, by definition, applications with very large computing times and/or memory requirements, that target HPC hardware. HPC applications have other specific characteristics which are detailed in the present section.

Application domains HPC applications come from a variety of science and engineering domains. Typical examples of HPC applications include:

- climatology (e.g., [59]);
- engineering simulations (e.g., nuclear-waste disposal [80]);
- biology (e.g., DNA alignment [5], biomolecular dynamics [56]);
- physics (e.g., astrophysics [98, 94]);
- computational chemistry (e.g., quantum chemistry [3]).

A study conducted by PRACE (Partnership for Advanced Computing in Europe, a European organisation regrouping most Europe's world-class HPC infrastructures) [90] shows that, in practice, on large computers, physics and chemistry constitute the majority of applications.

Scalability Since HPC infrastructures are very parallel (up to millions of cores, see Section 2.1.1), HPC applications must be capable of such a level of parallelism.

Most HPC applications are moldable, i.e., the level of parallelism can be configured per-run at the start of the application or at compilation. Technically, any such application can run at arbitrarily high parallelism. However, the very point of parallelism is to improve computing

time as the number of cores/threads/processes increases, which is difficult. The relation between the amount of computing resources available to an application and its performance is called *scalability*. At the very best, the performance of an application (number of operations/time unit) can scale linearly with the number of computing resources [1].

In practice linear scaling is difficult to attain for high resource counts as many phenomenons hamper scalability. At the very least, applications are held by Amdahl's law [6], which states that an application's performance has a sequential component which becomes dominant as the resource count go up, preventing linear scaling. Many other problems arise that hamper scalability, including:

- Algorithmic limitations: some problems cannot easily be parallelised (e.g.,, irregular meshes).
- Saturation of resources that arise with intensive use, e.g., network, memory bandwidth.
- Imperfect use of resources: it is difficult to ensure that every core/process/thread has work to do 100% of the time. This problem is called *load balancing*.
- Management/synchronisation overhead: managing thousands/millions of cores/processes/threads is no trivial task and may be imperfect or require resources in itself.

Scalability is a very central issue in HPC, a large part of the literature aiming at improving scalability through technologies, algorithms, and software optimisations.

In terms of figures, the largest HPC applications target million-cores architectures (see Section 2.1.1), and execution times of a few years at most. This means that the largest HPC applications require millions of *cpu.month*, i.e., millennia of cpu time.

Languages and technologies Most HPC applications are written in Fortran, C, or C++ [90]. All these languages have traditional call-stack-based executions, and a C-style memory model with pointers. From a software engineering perspectives, these languages feature functions, and module/objects for some of them.

Since HPC applications need to be very parallel, several technologies were developed to allow/ease parallel programming in C/C++/Fortran. The most prominent of these technologies are:

- MPI [50] (Message Passing Interface) is a widely-used standard in C++/Fortran HPC applications. It handles parallelism through message passing and has a distributed memory model, i.e., a MPI application is composed of several *processes* with their own memory which communicate by sending and receiving messages. A noteworthy feature of MPI is collective communications, i.e., procedures for communications involving many processes. Collective communications are usually very well-optimised and can bring a very noticeable increase in performance compared to using only point-to-point communications.
- Shared-memory technologies, such as OpenMP [38] and Intel TBB [84], rely on a shared memory approach, i.e., the application has several control threads that can access the same memory and must synchronise to avoid conflicts.
- Traditional thread libraries, e.g., PThreads [77], are low-level libraries that provide the basic primitives to create threads and synchronise them.
- Partitioned Global Address Space (PGAS) languages, e.g., UPC [47], Co-array Fortran [78], HPF [72], or XcalableMP [71]. These languages propose to abstract all the memories available in a (possibly distributed) system, as a single partitioned memory (the partition represents the various memory locations). This allows a shared-memory programming style, while still permitting to distinguish between memory locations.

[1] Technically, superlinear performance increase with the number of computing cores exist, but is generally a side-effect of having other resources that come with the additional cores (e.g., memory).

In addition to those technologies, which are mostly aimed at CPU-based architectures, there exist technologies dedicated to accelerators such as CUDA [79], OpenCL [53] or FPGA suites.

A synthesis and detailed classification of these technologies can be found in [45].

Complexity HPC applications are very complex in several respects.

First, because of the need to increase performance as much as possible, HPC applications rely on noticeable amounts of highly-optimised low-level code. Such low-level code is not only verbose, but also difficult to write and error-prone. Examples of difficult code issues that HPC programmers must face include manual fine-grain memory management (possibly down to the bit), or thread-level synchronisation (very error-prone).

Second, HPC programmers need to have a variety of non-trivial skills. Indeed, HPC typically requires knowledge of parallelism, non-standard hardware, and understanding of the functional part of the application (which typically involves state-of-the-art science). For this reason, HPC applications are difficult to develop and maintain, as programmers with such skill combinations are difficult to find.

In addition, HPC applications need to be adapted to different hardware in the course of their life. Indeed, the lifetime of HPC applications (e.g., TURBOMOLE which was introduced in 1989 [3], was still used in 2008 [91]), is shorter than the lifetime of HPC hardware (years). As many HPC applications rely on hardware-dependent optimisations to make the most of state-of-the-art hardware, it is necessary to adapt these applications when moving to new hardware.

Moreover, as the power of HPC hardware grows, possibilities emerges of combining several existing codes. For example, it is now possible to combine a heat simulation code with a fluid physics code to perform a heat simulation on a dynamic fluid, which would have been too costly to compute twenty years ago Combining several existing code is often preferred to writing a new one from scratch, as it saves time and builds on already well-known and accepted tools. However, this process, referred to as *code coupling*, leads to applications made of several large and complex glued-together parts, increasing even further the skills and knowledge required to have a good understanding of the whole application.

Finally, HPC applications are large in terms of code size, typically in the hundreds of thousands/million range in terms of lines of code (LOC). While these sizes are not particularly noteworthy in themselves—a typical OS or car software ranging in the tens or hundreds of millions LOC—they can still be problematic, as they are very complex and often developed/maintained by very small academic teams.

Dynamicity Many HPC applications have some sort of dynamic structure that evolves during execution. As HPC applications are very parallel and very complex, it is often difficult to predict its behaviour and performance perfectly, and thus several optimisation techniques rely on decisions taken at runtime to optimise performance.

A first example of dynamicity is dynamic load balancing. To maximise performance, a HPC application must make sure every core/thread/process has work most of the time. Since it is often difficult to estimate the computation time of everything, some application rely on *dynamic load balancing*, i.e., they attribute work at runtime, based on the observed workload.

Another common example of dynamic behaviour in HPC application is resiliency, or fault tolerance. As HPC applications can take centuries of cpu time, hardware faults that would otherwise be negligible become a real problem. Such large application require software mechanisms to ensure it completes its computation without error. While some techniques to handle fault tolerance are not very dynamic (e.g., checkpointing), some HPC works advocate dynamic approaches, e.g., [48] proposes to proactively migrate tasks away from nodes that are about to fail.

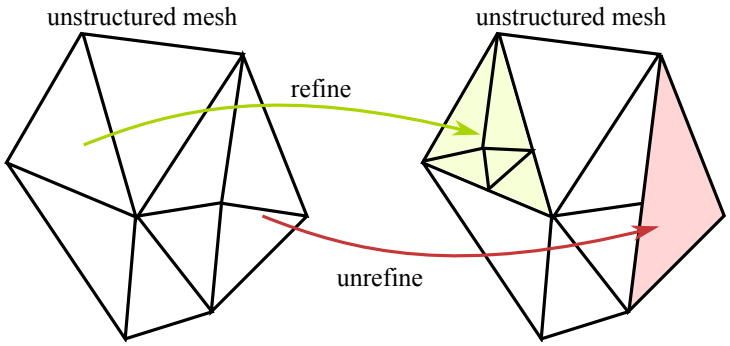


Figure 2.2: Illustration of mesh refinement and unrefinement on an unstructured mesh. Refinement adds nodes to the mesh, while unrefinement fuses several mesh elements together.

In addition, in order to use HPC platforms to their maximum, it can be interesting to have *malleable* HPC applications, i.e., applications whose resource allocation can change mid-execution. Some manner of adaptation of reconfiguration mechanism is required in such an application in order to make use of extra resources mid-execution, or to avoid crashing if resources are removed. Works advocating malleable approaches include [100, 44].

Finally, some HPC applications have a dynamic behaviour at functional level. Adaptive mesh refinement is an example of such application, and is presented in details in the next section.

Concluding remarks In this section on HPC applications, we have presented common characteristics of HPC applications, and the technologies that are commonly used. HPC applications are often very complex and, because of their long lives and the evolution of hardware, require important development, maintenance and adaptation efforts. Among HPC applications, those with dynamic structure are the most complex.

2.1.3 Focus: Adaptive Mesh Refinement

Adaptive Mesh Refinement (AMR) is a computing technique which involves dynamic modification of a data mesh, in order to optimise both performance and precision.

Principle Many simulation applications rely on meshes as discrete representations of space. For example, the universe might be represented by a 3D grid, or a solid object might be represented by an object-shaped mesh. Simulation of a phenomenon on such a mesh typically involves attaching physical values to the vertices (e.g., temperature, electromagnetic field) and simulating the passage of time by computing step-by-step evolution of these values.

In such a context, the resolution of the mesh, i.e., the number of mesh elements per unit of simulated space, determines the precision of the simulation. Obviously, high-resolution meshes are more costly computing-wise, to the point where computing time (or memory) became a limit for high-precision simulations.

However, many physical simulations do not require high precision everywhere on their mesh. For example, density of matter in the universe might present high gradients inside galaxies (which requires high precision to properly simulate), but be very homogeneous in intergalactic space (which does not require high precision). Note however that high-gradient regions might move as time passes (e.g., a storm in a climate simulation).

In order to take advantage of this fact, a technique was introduced which consists in adapting the resolution of the mesh locally and dynamically, depending on gradient (i.e., on required

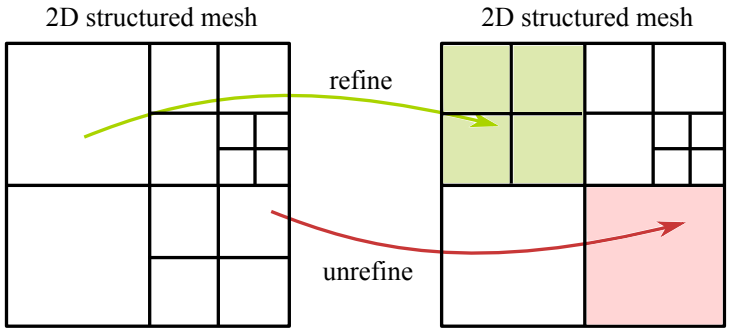


Figure 2.3: Illustration of mesh refinement and unrefinement on a structured 2D mesh.

precision). This approach allows to optimise the mesh resolution to have both high precision where it matters, and save on computing resources elsewhere. Figure 2.2 illustrates what mesh refinement means on an example unstructured mesh. This technique is called Adaptive Mesh Refinement, as it consists in changing the mesh refinement (i.e., its resolution) by adapting to simulated phenomenons.

Unstructured meshes For some applications, meshes are said to be *unstructured*, i.e., they can have pretty much any shape. For example, 3D models of objects are unstructured meshes. The mesh from Figure 2.2 is also unstructured.

Since unstructured meshes lack structural properties, they are difficult to work with. For example, it is difficult to have coordinates within a dynamic unstructured mesh. This makes it hard to provide application-agnostic support for such applications. However, there exist frameworks and libraries that targets unstructured mesh AMR applications, such as libmesh [66]. Such frameworks provide support for local reconfiguration and communication (e.g., communicating with neighbours in the mesh, local load balancing), along with global tools, e.g., for mesh partitioning.

Structured meshes In other cases, mesh are structured. The most common occurrence of structured mesh is 2D/3D grids in which cells can be further subdivided in 2D/3D grids respectively. Figure 2.3 presents an example of such a mesh: a 2D grid with local subdivisions. The number of times a cell has been recursively subdivided is called the *level of refinement*.

Since structured meshes have strong structural properties, it is easier to devise application-agnostic optimisations and tools for structured AMR. In particular, AMR framework that support structured meshes feature optimised data structures to represent the current state of the grid. Because of the recursive nature of structured AMR, these implementations, and related optimisations, are often based on quadtrees (for 2D AMR) or octrees (for 3D AMR) to represent the state of the mesh; those optimised tree structures are stored on every process and are used to locate neighbours. Examples of such frameworks include RAMSES [98], PARAMESH [73], and P4est [34].

One particular sub-category of structured AMR is meshes which follow the 2:1 rule [61]. The 2:1 rule is a constraint on refinement level which forces adjacent cells in a structured AMR mesh to have at most a difference one level of refinement. This means that a cell has at most 2 neighbours in a given direction (hence the 2:1 name).

Figure 2.4 illustrates one possible way to implement 2D structured AMR that follows the 2:1 rule. Such an application can be built from computing cells, each responsible for the computation on a square of data of fixed resolution. When refining, as illustrated on the figure, one cell is

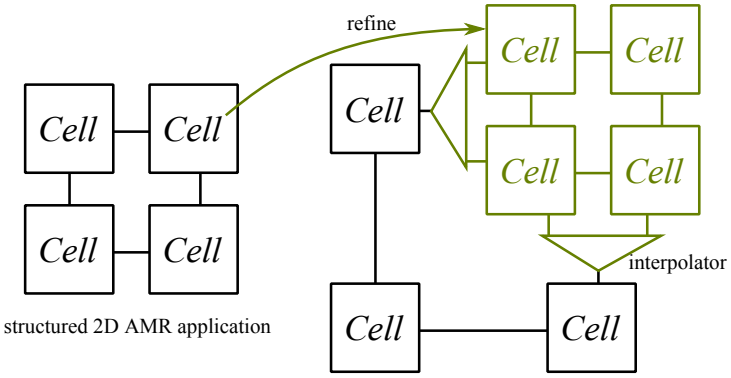


Figure 2.4: Application architecture before and after refinement of a 2D AMR structured mesh. Each cell is identical to every other, apart from the data it holds. All cells have the same resolution. Note the addition of interpolators to connect to adjacent cells with different levels of refinement.

replaced by four identical cells, and those cells are given interpolated data based on the data of the original cell before refinement. Interpolators are added between adjacent cells with different refinement levels, and are responsible for interpolating data so that each cells gets data with the correct resolution. Note that the 2:1 rule ensure that interpolators are at most two-to-one. While this way of implementing AMR is interesting because it builds upon reusable fixed-resolution cells, it also means refinement and unrefinement must modify the structure of the application.

Scalability issues Most traditional AMR approaches scale well up to a few thousands of cores, but struggle to scale further. This is due in most cases to an over-reliance on global representations and operations. For example, global partitioning steps in non-structured meshes are non-scalable because they require a global synchronisation. Another example is the use on quadtrees/octrees in traditional structured AMR implementations, which incurs a $O(P)$ storage cost on each node, and a $O(log\ P)$ time to locate neighbours, where P is the number of processes.

While it is possible to have AMR applications that scale to large processor counts [68, 33], it is particularly difficult. Recent works [68] advocates the use of more distributed approaches, with less reliance on global representations. Not only does such an approach address some of the programmability concerns raised by highly-complex traditional implementations, but it scales more easily at an algorithmic level.

Complexity AMR applications are extremely complex to write, maintain, and adapt. Indeed, not only is runtime adaptation difficult in itself, but AMR specifically needs to have very high performance (otherwise, a simple non-adaptive mesh could be used instead). Currently, the programmer of an AMR application needs to be skilled in distributed computation, non-trivial data structures (octrees/quadtrees), and the functional part of the application (whatever is actually simulated), which often involves non-trivial maths (differential equations).

The dedicated frameworks mentioned above help separating concerns and removing some of the optimisation work from the user. However, by taking away control from the user, they also limit application-specific optimisations, constrain conjoint use of other HPC technologies, and make code coupling more complex.

There has been studies that applied component-based software engineering techniques to structured AMR [85, 82]. These techniques proved particularly natural to represent recursive

refinement as recursive transformation of component assemblies. However, current HPC component models are ill-suited to runtime reconfiguration, and several problems remain unsolved.

Concluding remarks We have presented AMR in detail as it is a motivating and difficult example of application reconfiguration at runtime. Indeed, not only does AMR constitute an example of actual application-driven reconfiguration, but it also has high performance and scalability requirements, and is complex from a software engineering standpoint. For these reasons, AMR ticks all the boxes to be a motivating use case for the thesis.

2.2 Component Models

As discussed in Section 2.1.2, HPC applications are often very complex and need to be maintained and adapted throughout their long lives. Such applications could benefit greatly from software engineering techniques, easing high-level adaptation and separation of concerns. This section presents one such technique.

This section deals with component models, which are one of the domains at the intersection of which the subject of the thesis is situated. This section is divided into Section 2.2.1 which presents common component concepts such as connectors or hierarchy, and Section 2.2.2 which lists noteworthy models.

Definitions Before getting to these two sections, we propose to define what exactly component models are. There are various ways to define components, a few of which are listed below:

A first way to define components, which is the way we used in Chapter 1, is to start with the definition of *component-based software engineering* (CBSE). It is a software engineering paradigm which proposes to use composition as the primary means of code reuse. The entities that are being composed to implement CBSE are called *components*. While it is technically possible to use CBSE as a design paradigm in many languages, such as object-oriented languages, the best way to practice CBSE is to use a dedicated language or model. Such languages or models provide abstractions and concepts which are designed to help with composition. A model defining what a component is is called a *component model*, while a language designed to compose components is called an *assembly language*. The process of writing an application as a composition of components is called *component-based programming*.

Another way to define component models is to start by defining the components themselves. A classical definition has been proposed by Clemens Szyperski [96]: *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.* This definition is similar to the previous one in that it defines components as a unit of composition, but is more precise regarding what a component is and how it is defined. There are two important characteristics of components given in this definition which are worth highlighting. First, components must have *interfaces* and *explicit context dependencies only*; this is an important constraint which ensures components are easy to compose by explicitly declaring all its dependencies in an interface. Second, components are subject to *third-party composition*, which means components (and their interface) must be written while keeping in mind they might be reused by persons other than the programmer. Third parties can reuse components easily because all their dependencies are contained in their interface (by definition).

Variations exist on the second definition approach. One important distinction is whether components are primarily runtime entities, or design-time entities. Some works define components as runtime entities (e.g., [86]) which means that the units of composition used to build an application must correspond to an actual runtime entity which is called a component. Other

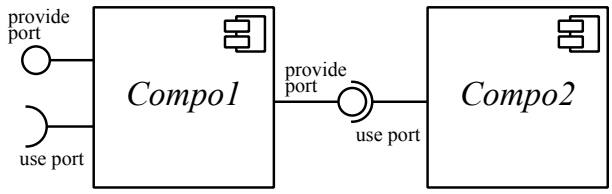


Figure 2.5: Example of components with use/provide ports. Components are represented by squares. The symbol at their top-right means they are software components.

works (e.g., HLCM [20]) present components as design-time entities which might not exist at runtime. For example, a component-based application might be compiled into a non-component-based language.

Properties The consequence of these definitions is that component-based approaches enforce, by construction, two important software engineering properties: reuse and separation of concerns.

Reuse is the process of using code written by third parties, which is a goal of components by definition. Reuse is an important property to minimize the cost of using third-party code, and minimize compatibility problems. Ideally, third-party components should be reusable as-is without modifying their code and with minimal extra "glue" code.

Separation of concerns (SoC) is the process of dividing an applications into parts (in this case: components) in such a fashion that each part deals with a small number of specific "concerns", i.e., aspects of the code related to a specific skill or characteristic of the application. In a context where an application is written by several persons, separation of concerns helps avoiding unnecessary conflict and helps minimizing the skill and knowledge required to write individual parts of the application.

2.2.1 Common Concepts and Features

The literature on component models is vast and many concepts have been proposed over the years. This sections reviews some of the most useful and common concepts used in existing component models. These concepts have been divided in three parts: first, essential concepts and definitions; second, common interface definitions; and then hierarchy and genericity which are two high-level ways to specify components.

Basic Concepts While, as illustrated by the definitions above, there are various ways to define components and component models, we provide a set of definitions of the basic concepts so that there is no ambiguity in the rest of the thesis.

Definition 1 (Component)**.** A component is a software unit, equipped with an interface, which is meant to be composed with other components using only the information contained in its interface.

Definition 2 (Component assembly)**.** A component assembly is the result of the composition of components.

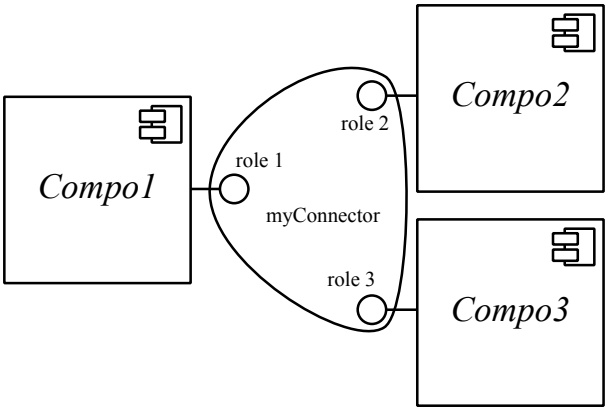


Figure 2.6: Example of components connected using a ternary connector with three roles.

Definition 3 (Component model)**.** A component model is a model which defines components, interfaces, and a way to compose components using their interfaces. In particular, composition semantics must be defined.

Interfaces The definitions given so far of components specify they must have interfaces but do not say what form these interfaces must have. There are various approaches to interfaces in the literature, two of which are presented below.

First, many component models use the notion of *ports* and *connection*, by analogy to physical plugs or electronic components. In such models, an interface is a list of ports, and components are composed by connecting their ports to ports of other components. A component model with ports provides a list of port types, details what connections are allowed between ports and what they mean.

The most common way to define ports is to have *use ports* and *provide ports*. A provide port means that a component provides a specific service, and a use port means that a component requires an external service to work. A use port is meant to be connected to a provide port, and an assembly is typically required to have no unconnected use port. Depending on the model, provide may be used by several use ports or not, and use ports might use multiple provide ports or not. Component models with use/provide ports are easy to represent graphically; Figure 2.5 shows how such connections are usually depicted.

A more generalist approach to ports is to have *connectors*. Connectors are devices used to connect components, but as opposed to the use/provide approach, connectors may be typed and more than two ports may participate in a connector. The type of a connector can be used to encode different sorts of port connexions. For example, two given ports might be connected either by a local connection (e.g., a C++ pointer to an object interface), or by a distant one (e.g.,, a message-passing connection). Non-binary connectors can be used to encode non-binary port interactions such as collective communications. Examples of models with connectors include BIP [12, 13] and HLCM [20]. Figure 2.6 shows one possible convention to represent connectors.

Hierarchy Some component models from the literature allows to implement components using component assemblies. Such component models are called *hierarchical*, as components can be implemented by other components, which can in turn be implemented by other components, and so on, leading to a tree-like hierarchy of components.

In hierarchical component models, components which are implemented by component assemblies are called *composite components*, while components with direct implementations are called *primitive components*.

Hierarchy in component models is a high-level feature which allows to group components into higher-level components, possibly until the entire application is a single composite components. The structure given by the nesting of components is a tree whose leaves are primitive components, and whose non-tree vertices are composite components which model some logical part of the application. Such a structure is useful, as it allows modelling the structure of the application at various levels.

Examples of hierarchical component models include FRACTAL [30], BIP [12], HLCM [20], SOFA2 [32].

Genericity In some cases, it may be useful to parametrise components with types or value so as to make them more generic. For example, let us consider a composite component which is implemented by a set of workers; it would be natural to parametrise such a composite component with the number of workers, instead of having to specify a composite component for each value. Some component models provide the necessary concepts to parametrise components in such a fashion. Such models are called *generic*, and improve genericity as parametrised components are more versatile [22].

Examples of component models featuring genericity include HLCM [20].

2.2.2 Examples of Component Models

Since components were first proposed by [74], tens of component models have been proposed, ranging from tried and tested industrial models focused on reliable implementation, to formal academical component models. Classifications of component models exist in the literature, e.g., [70, 37], but we provide our own overview below, which focuses on some noteworthy models, include some models relevant specifically to the thesis.

Fractal Fractal [30] is a hierarchical component model with runtime components, that has been the focus of many research efforts. Fractal is noteworthy for its *membrane* concept and for its various implementations.

In Fractal, components (which are runtime entities) are divided into a membrane, an interface, and an implementation. The membrane is a set of components responsible for managing the lifecycle of the component and for providing other non-functional services. The membrane separates the component from the rest of the assembly, and is notably responsible for connecting the interface of the component to its implementation.

Fractal is a component model in the truest sense, i.e., it is not linked to a specific implementation. Many implementations of Fractal exist, based on various underlying technologies. Implementations of Fractal include, for example:

- Julia [30], based on Java;
- ProActive [15], a distributed implementation based on active objects;
- FracNet [88], a .NET implementation.

Fractal is discussed further in the next chapter, as it is a reconfigurable model.

BIP BIP [12, 13] is an academical hierarchical model built in the first place to allow formal proofs on component behaviour and composition. While BIP is not reconfigurable, and has no direct relation to the thesis, it is presented here because it features several noteworthy concepts/

BIP stands for Behaviour, Interaction, Priorities, which corresponds to a division of component assemblies into three layers. Each BIP components features a model of its behaviour as a finite state automaton; these component behaviours collectively form the Behaviour layer. In addition, components have ports which can be connected using connectors, forming the Interactions layer. Connectors in BIP are noteworthy in that they are built from a connector algebra, which allows the encoding of a variety of collective synchronisations. Finally, BIP assembly feature a Priorities layers composed of rules determining which inter-component interactions should be triggered first.

Another noteworthy feature of BIP is that it allows to fusion of connected components into larger components. The automaton-based behaviours connected through composite connectors can be compiled into larger automatons with an equivalent behaviour.

HLCM HLCM [20] is a hierarchical and generic academical component model, which is meant to be "compiled" into a lower-level model or language, through a semi-automated transformation phase.

HLCM was initially designed for HPC applications, as a way to reconcile high-level features (provided by HLCM) and performance (to be obtained after transformation to a low-level language). The high-level language in which HLCM assemblies are written is called HLA (High-Level Assembly). HLA is hierarchic, generic, based on connectors, and allows *abstract* components and connectors (i.e., components/connectors whose implementation has not been chosen yet).

HLA assemblies are compiled into LLA (Low-Level Assemblies) which can then be transposed to various backend languages. Currently implemented backend languages are L2C, described below in more details, which is a zero-overhead low-level language designed for HPC, and Gluon++ which is a version of L2C based on Charm++ (a HPC language described in more details in the next chapter in Section 3.3.2). The transformation process (from HLA to LLA) consists in making individual specialisation decisions (e.g., choosing the implementation of an abstract connector or component, setting the value of a parameter, replacing a composite by its implementation), until the assembly is fully primitive.

HLCM is noteworthy because its transformation phase theoretically allows some optimisation or adaptation decisions to be taken automatically during the transformation phase. HLCM was developed in the Avalon team, as was L2C, and was a strong inspiration for the contribution presented in Chapter 6.

HPC component models Some component models have been specifically designed for use in HPC applications, or distributed applications in general. The CORBA Component Model [24] (CCM) and the Grid Component Model (GCM) [14] are notable examples of distributed models. However, they generate runtime overheads [102] that are acceptable for distributed application but not for HPC. The Common Component Architecture [41] (CCA) is the result of an US DoE effort to enhance composability in HPC. However, CCA is mainly a process-local standard that relies on external models such as MPI for inter-process communication. As a consequence, such interactions do not appear in component interfaces.

The Low Level Components [21] (L2C) is a minimalist HPC component model built, designed to have a negligible overhead at runtime. L2C is built on C++, MPI and Corba, which are common HPC technologies. Low overhead is achieved by removing all component-related infrastructure after instantiation of the components: two L2C components are, at runtime, C++ objects directly connected through C++/Corba connections or MPI communicators. As described in [21], L2C has been successfully used to describe a stencil-like application with performance similar to native implementations.

Non-HPC industrial models The more widely-used component models in the industry seldom provide a model in the formal sense, and instead focus on a specification and an implementation, often based on some existing widely-used language or technology. For example, JavaBeans [52] and OSGi [4] are based on Java, COM [26] was developed by Microsoft, and KobrA [8] is based on UML. Since those models are far from the subject of thesis, we do not present them in detail.

2.3 Conclusion

In this section, we have presented high-performance computing and component models, which make up the necessary context to understand the problem of the thesis and the contributions.

First, we have presented high-performance computing, notably HPC hardware architectures and HPC applications. We have introduced the main challenges of HPC which are scalability and managing the complexity of applications, and have presented Adaptive Mesh Refinement (AMR), a computing technique which serves as a motivating example for the contributions of the thesis.

Second, we have presented the general concepts and literature around component models. In particular, we have provided several definitions of "component"; we have presented common component concepts such as ports, connectors or hierarchy; and we have presented a handful of noteworthy component models including some models dedicated to HPC such as L2C or CCA.

Chapter 3

Related Works

Contents

Application reconfiguration in the literature happens in a variety of contexts for a variety of reasons. Several very different approaches to reconfiguration exist that can vary widely in terms of goals and methods. Examples of reconfiguration are numerous and include, for example, updating the code of a web application without stopping it, reconfiguring a P2P network in the presence of churn, or adapting an embedded application to its environment.

As reconfiguration is the primary focus of the thesis, we consider the reconfiguration literature to be related works, as opposed to HPC and component concepts which are considered context. This chapter presents related work around reconfiguration, whether it is component-based or not, as ideas relevant to the thesis may come from non-component literature.

First, Section 3.1 gives a broad overview of reconfiguration approaches sorted by abstraction level, and discusses which are relevant to the thesis. Second, Section 3.2 discusses issues related

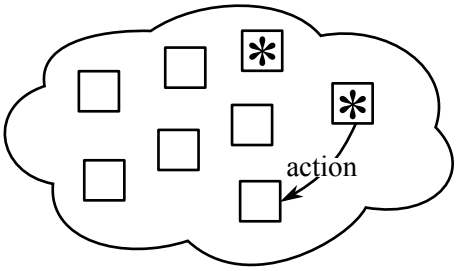


Figure 3.1: Application-level reconfiguration. The cloudy shape denotes the reconfigurable application, while the stars denote where reconfiguration logic is located, and the squares denote parts of the application.

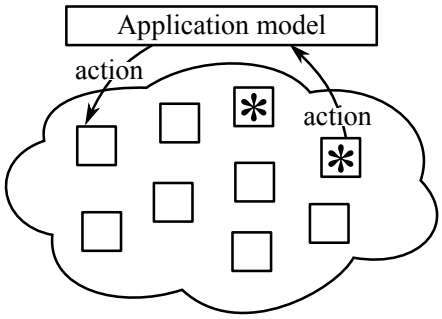


Figure 3.2: Model-level reconfiguration. The cloudy shape denotes the reconfigurable application, while the stars denote where reconfiguration logic is located, and the squares denote parts of the application.

to the specification of application transformations, and strategies from the literature to address these issues. Third, Section 3.3 discusses issues related to the execution of reconfigurations, such as consistency problems and application locking. Finally, Section 3.4 presents actual reconfigurable component models from the literature, and analyses their limitations regarding our specific HPC use case.

3.1 Brief Taxonomy

This section provides a simple coarse-grain taxonomy of reconfiguration approaches. There exist a variety of approaches in the literature that we have divided into broad categories, of increasing abstraction: application-level reconfiguration (Subsections 3.1.1), model-level reconfiguration (3.1.2), and autonomic reconfiguration (3.1.3). The relevancy for the thesis of each of these broad approaches is discussed in Section 3.1.4.

3.1.1 Application-Level

A first, seemingly easy, *ad hoc* approach to reconfiguration consists in writing the reconfiguration logic at the same level as the rest of the application. The programming model used for the application likely features the primitives which allow reconfiguration, things like object destruction and creation at runtime. Those primitives can be used by the application to reconfigure itself. We call this approach *application-level*, as reconfiguration is expressed and carried out at the same level of abstraction as was used to write the application itself. Figure 3.1 shows a simple diagram illustrating this kind of reconfiguration.

Example: *ad hoc* reconfiguration Some applications perform reconfiguration directly without relying on a third-party technology such as a framework or a language. We call this kind of reconfiguration *ad hoc* as it does not make use of reconfiguration-dedicated abstractions or technologies. Examples of such applications include AMR applications that do not use a dedicated framework such as the pkdgrav/gasoline computational astrophysics codes [95].

Discussion This approach is the most general as it does not make any assumption regarding how reconfiguration is decided or carried out. Applications with unusual reconfiguration patterns

might work, by default, only at that level of abstraction. In addition, this kind of approach does not restrict the kind of optimizations that can be performed, which can be useful in a HPC context to get maximum performance.

However, this approach is limited from a software engineering standpoint. Indeed, the absence of model-level enforced structure means that reconfiguration logic has to be *ad hoc*. One notable implication of relying on *ad hoc* reconfiguration is that reuse of third-party reconfiguration code is difficult, as no general framework for reconfiguration exists.

3.1.2 Model-Level

Another possible approach to reconfiguration is to have an abstraction level above the application itself, which provides a model of the applications parts and ways to rearrange those parts. The reconfiguration logic itself might or might not be part of the reconfigurable parts, but all reconfiguration operations are expressed at model-level instead of application-level. We call this approach *model-level control of reconfiguration* as it relies on a dedicated model to express reconfiguration. Figure 3.2 shows a simple diagram illustrating this kind of reconfiguration, to be put in opposition with the diagram from Figure 3.1.

This approach brings several advantages. By relying on an intermediate model dedicated to reconfiguration, reconfiguration code is easier to write from the point of view of the developer. A reconfiguration-dedicated model is also much more likely to be easy to analyse for a program (e.g., for model-checking or automatic proof) than a general-purpose programming language (which is often not analysable at all). Finally, because such an approach sets conventions regarding how reconfiguration is performed, reusing third-party reconfiguration code is easier, as hypotheses can be made regarding how it operates.

Example: reconfiguration DSLs A *Domain-Specific Language* (DSL), is a programming language tailored for a specific task, as opposed to a general purpose language. DSLs trade their generality for specialized abstractions, and are typically easier to use and easier to analyse than general-purpose languages.

There exist reconfiguration DSLs, that is, DSLs made specifically to express reconfiguration in dynamic applications. By definition, reconfiguration DSLs rely on an application model dedicated to reconfiguration, and as such constitute model-level reconfiguration.

A good example of a such a reconfiguration DSL is FScript [39], which is used to reconfigure FRACTAL component assemblies. FScript provides abstractions and tools that help reconfiguration, such as the FPath tool, which helps with pattern-matching in FRACTAL assemblies. FPath is discussed in more details in the section dedicated to component models below (Section 3.4).

Example: specialised frameworks Other examples of model-level reconfiguration include frameworks specialised for a certain kind of dynamic applications. For example, AMR frameworks such as RAMSES [98] belong to this category. By providing abstractions tailored for a very specific type of applications, such frameworks are particularly efficient for writing such applications, but are also less generic which might be problematic when combining several forms of reconfiguration in a single application.

3.1.3 Autonomic

A possible next step towards further abstraction is to use concepts from *autonomic computing* to get a general framework for reconfiguration. Application reconfiguration can be achieved using

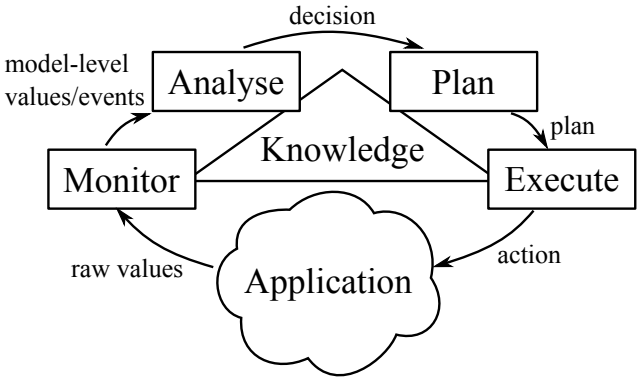


Figure 3.3: The overall structure of a MAPE loop.

autonomic loops, that is, closed control loops which use principles from autonomic computing to decide reconfigurations.

A widely-used autonomic loop is the *MAPE* loop, which divides reconfiguration into four distinct steps: Monitor, Analyse, Plan, and Execute. Figure 3.3 presents the overall MAPE loop and the four steps in question. These steps are:

- **Monitor:** some part of the reconfiguration infrastructure is responsible for monitoring the application, and getting relevant data to the next step.
- **Analyse:** the data from the Monitor step is analysed to determine if a reconfiguration must take place or not.
- **Plan:** if a reconfiguration has been decided at the previous step, a model-level reconfiguration plan is devised and transmitted to the next step for execution.
- **Execute:** this last step is responsible for executing a model-level plan transmitted to it, i.e., to actually act on the running application.

A MAPE loop can either rely on a centralized application representation, which is an inherently non-scalable approach, or can be distributed [103] (e.g., several local MAPE loops running concurrently) which poses specific problems such as consistency across concurrently-managed subsets of the application.

While MAPE provides a general-purpose approach to reconfiguration, in many cases not all its steps are required and performance may benefit from skipping them. For example, the Monitor and Analyse steps are unnecessary if reconfiguration is triggered directly by the application (as is the case for AMR). Also, in the case where reconfiguration is easy to predict, the Plan step may be replaced by pre-planned reconfigurations.

3.1.4 Discussion

While application-level reconfiguration might be natural in a HPC context, as it would ease the optimisation of applications, it does not help with the software engineering issues identified in Chapter 1 and Chapter 2. Notably, reuse of third-party code is difficult without a reconfiguration framework.

MAPE loops are a useful formalism which brings insight regarding the important facets of reconfiguration. However, actually running the MAPE steps in a reconfiguring program might not be the most efficient approach, especially as not all steps may be relevant to all applications. For example, the AMR use case (see Section 2.1.3) already provides the M (Monitoring) and A (Analyse) steps as functional parts of the application. In a HPC context where performance is paramount, these extra steps could be problematic. The P (plan) and E (Execute) steps

raise, however, several open issues which are relevant to the thesis. The rest of the chapter does not focus on autonomic loops, but uses the MAPE terminology to help structuring the various issues.

In the rest of this chapter, we focus on issues related to model-based reconfiguration.

3.2 Specification and Related Issues

Now that a brief overview of reconfiguration approaches has been presented, enough context as been laid out to present more specific problems and sub-problems related to reconfiguration.

This section focuses on issues related to reconfiguration specification ans selection (the "Plan" step, in MAPE terms).

Reconfiguration specification is the process of specifying the actual transformation that must take place in a reconfigurable application. Various approaches to this problem exist with various software engineering properties, and model-level analysability properties.

3.2.1 Code-based

A first, natural approach to reconfiguration specification is to write a reconfiguration code. For example, reconfiguration can be described using the language that is used for the functional parts of the application. Alternatively, it can be written in a dedicated reconfiguration language.

A first category of code-based approaches is those with entirely *ad hoc* reconfiguration. Those approaches are typically code-based as they rely on no reconfiguration framework. These approaches have poor software engineering properties, notably reuse, as they have no established framework to rely on to provide compatibility with third-party code. In addition, general-purpose code is a difficult thing to analyse (e.g., for model checking).

There exist in the literature languages dedicated to reconfiguration. The FScript DSL presented in Section 3.1.2 is an example of one such language. These languages are better than pure *ad hoc* approaches, as they are easier to use and to analyse.

3.2.2 Formal transformations

Another possible approach is formal reconfiguration specification. It consists in specifying reconfiguration as formal objects which might be quite different from typical code.

Graph transformations A first example of formalism which can be used to specify reconfiguration is *graph transformations*. Graph transformations are formal objects which describe modifications of the structure of a graph. While there exist several formalisms, the following formalism is common in software applications:

Definition 4 (Graph rewriting rule)**.** A graph rewriting rule is of the form $(L \leftarrow K \rightarrow R)$, which is composed of:

- A left-hand side L.
- A gluing graph K.
- A right-hand side R.
- A graph morphism[a] from K to L (denoted by an arrow in the rule).
- A graph morphism from K to R (denoted by an arrow in the rule).

[a] Defining all graph concepts properly is far beyond the scope of this section. A graph morphism is basically a mapping between two graphs which preserve structure (i.e., edges between nodes).

The left-hand side corresponds to a pattern to which the rule can be applied. The right-hand side corresponds to what the left-hand side is replaced by, when the rule is applied. The gluing graph and the two morphisms specify the relationships between vertices from the left-hand and right-hand side. The gluing graph can be thought of as a namespace used to identify certain vertices from the left and right-hand sides.

Such a rule can be applied to a sub-graph matching the structure of the left-hand side, and verifying certain conditions. The semantics of graph transformation rule application define not only how to replace the target sub-graph with the right-hand side, but also how to reconnect the newly added sub-graph so as to match the connections of the removed target sub-graph.

Graph transformation languages If one wants to use graph transformation in practice (i.e., in a computer program), some programmer-friendly way of specifying transformation is required. Graphical tools exist, but such tools do not necessarily scale well to complex use cases, and are pretty far from programmers' text-based practices. Text-based graph transformation languages and tools also exist; examples include the VIATRA2 language [101], DSLtrans [11], the GReTL language [60], or delta-modelling [54].

Target selection Such rules are often used to perform graph rewriting, e.g., for model transformation. In such a context, a set of rules and a starting graph are given; pattern-matching is performed to find valid rule applications until no more rule application is possible. Consequently, most frameworks supporting graph transformation use such pattern-matching as the default way to trigger transformations and select a target subgraph. However, since pattern-matching is a costly, it might be better to avoid such a step when it is not required (e.g., if transformations targets are easy to specify directly) to save on computing time.

3.2.3 User-Driven

Another important issue in reconfiguration control is what we call *user-driven reconfiguration*. User-driven reconfiguration is the process of having parts of the reconfiguration process decided by humans.

User-driven reconfiguration typically occurs with low frequencies, possibly as low as once every few months or years. Indeed, humans cannot take decisions as fast as computers. This means that user-driven reconfiguration does not concern itself with optimizing the reconfiguration process to gain microseconds, but is more concerned with other issues.

DSU One typical example of user-driven reconfiguration is Dynamic Software Updating (DSU). It is a common reconfiguration problem which deal with updating a software (i.e., update its code) without stopping it. While DSU can be automated to some degree to rely little on human intervention, it always retains the characteristics of user-driven reconfiguration: external decision and slow characteristic time. In DSU, service continuity and consistency across code updates are paramount.

DSU was formalised in the early 2000s with works such as [58, 57]. Recent studies interested in DSU include, for example, the Coqcots/Pycots component model [31].

Discussion User-driven reconfiguration is mostly irrelevant in an HPC context. Indeed, HPC applications are very complex and would be very difficult to reconfigure without raising consistency problems. HPC applications usually undergo a lot of testing at small scale before being executed at large scale; changing the code of an application running on a large-scale HPC architecture would be a very complex undertaking, even if it was feasible to modify the code on

the fly without problems. Also, issues continuity of service is irrelevant in HPC as only the final result usually matters. For these reasons, we do not consider DSU-like constraints and goals in this paper.

3.2.4 Discussion

For our purposes, i.e., to design tools for HPC programmers, a code-based approach would present several advantages. First, it would be closer to what HPC programmers are already familiar with (i.e., traditional programming languages), which is important for adoption, especially other unusual concepts (components) are already involved. Second, a code-based approach would make existing HPC optimisations easier to implement, as such optimisations have typically been devised in the first place with traditional programming languages.

However, code-based approaches have two important drawbacks: it is difficult to define their semantics unambiguously, and code is a difficult thing to analyse. In addition, formal specification such as graph transformations is relevant to reconfiguration in component models, as component assemblies have graph-like structures.

Finally, user-driven reconfiguration is mostly irrelevant in a HPC context, we thus do not consider it in the rest of the chapter.

3.3 Execution and Related Issues

Another important issue with reconfiguration is the execution of a reconfiguration plan. Indeed, once a reconfiguration has been decided and specified, its execution might be far from trivial. Changing the structure of an application that is currently executing can lead to various consistency problems, such as incorrect references, invalid call stacks, inconsistent application representation, deadlocks...

Various approaches to address or circumvent this issue exist in the literature, three of which are presented below in order of increasing complexity. Section 3.3.4 discusses a related issue which is application representation at runtime.

3.3.1 Global Synchronization

A simple solution to the execution problem consists in having the entire application stop every time there is a reconfiguration. That way, all the problems related to concurrency are safely avoided: application representation is easy to maintain, the application state cannot change while it is being reconfigured, and only one reconfiguration can happen at a time.

This approach has, however, an important drawback: global synchronization does not scale. Not only is stopping a whole application for every small reconfiguration inefficient, but global synchronization itself is known not to scale. For this reason, global synchronisation approaches are not compatible with HPC performance and scalability requirements.

3.3.2 Appropriate Execution Model

Another way to avoid reconfiguration execution problems is to use an execution model with which it is easy to stop parts of the application and manipulate the execution state. If a part of a running application can be easily stopped and isolated from the rest of the application, then safe reconfiguration is only a matter of making sure the execution state is in order before releasing control.

Example: active objects For example, *active objects* [2] are one such execution model. An application built using this approach is composed of active objects which communicate with each other using asynchronous method calls; each active object has a "mailbox" of incoming unprocessed method calls in which it selects the next non-blocking method to execute. Depending on the precise constraints of what methods are allowed to do and how messages work, locking can be easy to perform (wait for the current non-blocking call to end then lock the object). In addition, such models provide an explicit execution state (the mailboxes) that is easy to copy and manipulate.

Charm++ [64] is an example of a language which uses this approach, and is of particular interest to us at is targets HPC applications. Charm++ is built on top of C++, and provides a concept of *chares*, which are essentially active objects [7]. Charm++ takes advantage of its execution and programming models to provide high-performance migration [35] and load-balancing [106] capabilities, illustrating the ease of reconfiguration with active objects.

Other examples of active object literature include works around Proactive, a distributed middleware targeting grid environments, which is notable in the context of the thesis for being used in a FRACTAL implementation [15].

Example: agent-based systems *Agent-based software engineering* is a software engineering paradigm that proposes top build systems as a collection of *software agents*, i.e., application parts which communicate with each other with an expressive dedicated communication language [51].

With appropriate hypotheses on the behaviour of agents, general-purpose reconfiguration algorithms have been devised [43]. Note in particular that [36] discusses how their reconfiguration algorithms relies on specific properties of the agents' execution model, such as the atomicity of agent computation.

Discussion While having such execution models leads to reliable reconfiguration, the constraints imposed by the execution model might be problematic for some applications. For example, a lot of the HPC literature relies on executions models with blocking operations (such as can be found, e.g., in MPI); while it would be possible to rewrite functionally equivalent programs with agents or active objects, those programs would not necessarily be equivalent performance-wise and known optimisations might be difficult to apply to them. The non-blocking/atomicity hypothesis that can be found in some agents/active objects models, is a strong constraint on programming style, which considerably limits the possible synchronisation patterns inside applications. Although such models can be used in HPC—as exemplified by Charm++—they constrain programming style and require specific algorithms and optimisations.

3.3.3 Locking

When synchronisation is not global, and when the underlying execution model is not favourable to easy reconfiguration, then reconfiguration becomes very difficult, and typically relies on *stopping* and *locking* parts of the application before reconfiguration.

For example, in a call-stack-based object-oriented application (e.g., a C++ application), removing or modifying an object without care can lead to all manners of problems such as incorrect pointers or invalid call stacks. A possible approach then is to *stop* the target object before reconfiguration so as to be sure its modification does not break anything, i.e., make sure the object is executing no code, make sure that no call stack contains the object, and prevent further calls to the object. While stopping an object is indeed a way to ensure that reconfiguration causes no problems, stopping the object might either be difficult or cause deadlocks in the application.

In component models the process of stopping components before reconfiguration is called reaching a *quiescent state* [67]. Examples of work which deal with reaching a quiescent state include [55].

Apart from the component literature, there exists a rich literature on deadlock prevention and detection. However, component models introduce the added constraint that components are blackbox entities which only provide guarantees through their interfaces. While interfaces can provide behavioural information (i.e., a model of a component's behaviour) through their semantics [37], they contain less information than full component code, and vary from component model to component model. This means that works which rely on direct code analysis (e.g., [63, 27]) have to be adapted to work with the level of behaviour information provided by interfaces.

3.3.4 Application Representation at Runtime

The last reconfiguration issue to be presented in this section is the problem of *application representation at runtime*. In reconfiguration approaches relying on models, the application model needs to be consistent with the actual application. Maintaining this consistency across execution, especially in the presence of concurrent reconfiguration, can be a difficult problem. Consequently, various approaches exist in the literature regarding how such a representation should be structured and maintained.

Global Representation The easy approach to the reconfiguration problem is to have a single representation of the full application, and to use that representation for all purposes. Such a representation is said to be *global*, as it encompasses the whole application.

Such a representation is useful for tools and algorithms that analyse the application structure, as they have access to a complete view of the application. Examples of such tools and algorithms include reconfiguration proof, as is for example performed in Coqcots/Pycots [31].

To our knowledge, existing global representation approaches are fully centralised, i.e., the application representation is kept at a single place and reconfigurations are performed sequentially. While being centralised is not technically required for a global representation, it is an easy way to avoid consistency problems by centralising everything. However, such an approach is inherently non-scalable, as already discussed in Section 3.3.1 about global synchronisation.

Local Representation Another approach to application representation at runtime is *local representation*, i.e., having several separated representations of parts of the application. For example, in a distributed application, there can on each process a representation of the part of the application located on this process.

One example of this approach is hierarchical component models with controllers, such as FRACTAL. These component models are discussed further in Section 3.4.2.

Local representation is better than global representation from a scalability/performance point of view, as each local representation can be managed independently, leading to more concurrency. However, reconfigurations which cross the boundaries of the local representations are problematic, as locking several local representations can lead to deadlock or performance problems, and as breaking them down into local reconfigurations can lead to inconsistent behaviours.

Moreover, most approaches based on local representation constrain the shape of the local representations (e.g., hierarchical component models are limited to composite components). Exception to this include approaches based on aspects, such as aspectual components (e.g., FAC[81]), in which custom scopes can be specified through the use of aspects.

Discussion Overall, global representation is to be avoided in the context of the thesis as it is inherently non-scalable. Local representation seems more appropriate but it raises execution problems, and many existing approaches do not allow arbitrary partition of the application into local representations.

3.3.5 Concluding remarks

In this section we have presented existing issues, and literature regarding reconfiguration execution.

Reconfiguration execution can lead to consistency and synchronisation problems, which can be either circumvented by relying on global synchronisation / a favourable execution model, or must be addressed in their most difficult form otherwise. Unfortunately, HPC scalability constraints excludes the use of the global approach, and "nice" execution models means many algorithms and optimisations from the literature could not be used.

This section also discussed the related issue of assembly representation at runtime, and concluded that while local representation is obviously the way to go for scalability, there exist no catch-all way to decide the scope of local representations.

3.4 Reconfigurable Component Models

This section focuses specifically on *reconfigurable component models* which are the actual subject of the thesis. These models are, more than the reconfiguration literature evoked so far, directly competing with our contributions and deserve thus a section of their own.

3.4.1 *Ad hoc* reconfiguration

Many component models offer basic low-level reconfiguration primitives (i.e., create, destroy and connect operations) through APIs or dedicated components, without higher-level reconfiguration support. In practice, many component model implementations based on objects offer these operations whether or not they appear in the model itself.

In L2C [21], *ad hoc* reconfiguration can be done by using the deployment infrastructure. Indeed, L2C programs are deployed using an dedicated deployment assembly, responsible for instantiating and connecting the components. Among other things, this deployment assembly contains one *Host* component per MPI process which is responsible for local instantiation, and references, and which uses a basic namespace. One can connect user-defined components to these *Host* components, and use the instantiation/reference services they provide to reconfigure a running application.

Languages in which component are runtime entities, e.g., CCA [19], often provide basic reconfiguration primitives,e.g., as methods of the component itself (e.g., by inheriting from a generic component class). These primitives can be used for reconfiguration, and present the advantage of being fully local (i.e., no reliance on any form of global service).

Although it is possible to program reconfiguration in these implementations, and although it may be well-suited to some cases, this approach has two drawbacks. First, it does not provide any tool or model to help with the reconfiguration logic. In a distributed case, performing non-local reconfiguration can be difficult. Second, this approach does not allow third-party reuse in a concurrent context. Indeed, since there are no conventions or rules regarding synchronisation between concurrent reconfigurations, combining two third-party reconfiguration codes would break the application.

3.4.2 Controllers

Component models with hierarchical runtime components, e.g., FRACTAL [30] or SOFA2 [32], often provide life-cycle and reconfiguration services through components called *controllers*, attached to functional components. Controllers can be used to reconfigure the assembly, and

Controllers-based approaches present one main advantage compared to *ad hoc* approaches presented above: they solve the synchronisation/third-party reuse to a great extent, by providing a central synchronisation point (the controller) for reconfiguration on a composite. Indeed, since the controller is responsible for performing reconfigurations, two concurrent reconfigurations on the same component have to go through the same controller, which can at the very least make sure they are executed in sequence. In addition, controllers can easily provide and maintain a representation of their implementation if they are composite.

These models can be used to implement efficient distributed reconfigurations while preserving consistency during execution. However, the reconfiguration scope is restricted to composites which does not fit the structure of many HPC applications. For example, AMR will require neighbour-to-neighbour synchronization on a mesh structure which makes it complex to manage inter-branch communications when mapped on a tree-like component hierarchy.

3.4.3 Global Reconfiguration

Another approach to reconfiguration in component assemblies is *global reconfiguration*, i.e., relying on global synchronisation as discussed in Section 3.3.1, and global representation as discussed in Section 3.3.4.

A first example of models which use this approach is reconfiguration DSLs as discussed in Section 3.1.2. Examples of such languages include FScript [39] for FRACTAL assemblies, and GScript [105] for COM/Corba. These DSLs provide high-level features to ease reconfiguration, such as, for example, the use of the FPath [40], query language which helps with navigation and pattern-matching in FRACTAL assemblies. Such DSLs might support distributed systems, such as is the case for FScript [17, 86], although in this case, representation and execution are still centralised.

Any existing model which relies on full assembly analysis uses the global approach. Examples of such models include the already mentioned Pycots/Coqcots model [31] which focuses on proved DSU (see Section 3.2.3).

As previously discussed in Sections 3.3.1 and 3.3.4, this approach addresses representation and execution problems, as everything is centralised. However, while this approach makes sense for moderate-scale distributed systems, HPC is now targeting very large scales to a point where global synchronization becomes too costly. Applications such as AMR do not inherently require global synchronization since reconfiguration is functionally local. In such cases, centralized reconfiguration introduces a non-scalable overhead which is not functionally necessary.

3.4.4 Concluding remarks

Overall, none of the approaches in the literature completely fits our HPC use case. Indeed, *ad hoc* reconfiguration allows all manners of optimisations and concurrency but is poor from a software engineering standpoint, as it is a very low-level approach, and as it prevents third-party reuse. Global reconfiguration is good from a software engineering standpoint but is inherently non-scalable, and thus completely unfit for HPC usage in HPC. Controller-based approaches are the closest to a solution to our problem, as they allow concurrency while having good software engineering properties. However, those solutions rely on composite components as their unit for local representation, which does not fit all HPC application (e.g., AMR).

3.5 Conclusion

In this chapter, we have presented an overview of software reconfiguration, evoked various related issues, and dwelt more specifically of reconfigurable component models, which are the subject matter of the thesis. Each section had a section discussing what parts of the literature are, or are not, relevant to our specific HPC context.

In Section 3.1, we have presented a first coarse-grain classification of reconfiguration approaches based on abstraction level. We have concluded that what we call *model-level* is the most suited to our problem, as application-level is poor from a software engineering standpoint and autonomic approaches are too constraining as-is for HPC.

In Sections 3.2 and 3.3, we have presented various issues related to application reconfiguration, and existing solutions in the literature for each of them. Regarding reconfiguration specification, it seems that formal approaches are the most promising from a software engineering point of view. Regarding reconfiguration execution, it seems that the stopping problem is difficult to avoid in a HPC context, although it is a very complex issue. Chapter 5 discusses this problem more in-depth. Regarding application representation at runtime, we have discussed scalability problems posed by global approaches, and how local representations are required.

In Section 3.4, we have presented existing component models and shown that none of them was fully suited to use cases such as the AMR. Indeed, global approaches are not scalable and non-global approaches either have good engineering properties but do not allow the desired concurrency, (as is the case for controller-based approaches), or allow any level of concurrency but are low-level and do not allow third-party reuse (*ad hoc* reconfiguration).

Part II

Contributions

Chapter 4

The DirectMOD Component Model

Contents

This chapter presents our main contribution: the DirectMOD formal component model. DirectMOD takes the form of a formal definition of its syntax and semantics. The goals of DirectMOD are to be scalable, to ease reuse of reconfiguration code and to separate the safe execution of reconfiguration code from other concerns.

This chapter begins by presenting a simple and generic preliminary component model. This preliminary model serves as a base for DirectMOD and for the contribution of Chapter 5. Then, the DirectMOD model itself is presented with its syntax and semantics. A series of examples, inspired by AMR or stencil applications, are used to illustrate the model. Finally, the properties of the model are discussed, and the model is evaluated using an implementation called DirectL2C. This implementation was used to implement an AMR benchmark, whose code-level properties and performance are presented and discussed.

The model presented in this chapter is a slightly reworked and improved version of the model published in [69].

4.1 Preliminary Model

In this section, we present a simple component model which serves as a basis for two of our contributions. Since all these contributions deal with component models one way or another, it makes sense to factorize some definitions instead of providing several fully disjoint models.

In addition, the model we provide can be seen as a generalization of the L2C and CCA models. This means that our different contributions can be used with those existing HPC technologies. Our own implementation, DirectL2C, is an example of such a use (see Section 4.3.2).

Moreover, this generic model has a call-stack-based semantics which models the behaviour of HPC languages such as C++/Fortran, which also feature call stacks. Targeting these languages is important for implementation purposes, since they are the most commonly supported by HPC libraries and platforms.

The generic model we propose has the following characteristics:

- it is not reconfigurable (L2C and CCA provide API-based reconfiguration, see Chapter 3, but we are interested in providing more powerful reconfiguration);
- it is not hierarchical (in order to keep the complexity of our first models manageable);
- it has point-to-point connections (like in L2C and CCA);
- it has a call-stack-based semantics (like C++ connections in L2C);
- it provides a resource model (similar to L2C processes).

4.1.1 Syntax

In order to define a component model, we need to define *components* as well as a way to *compose* them together to form a *component assembly*. Also, since it is one of our goals for this specific model, we need to provide a simple resource model.

Notation Conventions First, let us introduce a set of conventions for our notations. These conventions hold for all chapters.

- Operators are denoted with full caps, e.g., $INSTANCE$, USE;
- pre-defined sets (i.e., sets given by hypothesis) are denoted with rounded uppercase letters, e.g., $\mathcal{N}$, $\mathcal{C}$;
- user-defined sets (e.g., sets of instances in an assembly) are denoted with Latin uppercase letters, e.g., C, P;
- functions are denoted with lowercase Latin letters, e.g., o, r;
- user-constructed objects (e.g., assemblies) are denoted with lowercase Greek letters, e.g., α, β;
- other variables (e.g., inside a definition) are denoted with explicit lowercase words or abbreviations, e.g., $name$, ref;
- utility functions and sets (mostly notations introduced for readability) are denoted with words whose first letter is uppercase, e.g., Ref, $Name$.

Pre-defined sets First, we assume the existence of a set of components and a set of resources, which represent the available code (as instantiable components, i.e., component types, on the shelf) and hardware respectively. We denote $\mathcal{C}$ the set of available component types and $\mathcal{R}$ the set of available resources (eg, CPU cores, computing nodes, ...).

In addition, let $\mathcal{N}$ be a set of names (i.e., identifiers). These names are used to uniquely identify components and ports.

Note that those three sets ($\mathcal{C}$, $\mathcal{R}$ and $\mathcal{N}$) can be infinite. This allows unbounded behaviours such as: unbounded number of components at execution and dynamic resources.

Component instances A component instance is built from a type and a name using the $INSTANCE$ binary operator, as defined here:

Definition 5 (Component instances)**.** The set of component instances using the component shelf $\mathcal{C}$ and the nameset $\mathcal{N}$ is given by:

$$Instances(\mathcal{C}, \mathcal{N}) = \{INSTANCE(name, type) \mid name \in \mathcal{N}, type \in \mathcal{C}\} \tag{4.1}$$

In this model, components are instances of the components types, i.e., instances of the components in $\mathcal{C}$. The components on the shelf are encapsulated code and can thus be instantiated multiple times. In addition, each component has a name which is used to identify it.

In the rest of the thesis, instances of component types are simply called *components*.

Note that component types are not described. There are assumed to exist ($\mathcal{C}$) but it is not the purpose of this chapter to model anything regarding component types.

Ports In order to connect components together, i.e., to compose them, those components must have *interfaces*. In DirectMOD, interfaces are *ports*, which are constructed using the USE binary operator and the $PROVIDE$ unary operator, as defined here:

Definition 6 (Ports)**.** The set of ports using the nameset $\mathcal{N}$ is given by:

$$Ports(\mathcal{N}) = \{USE(name, ref) \mid (name, ref) \in \mathcal{N}^2\} \cup \{PROVIDE(name) \mid name \in \mathcal{N}\} \tag{4.2}$$

Those ports have names and can be connected by oriented point-to-point connections. Connections are encoded in the form of references, i.e., a port $port_0$ is connected to another port $port_1$ if $port_0$ holds a reference to $port_1$. A port can hold zero or one reference.

Depending of if they hold a reference or not, ports are either called *use ports* or *provide ports*. A provide port holds no reference and exists so other ports can hold references to it. A use port is meant to hold a reference to another port which can be a provide port or another use port [1].

Assemblies Now that components and ports have been defined, everything necessary for the definition of assemblies has been laid out. Obviously, an assembly contains a set of component instances and a set of ports. In addition to that, a correspondence between ports and components must be established in order to determine which port belongs to which component.

Definition 7 (Assembly)**.** An assembly α is a triplet:

$$\alpha = (C, P, o) \tag{4.3}$$

where:
- $C \subseteq Instances(\mathcal{C}, \mathcal{N})$ is a set of *component instances*;
- $P \subseteq Ports(\mathcal{N})$ is a set of *ports*;
- $o : P \rightarrow C \cup \{\perp\}$ is the *owner* function.

[1] Connecting a use port to another use port is unusual but poses no problem as use ports have names too.

The correspondence between instances and ports is given by an *owner function* which, to each port, associates either a component (its owner), or $\perp$ (meaning the port has no owner).

Allowing some ports to have no owners permits incomplete assemblies, e.g., subassemblies comprising ports without their owners. Subassemblies are largely used in the remaining of the chapter and them being assemblies is a useful simplification.

Resources We now propose a new definition, which extends the assembly definition so as to add a simple resource model. Given a set of available resources $\mathcal{R}$, all that is required for component localization is a mapping between components, ports and resources:

Definition 8 (Localized assembly)**.** Let $\beta = (C, P, o)$ be an assembly. A localized assembly α is a triplet (β, R, r) which is denoted, for convenience, as the quintuplet:

$$\alpha = (C, P, o, R, r) \tag{4.4}$$

where:
- $R \subseteq \mathcal{R}$ is a set of *resources*;
- $r : C \cup P \to \mathcal{P}(R) \cup \{\perp\}$ is the *localization* function[a].

[a] where $\mathcal{P}(S)$ is the power set of set S.

The mapping from components and ports to resources is given by the localization function. In this model, each port and component can have several localizations (i.e., one port or component can be located on several resources at once [2]). Ports do not need to be on the same resource as their owner. Finally, components and ports can have no localization (by being mapped to $\perp$) which is useful for partial assemblies, as discussed above for the owner function.

Additional notations

Notations 1 (Assembly sets)**.** The set all assemblies with component shelf $\mathcal{C}$ and nameset $\mathcal{N}$ is denoted $A(\mathcal{C}, \mathcal{N})$.

The set all localized assemblies with component shelf $\mathcal{C}$, nameset $\mathcal{N}$ and resource set $\mathcal{R}$ is denoted $A_l(\mathcal{C}, \mathcal{N}, \mathcal{R})$.

Note that $A(\mathcal{C}, \mathcal{N}) = A_l(\mathcal{C}, \mathcal{N}, \emptyset)$. That means that non-localized assemblies are a special case of localized assemblies. In the remainder of the thesis, properties and definitions are given for localized assemblies and also apply to non-localized assemblies (since they are a special case of localized assemblies).

4.1.2 Graphical Conventions and Example

Graphical representation In order to present examples in a compact and legible fashion, a graphical representation of component assemblies is proposed. The following conventions are used for graphical representation of assemblies:
- components are represented by squares and feature an UML-style component symbol;
- ports are represented by circles;
- the owner relation is represented by solid lines;

[2] Note that while this makes sense for components, being located on several resources is strange for ports. While it might not make sense for target languages such as C++/Fortran, it can be used to model MPI communicators, for example.

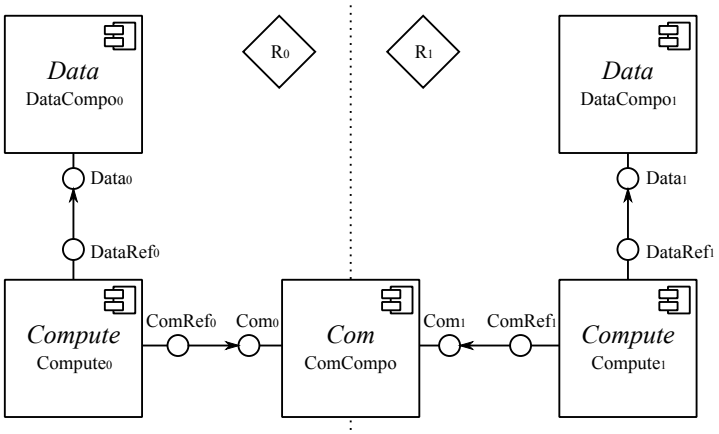


Figure 4.1: Graphical representation of a component assembly.

- the fact that a port $port_0$ holds a reference to port $port_1$ is represented by a solid arrow from $port_0$ to $port_1$;
- resources are represented by diamonds;
- dashed lines divide the assembly into sections, each containing at most one resource; the ports and component in one of these sections are localized on the corresponding resource;
- component types are in italics;
- names may or may not be present; if names are omitted then it means that each component or port without name has a unique name but the name itself is irrelevant.

Example: Figure 4.1 is a simple example of component assembly, consisting of two `Compute` components, each using a `Data` component, and connected to each other by a `Com` communication component. Each `Compute` component and its associated `Data` component are located on a different resource, while the `Com` component is located on both resources.

Text example For the sake of demonstrating the formalism, a complete text version of the graphical example from Figure 4.1 is presented.

Example: The assembly from Figure 4.1 requires three component types.

$$\mathcal{C} = \{Compute, Com, Data\} \tag{4.5}$$

where `Data` is a component which provides access to local data; `Com` is a component which allows point-to-point communication between two processes; and `Compute` is a component which, using `Data` and `Com` components, performs the computing required on a single resource.

The assembly in the figure is composed of two `Compute` components, two `Data` components and one `Com` component. Each of these components requires a unique name. This gives the following component instance set:

$$\begin{aligned} C \ = \{&INSTANCE(Compute_0, Compute),\\ &INSTANCE(Compute_1, Compute), INSTANCE(ComCompo, Com),\\ &INSTANCE(DataCompo_0, Data), INSTANCE(DataCompo_1, Data)\} \end{aligned} \tag{4.6}$$

In the figure, each Compute component has a connection to the corresponding Data component and to each adjacent Com component. In addition to that, every Com component and Data component provides its functionality through provide ports. For two processes, this gives the following set of ports:

$$\begin{aligned} P \ = \{&PROVIDE(Data_0), PROVIDE(Data_1),\\ &PROVIDE(Com_0), PROVIDE(Com_1),\\ &USE(ComRef_0, Com_0), USE(ComRef_1, Com_1),\\ &USE(DataRef_0, Data_0), USE(DataRef_1, Data_1)\} \end{aligned} \tag{4.7}$$

with the following owner function (where components and ports are represented by their name for brevity):

$$\begin{aligned} o \ &: P \to C \cup \{\bot\}\\ Data_0 \ &\mapsto DataCompo_0\\ Data_1 \ &\mapsto DataCompo_1\\ Com_0 \ &\mapsto ComCompo\\ Com_1 \ &\mapsto ComCompo\\ ComRef_0 \ &\mapsto Compute_0\\ ComRef_1 \ &\mapsto Compute_1\\ DataRef_0 \ &\mapsto Compute_0\\ DataRef_1 \ &\mapsto Compute_1 \end{aligned} \tag{4.8}$$

To model the two processes, two resources are required:

$$\mathcal{R} \ = R = \{R_0, R_1\} \tag{4.9}$$

Each resource hosts one Data component, one Compute component and all their ports. The case of Com components is more complex and can be modelled by using localization to multiple resources, as allowed by the model. Since one Com component is used to communicate between two processes, its implementation will likely be spread on the two processes, and it thus makes sense to have it bi-localized on the two said processes. Overall, this gives the following localization function (where components and ports are represented by their name for brevity):

$$\begin{aligned} r \ &: P \cup C \to \mathcal{P}(R) \cup \{\bot\}\\ Data_0 \ &\mapsto \{R_0\}\\ Data_1 \ &\mapsto \{R_1\}\\ Com_0 \ &\mapsto \{R_0\}\\ Com_1 \ &\mapsto \{R_1\}\\ ComRef_0 \ &\mapsto \{R_0\}\\ ComRef_1 \ &\mapsto \{R_1\}\\ DataRef_0 \ &\mapsto \{R_0\}\\ DataRef_1 \ &\mapsto \{R_1\}\\ ComCompo \ &\mapsto \{R_0, R_1\}\\ Compute_0 \ &\mapsto \{R_0\}\\ Compute_1 \ &\mapsto \{R_1\}\\ DataCompo_0 \ &\mapsto \{R_0\}\\ DataCompo_1 \ &\mapsto \{R_1\} \end{aligned} \tag{4.10}$$

Finally, the localized assembly α of this two-processes example is given by:

$$\alpha = (C, P, o, R, r) \tag{4.11}$$

Since this text approach is—as just illustrated—verbose and complex, the rest of the thesis uses the graphical representation to present examples.

4.1.3 Additional Definitions

This section introduces a few extra definitions that will be useful further on.

Additional notations In order to ease the manipulation of ports, the two following notations are introduced:

- **Name** if $port = PROVIDE(name)$ is a provide port then $Name(port) = name$, if $port = USE(name, ref)$ is a use port then $Name(port) = name$.
- **Ref** if $port = use(name, ref)$ then $Ref(port) = ref$

In order to more easily refer to internal elements of assemblies, extra notations are introduced thereafter. Let $\alpha = (C, P, o, R, r) \in \mathcal{A}_l(\mathcal{C}, \mathcal{N}, \mathcal{R})$ be a localized assembly.

- **Support** the support of an assembly is the union of the components, the ports and the resources: $Support(\alpha) = C \cup P \cup R$

Subassemblies First, we introduce the definition of subassemblies. A subassembly is an assembly which is a sub-part of another assembly.

Definition 9 (Subassembly)**.** $\alpha = (C, P, o, R, r)$ is a subassembly of $\beta = (C', P', o', R', r')$ iff

- $C \subseteq C'$
- $P \subseteq P'$
- $R \subseteq R'$
- $\forall port \in P, \quad o(port) = o'(port) \ \vee \ o(port) = \perp$
- $\forall element \in C \cup P, \quad r(element) = r'(element) \ \vee \ r(element) = \perp$

Also note that a subassembly can feature a port without its owner (even if said port does have an owner in the enclosing assembly), and can feature ports and/or components which are not localized (even if they are localized in the enclosing assembly).

Well-formed references With the assembly definition given so far, there is no guarantee that assemblies are correctly constructed. Indeed, given a use port $port = USE(name, ref)$, there is no guarantee that ref is actually the name of a port in the assembly.

In order to avoid the case of ill-formed port references, we introduce the definition of having *well-formed references*:

Definition 10 (Well-formed references)**.** A localized assembly $\alpha = (C, P, o, R, r)$ has well-formed references iff all use ports hold valid references, i.e., iff

$$\forall port \ \in P, \quad port \ = USE(name, ref) \quad \Rightarrow \quad \exists port' \in P \mid Name(port') = ref \tag{4.12}$$

4.1.4 Call-Stack Semantics

This section describes an operational semantics for the execution of a localized assembly. This semantics serves to model the behaviour of an assembly without getting into the details of what happens inside the components themselves. Basically, this semantics implements multithreaded method call stacks with non-deterministic method calls.

Assembly state In order to have call-stack semantics, assemblies need to have a state which contains information about which component is currently executing what. To that end, we introduce the following assembly state grammar:

Definition 11 (Assembly state)**.** An assembly state is defined by the following grammar:

$\langle thread \rangle$ ::= $\bot \mid \langle thread \rangle$-$\langle port \rangle$

$\langle callState \rangle$ ::= set of $\langle thread \rangle$

$\langle state \rangle$ ::= $(\langle assembly \rangle, \langle callState \rangle)$

An assembly state is made of an assembly and a call state. An call state is a set of threads, each thread being a stack of "method calls". A method call represent some work being performed by a port provided by a component. Each stack is an ordered set of method calls which correspond to nested calls, outermost call on the left. For example if a thread executing *port* calls *port*′ then the resulting call stack is $\bot - port - port'$.

Not all possible call stacks are valid. Indeed, the point of use ports and references is to determine which component has access to which provided port. Thus, not all ports are accessible from anywhere in an assembly and not all successions of calls in a thread are valid. So as to represent this limitations on threads, a definition of a well-formed thread is provided below:

Definition 12 (Well-formed thread)**.** Let $n \in \mathbb{N},\ n > 0$. A thread

$$thread\ = \bot - port_1 - port_2 - \ldots - port_n$$

where $compo_i = o(port_i)$, is well-formed inside an assembly $\alpha = (C, P, o, R, r)$ iff

- all components and ports belong to the assembly, i.e.,
  $\forall n \in \{1, \ldots n\},\ compo_n \in C\ \wedge\ port_n \in P$
- calls are either local or require a connection, i.e.,
  $\forall n \in \{1, \ldots n-1\}$ either $compo_n = compo_{n+1}$ (local call)
  or $\exists (port, port') \in P^2 \mid o(port) = compo_n \wedge o(port') = compo_{n+1} \wedge Ref(port) = port'$
  (distant call).

In addition, if $n = 0$ then the thread is well-formed.

Note that non-local calls (say, a call from p_i to p_{i+1} owned by two distinct components) requires only that the owner of p_i owns a use port with a reference to a port with the same owner as p_{i+1}.

Operational semantics In order to model the behaviour of components at assembly-level, an operational semantics is introduced in the form of a set of six rules.

Let α, β be assemblies and let σ, π be call states. A rule of the form

$$\frac{precondition}{(\alpha, \sigma) \ \rightarrow (\beta, \pi)} \tag{4.13}$$

means that, provided *precondition* is true, the application of the rule to a system whose state is (α, σ), changes its state to (β, π).

Let $\alpha = (C, P, o, R, r)$ be a localized assembly and let σ denote an call state.

Definition 13 (Operational semantics)**.** The operational semantics of localized assemblies is defined by the six following rules:

- **Terminate** an empty thread can be removed from the call state:

$$\frac{}{(\alpha, \sigma \cup \{\bot\}) \ \rightarrow (\alpha, \sigma)} \tag{4.14}$$

- **Return** a non-empty thread can return from the current method call to the parent method call:

$$\frac{}{(\alpha, \sigma \cup \{thread - port\}) \ \rightarrow (\alpha, \sigma \cup \{thread\})} \tag{4.15}$$

- **Spawn** a new thread can be created on a component currently executing something:

$$\frac{port' \ \in P \qquad o(port) = o(port')}{(\alpha, \sigma \cup \{thread - port\}) \ \rightarrow (\alpha, \sigma \cup \{thread - port\} \cup \{\bot - port'\})} \tag{4.16}$$

- **Local call** a thread can call any port on the same component as the current method call:

$$\frac{port' \ \in P \qquad o(port) = o(port')}{(\alpha, \sigma \cup \{thread - port\}) \ \rightarrow (\alpha, \sigma \cup \{thread - port - port'\})} \tag{4.17}$$

- **Distant call** a thread can call a distant port provided there is a connection between the component of the current method call and the distant port:

$$\frac{\begin{array}{c} port' \in P \qquad o(port) = compo \qquad o(port') = compo' \\ \exists use \in P \mid o(use) = compo, \ Ref(use) = port' \end{array}}{(\alpha, \sigma \cup \{thread - port\}) \ \rightarrow (\alpha, \sigma \cup \{thread - port - port'\})} \tag{4.18}$$

- **Parallel composition** two valid rule applications on two distinct threads can be composed in parallel:

$$\frac{(\alpha, \sigma) \ \rightarrow (\alpha, \sigma') \qquad (\alpha, \sigma'') \rightarrow (\alpha, \sigma''')}{(\alpha, \sigma \cup \sigma'') \ \rightarrow (\alpha, \sigma' \cup \sigma''')} \tag{4.19}$$

Execution model Given a starting assembly and a call state, the execution of the assembly is a non-deterministic application of the rules from Definition 13. The non-determinism of rule application is absolute and no hypothesis regarding those rule applications can be made.

Non-determinism means that this model provides no behaviour model for components (as opposed to models like BIP). This is in line with our goal to be close with languages such as L2C, which do not provide behaviour information in interfaces.

The absence of such a behaviour model is problematic to guarantee properties such as deadlock-freedom or termination of parallel assemblies. Chapter 5 proposes an extension of the present model which add minimal behavioural guarantees at interface level, in order to ease subassembly locking.

Preservation of well-formedness Considering the above execution model, and a well-formed starting state (i.e., well-formed assembly and well-formed threads), the fact that rule applications preserve well-formedness is not a given.

Since there is no reconfiguration (so far), the preservation of the well-formedness of an assembly is immediate.

Preservation of the well-formedness of threads is given by the following theorem:

Theorem 1 (Preservation of thread well-formedness)**.** Let $\alpha \in \mathcal{A}_l(\mathcal{C}, \mathcal{N}, \mathcal{R})$ be a well-formed localized assembly, and σ be a call state with well-formed threads, then the application of transition rules from Definition 13 preserves the well-formedness of the threads.

Proof. We proceed by induction on the application of the reduction rules presented in Definition 13.

First, note that removing method calls and adding/removing empty threads preserves well-formedness. Indeed, empty threads are well-formed by definition (see Definition 12). In addition, all three conditions for well-formedness are properties for all method calls, so removing one method calls cannot break well-formedness.

For this reason, the **Terminate**, **Return** and **Spawn** rules preserve well-formedness.

Secondly, the first two conditions of Definition 12 (belonging to the assembly and having the correct owner) are guaranteed by the preconditions of the **Local call** and **Distant call** rules. The third condition (call local or along connection) is immediate for the **Local call** rule which, consequently, preserves well-formedness. For the **Distant call** rule, the third condition is guaranteed by the precondition.

Finally, parallel composition of rules which preserve well-formedness also preserves well-formedness since the well-formedness of threads are independent. □

4.2 The DirectMOD Component Model

This section describes DirectMOD, a component model whose goals are to enable scalable assembly reconfiguration at runtime, and to have good software engineering properties. DirectMOD stands for *DIstributed REconfigurable ComponenT MODel*.

To achieve these goals, DirectMOD proposes several new ideas which can be summed up by the programming model presented in Figure 4.2. This programming divides work into four roles.

Component programmers write traditional components. DirectMOD propose no special feature for this role as it is not its focus. From the point of view of a component programmer, DirectMOD is no different than a traditional non-reconfigurable component model such as our preliminary model from Section 4.1.

Transformation programmers write pieces of reconfiguration code called *transformations*. Transformations are written using a graph-based assembly-level language. This language has an original semantics which make transformations less dependent on context than traditional graph transformations, and consequently eases reuse. In addition to that, DirectMOD proposes the concept of *transformation adapters* which allow explicit mapping of transformations to their target subassembly, avoiding any possibly time-consuming pattern-matching step in executing transformations.

One *domain programmer* writes devices called *domains* which are responsible for safely executing transformations. Most of the functionality of domains is provided directly by DirectMOD and the domain programmer is only responsible for writing the *locking* part of the code. DirectMOD adopts the quiescent state approach, i.e., to reconfigure a part of the assembly this

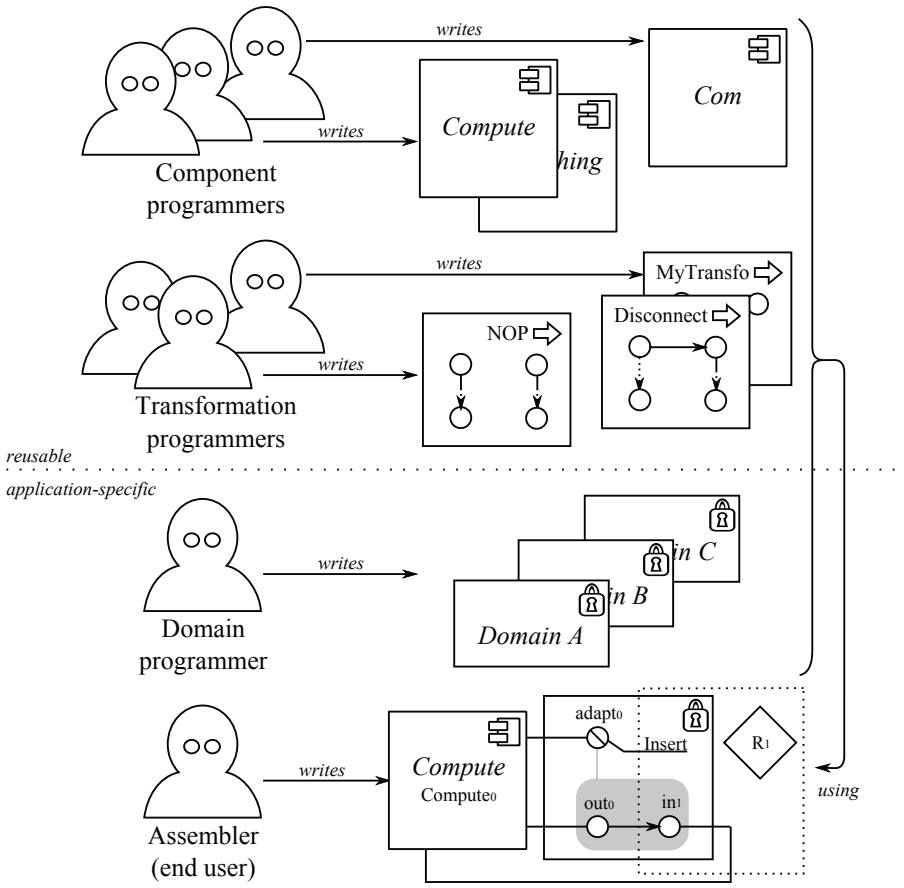


Figure 4.2: The DirectMOD programming model.

part needs to be stopped/locked. Locking an assembly at runtime is both performance-critical and a complex problem whose solution might vary greatly from application to application. Since providing a general-purpose solution to the locking problem is very unlikely, DirectMOD provides only the interface to separate this concern from the rest of the programming. Chapter 5 studies this problem and proposes a specialized solution for some application classes.

Finally, the *assembler*, can assemble components, transformations and domains into a DirectMOD assembly.

Concretely, DirectMOD is an extension of the preliminary model presented in Section 4.1. This section presents the new DirectMOD devices in turn (transformations, adapters, domains), then updates the assembly definition, and finally presents the updated semantics.

4.2.1 Transformations

Reconfiguration in component models is the process of changing the assembly at runtime. With this in mind, expressing reconfiguration at assembly level is natural. While this could be hidden inside dedicated components using an API (e.g., allowing creation/destruction of components and connection/disconnection of ports), we prefer to extensively describe reconfiguration using assembly-level abstractions so as to be able to control its semantics.

Definition From a formal standpoint, reconfiguration code in DirectMOD is expressed in the form of *transformations* defined as follows:

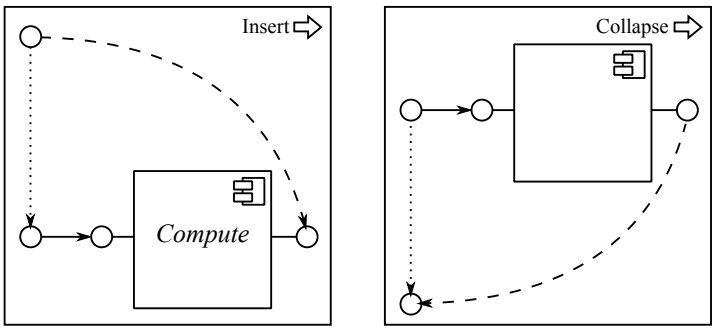


Figure 4.3: Graphical representation of two DirectMOD transformations.

Definition 14 (Transformation)**.** Let $\mathcal{A}$ be a set of assemblies and $\mathcal{N}$ a set of names. A *transformation* τ is of the form:

$$\tau = (name, \alpha, \omega, s, t) \tag{4.20}$$

where:
- $name \in \mathcal{N}$ is the name of the transformation;
- $\alpha \in \mathcal{A}$ is the *origin* of the transformation;
- $\omega \in \mathcal{A}$ is the *destination* of the transformation;
- $s : Support(\alpha) \to Support(\omega) \cup \{\bot\}$ is the *state mapping*;
- $t : Support(\alpha) \to Support(\omega) \cup \{\bot\}$ is the *topology mapping*.

The set of all transformations on $\mathcal{N}$ and $\mathcal{A}$ is denoted $Transfos(\mathcal{N}, \mathcal{A})$.

These transformations resemble graph transformations as can be found in the literature. Transformations have an origin (a part of the assembly they are applied to), a destination (what they replace their origin with) and several mapping between origin and destination which specify that some things are carried over during transformation. The *state mapping* denotes which component and ports have their state copied over through the transformation (those mapped to $\bot$ are destroyed). The *topology mapping* denotes which component takes the role of which one in the final assembly. The detailed behaviour of those mappings is defined in Section 4.2.6 which presents the semantics of DirectMOD.

Additional notations Let us denote $T(\mathcal{A}, \mathcal{N})$ the set of all transformations on assemblies $\mathcal{A}$ with nameset $\mathcal{N}$. In addition, extra notations are introduced to ease the manipulation of transformations in formulas. Let $\tau = (name, \alpha, \omega, s, t)$ be a transformation:
- **Name** $Name(\tau) = name$
- **Origin** $Origin(\tau) = \alpha$
- **Destination** $Destination(\tau) = \omega$
- **State** $State(\tau) = s$
- **Topo** $Topo(\tau) = t$.

Example and graphical conventions The graphical conventions for transformations are the following:
- the origin is represented on the top;
- the destination is at the bottom;

- the state mapping is represented by dashed arrows;
- the topology mapping is represented by dotted arrows;
- when the state and topology mapping coincide, they are represented by an arrow with both dots and dashes;
- the transformation is enclosed in a square featuring an arrow symbol in the top-right corner.

Example: Figure 4.3 presents two transformations. The first (on the left), called ⇨`Insert`, is a simple transformations which inserts a new component in the place of a point-to-point connection.

First note that the origin consists only of a single ports. Obviously, this port is not meant to be a complete assembly but instead it is a small part of a bigger assembly on which the transformation is supposed to be applied.

In the example, the topology and state mapping are used to connect the two ports of the newly added component to the port in the origin. A detailed description of the mappings' meaning is given in the semantics Section 4.2.6.

Finally, note that the second transformation, called ⇨`Remove`, looks like ⇨`Insert` upside-down. This transformation actually performs the reverse operation compared to ⇨`Insert`: it removes a component from a chain and connects the chain back.

4.2.2 Transformation Adapters

Transformations do not exist in a vacuum, and are meant to be applied to actual assemblies. Since transformations such as the ⇨`Insert` example above apply to only parts of an assembly, some way to specify the target subassembly is required.

Traditionally, graph transformations frameworks provide automated pattern-matching to find possible applications of graph transformations. Transformations in this context are often meant to be applied as many times as possible to perform some computation.

In the case of HPC applications, this pattern-matching step is not always desirable. First, the wanted behaviour may require that transformations are not always applied if possible. In this case, a way to specify when and where to perform a transformation is required. Secondly, pattern-matching an assembly can be a costly process which is not always necessary. When the target subassembly is known beforehand, a way to specify it directly could avoid performing pattern-matching.

DirectMOD provides a special kind of port, called a *transformation adapter* which serves to link a transformation to its target subassembly.

Definition A transformation adapter is a new kind of port built using the $ADAPT$ ternary operator, and is defined as follows:

Definition 15 (DirectMOD ports)**.** The set of DirectMOD ports on nameset $\mathcal{N}$, with assembly set $\mathcal{A}$, is defined by:

$$\begin{aligned} Ports_{dmod}(\mathcal{N}, \mathcal{A}) = &\{USE(name, ref) \mid (name, ref) \in \mathcal{N}^2\} \\ &\cup \{PROVIDE(name) \mid name \in \mathcal{N}\} \\ &\cup \{ADAPT(name, transfo, \alpha) \mid (name, transfo, \alpha) \in \mathcal{N}^2 \times \mathcal{A}\} \end{aligned} \tag{4.21}$$

where :

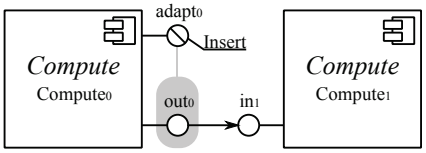


Figure 4.4: Graphical representation of a DirectMOD transformation adapter.

- provide and use ports are defined as for the preliminary model;
- the $name \in \mathcal{N}$ in the $ADAPT$ operator is the name of the adapter;
- $transfo \in \mathcal{N}$ is the transformation reference of the adapter;
- α is the target assembly.

Note that, just as the ports from the preliminary model, transformation adapters have names and thus can be the target of use ports. Also note that a transformation adapter holds a transformation *reference*, not a transformation. Of course, a transformation reference is meant to match the name of an existing transformation, which it references. Details about how transformation adapters and transformations are put together are given below in Section 4.2.4.

Example and graphical conventions Graphical conventions for transformation adapters are the following:

- a transformation adapter is represented by a circle, like other ports, but is struck through to mark its adapter nature;
- the transformation reference is denoted with a transformation name connected to the adapter with a solid line;
- the target subassembly is encircled in a grey area with rounded corners; this area is connected to the transformation adapter using a grey line.

Example: Figure 4.4 shows an example of a transformation adapter on $\mathcal{A}_l(\mathcal{C}, \mathcal{N}, \mathcal{R})$ for the ⇨`Insert` example transformation from figure 4.3.

Note on this example that the target subassembly is made of a single port. This is because the ⇨`Insert` transformation's origin is made of a single port. It would not make sense to try to apply the ⇨`Insert` transformation to, say, a subassembly made of a single component.

4.2.3 Domains

So far we have defined transformations and adapters which conjointly allows the specification of reconfiguration code and the mapping to a target assembly.

However, the question of how to actually execute transformations in a scalable way is a difficult one. Indeed, an application such as the AMR may try to perform many transformations concurrently. Sequentialising all transformations or executing all of them at once through a global synchronization are not scalable options, and executing them concurrently is difficult.

Concurrent execution of transformations requires some amount of synchronization to avoid incorrect behaviours. Indeed, transformations are non-atomical modifications on the assembly; several of them executing concurrently on non-disjoint subassemblies can very well interfere. Such interferences, i.e., modifications on the assembly performed in the course of a transformation execution, should either be avoided or be carefully taken into account in the transformation semantics. Either way, some form of synchronization is required.

In addition, representations of the assembly must be available for the transformation execution. The subassembly data contained in the transformation adapters is not sufficient to fully execute a transformations as defined above. Indeed, the topology mapping requires knowledge of the components and ports immediately surrounding the transformation origin. The usefulness of this feature is discussed extensively in Section 4.3.1 below. Even if the adapter data was sufficient, it would need to be updated when other transformations change it in any way.

To solve these two problems (synchronization and assembly representation) DirectMOD proposes special components called *domains*. Domains are tied to specific subassemblies, are responsible for executing transformations on those subassemblies, and maintain representations of those subassemblies.

Domains differ from other approaches from the literature in that the size and shape of the subassemblies they manage are completely up to the programmers. This is important in a HPC context as having fine control on synchronization and concurrency is performance-critical.

The flip side of this coin is that domains require application-specific code in order to perform their task. More specifically, domains must be capable of stopping the execution of the subassembly they manage without creating deadlocks. This is a difficult problem which is discussed further in Chapter 5. For the current chapter, we assume domains are correctly programmed.

Definition Domains are a new special kind of component which is defined as follows, using the $DOMAIN$ ternary operator:

Definition 16 (DirectMOD components)**.** The set of DirectMOD component instances using the component shelf $\mathcal{C}$, the nameset $\mathcal{N}$ and the assembly set $\mathcal{A}$ is given by:

$$\begin{aligned} Instances_{dmod}(\mathcal{C},\mathcal{N},\mathcal{A}) = \{&INSTANCE(name, type) \mid name \in \mathcal{N}, type \in \mathcal{C}\} \\ \cup\, \{&DOMAIN(name, \alpha) \mid name \in \mathcal{N}, \alpha \in \mathcal{A}\} \end{aligned} \tag{4.22}$$

where:
- instances are defined as for the preliminary model;
- *name* is the name of the domain;
- α is the assembly managed by the domain.

Note that, using an assembly definition such as the one from the preliminary model, the subassembly managed by a domain can have a large variety of shapes and sizes. For example, it can include ports without their owner components and it can be as small as a single port. Also note that domains can manage resources using this definition.

Example and graphical conventions Graphical representation of domains follow the following conventions:
- domains are represented with rectangles adorned with a bolt symbol at the top right;
- the managed subassembly of each domain is enclosed within the rectangle.

Example: Figure 4.5 presents two examples of assemblies featuring domains.

Both domain feature the same ring of `Compute` components, but have two different domain layouts. The first assembly (on the left) has only one domain which encloses the entire assembly. The second assembly (on the right) has one domain per `Compute` component which encloses the component itself, its outgoing port and the provide port of the next component. Note that the second example illustrates the fact that domains can have non-trivial shapes.

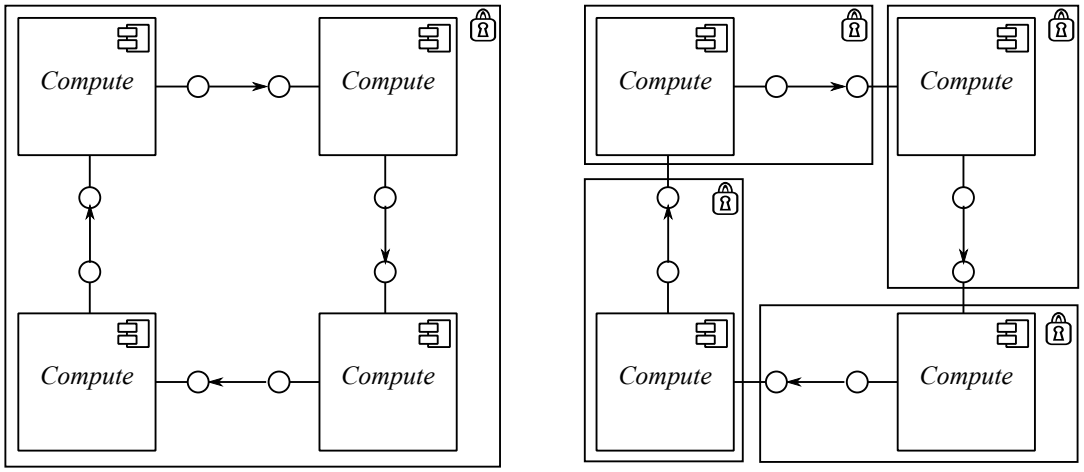


Figure 4.5: Graphical representation of two DirectMOD component assemblies featuring domains.

Since domains are responsible for reconfiguration and assembly representation, these two examples can be interpreted in synchronization terms. The example on the left implements global reconfiguration and assembly representation, while the example on the right implements concurrent reconfiguration and local assembly representation.

4.2.4 Full Assembly

Now that the new concepts proposed by DirectMOD have been introduced, a new assembly definition is introduced which makes use of them. Transformations are introduced as new set in the definition while transformation adapters and domains are new kinds of ports and components, respectively.

Definition 17 (DirectMOD localized assemblies)**.** The set of localized DirectMOD assemblies with component shelf $\mathcal{C}$, with nameset $\mathcal{N}$, and with resource set $\mathcal{R}$, is the set denoted $\mathcal{A}_{dmod}(\mathcal{C},\mathcal{N},\mathcal{R})$ composed of all the sextuplets:

$$\alpha = (C, P, o, R, r, T) \tag{4.23}$$

where:
- $C \subseteq Instances_{dmod}(\mathcal{C},\mathcal{N},\mathcal{A}_{dmod}(\mathcal{C},\mathcal{N},\mathcal{R}))$ is a set of component instances;
- $P \subseteq Ports_{dmod}(\mathcal{N},\mathcal{A}_{dmod}(\mathcal{C},\mathcal{N},\mathcal{R}))$ is a set of ports;
- $o : P \rightarrow\ C \cup \{\perp\}$ is the owner function.
- $R \subseteq \mathcal{R}$ is a set of resources;
- $r : C \cup P \rightarrow \mathcal{P}(R) \cup \{\perp\}$ is the localization function;
- $T \subseteq Transfos(\mathcal{N},\mathcal{A}_{dmod}(\mathcal{C},\mathcal{N},\mathcal{R}))$ is a set of transformations.

Note that non-localized DirectMOD assemblies can be easily obtained by choosing $\mathcal{R} = \emptyset$. In this case, the definition can be simplified so that assemblies are quintuplets instead of sextuplets. Similarly, the preliminary assembly definition is a special case of a DirectMOD assembly where $T = \emptyset$, where there is no domains and where there is no transformation adapter.

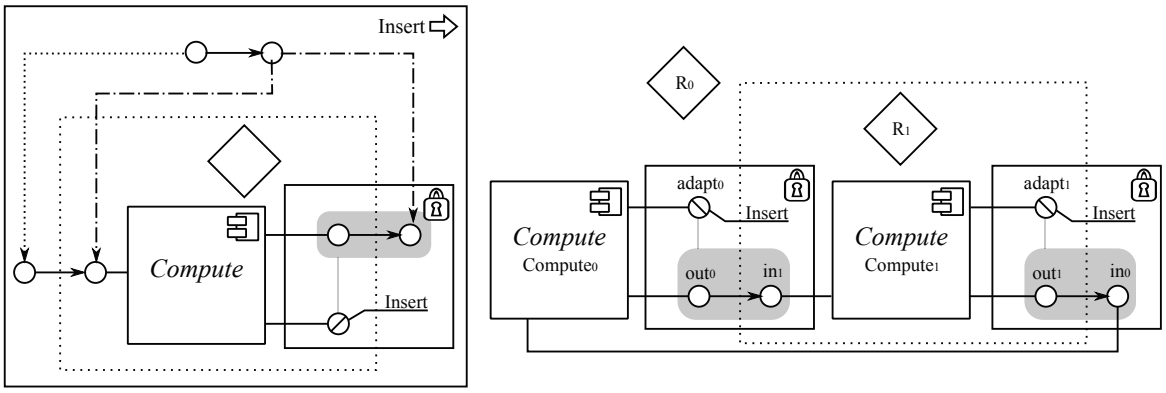


Figure 4.6: Graphical representation of a DirectMOD component assembly implementing a self-extensible ring. The arrows with dashes and dots denote mappings for both state and topology.

Example

Example: Figure 4.6 presents a complete example of a reconfigurable assembly implementing an extensible ring of computing components. All necessary graphical conventions have been introduced in the previous sections.

This assembly is composed of a ring of `Compute` components. Each of those components has an `in` port and an `out`. The `out` port of each component is connected to the `in` port of the next component in the ring. In addition, each component has a transformation adapter called `insert`, which connects the `Insert` transformation to the adequate subassembly for execution.

In addition, this assembly features one domain per computing component. Note that each of those domains includes the `in` port of the next component in the ring but not the `in` port of the component it manages. The reason for this is to make sure that the subassemblies pointed by the domain adapters do not cross the domain boundaries, so that the assembly is well-formed.

Note also that transformations feature domain modifications and adapter notifications. This is done so that transformations preserve the structure of the ring (i.e., one domain per computing component and adapters pointing to relevant subassemblies). This needs to be done explicitly, and it is possible because the origin and destinations of transformations are reconfigurable assemblies themselves; they can thus feature domains and adapters.

4.2.5 Additional Notations and Definitions

Compatible mappings We introduce the notion of *compatible mapping* which is useful in finding element-to-element mapping between a transformation origin and its target subassembly:

Definition 18 (Compatible mapping)**.** Let $\alpha = (C, P, o, R, r, T)$ and $\beta = (C', P', o', R', r', T')$ be two assemblies, a mapping $m : Support(\alpha) \to Support(\beta)$ is said to be a *compatible mapping from α to β* if the following conditions are met:

- m is bijective;

- m and m^{-1} preserve nature (a component maps to a component and so on);
- β has more structure than α, i.e., for all $(element, element') \in Support(\alpha)^2$:
  - **Owner:** if $element$ is a component, $element'$ is a port, and $o(element') = element$ then $o'(m(element')) = m(element)$;
  - **Localization:** if $element$ is a resource, and $r(element') = element$ then $r'(m(element')) = m(element)$;
  - **Reference:** if $element$ is a use port or an adapter, and $Ref(element) = Name(element')$ then $Ref(m(element)) = Name(m(element'))$;
  - **Belonging to domain:** if $element = DOMAIN(name, \gamma)$ is a domain, and $element' \in \gamma$ then $m(element) = DOMAIN(name', \delta)$ and $m(element') \in \delta$;
  - **Belonging to target subassembly:** if $element = ADAPT(name, transfo, \gamma)$ is a transformation adapter, and $element' \in \gamma$ then $m(element) = ADAPT(name', transfo', \delta)$ and $m(element') \in \delta$.

Basically, the existence of a compatible mapping between α and β means that they have identical supports (up to naming) and that β may contain more structure (localization, ownership...) than α but not less. This is useful to map under-specified origins to target subassemblies (e.g., the ⇨`Insert` example from Figure 4.3 does not specify resources and localization but we may want to apply it to an assembly with resources).

When there exists a compatible mapping between assemblies α and β, we denote $\alpha \preceq \beta$.

Default mapping Several compatible mappings may exist between two assemblies, in which case we distinguish on of them as the **default mapping** between those two assemblies. Assuming a complete order on $\mathcal{N}$, one possible way to select a default mapping is to order compatible mappings (e.g., lexicographical order since mappings are basically permutations) and select the smallest.

Well-formed assemblies The definitions given so far impose very little by-construction constraints on the use of domains and adapters. So as to forbid ill-formed assemblies which could not be executed or be reconfigured, a set of structural constraints is provided through a notion of *well-formed assembly*.

Definition 19 (Well-formed DirectMOD assembly.)**.** A DirectMOD assembly

$$\alpha = (C, P, o, R, r, T)$$

is said to be well-formed if the following conditions are met:

- **Well-formed references** each reference to a port or transformations corresponds to an actual port or transformation, i.e.,

$$\begin{aligned} &\forall port \in P, \quad port = USE(name, ref) \;\Rightarrow\; \exists port' \in P \mid Name(port') = ref \\ &\quad \wedge \quad port = ADAPT(name, ref, \alpha) \;\Rightarrow\; \exists transfo \in T \mid Name(transfo) = ref \end{aligned} \tag{4.24}$$

- **Disjunction of domains** domains are disjoint, i.e.,

$$\begin{aligned}\forall element \in Support(\alpha),\\ \big(\quad & \exists domain_0 = DOMAIN(name, \beta) \in Support(\alpha)\\ & \wedge \exists domain_1 = DOMAIN(name', \beta') \in Support(\alpha)\\ & \mid element \in Support(\beta) \wedge element \in Support(\beta') \quad \big)\\ & \Rightarrow \; domain_0 = domain_1\end{aligned} \tag{4.25}$$

- **Well-formed domains** let β be the assembly managed by a domain, β is a subassembly of α.
- **Well-formed targets** let β be the target of an adapter, β is a subassembly of α.

Nothing too fancy here: well-formed references, domains and target are straightforward constraints to make sure everything references is actually part of the assembly; disjunction of domains is here to ensure any element is managed at most by one domain, to avoid having to deal with cross-domain consistency issues.

4.2.6 Full Semantics

Now that a reconfigurable assembly definition has been provided, its behaviour can be specified as an extension of the semantics of the preliminary model. This extension is divided into two parts: first, the semantics of transformations and then a new rule (added to those of Definition 13)

Auxiliary Functions First, we define a few auxiliary functions to simplify the semantics definition:

- **SubMap:** Let set, set' and set'' be three sets and $func$ a function from set' to set''. The subset mapping function of set and set' according to $func$ is defined by $SubMap(set, set', func) = (set \setminus set') \cup \{func(element) | element \in set \cap set'\}$.
- **SubAMap:** Let $\alpha = (C, P, o, R, r, T)$ and $\beta = (C', P', o', R', r', T')$ be two assemblies. Let $func$ be a function from $Support(\alpha)$ to $Support(\beta)$. The support subassembly mapping function is defined by:

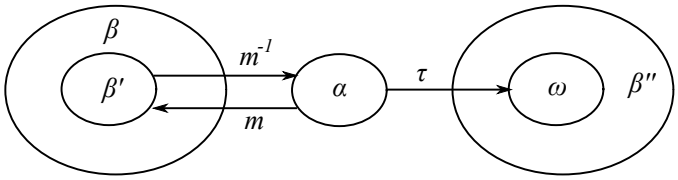
$$\begin{aligned}SubAMap(\alpha, \beta, func) = \big(&SubMap(C, C', func), SubMap(P, P', func),\\ &\perp_P, SubMap(R, R', func), \perp_R, SubMap(T, T', func)\big)\end{aligned} \tag{4.26}$$

  where $\perp_P$ and $\perp_R$ are empty owner and localization functions (i.e., everything mapped to $\perp$).

The $SubAMap$ function is used in the transformation definition below to help modify subassemblies by replacing some specific elements by their image by the topology mapping.

Transformation semantics Formally, the behaviour of transformations is given by Definition 20 below. Inline figures within the definition provide a view of the various sets involved.

Definition 20 (DirectMOD transformation semantics)**.** A transformation $\tau = (\alpha, \omega, s, t)$ applied to a subassembly $\beta' = (C', P', o', R', r', T')$ of an assembly $\beta = (C, P, o, R, r, T)$ such that $\alpha \preceq \beta'$ and $\beta' \subseteq \beta$ results in an assembly $\beta'' = (C'', P'', o'', R'', r'', T'')$ denoted $\beta'' = Apply(\beta, \beta', \tau)$. Let us denote m the default mapping from α to β' and m^{-1} its inverse.



β'' is obtained by applying the following operations in sequence:

Support A new subassembly γ is created with the following support:

$$Support(\gamma) = (Support(\beta) \setminus Support(\beta')) \cup Support(\omega) \tag{4.27}$$

this assembly does not have a owner function or a localization function (i.e., everything is mapped to $\perp$).

State mapping Then, assembly γ' is built by replacing every component/port that has an antecedent for $s \circ m^{-1}$ (i.e., that is the target of a state mapping) by its antecedent.

Topology mapping (references) Then, assembly γ'' is built by replacing every reference (i.e., every reference in a use port or transformation reference in a transformation adapter) to a port or transformation in β', by a reference to the image under $t \circ m^{-1}$ of said port or transformation.

I.e., if γ' contains a reference to a transformation/port *element* in β', then this reference is replaced by $Name(t(m^{-1}(element)))$.

Topology mapping (subassemblies) Then, assembly γ''' is built by replacing every element from β' which appear in a domain of transformation target by its image under $t \circ m^{-1}$.

This can be achieved using the auxiliary function $SubAMap$, by replacing every domain/target δ in γ'' by:

$$SubAMap(\delta, \beta', t \circ m^{-1}) \tag{4.28}$$

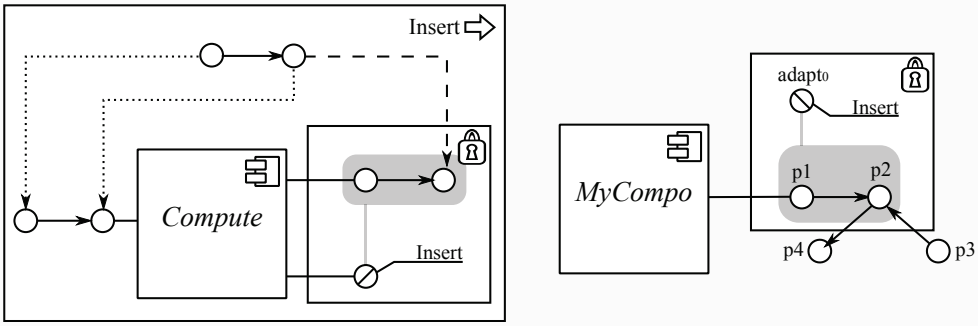
Owner and resources Finally, assembly β'' is obtained by constructing the owner and resource functions. Let *element* be a port or component in γ''', its owner is given by:

- if $element \in (\beta \setminus \beta')$
  - if $o(element) \in (\beta \setminus \beta')$, then $o''(element) = o(element)$
  - else if $o(element) \in \beta'$ and $t(m^{-1}(o(element)))$ is defined, then $o''(element) = t(m^{-1}(o(element)))$
  - otherwise, $o''(element) = \perp$
- if $element \in \omega$
  - if *element* has a owner $element'$ in ω, different from $\perp$, then $o''(element) = element'$
  - else if *element* has an antecedent $element'$ for $t \circ m^{-1}$, then $o''(element) = o(element')$, or $o''(element) = t(m^{-1}(o(element')))$ if $o(element) \in \beta'$
  - otherwise, $o''(element) = \perp$

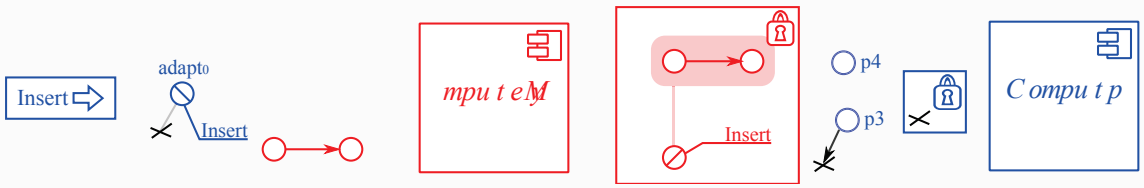
the localization function r'' is defined the same way.

Example:

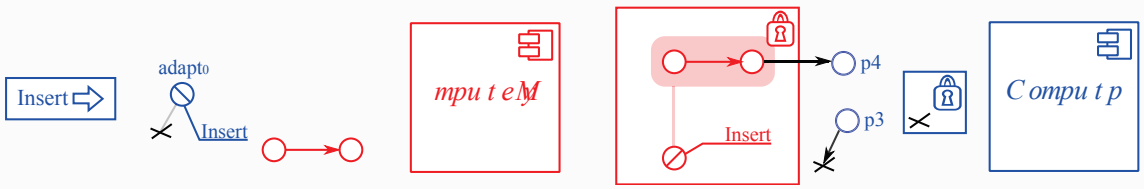
This example is a simplified version of the example given before in Figure 4.6, with the resources removed to keep things simple. Let us assume one of the ⇨`Insert` transformations is triggered. Here is the subassembly as seen from the point of view of the executing domain (the transformation is presented in full in this step as a reminder but is not detailed in further steps to save space):



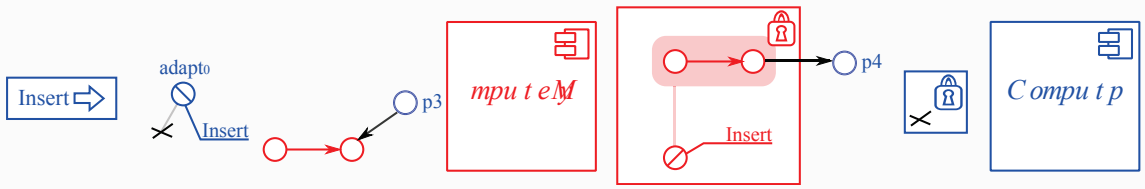
First, an assembly γ with only a support (i.e., no owner or resource) is built. In red are the elements parts of the transformation destination; in blue are the elements parts from the original assembly. At the moment, the adapter and the domain have invalid subassembly targets (denoted by black crosses). The $p3$ port also has an invalid reference.



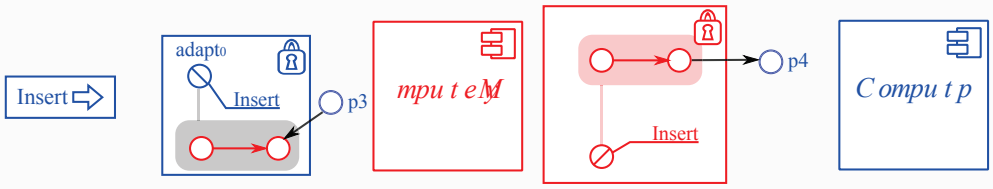
The *state mapping* means that, since the port on the right of the origin is mapped to $p2$, then $p2$ is copied to the port pointed by the arrow with dashes (topology mapping). In that case, the state of $p2$ is its reference to $p4$. This operation results in the following assembly:



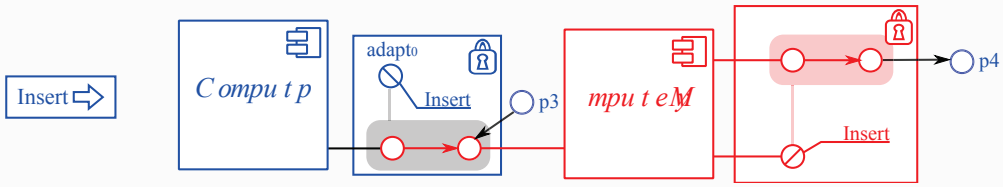
The *topology mapping (references)* means that every reference to a port p removed in the first step is replaced by a reference to the port pointed by the topology mapping. In this case, $p3$ holds a reference to $p2$ which was removed. Since $p2$ was mapped to the right port in the origin, and since this port is mapped to a port in the destination using a dotted arrow (topology mapping), then the reference to $p2$ is replaced by a reference to this pointed port:



The *topology mapping (subassemblies)* step means that every port/component featured in a subassembly (in a domain specification or adapter target) that was removed in the first step, is replaced according to the topology mapping. In this case, it means that the ports $p1$ and $p2$ are replaced by the two ports pointed by the dotted arrows, both in the invalid domain and the invalid adapter. This results in:



Finally, the *owner and resources* step updates the owner and localization functions according to the topology mapping. In our case, the owner relations mapped in red are obtained directly from the destination (case "$element \in \omega$" and "owner in ω different from $\perp$") while the black owner line is derived from the topology mapping (case "$element \in (\beta \setminus \beta')$" and "owner of *element* has an image under $t \circ m^{-1}$").



This is the final state of the assembly after the transformation application. A new compute component was indeed inserted and a new domain was set up to provide this new component with the same ⇨`Insert` transformation capability than the other `Compute` components. The connections to and from the $p3$ and $p4$ components have been changed according to the topology mapping.

New operational rule Now that transformation semantics, i.e., the effect of transformation application on assembly, have been determined, some way to trigger transformations is required in addition of the operational semantics rules of the preliminary model (see Definition 13):

Definition 21 (New operational rule)**. Transformation Execution** a transformation can be triggered through a transformation adapter provided enclosing domains do not contain

call stack ports, and provided the owner of the adapter is executing code:

$$\frac{\begin{array}{c}\exists adapter = ADAPT(name, \tau, \beta') \in Support(\beta) \\ \tau = (\alpha, \omega, s, t) \qquad \alpha \preceq \beta' \qquad o(adapter) = o(port) \\ \forall port' \in Enclosing(\beta', \beta),\ port' \notin \sigma \cup \{thread\}\end{array}}{(\beta, \sigma \cup \{thread - port\}) \ \rightarrow (Apply(\beta, \beta', \tau), \sigma \cup \{thread - port\})} \tag{4.29}$$

where $Enclosing(\beta', \beta)$ denotes the union of domains of β which intersect β', i.e.,

$$Enclosing(\beta', \beta) = \bigcup \{\alpha \mid \exists domain = DOMAIN(name, \alpha) \in Support(\beta), \ \alpha \cap \beta' \neq \emptyset\} \tag{4.30}$$

There are two important preconditions to this rule. First, there must be some code running on the component owning the adapter. This models the fact that transformations are not triggered out of nowhere but are called by regular component code. Second, all the ports in all of the enclosing domains must not appear in any call stack. The enclosing domains are the union of all the domains comprising at least one element from the target subassembly. Since the transformation might modify any subassembly managed by any of these domains, it is important that no call stack traverses any of these domains to avoid invalid call stacks. For example, if there is a call stack featuring a call to port *port*, and if *port* is removed by the transformation, then the call stack would have become ill-formed (in virtue of the first bullet of Definition 12).

Note that this definition implies that a transformation adapter can cross domain boundaries, and thus have several enclosing domains. We call such transformations *multi-domain* transformations.

If the preconditions are met, the modification to the assembly is a simple application of the *Apply* function defined by the transformation semantics (see Definition 20).

4.3 Discussion and Evaluation

This section analyses and evaluates DirectMOD, first through a series of model-level examples (Section 4.3.1), and second through an implementation called DirectL2C whose performance and code-level properties are discussed (Section 4.3.2).

4.3.1 Model-level Discussion

Now that the DirectMOD formalism has been extensively laid out, several questions can be asked regarding what it can express, how easy it is to do so, and its software engineering qualities.

Expressiveness Regarding component assemblies themselves, DirectMOD is as expressive as any other non-hierarchical, component model with point-to-point connections, with the unusual possibility to describe resources in the assembly (L2C is the only other example that we know of).

Transformation expressiveness is not so easily characterised, as the formalism is unusual, most notably the necessity for explicit transformation adapters.

So as to evaluate transformation expressiveness, we have first devised a series of very simple examples presented in Figure 4.7. examples are DirectMOD transformations which implement the following basic operations (in order, from left to right): component migration, connecting two ports, disconnecting two connected ports, and changing the type of a component while keeping its connections in the assembly. Note that all those operations have simple DirectMOD

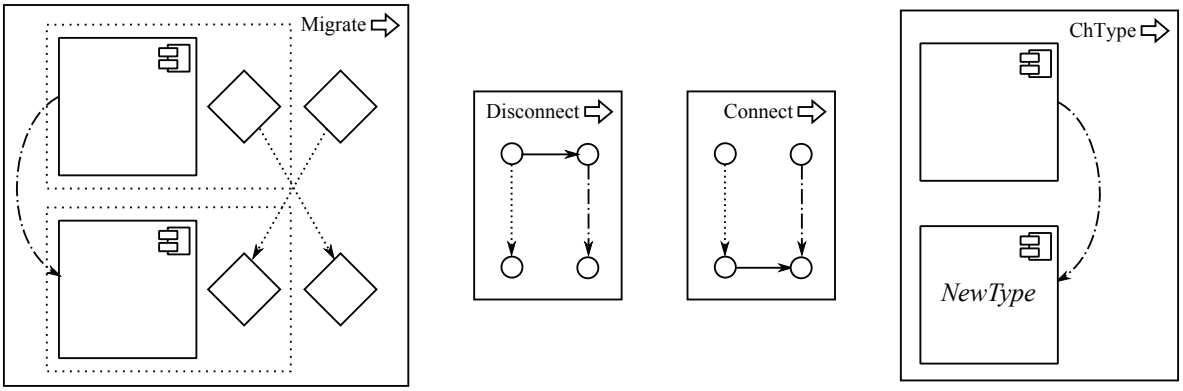


Figure 4.7: Examples of transformations implementing basic reconfiguration operations.

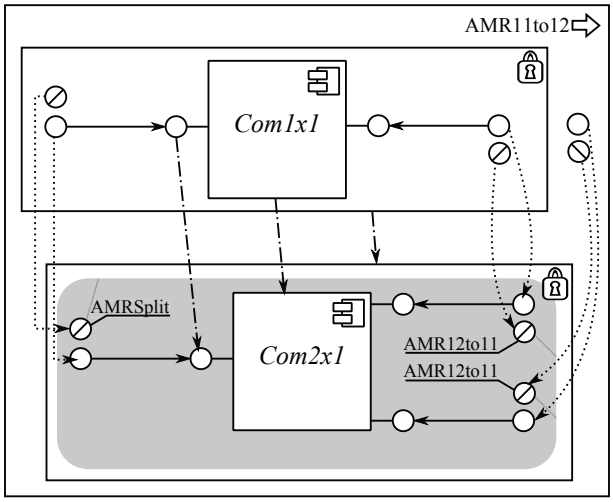


Figure 4.8: Example of an DirectMOD transformation implementing an AMR connection transformation.

implementations, and that the connect/disconnect transformations are, as can be expected, mirrors of one another. This first series of examples show that common reconfiguration operations can be expressed in DirectMOD, and that their expression is simple enough.

We have also studied a more complex example: an 2:1 2D AMR assembly, as described in Section 2.1.3. While we have chosen not to present the full example here (as it is quite large), Figure 4.8 presents one transformation that is typical of how AMR can be handled using DirectMOD. This transformation is used to reconfigure the communication infrastructure between two neighbours, going from a 1-to-1 connection to a 1-to-2 connection. Although this examples is complex enough to show the limits of the graphical representation, it also means that the AMR, the use case we set as a goal in Chapter 2, can actually be done using our approach.

Another point illustrated by the AMR example (Figure 4.8), and our earlier ring example (Figure 4.6), is that having to explicitly maintain domains and transformation adapters across transformations is actually responsible for most of the complexity of transformations.

Reuse, separation of concerns Components in DirectMOD have no special features that make them more or less reusable than in any other component model. Domains and transfor-

mations, however, are concepts specific to DirectMOD, and their properties regarding reuse and separation of concerns are not obvious.

Domains, although they appear reusable *a priori*, are actually tied to specific assemblies or assembly structures so as to be able to perform. On the one hand, DirectMOD does not mechanically restrict the use and reuse of domains. On the other hand, though, domains need to be able to ensure transformations can execute without concerns on the subassembly they manage, which is actually a complex problem with application-specific solutions (more on that problem in Chapter 5). For domains to be able to operate, they must make some assumptions regarding their managed subassembly, which consequently limits their reuse to subassemblies which actually verify these hypotheses.

Transformations have two main characteristics regarding reuse and separation of concerns.

First, they depend little on context and are easy to reuse in different assemblies. Consider, for example, the ⇨`Connect` and ⇨`Migrate` transformations in Figure 4.7, which are very general. Such general transformations are possible because of the possibility to have ports without their owner—or components without their ports—appear in transformations, and because the state and topology mappings allow some control over the connexions to the surrounding assembly. Having separate state and topology mappings, as opposed to just one mapping (e.g., the graph morphism between destination and origin in traditional graph transformations), allows transformations that would not be possible otherwise, such as the ⇨`Insert` transformation from Figure 4.3. Having an even finer control over this mapping (i.e., more than just a topology and a state mapping) may be desirable (and possibly more elegant from a formalism standpoint) but we have not encountered yet a use case which would require it.

Second, while transformations are fairly easy to reuse in different contexts, the transformations themselves lack flexibility. Consider the ⇨`AMR11to12` example from Figure 4.8: while this transformation depends very little on context (which is good for reuse), it is also very specific; a full 2:1 AMR assembly requires two different transformations for refinement (from a 1-to-1 to a 1-to-2 connection, or from a 1-to-2 to two 1-to-1 connections) and other factors such as resources may increase further the number of specific transformations to be written. Mechanisms for transformation variability are lacking in DirectMOD itself; Chapter 6 addresses this problem, in Section 6.3.2.

Domains, parallelism Domains are the DirectMOD feature that addresses the problems of concurrent reconfigurations and assembly representation at runtime. Several questions can be asked about how well domains actually work.

First, the consistency of assembly representation at runtime is adequately addressed by DirectMOD domains. Indeed, since each element in the assembly is managed by at most one domain (see Definition 19), and since this domain is the only one that can modify said element, maintaining a representation at runtime is easy from the point of view of the domain.

Second, domains indeed allow concurrent execution of transformations (at least in theory). Indeed, transformation execution (see transformation semantics, i.e., Definition 20) requires only local knowledge, and the semantics are unambiguous regarding concurrent transformations (i.e., they can only be done in sequence).

Third, domains provide separation of concerns, in the sense that they separate the synchronization/execution concerns from the other concerns. Even if a domain cannot work with any subassembly, the way DirectMOD maps domains to managed subassemblies allows variations.

Finally, domains are actually very difficult to implement. As discussed in more details in the next chapter (Chapter 5), guaranteeing the correct execution of assembly transformations in a concurrent context is actually a very difficult problem, especially so if little hypotheses can be made regarding the assembly of application.

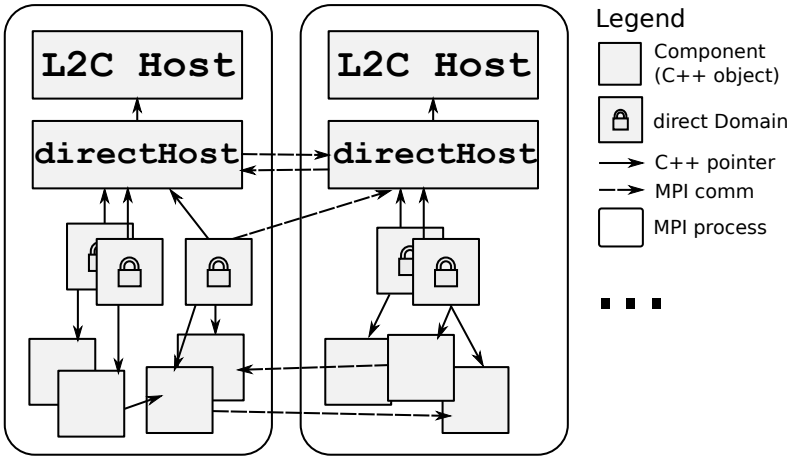


Figure 4.9: An overview of the DirectL2C architecture.

Multi-domain transformations Assemblies with multi-domain transformations (see Definition 21) pose a particularly complex problem. Indeed, reconfiguring a single domain is a difficult problem in itself, but reconfiguring several domains at once might cause application-wide deadlocks that would not have arisen otherwise. Writing domain code that can safely support multi-domain reconfiguration may be too much to ask to domain developers.

Multi-domain transformations are, however, useful. Indeed, when implementing something like AMR unrefinement (not refinement), it is important to be able to fuse several domains together, which requires multi-domain transformations (as otherwise transformations can create domains but not remove them). While implementing domains supporting this kind of transformation is possible (although non-trivial) in the very structured case of the AMR, it might not be possible at all in other less structured cases.

4.3.2 Implementation and Evaluation

This section presents DirectL2C, an implementation of the DirectMOD model based on C++ and MPI. A preliminary evaluation of DirectL2C performance is performed through a set of experiments on the Grid'5000 platform [10]. In addition, the complexity and size of the code of DirectL2C benchmarks are analysed and discussed.

The DirectL2C Implementation We have implemented the DirectMOD model by extending the L2C C++ implementation called LLCMC++ [21]. LLCMC++ provides zero-overhead components on top of C++ objects as well as distributed deployment and remote component connections using in particular MPI communicators. Basic local reconfiguration capabilities (i.e., create, destroy, connect) are provided by LLCMC++ `Host` components, one of which is deployed on each MPI process.

DirectL2C extends the functionality of LLCMC++ `Host` components by interposing `directHost` components which provide the same local functionality but are also connected to each other using MPI to provide remote operations. `directHost` components implement basic remote reconfiguration operations as well as basic remote method call and remote locking. `directHost` components required an efficient multithreaded MPI implementation which is provided in our tests by MadMPI [99].

Reconfiguration logic (i.e., implementation of the transformation semantics) is provided by DirectL2C domains which are connected to the relevant `directHost` components. Locking

```
Inputdata: portName, reconfPortName, resourceName
Direct::transformation insert();
insert.create("Cp", "newCp", resourceName);
//ports of created component have implicit names
insert.connect(portName, "newCpLeft");
insert.statePort(portName, "newCpRight");
insert.topoPort(reconfPortName, "newCpReconf");
```

Figure 4.10: Code of the ⇨`Split` transformation.

Function	**C++ LOC**
Transformation	8
Non-functional synchronization	20
Code instrumentation	13
LLCMC++ overhead	7
DirectL2C overhead	6
Functional code	31
Other	3
TOTAL	88

Table 4.1: C++ LOC breakdown for the ring example.

code is up to the programmers of both components and domains but standardized locking interfaces are assumed by DirectL2C.

Code Size and Complexity We have implemented a ring example similar to that of Figure 4.6 with fully distributed domains. The code of the ⇨`Split` transformation, which is displayed in Figure 4.10, is very close to the expected pseudocode. In practice, although the transformation code is simple, the implementation of the ring example was not trivial due mostly to synchronization and locking code which is difficult to write and error-prone.

Table 4.1 reports a line of code breakdown of the code for the ring example with only the split transformation, only C++ code lines with semicolons have been taken into account. Although the transformation code itself is short, the non-functional code and DirectL2C/LLCMC++ overhead still represent a sizeable fraction. In practice, the *non-functional synchronization* part, i.e., mostly locking logic, was by far the harder part to write and to debug.

There is also ongoing work to implement an AMR-like example similar to that presented in Section 2.1.3. Preliminary coding and testing indicate that the implementation is, as expected, more complex than it was for the ring example mostly because multiple cases must be implemented separately. The locking code, i.e., the domain implementation, in the AMR case was also a lot of work and proved bug-prone but in the end, the desired synchronization scheme was successfully implemented. Although low-level synchronization is inherently a complex task, the level of control offered by DirectL2C allows to optimize the performance of the reconfiguration synchronization.

Performance We have conducted experiments to measure the performance of our DirectL2C ring implementation. Experiments were conducted on the Grid'5000, on clusters Edel and Graphene whose characteristics are given in Table 4.2.

	Graphene	Edel
Processors	Intel Xeon X3440	Intel Xeon E5520
Memory	16 GB	24 GB
Network	Infiniband 20G	Infiniband 40G

Table 4.2: Grid'5000 cluster description.

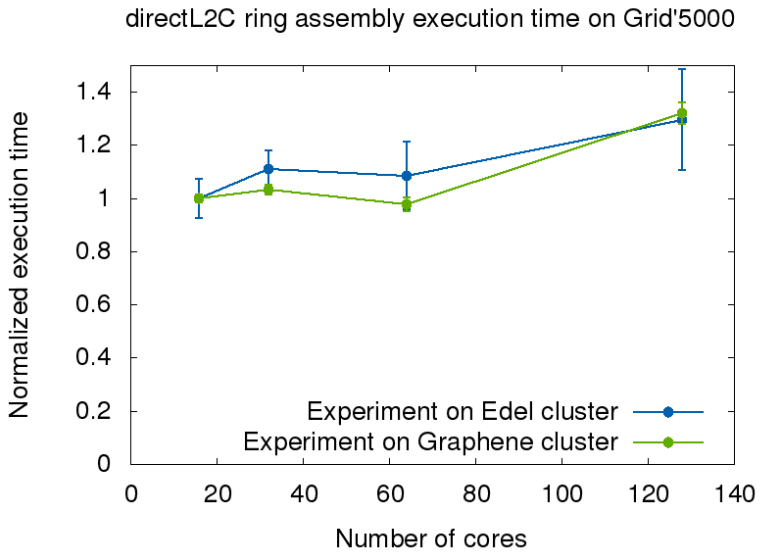


Figure 4.11: Scalability experiments of the ring assembly implementation on Grid'5000 clusters. Computing time is normalized using the computing time for 16 cores on 1 node.

We have first measured the overhead introduced by our implementation outside of any reconfiguration. Contrary to LLCMC++ which introduces zero overhead at runtime, our model requires connections to be lockable and thus adds a mutex lock for each method call through a port, even when reconfiguration is not occurring. Our measurements indicate that the overhead introduced by our implementation is approximately 5 ns per call through a port. Such an overhead is negligible outside of very fine-grain applications, but such applications are not likely to be the kind of applications featuring reconfiguration.

We have also conducted a series of preliminary scalability experiments on Grid'5000 using a simple version of the ring assembly which, starting from a small ring, inserts new computation components until there is one computation component per physical core. Weak scaling results, presented in Figure 4.11, show that iteration time stays roughly constant when the number of cores increases which shows that our code scales up to 128 cores. Larger experiments are needed to fully validate the scalability of our approach but, baring technical problems and since only local synchronization is ever done, the program should scale.

4.4 Conclusion

This chapter has presented two component models and evaluated one of them.

First, a preliminary component model has been defined. This first model is a partial formalization of models such as L2C or CCA. This model is non-hierarchical, non-reconfigurable,

features point-to-point communications, and has a call-stack-based semantics.

Second, the DirectMOD model was defined by adding concepts on top of the preliminary model. DirectMOD features transformations (to describe reconfiguration), transformation adapters (to link transformations to their targets) and domains (to execute transformations). The semantics of the preliminary model was expanded to allow transformation application.

Third, a preliminary evaluation of DirectMOD was performed in two steps: the model itself is applied to a ring assembly use case, and an implementation of the model, called DirectL2C, is evaluated. It appears that DirectMOD provides good separation of concerns, and that preliminary experiments with DirectL2C scale well. On the other hand, DirectL2C transformations tend to be verbose, and domain implementation is very difficult.

Chapter 5

Mutex-based Locking of Component Assemblies

Contents

The component model presented in the previous chapter requires that assembly transformations can be performed as if they were atomic. Making sure this property holds in practice is the job of the DirectMOD domain programmer, but it is not an easy job. Indeed, concurrent reconfiguration can easily lead to assembly consistency problems (incorrect pointers, incorrect assembly representation) and/or synchronization problems (e.g., deadlocks).

The literature provides the concept of *quiescent state* as a way to approach this problem. The quiescent state approach consists in stopping every activity in a subassembly, and making sure it stays stopped, before performing a reconfiguration. The process of stopping activity (executing code, other reconfigurations) in a subassembly, and preventing further activity for some time, is referred to as *locking*. Once a subassembly has been locked, reconfiguration on that subassembly can occur without concerns as no other code can interfere.

Unfortunately, locking a subassembly *in a safe manner* in a concurrent application is a difficult problem in itself. Stopping parts of an application without proper care can easily lead to deadlocks and other synchronization problems.

In addition, locking in a HPC application can be performance-critical. Indeed, for some HPC applications the latency of a synchronization can have a non-negligible cost. Poorly optimized or unnecessary synchronization for the purpose of reconfiguration could lead to performance drop. So, not only must locking be safe (which is difficult), but it must also be efficient (which is even more difficult).

On top of that, component-based programming requires that code must be accessed through interfaces, which complicates the problem further. Indeed, there exist approaches to locking in the literature which rely on code analysis to work. Such approaches are difficult to put in

practice in a component context. Indeed, such approaches cannot operate on interfaces alone (they need actual code), and ignoring interfaces would negate the benefits of the component approach. Additional behavioural information could be added to interfaces, but exposing too much of component behaviour also defeats the purpose of interfaces.

Overall, efficient locking of component subassemblies is a major roadblock before component models can support dynamic HPC applications. On one hand, existing HPC component models leave synchronization entirely up to the programmers. On the other hand, existing non-HPC approaches rely on costly over-synchronization (e.g., global lock) to solve the locking problem. Moreover, these approaches typically rely on underlying execution models with good properties (e.g., asynchronous messaging, continuations) which are difficult to implement without performance loss on top on traditional HPC languages such as C++ or FORTRAN.

The locking problem with all these constraints is a very difficult one; instead of tackling it in the general case, we propose to approach it from a specific angle to see what can be done. More specifically, we propose to extend component interfaces with simple method-level control metadata which correspond to the actual method-level control understanding of HPC programmers.

Concretely, this chapter proposes a minimalistic metadata scheme as an extension of the preliminary model presented in Chapter 4. This scheme consists in method-level behavioural information about the termination dependencies of code associated to ports. This chapter then proposes a locking paradigm and algorithm based on this extra information. The locking algorithm is a specialized two-phase protocol [49]. An evaluation of the approach is also presented, using a DirectL2C implementation.

5.1 Model and Algorithm

This section presents our contributions for this chapter. Those contributions consist of a control metadata scheme (Section 5.1.1), a locking paradigm (Section 5.1.2), and an algorithm (Section 5.1.3).

Those contributions are built on top of the preliminary model presented in the previous chapter (see Section 4.1).

5.1.1 Control Metadata

Motivation In the preliminary model introduced in the previous chapter, interfaces are mostly syntactic entities and provide little to no information about the behaviour of components. Basically, at assembly level, all that is known about a component is the name of its type and the list of its ports. This remark applies as well, to some extent, to other component models such as L2C or CCA. In theory, no extra hypothesis can be made regarding the behaviour of components in such models.

In practice however, to use third-party components in, say, a HPC application with non-trivial control, a type name and a list of ports is not sufficient. Since component interfaces in models like, e.g., L2C are largely under-specified, the end user performing the assembly must actually make hypothesis about their behaviour, which breaks the principle that only interfaces should be used.

Constraints Some models provide ways to specify component behaviour in interfaces, but they incur non-trivial extra work for component programmers. BIP [12], for example, provides an automaton-based model of the behaviour of components. While this is precisely the kind of information required, it requires the component programmer to devise such an automaton for

every component. Not only is this extra work, but it's also pretty far from the typical skillset of a HPC programmer. In addition, it might not easily model some unusual behaviours such as non-deterministic control.

Another possible excess of interface content that we wish to avoid is over-specification. It would technically be possible to put a lot of information in interfaces so as to have enough for control analysis purposes. At the limit, this could very well mean putting the whole code of the component into its interface. Obviously, this defeats the purpose of having an interface. Without necessarily going to such extremes, this can serve to illustrate several possible issues with over-specification: poor hiding of the component's complexity (i.e., it is a bad model), unnecessary constraints (e.g., implementation-dependent) which might hamper component evolution (i.e., poor encapsulation).

Proposition So as to avoid those pitfalls, we propose an intermediary approach: minimalist metadata that requires little to no extra effort for component programmers with HPC skillsets.

Definition 22 (Localized assembly with control metadata)**.** Let $\alpha = (C, P, o, R, r)$ be a localized assembly as defined in Section 4.1, a localized assembly with control metadata is defined by

$$\beta = (\alpha, m) \tag{5.1}$$

where m is the dependency function, defined on:

$$m \ : P \to (\{CALL(port) \mid port \in P\} \cup \{TERM(port) \mid port \in P\}) * \tag{5.2}$$

which verifies

$$\forall port, port' \mid CALL(port') \in m(port) \vee TERM(port') \in m(port),\ o(port) = o(port') \tag{5.3}$$

These metadata are made of port-to-port dependencies of two distinct kinds, represented by the unary operators $CALL$ and $TERM$

First, a *call dependency*, i.e., a dependency of the form $m(port) = CALL(port'))$, means that ○`port` may call ○`port'` during execution. For example, this could be the case if ○`port` was a method which calls ○`port'`.

Second, a *termination dependency*, i.e., a dependency of the form $m(port) = TERM(port'))$, means that ○`port` is potentially blocking and is dependent of activity on ○`port'` for the termination of blocking calls. For example, this could be the case for a `Com` communication component, with ports ○`blocking_send` and ○`receive`, where calls to ○`blocking_send` are blocking and require a call to ○`receive` to terminate.

The condition at the end of the definition forces dependencies to be internal to components. This represents the fact that dependencies are meant to be specified by component programmers, which cannot make assumptions regarding how the components are used.

Discussion: cost for programmers One the one hand, this kind of metadata is easy to specify for HPC programmers. Indeed, the list of blocking functions/methods and the required calls to unlock them are typically the kind of information that is found in the documentation of HPC technologies such as MPI [50]. All considerations of component models aside, knowing which calls are blocking and which are not is fundamental for HPC programmers, and should not require extra work.

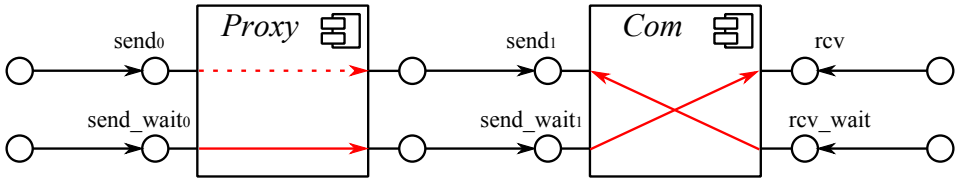


Figure 5.1: A subassembly implementing an asynchronous connector and a proxy, equipped with their control metadata. Red dotted arrows represent call dependencies while solid red arrows represent termination dependencies.

On the other hand, this approach might result in a decomposition of interfaces into ports with a finer grain that what might be natural in such a model. For example, having a single ○ `com_port` communication port for the ▭ `Com` component mentioned above does not work, because dependencies between the communication calls (e.g., blocking send/receive and corresponding receive/send) cannot be expressed separately. We argue, though, that this fine-grain decomposition into ports is necessary anyway, since those fine-grain ports are meaningful separate entities for control purposes.

Example: Figure 5.1 displays a subassembly implementing an asynchronous connection and a proxy. Ports ○ $\texttt{send}_0$, ○ $\texttt{send}_1$ and ○ `rcv` are non-blocking while ports ○ $\texttt{send\_wait}_0$, ○ $\texttt{send\_wait}_1$ and ○ `rcv_wait` are blocking and depend respectively on ○ $\texttt{send\_wait}_1$, ○ `rcv` and ○ $\texttt{send}_0$ to terminate.

The ○ $\texttt{send\_wait}_1$ port for example, has a termination dependency on ○ `rcv`, since calls on ○ $\texttt{send\_wait}_1$ depend on ○ `rcv` to terminate.

The ○ $\texttt{send}_0$ port has a call dependency on the use port pointing to the ○ $\texttt{send}_1$ port; this simply means that the functionalities provided by the ○ $\texttt{send}_0$ port of the proxy (e.g., a method) may use the reference to the ○ $\texttt{send}_0$ port of the ▭ `Com` component (e.g., to call a method provided by this other port).

5.1.2 Locking Paradigm

The semantics of the preliminary model (see Section 4.1) models call-stack-based execution but provides no way to act on an executing assembly. So as to be able to stop and lock an executing assembly, further assembly-level operations must be introduced.

Basic principle Locking a subassembly is a complex task which requires stopping the subassembly itself and making sure neighbouring computing components will not call any port inside the connection. Since it is not possible to stop a whole assembly at once with a single atomic operation, the locking process has to be divided into several operations, to be run sequentially or in parallel.

One possible approach, and the approach this chapter adopts, is to decompose the locking of a subassembly into the locking of several smaller parts of the subassembly. A prerequisite for this approach is, of course, that those small parts can be locked in an atomic fashion.

Granularity A first possible approach to locking a component subassembly is global locking, e.g., by having all the components perform global barriers during execution. This approach has the advantage of avoiding conflicts between concurrent locking attempts and of allowing global reconfigurations. However, global synchronization is known to cause scalability problems. Also,

in the case where reconfiguration is unpredictable, unnecessary global synchronization may be performed, provoking even more performance loss.

Although possibly efficient in certain cases, global reconfiguration is progressively abandoned by the literature as it is an obstacle for scalability. For this reason, the rest of this work focuses only on non-global approaches.

A second possible approach to locking is component-wise locking, i.e., locking where the smallest locking unit is the component. This approach is used by some existing reconfigurable component models [28]. While this approach allows concurrent reconfiguration, it means that a component must be fully stopped in order to prevent it from making outgoing calls, possibly leading to over-synchronization. For example, let us consider a component with four use ports pointing to as many neighbours; if the connection to one neighbour must be reconfigured, then the whole component must be stopped, which not only needlessly locks three use ports, but might also delay the computing done by the three corresponding neighbours. Stopping components that way can lead to a cascade of stopped components, possibly stopping large parts of an assembly. Moreover, several concurrent locking attempts touching the same component (e.g., several use ports of the same component) can conflict which would provoke further delays in the computation.

A third possible approach, which this paper advocates, is port-wise locking, i.e., fine-grain locking where it is possible to lock individual ports. Other recent works advocate this approach for performance reasons [31]. While this finer-grained approach might induce slightly higher overhead, it is more efficient in high-conflict cases. Indeed, the finer grain means reconfigurations have smaller footprints and that more of them can be done concurrently.

Model As just discussed, port-wise locking is likely an efficient approach but a precise definition of what it means to lock a port must be provided. Still building on top of the preliminary model from Section 4.1, notably on the execution model, we propose a locking paradigm (i.e., a definition of locking) called *mutex-based port-wise reconfiguration*.

First, an extension of the assembly state of the preliminary model is provided to add locking information to the runtime state:

Definition 23 (Assembly state with locking)**.** An assembly state is defined by the following grammar:

$$\langle thread \rangle ::= \bot \mid \langle thread \rangle\text{-}\langle port \rangle$$

$$\langle callState \rangle ::= \langle set\ of\ threads \rangle$$

$$\langle lockState \rangle ::= \langle set\ of\ ports \rangle$$

$$\langle state \rangle ::= (\langle assembly \rangle, \langle callState \rangle, \langle lockState \rangle)$$

Basically, this definition just adds a set of ports to the assembly state. The ports in this set are the ports currently locked.

Now, a modification and extension of the semantics of the preliminary model is provided so as to define new locking operations and their impact on pre-existing operations.

Definition 24 (Operational semantics with locking)**.** The operational semantics of localized assemblies with locking is defined by the following rules:

- **Terminate** an empty thread can be removed from the call state:

$$\frac{}{(\alpha, \sigma \cup \{\bot\}, \lambda) \ \rightarrow (\alpha, \sigma, \lambda)} \tag{5.4}$$

- **Return** a non-empty thread can return from the current method call to the parent method call:

$$\frac{}{(\alpha, \sigma \cup \{thread - port\}, \lambda) \ \rightarrow (\alpha, \sigma \cup \{thread\}, \lambda)} \tag{5.5}$$

- **Spawn** a new thread can be created on a component currently executing something:

$$\frac{port' \ \in P \qquad o(port) = o(port')}{(\alpha, \sigma \cup \{thread - port\}, \lambda) \ \rightarrow (\alpha, \sigma \cup \{thread - port\} \cup \{\bot - port'\}, \lambda)} \tag{5.6}$$

- **Local call** a thread can call any port on the same component as the current method call:

$$\frac{port' \ \in P \qquad o(port) = o(port') \qquad port' \notin \lambda}{(\alpha, \sigma \cup \{thread - port\}, \lambda) \ \rightarrow (\alpha, \sigma \cup \{thread - port - port'\}, \lambda)} \tag{5.7}$$

- **Distant call** a thread can call a distant port provided there is a connection between the component of the current method call and the distant port:

$$\frac{\begin{array}{c} port' \in P \qquad o(port) = compo \qquad o(port') = compo' \\ \exists use \in P \mid o(use) = compo, \ Ref(use) = port', \ use \notin \lambda \end{array}}{(\alpha, \sigma \cup \{thread - port\}, \lambda) \ \rightarrow (\alpha, \sigma \cup \{thread - port - port'\}, \lambda)} \tag{5.8}$$

- **Port lock** a port not already locked can be locked:

$$\frac{element \ \in P}{(\alpha, \sigma, \lambda) \ \rightarrow (\alpha, \sigma, \lambda \cup element)} \tag{5.9}$$

- **Port unlock** a locked port can be unlocked:

$$\frac{}{(\alpha, \sigma, \lambda \cup element) \ \rightarrow (\alpha, \sigma, \lambda)} \tag{5.10}$$

The two **Port lock** and **Port unlock** operations respectively add and remove ports to the list of locked ports. Locked ports have two impacts on the assembly behaviour:

- locked use ports cannot be used for distant calls;
- locked ports cannot be called locally.

Note that a locked use port can still be the target of a distant call. The motivation behind this is to ensure the precondition requires only a local check for the prospective caller. If such a call would happen, it would not be locked at all; preventing such calls requires locking the use provides with a reference on the provide port.

Now that the notion of port locking has been introduced, we can define the locking a subassembly:

Definition 25 (Locked subassembly)**.** A subassembly $\beta = (C', P', o', R', r')$ within assembly $\alpha = (C, P, o, R, r)$ with state $(\alpha, \sigma, \lambda)$ is said to be *locked* if the following conditions are met:

- **Locked internal ports** $\forall port \in P', \ port \in \lambda$
- **Locked incoming ports** $\forall port \in P \setminus P', (\exists port' \in P' \mid Ref(port) = port') \Rightarrow port \in \lambda$

- **No call stack** $\forall callstack \in \sigma,\ \forall port \in callstack,\ port \notin P'$

This definition imposes three conditions for subassembly locking. This first is that the ports inside the locked subassembly are themselves locked; this prevents local calls to them. The second conditions is that all references to ports in the subassembly are also locked; this prevents distant calls to ports in the subassembly. This second condition is a direct consequence of the fact that, in the semantics given above (Definition 24), distant calls can only be prevented by locking the use port. The third condition is that there is no call stack currently traversing the subassembly, so as to avoid thread well-formedness problems during reconfiguration.

Discussion The basic operations behind the paradigm itself are easy to implement in an efficient fashion. This paradigm is called "mutex-based" as it is equivalent to having a mutex per port which must be held in order to call or lock said port. The mutex check to call a port is always local. Indeed, a call is either intra-component (in which case everything is local) or distant, in which case it requires only to check the locking state of the use port (which is local to the caller).

While such an approach is not technically novel, it has not been used to our knowledge for port-wise locking of component assemblies. Moreover, the model presented here is tailored to our specific component model, and non-trivial design decisions were made, notably regarding its semantics.

However, while this paradigm provides basic easy-to-implement operations, it does not specify how to use said operations to actually lock subassemblies. Locking procedures which use these operations must be devised in order to lock subassemblies, and devising can possibly be complex.

5.1.3 Locking Algorithm

In order to bridge the gap between the basic locking operations described just above (in Section 5.1.2) and actually locking subassemblies, this section presents an algorithm which uses the metadata presented in Section 5.1.1 to devise locking procedures provided the target subassembly verifies certain properties.

This algorithm is a two-phase algorithm [49], specialized for our particular problem. A two-phase algorithm is a locking protocol in which individual locks are requested (growing phase) until a satisfying locking state is achieved (in our case, a particular subassembly is locked); then all locks are released in turn (shrinking phase).

Termination dependencies Ultimately, what we want is to know which port must be locked before which other port to avoid provoking deadlocks. To this end, we propose to introduce a relation between use ports which basically amounts to "use port p_0 must be locked before use port p_1 otherwise a deadlock may occur". There are two reasons for which the stopping of a port could deadlock another one.

First, a port p_0 may be blocking and depend on port p_1 being alive to terminate. In this case, not only must p_1 not be locked before p_0, but other ports that might call p_1 or that p_1 might call must also not be locked (so that p_1 is still alive). This relation is captured by the following rule:

> **Rule 1 (liveness of termination dependency):** A use port p_0 is connected to another use port p_1 if there exists in the assembly a path from p_0 to p_1 which is composed of at least one termination dependency oriented from p_0 to p_1 and any number of use ports and call dependencies with any orientation.

Second, a port p_0 might be able to call a port p_1, for example through a call dependency, and needs p_1 to not have been locked. Not only that, but p_1's own call dependencies must also not have been locked and so on. This second relation is captured by the following rule:

> **Rule 2 (indirect call dependencies):** A use port p_0 is connected to another use port p_1 if there exists in the assembly a path from p_0 to p_1 which is composed of at least one call dependency oriented from p_0 to p_1 and any number of use ports with any orientation.

These two rules can be used to compute a graph of termination dependencies between the use ports in a subassembly.

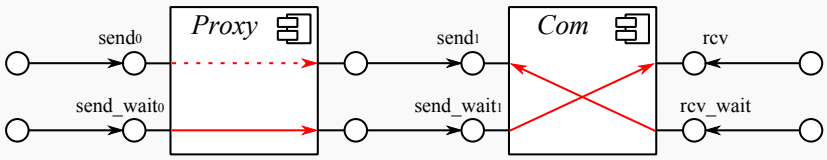
Algorithm Now that indirect dependencies have been properly defined, they can be used to compute a valid locking order using the following algorithm:

```
1.    function lock_order(subassembly S):
2.      Compute a graph of termination dependencies from S (see below)
3.      Compute a topological order R of this graph.
4.      If the topological sort terminated:
5.            return R
```

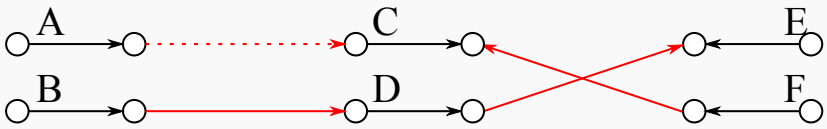
First, remember that the graph of termination dependencies is a graph of use ports only, and thus the result is a list of use ports. This is because of the semantics of mutex-locking: locking provide ports does not prevent distant calls, and thus the corresponding use ports must be locked instead.

Second, notice that the algorithm relies on a topological sort of the dependency graph. Topological sort returns an order of the vertices such that, if v_0 and v_1 are two vertices such that $v_0 < v_1$ for the topological order, then there is no edge from v_1 to v_0, i.e., all edges are in the direction of increasing order. Using a topological sorting guarantees that no use port is locked before a port that has a termination dependency on it, thus ensuring a correct ordering of locks. However, topological sort terminates iff there is no loop in the graph, i.e., if there is no dependency cycle.

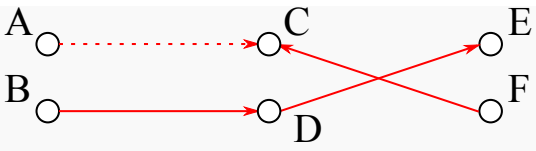
Example: Let us consider the asynchronous connector subassembly from Figure 5.1, and apply the algorithm to it:



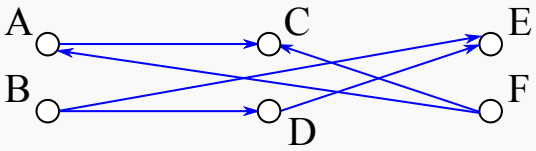
Since the components themselves do not matter for the purpose of this algorithm, let us remove them and consider only the graph of ports (and name them for clarity):



Now, let us keep only the use ports, by collapsing the use/provide connections. While technically, the algorithm computes the graph of termination dependencies directly, it can be done in two steps to better understand what happens:



Now, termination dependencies must be computed. Direct dependencies are simply the union of termination and call dependencies, while indirect dependencies are given by the two rules presented above. The edge from A to F results from Rule 2.



This graph represents relations of the type "must be stopped before". Consequently, a topological order of the use ports, if one such order exists, is a valid stopping order. Let us compute said topological order:

Now, since the topological sort did not fail, then this order is a valid stopping order and can be returned.

Correction The algorithm gives a locking order which is guaranteed to terminate eventually under the following conditions:

- use ports from outside to ports in the subassembly are always alive;
- the subassembly does not contain any dependency loop (otherwise the topological would fail).

The correction of this algorithm relies on the fact that the two construction rules given above capture all indirect dependencies. Indeed, there are only two possible reasons for a call to a port p_0 to be blocking: either its implementation is blocking in which case a termination dependency must be present in the metadata and this dependency must be locked after p_0 (Rule 1) or its implementation is non-blocking but it can call an external port (which might be locked) in which case said port must be locked after p_0 (Rule 2).

Complexity This algorithm has a polynomial worst-time complexity. A fairly straightforward upper-bound is $O(|E|*(|E|+|V|^2))$ to compute indirect termination dependencies and $O(|E|+|V|)$ to compute the topological sorting using a classic algorithm.

Here is how the indirect dependencies given by Rule 1 can be computed:

```
1.    For each termination dependency d=(v0,v1):      (O(|E|) dependencies)
2.       Remove d from the graph
3.       Compute V0 and V1, the lists of vertices in the connected\
           components of v0 and v1 respectively        (O(|E|) time using a BFS)
4.       Add all couples from V0*V1 to the list of termination dependencies
                                                       (O(|V|^2) couples)
```

where BFS stands for Breadth-First Search.

Hence an overall complexity of $O(|E|*(|E|+|V|^2))$ for Rule 1 dependencies. Rule 2 is a simpler case of the same algorithm.

5.1.4 Discussion

The two conditions for the algorithm to work (no dependency loop and inbound alive dependencies) are strong conditions which are very limiting in the general case. Moreover, even in the case where it terminates there is no guarantee on the time it can take to terminate.

Stencil applications However, this algorithm works well for certain classes of assemblies. One example of such a class is communication subassemblies in stencil applications.

Stencil codes are a commonly-used pattern in high-performance computing. Various definitions of stencil applications can be found in the literature; see e.g., [97]. For the purpose of the chapter, we consider a component-specialized definition:

Definition 26 (Stencil assembly)**.** A *stencil assembly* is a set of *computing components* which expose *communication ports*. Communication ports are connected following a given topology and using subassemblies dedicated to communication (e.g., proxies, connectors) called *communication subassemblies*.

In a stencil assembly there is exactly one thread per computing component which implements the following algorithm: until a certain termination condition is met, do some computation on local data then exchange data with neighbours and then start over.

Indeed, the first condition (non dependency loop) will be true for a lot of communication subassemblies since they have no internal threads and they typically have a tree structure (e.g., 1-to-n interpolator, linear proxies). Even in the case where there is a dependency loop, it can often be split by over-decomposing the ports (e.g., Figure 5.1 would feature dependency loops if *send/receive* and *wait* ports were not separated). The second condition (alive inbound dependencies) is given by the computation component behaviour. Finally, the behaviour of computing components guarantees that their call stacks will exit the communication subassemblies at least once per iteration, giving an upper bound on locking time.

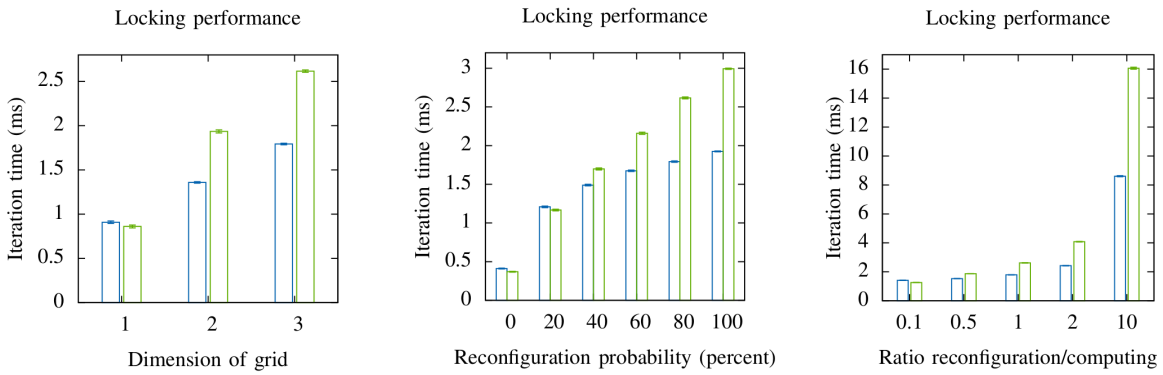
Comparison to literature Compared to the component literature, this work is the only one to our knowledge to deal with concurrent port-wise locking in call-stack-based assemblies.

Apart from the component literature, there exist a rich literature on deadlock prevention and detection. However, component models introduce the added constraint that components are blackbox entities which provide only the guarantees explicitly given in its interface or metadata. This means that works which rely on analysis of the code (e.g., [63, 27]) cannot be applied directly to component assemblies, and require some modification to work with interfaces instead of code.

5.2 Evaluation

The present section first evaluates our approach in terms of performance on a set of stencil benchmarks; then it presents a full reconfigurable example built using a dedicated component model called DirectMOD, and it discusses how it is easier to use compared to a traditional HPC component model.

Both sets of benchmarks have been implemented using DirectL2C, an implementation of DirectMOD based on L2C [21] and MadMPI [9] (which provides support for `MPI_THREAD_MULTIPLE`). Notably, DirectL2C implements mutex-locking through macros which add a mutex lock for each port call.



(a) Varying dimension of the grid. Number of components is always 64. Configurations are 64x1x1 (1D), 8x8x1 (2D) and 4x4x4 (3D). All other parameters have default values.

(b) Varying reconfiguration probability between 0 and 1. All other parameters have default values.

(c) Varying ratio between reconfiguration and performance varying from 1/10 to 10. All other parameters have default values.

Figure 5.2: Comparison between the performance of port-wise locking and component-wise locking. Each plots focuses on the variation of one parameter. For each plot, the blue bars (on the left) correspond to the performance of our approach (port-wise locking) while the green bars (on the right) correspond to component-wise locking.

Since both sets of benchmarks rely on a small number of communication subassemblies, the port-wise locking order for each subassembly has been computed by hand using our algorithm and hard-coded into dedicated locking components.

5.2.1 Locking Performance on Stencil Benchmark

In order to evaluate the performance of our approach we have programmed a stencil benchmark; we have also conducted a set of experiments to compare our approach (which allows individual locking of ports) to a component-wise locking approach as implemented by component models from the literature. The factors which can affect the performance of both approaches have been discussed in Section 5.1.2; the present section aims to confirm the expected behaviour and measure it. Since our approach focuses on call stacks, experiments are multithreaded and single-node.

Our benchmark assembly is a grid-based stencil assembly with "placeholder" computing components (i.e., computing components which do an active sleep instead of actually computing something), and asynchronous communications. The communication subassemblies used in our benchmark are composed of two 1-to-1 asynchronous connectors similar to the `Com` component from Figure 5.1. In addition to connectors, each communication subassembly includes a component dedicated to locking which also emulates a reconfiguration with an active sleep. Several parameters can vary:

- it can either be a 1D, 2D or 3D grid of components;
- the "computing time" of the components and the "reconfiguration time" of locking components can be changed;
- locking can be either port-wise using our approach (*PortLock* components) or component-wise (*CompoLock* components) using one mutex per component as discussed in Section 5.1.2;

- the probability that a reconfiguration occurs at the end of an iteration for each component is a parameter.

Experiments were conducted on the Grid'5000 platform [10], more precisely on a node of the graphite cluster [1] on the Nancy site. Each experiment has been conducted 10 times for each measurement and the data shown in the figures correspond to the median of these runs with standard deviation bars.

In order to evaluate the impact of each parameter on performance, a set of "default" non-extreme parameters have been selected and then three sets of experiments have been conducted, each changing one parameter while keeping the others at default values. The default parameters are a 4x4x4 grid of component (i.e., a 3D grid of 64 components) with 100 μs computing time and reconfiguration time (enough for overheads not to be noticeable and equal reconfiguration and computing times) and a probability of reconfiguration of 80% at the end of every iteration (low-probability reconfiguration is not interesting since we want to study conflicts between reconfigurations).

Figure 5.2 sums up the results of the experiments. Figure 5.2.a shows that as the dimension of the grid (and the number of neighbours of each component) increases, the overall iteration time increases (for both approaches) as expected as the number of connections increases; it also shows that port-wise becomes increasingly more efficient compared to component-wise locking (up to 46% less time for the 3D assembly) while the component-wise approach is slightly more efficient in the 1D case, as it does not require mutex locking on every call. Figure 5.2.b shows that, as the probability of reconfiguration increases, port-wise becomes more efficient (component-wise is up to 56% worse) while component-wise is slightly more efficient with very low probability. Similarly, Figure 5.2.c shows that port-wise is more and more efficient as reconfiguration time increases in comparison to execution time (component-wise is up to 87% worse) while component-wise is more efficient for very low reconfiguration time.

Overall, these results confirm the analysis from Section 5.1.2: port-wise is a more efficient approach when conflicts are likely while component-wise is simpler and slightly more efficient in low-conflict cases.

5.2.2 Software Engineering Properties on AMR Benchmark

This section evaluates the contributions of this chapter on a set AMR-based benchmarks, through experiments and code metrics.

We have implemented mutex-based port-wise locking into DirectL2C in the form of macros to add mutex locks to port calls. We have also implemented a 2D AMR benchmark following the 2:1 rule (see Section 2.1.3), using the proposed locking algorithm to devise reconfiguration algorithms. As in the previous section, these benchmarks implement "placeholder" computing in computing components but contrary to the previous section, these benchmarks implement actual reconfiguration (refinement). Apart from the added reconfiguration and the possibility of 1-to-2 connections, these two benchmarks implement the same assembly as the 2D version from the previous section. The DirectL2C assembly also reuses the components from the previous section.

Let compare this DirectL2C assembly, called *AMRDirect*, with another assembly written in L2C (as a traditional HPC component model with no support for reconfiguration), called *AMRL2C*, which implements exactly the same benchmark.

Table 5.1 presents the breakdown of the codes of both benchmarks. Overall, the code of *AMRDirect* (which uses features from DirectMOD and our locking approach) is much shorter (146 lines of code instead of 501). Also, *AMRL2C* includes by-hand synchronization code (using

[1] 2 Intel Xeon E5-2650 at 2.00 Ghz for a total of 16 cores

Component	*AMRL2C*	*AMRDirect*
Init	137	23*
Compute		16
Iter		29
AMR2D	166	52
Com	198	26
Transformation		19*
(ComHybrid)		(43)
(MpiProxy)		(16)
TOTAL MT	501	146

Table 5.1: Code size of both AMR assemblies. Sizes are expressed in number of significant lines of code (i.e., not a comment and not a block delimiter). All codes are C++ excepted those marked with a * which are DirectL2C transformations. The two components between parenthesis are components which are not part of the multithreaded implementation. TOTAL MT denotes the total number of line for the multithreaded implementation (i.e., not including the components between brackets).

mutexes and conditions) which is bug-prone and difficult to write while *AMRDirect* benefits from guaranteed locking from our algorithm.

It is also of note that the DirectL2C approach provides easier reuse. Indeed, in L2C the reuse of third-party components in such an assembly is very difficult since there are no guarantees on the behaviour of components and any third-party component might break the locking. Also, since there is no model-level convention for locking in L2C, extra work might be required to adapt third-party components to the locking interfaces. On the other hand, DirectL2C provides model-level locking interfaces and our locking algorithm provides a way to use third-party components without writing any new synchronization code (or at least it enables the detection of dependency loops).

An example of a situation where the reuse capabilities of DirectL2C can be useful is the modification of this 2D AMR benchmark for distributed assemblies. A simple way to do that is to implement a MPI proxy component and an asynchronous hybrid C++/MPI connector. Table 5.1 shows that the code of those two components is very short. Including them in the assembly is only a matter of writing a new assembly by using these two new components and reusing the components from the local assembly, writing new DirectL2C transformations (to describe the reconfiguration of the new connectors, see Chapter 4 for details) and executing our algorithm to compute the new locking algorithms.

5.3 Conclusion

This chapter has dealt with the problem of efficient subassembly locking in concurrent call-stack-based component models. It has presented an efficient but complex locking paradigm; to simplify it, it has presented an algorithm to automate locking in certain cases. While our approach has some strong limitations, it is efficient when applicable, and it is at least usable with stencil application, which are a relevant use case. Finally, the performance of our locking approach has been evaluated on a stencil-based multithreaded benchmark and the ease of development has been evaluated by comparing two AMR benchmark implementations with and without our approach.

Overall, on dynamic stencil assemblies (and possibly other classes of applications), this approach is more efficient than traditional component model locking paradigms and it is easier to write and adapt than existing HPC component models.

Chapter 6

A Specialization Model For Hierarchical Component Assemblies

Contents

Software specialisation is the process of modifying a general-purpose program in order to make it more efficient for a specific subset of possible inputs or use cases. Examples of software specialisation include partial compilation as well as application adaptation to specific hardware.

By extension, the process of building an application through successive implementation decisions can be seen as a form of specialisation from an (abstract) application description to a (concrete) implementation. With this more general definition, processes such as variant selection or instantiation from a feature model are a form of software specialisation.

In particular, specialisation of component assemblies has been the focus of extensive research and raises specific challenges. Examples include component-based feature models and automatic variant selection in component models. Challenges raised include efficient exploration of the decision space and composite specialisation in hierarchical models.

We argue that those component assembly specialisation processes share a common structure: a sequence of specialisation decisions are made until the assembly satisfies specific constraints. With this perspective, composition and reuse of such processes should be easy but the current lack of technological or formal conventions make it difficult.

So as to ease reuse and composition of component assembly specialisation processes, we propose a formal specialisation model for component assemblies, called SpecMOD. This model

takes the form of a calculus which formalizes partial specialisation in hierarchical component assemblies as an operation on a formal assembly. In this context, a specialisation process is an oracle which chooses which operation to trigger from a set of possible specialisation operations. Such oracles can easily be composed and reused. Generic oracles can even be written to implement generic decision tree exploration algorithms.

We evaluate the calculus following two directions. First, we consider several common component model features from the literature and show how to encode them in our approach. Second, we discuss specialisation processes from the literature, such as product lines and automatic variant selection. In addition, we show how those processes can be composed and we argue about the usefulness of such a composition.

The structure of this chapter is as follows: Section 6.1 introduces the calculus through a series of formal definitions and a running example; then, Section 6.2 shows the generality of the approach by encoding common component model features, whereas Section 6.3.1 discusses the usefulness of the approach, and Section 6.3.2 presents a DirectMOD-based use case; finally, Section 6.4 concludes.

6.1 SpecMOD, A Calculus for Assembly Specialisation

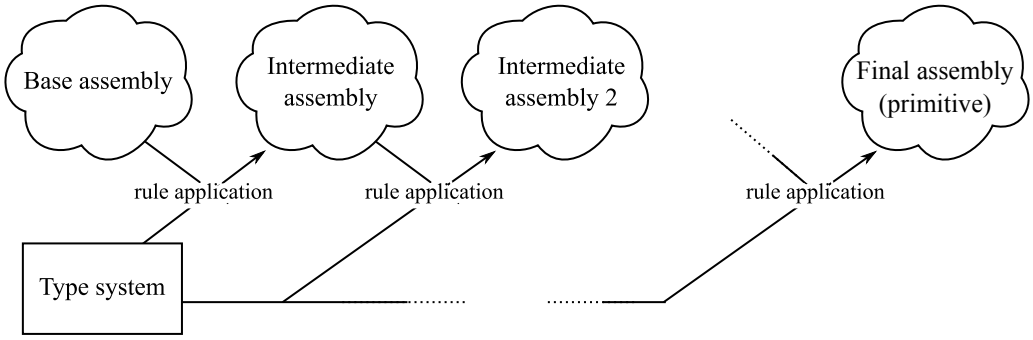


Figure 6.1: Overview of the use of the calculus. The type system is represented by a square while the cloud shapes represent component assemblies. The arrows represent the application of operations to transform the assembly step-by-step.

This section presents our proposal: SpecMOD, a generic calculus for component assembly specialisation. It is defined by an *assembly model*, a *type system*, and an *operational semantics*. Figure 6.1 presents how those three parts come together to perform assembly specialisation. The idea behind the calculus is to be able to perform step-by-step specialisation, one component at a time, until a working assembly is obtained (called primitive), using a specialisation relation provided by a type system.

The section is divided into six sections, the first four introducing the model and its semantics; the fifth, Section 6.1.5, presenting a complete example of how the model works; and Section 6.1.6 presenting a variant of the calculus that is slightly more complex but supports reversible operations.

6.1.1 Assembly model

We introduce an assembly model whose goal is to be as general as possible. Our approach to generality is to propose a simple model with few very general concepts and to introduce advanced concepts by encoding them in the assembly model. Examples of such encodings are given in Section 6.2.

Contrary to Chapter 4 and Chapter 5, this chapter does not make use of the preliminary component model introduced in Section 4.1. In the present chapter, we are not interested in runtime or control properties of components, so the semantics of the preliminary would have been of little use. In addition, SpecMOD is not meant to be HPC-specific, so the HPC-inspired features of the preliminary model make a lot less sense. Finally, SpecMOD is more mechanically complex than DirectMOD, so the base model needed to be as simple as possible to avoid unnecessary syntax-induced complexity.

Definition The definition of a DirectMOD component assembly is given below. We assume, for now, the existence of T_c and T_e, which are sets of component types and endpoint types respectively.

Definition 27. A *component assembly* is defined by:

- a component set C;
- an endpoint set E;
- a connection set X;
- an endpoint implementation set I;
- an implementation source function $i_s : I \to E$;
- an implementation target function $i_t : I \to E*$ [a];
- a connection source function $x_s : X \to C$;
- a connection target function $x_t : X \to E*$;
- a component type function $t_c : C \to T_c$;
- an endpoint type function $t_e : E \to T_e$.

[a] $*$ is the Kleene star.

The assembly model is a graph of *components* and *endpoints*. Components can represent either traditional components or be used to encode other architectural elements such as connectors. In that sense, component in the model can be thought of as generic architectural elements. In addition to that, *endpoints* are introduced; they aim to model interface constraints. Endpoints can be thought of as analogous to ports in classical component models.

To simplify the encoding of component interfaces (i.e., which outgoing edge corresponds to which endpoint), the definition is based on *multi-sorted list graphs* (as defined in [42]) instead of traditional graphs.

List graphs provide *ordered edges* that have one source and several (ordered) targets. These ordered edges allow the encoding of interfaces comprising several endpoints without having to resort to edge labels or extra vertices.

Example: Figure 6.2 gives an example of a possible assembly and some hints at what endpoints are used for. The presented assembly is a partially-specialised master-worker assembly that would typically be a step during a specialisation process. Note that the `1-to-2 use/provide` component is used to encode what would be a connector in a traditional component model (since our assembly model has no connector concept). Also note the endpoint hierarchy between the `1-to-2 use/provide` and `n workers` components. This hierarchy is here to serve as a buffer between two components which are not yet as specialised: the `1-to-2 use/provide` component is already specialised for the value 2 and it is connected to the adequate endpoints; the `n workers` component has not yet been specialised for the value 2 and it is still connected to the un-specialised root endpoint.

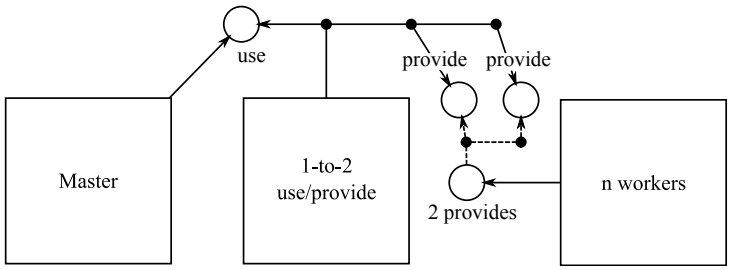


Figure 6.2: A half-specialised master/worker assembly. Components are represented with squares while endpoints are represented with circles. List edges are represented with lines and arrows similarly to [42]. The targets of list edges are ordered from closest to farthest from the source. Dotted list edges are implementations while solid list edges are interfaces.

6.1.2 Type System

To perform specialisation on black-box components, we propose to have a type system equipped with a specialisation relation which specifies which component type can be specialised into which other. This section defines such a type system for the calculus. In addition to a straightforward type hierarchy, type systems introduce composite component types (in the traditional sense) and composite endpoint types (which are used to replace a single endpoint with multiple ones, e.g., when specialising for a specific arity).

Definition 28. A *type system* is defined by the following:

- a set of component types T_c;
- a set of primitive component types $T_c^p \subset T_c$;
- a set of composite component types $T_c^c \subset T_c$;
- the signature constraint function $s : T_c \to T_e*$;
- a set of builders $B \in strans$;
- a set of endpoint types T_e;
- a set of composite endpoints $T_e^c \subset T_e$;
- a set of primitive endpoints $T_e^p \subset T_e$;
- the component implementation function $i_c : T_c^c \to B$;
- the endpoint implementation function $i_e : T_e^c \to P*$;
- a specialisation relation on component types $<_c$ (a partial order);
- a specialisation relation on endpoint types $<_e$ (a partial order).

Where *strans* is the set of transformations on SpecMOD assemblies. See Section 3.2 for a brief presentation of graph transformations. Graph transformation can be defined on SpecMOD assemblies as they are multi-sorted graphs with list edges (see [42, 83]). Section 6.1.3 gives constraints for these transformations and examples of such transformations are given in Figure 6.3 and explained below.

Component implementations (given by i_c) are meant to represent composites as traditionally understood in component models. The implementation of a composite component in the calculus is given by a *builder* which is a graph transformation which replaces the composite by its implementation in the system. Further details about the constraints on these transformations and how they are used are given in Section 6.1.4.

The implementation of an endpoint (given by i_e) is a list of endpoint types. Contrary to composite components, composite endpoints are not meant to be replaced by arbitrary assemblies

but can only be divided into several endpoints. Endpoint implementation can typically be used to resolve n-to-m or 1-to-n connections by replacing a single composite endpoint with n endpoints; Figure 6.2 illustrates it with an endpoint *2-provides* being implemented by two distinct *provide* endpoints.

There are different sets of components and endpoints (abstract, composite and primitive), which are meant to capture the level of abstraction of components and endpoints. Components and endpoints which are neither primitive nor composite are called *abstract*. The sets of abstract components and endpoints are denoted T_c^a and T_e^a respectively. Abstract component and endpoints are uninstantiable while primitive ones have a blackbox implementation and composite ones have an explicit model-level implementation (represented by i_c and i_e).

In addition, let us define primitive and abstract assemblies. This notion is useful for determining whether an assembly can be instantiated or not.

Definition 29. 1. An assembly is *primitive* iff

$$\forall c \in C, t_c(c) \in T_c^p \quad \textit{and} \quad \forall e \in E, t_e(e) \in T_e^p$$

2. An assembly that is not primitive is *abstract*.

Example: Figure 6.3 presents a complete example of a type system for simple master-worker assemblies such as the one from Figure 6.2. At the top-right is the hierarchy of connector component types which comprises various use/provide connectors; other component types include various masters, workers and worker sets; on the middle-left is the hierarchy of endpoints which comprises use and provide endpoints; at the bottom are the endpoint and component implementations.

Note that specialisation arrows can represent various kinds of specialisations: the arrow from *Worker* to *Worker A* corresponds to a variant selection while the arrow from *n identical workers* to *2 identical workers* represent setting a value for an integer parameter and the arrow from *n workers* to *n identical workers* represents the addition of an architectural constraint.

Also note that, if we wanted to have all possible worker set sizes, this type system would need to include an infinite number of component types (e.g., one *i workers* type for every integer i) along with an infinite number of corresponding endpoints and implementations. A more practical way to specify such an infinite system is given in Section 6.2.2 in the form of a parametric type system.

6.1.3 Well-Formedness

Not all the assemblies and type systems as defined above make sense. For them to be meaningful, certain constraints must be respected. We model these constraint by introducing *well-formed assemblies*.

First, we introduce the notion of *component signatures*. A component signature in an assembly is the list of the types of the endpoints it is connected to.

Definition 30. Let $c \in C$ be a component, if $\exists!\, i \in X, x_s(i) = c$ and $x_t(i) = \{i_0, \dots i_k\}$ then we define $sig(c) = \{t_e(i_0), \dots t_e(i_k)\}$ the *signature* of c.

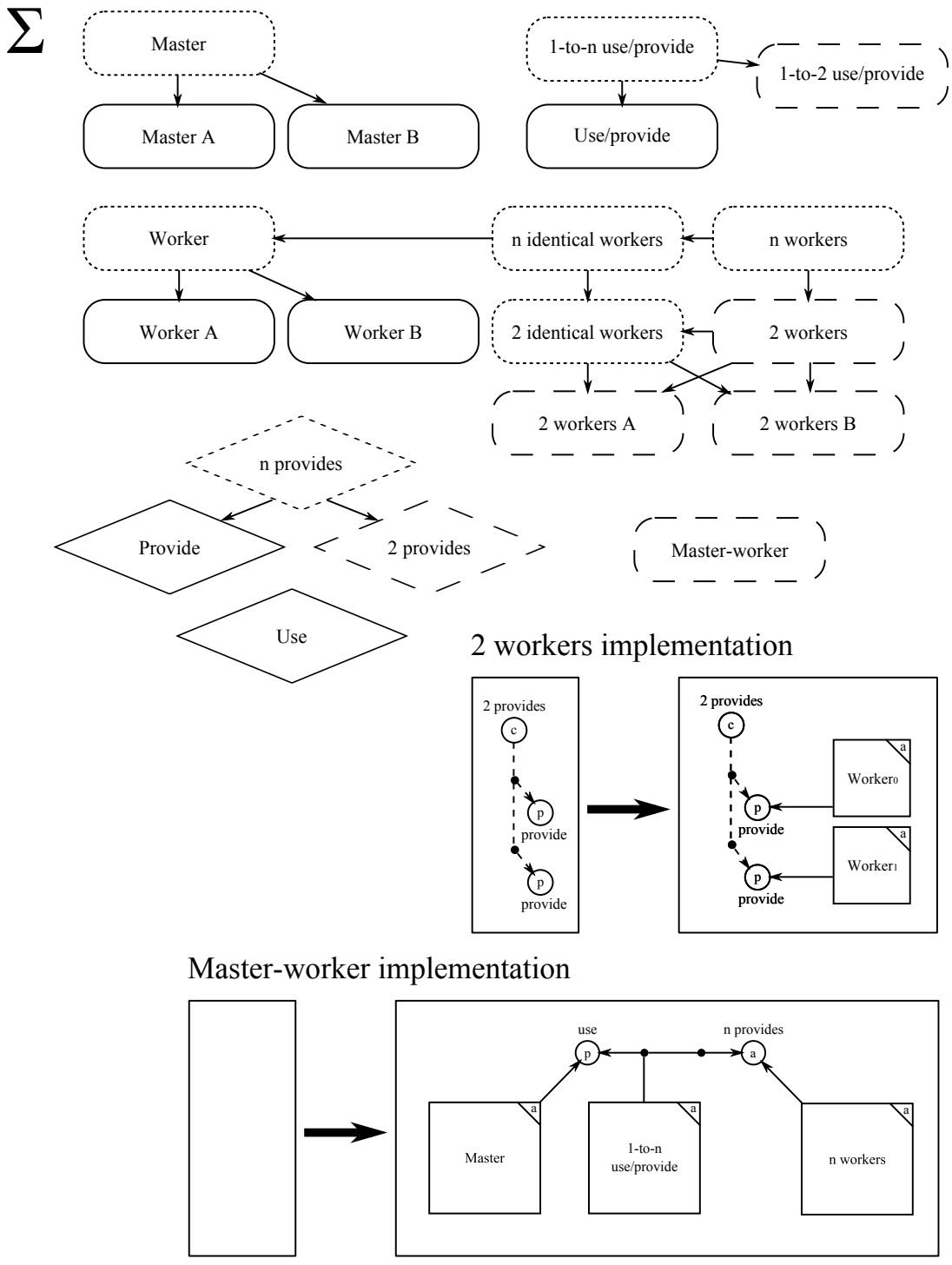


Figure 6.3: Example of a type system named Σ for master-worker assemblies. Component types are represented with rounded rectangles while endpoint types are represented with diamonds. Types with solid borders are primitive; those with dotted borders are abstract while those with dashed borders are composite. Arrows represent the specialisation relation (if there is an arrow from A to B then B is a specialisation of A). The full specialisation relation is the transitive closure of the relation presented here. The boxes at the bottom represent the builders for the composite endpoints and components as graph transformation with implicit mapping. For clarity, components and endpoints in the transformations have their abstraction level written on them (a for abstract, c for composite or p for primitive)

Definition 31. The specialisation relation on signatures $<_{sig}$ is defined by: let $r = \{r_0 \dots r_n\}$ and $s = \{s_0 \dots s_m\}$ be two signatures then $r <_{sig} s$ iff $n = m$ and $\forall i \in \{0 \dots n\}, r_i <_e s_i$.

where $<_e$ is the endpoint specialisation relation, see Definition 28.

Theorem 2. $<_{sig}$ is a partial order.

Now, we can define what is a *well-formed* assembly.

Definition 32. An assembly A is *well-formed* according to a type system Σ iff
- each component has at most one interface, i.e., $\forall c \in C, sig(c)$ is defined;
- the type of every endpoint and component in A appears in Σ;
- each component interface complies with the component's type signature constraint, i.e., $\forall c \in C, sig(c) <_{sig} s(t_c(c))$;
- composite implementations are well-formed, i.e., $\forall c \in T_c^c$ the origin of $i_c(c)$ is an assembly containing only the endpoints from $s(c)$ and the destination of $i_c(c)$ contains at least the endpoints from $s(c)$.

where s is the signature constraint function, see Definition28.

The first condition means that all connections from a single component must be ordered. If multiple connections per components were allowed, there would be no order between the endpoints connected by different connections. This order on connected endpoints is important to encode which one is connected to which port of the component.

The third condition means that components cannot be connected to any kind of endpoints and that they are constrained by the type system *signature constraint*. This constraint is important as a guarantee on the connected endpoints which can be used for composite implementation.

The fourth condition means firstly that the only hypothesis a composite implementation is allowed to make is that the endpoints it is connected to respect the signature constraints and secondly that a composite implementation cannot remove the endpoints it is connected to.

In the rest of the paper, we assume that assemblies and type systems are well-formed unless specified otherwise.

6.1.4 Operational Semantics

This section defines operations that can specialise an assembly according to a type system. These operations implement local specialisation (per component or per endpoint); they are meant to be used in multi-step transformations as presented in Figure 6.1.

Operations:

- Specialise endpoint ($\mathbf{sp_e}(\mathbf{e}, \mathbf{B})$): let e be an endpoint of type A and $B \in T_e$ such that $B <_e A$ then replace type of e by B.
- Specialise component ($\mathbf{sp_c}(\mathbf{c}, \mathbf{B})$): let c be an component of type A and $B \in T_c$ such that $B <_c A$ and such that $sig(c) <_{sig} s(B)$ then replace type of c by B.
- Remove unused endpoint ($\mathbf{rm}(\mathbf{e})$): let e be an endpoint such that there exists no $i \in X$ such that $e \in x_t(i)$ and there exists no $i \in I$ such that $e \in i_t(i)$ then remove e from assembly and remove its implementation if it had one.
- Implement composite ($\mathbf{im_c}(\mathbf{c})$): let c be a composite component, then apply $i_c(c)$ to the subassembly composed of c, its outgoing connection if there is one and all the endpoints it is connected to.
- Implement endpoint ($\mathbf{im_e}(\mathbf{e})$): let e be a composite endpoint, add a new implementation i such that $i_s(i) = e$; create new endpoints $e_0 \dots e_n$ such that $n = |i_e(e)|$, $\forall i \in \{1 \dots n\} t_e(e_i) = i_e(e)_i$ and $i_t(i) = \{e_0 \dots e_n\}$.

Definition 33. The *set of valid operations* on assembly A according to type system Σ is the set of all operations o, denoted $V(A, \Sigma)$, which verify:

- The target endpoint or component belongs to A.
- If o is a specialisation operation ($\mathbf{sp_e}$ or $\mathbf{sp_c}$) then the target type is from Σ.
- All conditions for the operation are met (e.g., the target type for specialisation is indeed a specialisation of the current type).

Definition 34. A *specialisation process* is an algorithm which takes as parameter a type system Σ and an assembly A and returns an operation from $V(A, \Sigma)$.

This definition means that a specialisation process is a black box responsible for selecting a valid operation at each step of the specialisation. It can be thought of as an oracle or a chooser which selects a path in the tree of possible specialisation decisions. Whether such specialisation processes exist in the literature and whether the calculus presented here is useful to reuse and compose them are questions discussed in Section 6.3.1.

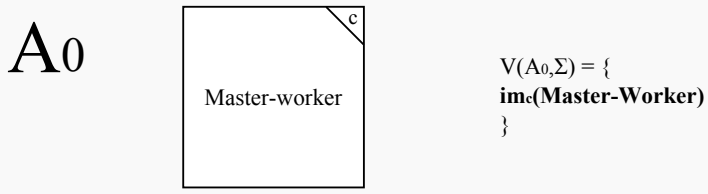
Theorem 3. The five operations preserve well-formedness.

Proof. In order of appearance in the well-formedness definition (see Section 6.1.3): the well-formedness of interfaces holds since no operation adds interfaces to an existing component; the new types all belong to the type system (constraint of sp_c and sp_e); the signature constraints are preserved because it is a condition for im_c and the builders are still well-formed because operations do not touch the type system. □

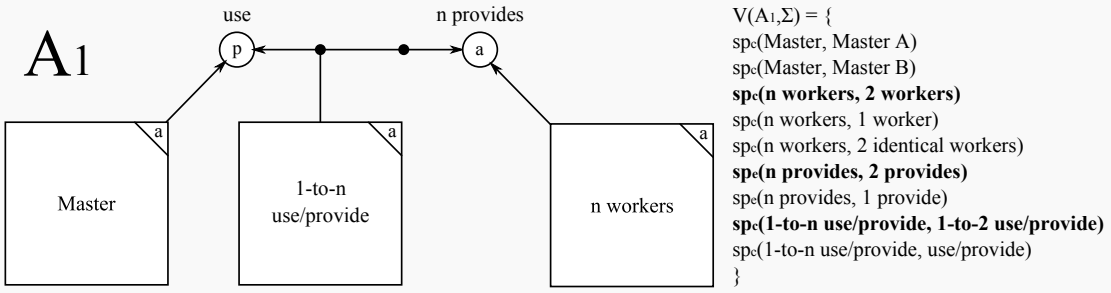
6.1.5 Full Example

This section presents a full example of an assembly specialisation from a very abstract single component assembly to a primitive multi-component assembly. Specifically, it deals with the example of a master-worker assembly built from the type system presented in Figure 6.3. Hereafter are presented the assembly, the set of valid operations and the chosen operation at each step of the transformation (in bold). Not all specialisation steps are presented in order to save space.

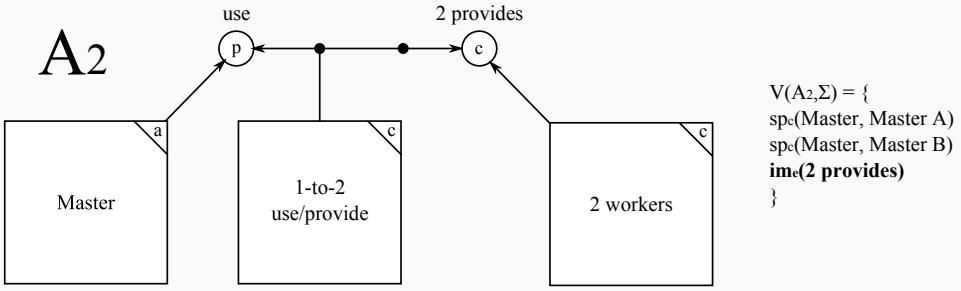
Example: At first (assembly A_0), there is only a single *Master-Worker* composite component:



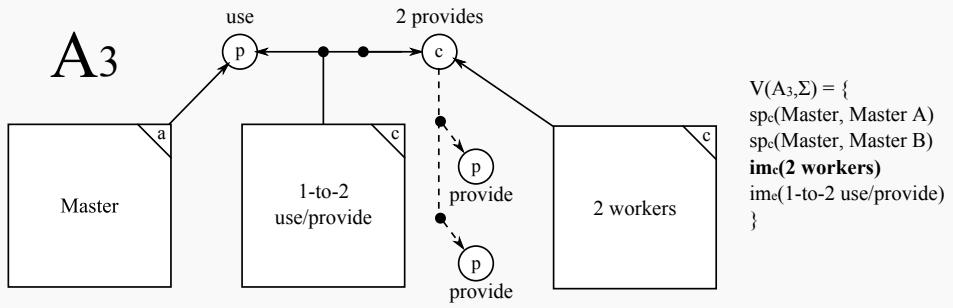
So far no architectural decision has been taken apart from the fact that is assembly must be a master-worker style assembly. The first operation is to implement the *Master-Worker* component by applying the builder from the type system; this results in a assembly with a *Master* component and an abstract set of workers which are connected with an abstract $1 - to - n$ *use* connection:



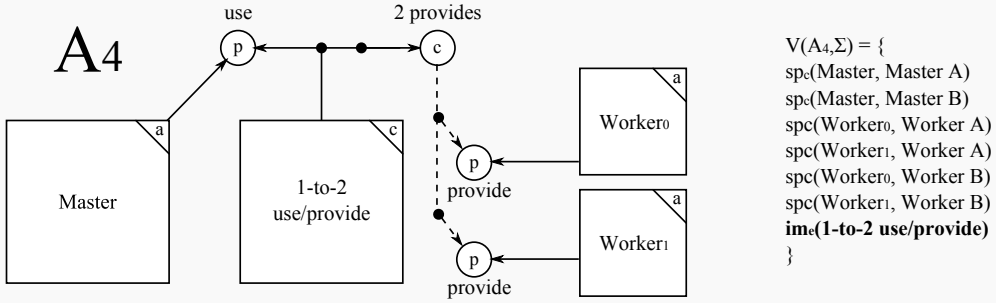
Then, it is decided that the number of workers will be 2 and all the relevant endpoints and components are specialised for this value (the connection, the worker set and the multiple provide endpoint) resulting in assembly A_2:



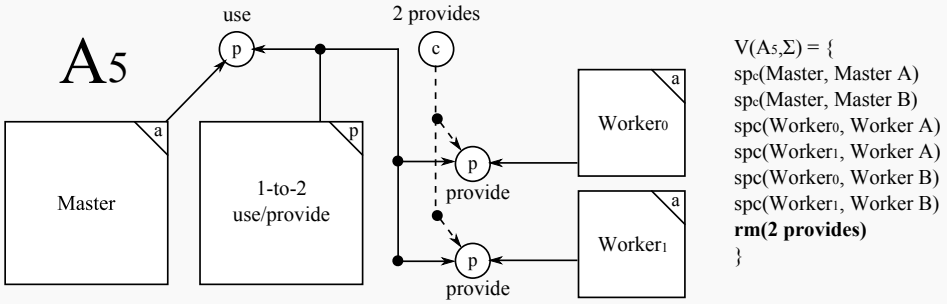
Note that the endpoint specialisation must be performed first so as to respect the signature constraints at all steps. None of the two composite components can be instantiated yet because they are connected to an uninstantiated composite endpoint. The composite endpoint in question is then instantiated resulting in assembly A_3:



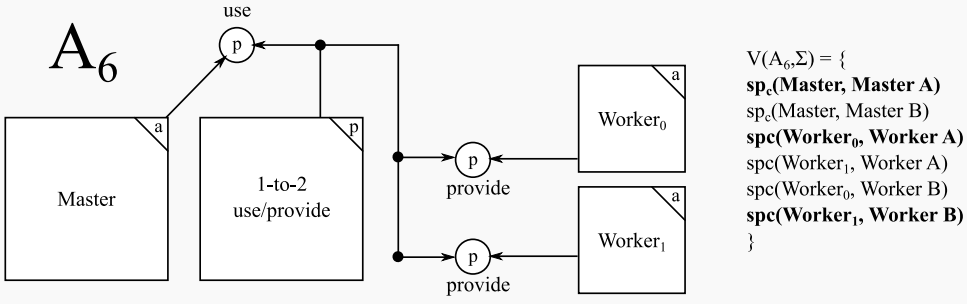
All the conditions are now met to instantiate the worker set which results in assembly A_4 (which resembles closely the example from Figure 6.2):



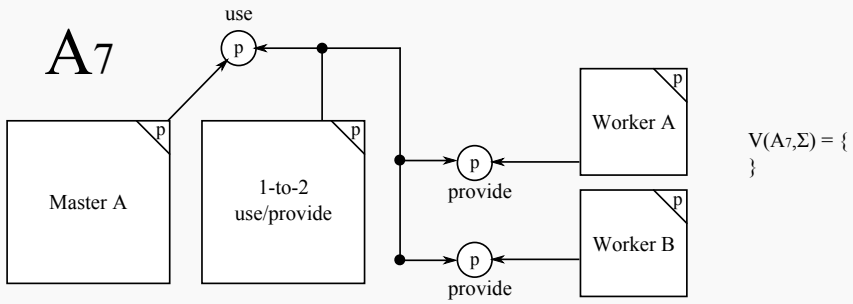
Note that the implementation of the endpoint is used to connect the two workers directly to *Provide* endpoints. The composite connection is then instantiated in the same fashion resulting in assembly A_5:



There is now an unused endpoint which is promptly removed with the appropriate operation resulting in assembly A_6:



Note that the two *Provide* endpoints are now fully independent. A final specialisation step is taken to choose a variant for every master and worker component resulting in assembly A_7 which is primitive:



One can notice how hierarchy is handled: the hierarchy of endpoint implementations serves as a buffer between two composites which will each require several endpoints to instantiate. Instead of requiring a simultaneous specialisation or instantiation of both components, the calculus permits to specialise and instantiate each one in turn.

A second remark is that two meaningful architectural decisions were made: first, the value of n was set (between A_1 and A_2) and second, the variants for all components were chosen (between A_6 and A_7). Also note that, at every step most valid operations correspond either to a mandatory step in resolving a previous meaningful decision (e.g., instantiating a composite) or to another meaningful decision (e.g., specialising to *2 identical workers* at A_4 would have put an additional architectural constraint).

A last point to note is that, in this specific example, there is no way to make an inconsistent decision (because of the signature constraints); a process choosing operations randomly in the valid operations set would always produce a primitive master-worker assembly. While this is not true in the general case, it shows that signature constraints can efficiently forbid invalid cases.

6.1.6 Calculus Variant: Reversible Operations

In the calculus as presented so far, not all operations are reversible. Indeed, $\mathbf{im_c}$ and **rm** remove elements from the assembly with no way to revert unless extra information is available. This limitation can be problematic for decision tree traversal. In order to remove this limitation, this section presents alternative operations which preserve the structure during specialisation, providing efficient reverse operations at the cost of extra unremovable nodes in the assembly.

The idea behind this calculus variant is to maintain a component hierarchy similar to the endpoint hierarchy and to forbid removal of endpoints and nodes. To this end, we need to extend the assembly model to allow component-to-component implementation edges and to modify the operations to maintain the implementation hierarchy.

Assembly Model Modification. To let implementation edges connect components, the definitions of i_s and i_t are modified as follows:

- $i_s : I \to (E \cup C)$;
- $i_t : I \to (E * \cup C*)$.

In addition, for an assembly to be well-formed, an implementation must now have a component as a source iff it has components as targets.

Reversible Operations.

- $\mathbf{sp_e}$, $\mathbf{sp_c}$ and $\mathbf{im_e}$ are unchanged.
- **rm** no longer exists.

- $\mathbf{im_c(c)}$ is not any longer allowed to remove the original component and, in addition, it creates a new implementation i whose source is c and whose targets are all the components created by the builder.

Now that the calculus keeps the hierarchy structure during specialisation, we need to define what additional data must be kept and how to reverse operations.

Additional Data. For operations to be reversible, the following data must be kept during specialisation:

- the list of operations made so far with their parameters;
- for specialisation operations (i.e., $\mathbf{sp_c}$ and $\mathbf{sp_e}$), the type of the endpoint or component *before* the specialisation (e.g.,, if specialised from type A to type B then the operation is $\mathbf{sp}_*(*, B)$ and A must be stored in addition).

Reverse Operations. At any time, the last operation that has been done can be reversed in the following way:

- if it was a $\mathbf{sp_e}$ or $\mathbf{sp_c}$ operation, then change the endpoint or component type back (all the necessary information is stored);
- if it was a $\mathbf{im_e(x)}$ or $\mathbf{im_c(x)}$ operation, then, considering $\exists i \in I, i_s(i) = x$, remove all components/endpoints in $i_t(i)$ and remove i.

These new reversible operations have the following costs: the list of operations must be stored with extra type information ($O(\#operations)$ in extra space), the hierarchy of composite components must be kept during specialisation ($O(\#components \times maxCompositeDepth)$ in terms of extra components and extra connections), and the time cost of reversing is $O(1)$ plus the actual modification of the assembly which is unavoidable.

In addition to providing easily reversible operations, this calculus variant keeps additional structural data regarding which component was instantiated by which composite. This can be useful to model nested composites (see Section 6.2.1 for details).

6.2 Encoding Additional Features

The first step in evaluating the calculus is to estimate how general it is, that is how many of the existing component models it can itself model. Benefits of being general include easy reuse across component models and a wide area of application. For the calculus and the assembly model in particular to be general, commonly found features of component models must be easy to encode. Examples of such features include hierarchy, connectors, and genericity. An example on how to encode connectors was presented in the running master-worker example from Section 6.1.

Finding an encoding for a new concept into the assembly model is easy but finding a good one is not. A good encoding is an encoding which does not increase dramatically the complexity of the operations on the encoded objects. In our particular case, a good encoding of a new concept or feature is one in which a meaningful architectural decision is easy to implement (in terms of number of operations and ratio of valid paths in the decision tree).

6.2.1 Hierarchy

Hierarchy is a commonly found feature in academic component models which allows to implement a component with a component assembly. Examples of models with hierarchy include [30, 32, 12, 23]. Depending on the model, hierarchy can be either just an efficient way to describe subassemblies in an otherwise flat model or hierarchy can be present in the assembly in the form of nested components.

The proposed calculus is capable of modelling both. The basic calculus features composite components with an implementation which can be used to replace the component by its assembly

implementation. In this case the assembly stays flat at all time (since a composite is destroyed when replaced by its implementation).

The calculus variant with reversible operations (see Section 6.1.6) is capable of modelling nested components. Indeed, in this variant of the calculus, composite components are not destroyed when implemented. They are connected to their implementation whose components can in turn be connected to their own implementations and so on, creating a tree structure. This tree structure can be thought of as a topograph (as in bigraphs [75]) and encodes exactly the nesting of composite components.

The operations and type systems have been made specifically with hierarchy in mind and are thus easy to use with it. One typical problem encountered in hierarchical models which is solved by this approach is the step-by-step specialisation of two abstract components connected by an abstract connection (e.g., two generic components connected with a n-to-m connector) which is handled here using endpoints as a buffer.

6.2.2 Genericity

Component model literature has proposed genericity [22] to ease reuse of components by having components parametrized by types or values.

In the case of the calculus, genericity (from a specialisation process's perspective) can be implemented in a type system by providing one type per possible parameter value. Specialisation operations such as setting an integer parameter must be encoded in type systems using an infinite number of types (as illustrated in Figure 6.3) which is impractical both for specifying the type system and enumerating possible operations at a given step in a transformation.

This section defines parametrized type systems which ease those two tasks by providing a model to describe an infinite type system in a finite way. Along with the type system definition, a mapping to a (non-parametric) type system is also provided.

Definitions Let us first define a simple parameter grammar. Parameters can either be endpoint/parameter types or values. A value can be an actual value (e.g., an integer), a value type (e.g., Integer), or it can be undefined.

Definition 35. Parameters are defined by:

$$P_t ::= TYPE$$

$$P_v ::= UNDEF \mid VTYPE \mid VVALUE$$

Where $TYPE$ is a component or endpoint type (i.e., from $T_c \cup T_e$), $UNDEF$ is a constant which denotes that nothing is defined, $VTYPE$ is a value type from a set of value types (e.g., integer, string, float) and $VVALUE$ is a value (e.g., 3, "$hello$", or 5.2).

Let us now define parametric component and endpoint types. The constraint, build, implementation and abstraction functions are just straight ports from non-parametric type systems while the two parameter lists are new.

Definition 36. A *parametric component type* is defined by:

- a list $p_t \in P_t*$ of type parameters;
- a list $p_v \in P_v*$ of value parameters;
- a constraint function $c : p_t \cup p_v \to P*$;
- a build function $b : p_t \cup p_v \to strans$;

- an abstraction function $a : p_t \cup p_v \to \{primitive, composite, abstract\}$.

Definition 37. A *parametric endpoint type* is defined by:
- a list $p_t \in P_t*$ of type parameters;
- a list $p_v \in P_v*$ of value parameters;
- a implementation function $i : p_t \cup p_v \cup P*$;
- an abstraction function $a : p_t \cup p_v \to \{primitive, composite, abstract\}$.

We can now define a parametric type system (PTS). It is an extension of a non-parametric type system (NPTS) where some abstract component and endpoint types are associated to parametric types.

Definition 38. A *parametric type system* (PTS) is defined by:
- a non-parametric type system S;
- the parametric component subset $T_c^{par} \subset T_c^a$;
- the parametric endpoint subset $T_e^{par} \subset T_e^a$;
- a set of parametric component types P_c^{par};
- a set of parametric endpoint types P_e^{par};
- the parametric component function $p_c : T_c^{par} \to P_c^{par}$;
- the parametric endpoint function $p_e : T_e^{par} \to P_e^{par}$.

With this approach, each parametric type models a whole family of non-parametric types. Having such a family be a specialisation of a non-parametric type makes sense but the reverse is not true. For this reason, we define well-formed PTSes to forbid it. We assume in the remaining of the section that PTSes are well-formed.

Definition 39. A PTS is said to be *well-formed* iff
- S is well-formed;
- $\forall t \in T_c^{par}, t$ is a minimum for $<_c$;
- $\forall t \in T_e^{par}, t$ is a minimum for $<_e$.

Being a minimum for the specialisation relation means that there can be no type which is more specialised, thus forbidding the case we wanted to forbid.

We also define a specialisation relation on parametric types. This relation is analogous to the specialisation relation on NPTSes.

Definition 40.
1. The specialisation relation on value parameters is defined by: a $VVALUE$ is a specialisation of the corresponding $VTYPE$ which is a specialisation of $UNDEF$.
2. The specialisation relation on parametric types is defined by: two parametric types A and B verify $A <_{par} B$ iff their parameter lists have identical lengths and A's parameters are specialisations of B's corresponding parameters.

$<_{par}$ is, fairly straightforwardly, also a partial order.

We can now define the NPTS generated by a PTS. Basically, it adds to the type system all the possible variations of the parametric types and updates everything necessary. To save space, we do not give a fully explicit definition but there is nothing complex or unexpected anyway.

Definition 41. The *generated non-parametric type system* of the parametric system

$$\Sigma = (S, T_c^{par}, T_e^{par}, P_c^{par}, P_e^{par}, p_c, p_e)$$

with

$$S = (T_c, T_c^p, T_c^c, s, B, T_e, T_e^c, T_e^p, i_c, i_e, <_c, <_e)$$

is defined by the following:

- T_c and T_e are respectively augmented with P_c^{par} and P_e^{par} and all types t such that $\exists t' \in P_e^{par} \cup P_c^{par}, t <_{par} t'$;
- $<_c$ is augmented with $<_{par}$, p_c and p_e;
- T_c^p, T_e^p, T_c^c and T_e^c are updated according to the a function of the newly added component and endpoint types;
- s is updated according to the c function of newly added component types;
- B and i_c are updated according to the b function of newly added component types;
- i_e is updated according to the i function of newly added endpoint types.

Discussion and Example Let us have a brief discussion about the genericity encoding described above.

First, Figure 6.4 shows the example of a *n workers of type T* component similar the the one found in Figure 6.3. The constraint, build and abstraction functions are described using ad-hoc pseudocode and a graphical representation of the build function is presented. This example shows that a simple parametric component is fairly easy to describe and that the encoding is usable in practice.

Second, parametric type systems can be used to ease the operation selection process. Indeed, compared to a non-parametric type system, a parametric one provides additional structure which can be used to sort possible operations in a meaningful way. For example, the specialisation of a component of type *n workers of type T* can be done either by setting the value of n or by specialising T. In a non-parametric type system these two sorts of decisions are indistinguishable and correspond to an infinity of types while the structure is apparent in a parametric type system.

6.3 Discussion and Use Case

While the previous section has discussed the generality of the approach with regards to component model features, this section gives a few elements of evaluation through a discussion about the existence and relevancy of specialisation processes (Subsection 6.3.1), and through the presentation of preliminary work that makes use of SpecMOD to specify a transformation from a high-level generic and hierarchical component model to DirectMOD (Subsection 6.3.2).

6.3.1 Specialisation Processes

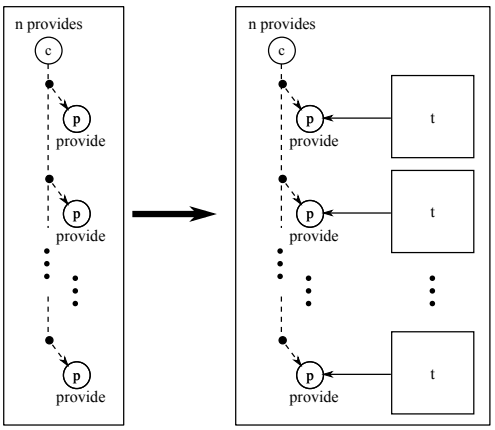
SpecMOD provides a framework to express specialisation processes. In order to evaluate the relevancy and importance of specialisation processes, this section discusses the existence of specialisation processes in the literature, and discusses the usefulness of generic processes.

Specialisation Processes in The Literature Several component models from the literature propose some form of assembly generation or assembly-level assembly optimization. Examples of domains where such approaches exist include scientific computing and cloud computing where performance and scalability are important focuses.

n identical workers

p_t = {Worker}

p_v = {INTEGER}




```
c: function c(Worker_type t, Integer n)
       if n is a value then
           return [endpoint_type(n+" provides")]
       else
           return [endpoint_type("n provides")]

a: function a(Worker_type t, Integer n)
       if n is a value then
           return COMPOSITE
       else
           return ABSTRACT

b: function b(Worker_type t, Integer n)
       if n is a value then
           for i in {1... n} do
               tmp_c = new Component(t)
               tmp_e = sig_constraint.implem[i]
               connect(tmp_c, tmp_e)
       else
           return NO BUILDER
```

Figure 6.4: Example of parametric system for a worker set. Functions are described using pseudocode and the builder function is also represented as a pseudo graph transformation for maximum clarity.

A first, simple form of automatic assembly optimization is *variant selection*. Variant selection consists in choosing, for each component in a provided assembly, one implementation or algorithm (called a variant of that component) so as to optimize a non-functional property such as performance of the assembly for a specific hardware. An example of a component model which propose automatic variant selection is PEPPHER [65, 16]. Variant selection can be modelled in the calculus presented here as a specialisation from an abstract component to a primitive one (the variant).

Other component models have a more general approach and propose assembly-level optimization (which can modify the assembly, as opposed to variant selection which only selects variant for already connected components). Examples of such approaches include BIP optimizations [25, 29]. The possible natures of optimizations in such a context may include changing the communication topology, selecting variants, deciding the size of component collections, merging components or placing the components on resources. Some of these forms of optimization can

be easily modelled by the calculus proposed here as they can be seen as local specialisations (e.g., variant selection, size of collections) while others might require more work as they are not local (e.g., merging components) or not strictly speaking specialisations (e.g., placement on resources). In the case where all optimizations can be modelled with our approach, the algorithm which chooses which optimization to perform can be made into a specialisation process.

The example from the literature which resembles our approach the most is HLCM [23]. HLCM is a connector-based generic hierarchical component model which proposes to generate a low-level flat component assembly from a high-level abstract assembly through an automatic generation process. This approach matches exactly the calculus –which was actually one of the motives behind this calculus– and requires the two extensions proposed here in order to be fully modelled (reversible operations and genericity). The default HLCM implementation implements a default chooser function which lists all the possible operations at a given step and chooses the first one. This default implementation (or alternative user-provided choosers) constitutes a specialisation process in the sense of our approach.

Another area of software engineering where specialisation processes exist is the study of product lines and feature models. This domain is concerned with modelling families of program using features as parameters. Some works in this domain deal with automatic feature selection with respect to a set of constraints and some of those works describe programs with component assemblies. Examples of such works include [87, 104]. Feature selection can be modelled in our approach using parametric type systems (see Section 6.2.2). While some global constraints might be complex to enforce, algorithms for automatic feature selection are good examples of automatic specialisation processes.

Generic Processes A first advantage of our approach is to allow reuse of specialisation processes across applications. While some specialisation processes might be too application-specific to be reused, some other processes might be generic or general enough to be used in a variety of contexts.

One example of such a generic process is the default HLCM implementation [23]. While this is only a default implementation and meant to be replaced by user-defined choosers, it can also be seen as a generic specialisation process, i.e., a specialisation process which can work with any type system and any assembly, although it does not guarantee termination.

As we have seen in the full master-worker specialisation example (see Section 6.1.5) a lot of meaningful architectural constraints can be captured by signature constraints. If signature constraints are strong enough to guarantee that a primitive assembly fits the user's purposes, then the challenge is to find a sequence of operations which results in a primitive assembly. There is a rich literature in exploration of decision trees which could be used directly to make generic specialisation processes which guarantee that they will produce a primitive assembly if possible. Such a generic process could be reused in a large variety of contexts.

Concluding remarks To sum up, there exist a variety of processes from the literature which fit our definition of specialisation process. In addition, generic specialisation processes are an interesting prospect, and relevant literature exists that could serve their implementation.

6.3.2 Use Case: Compiling a High-level Language to DirectL2C

As explained in Section 4.3.2 which presents DirectL2C, describing DirectMOD transformations and assemblies using an *ad hoc* syntax is extremely verbose, and provides no way to parametrise or factorise descriptions. For example, describing a $5 \times 5 \times 5$ 3D structured AMR using the default DirectL2C description language requires more than 6000 lines, as everything

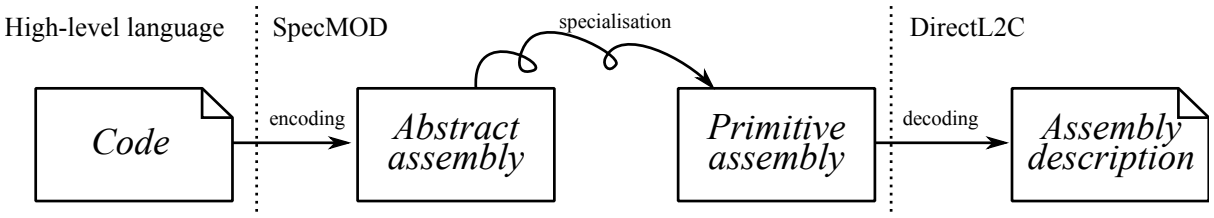


Figure 6.5: Overview of the compilation of the high-level language to DirectL2C using SpecMOD.

has to be described extensively. In practice such descriptions had to be generated using an external Python script, and the script itself was rather lengthy (approximately 500 lines). Moreover, a specialised DirectL2C assembly has to be written for every parameter variation (e.g., size of the 3D grid).

However, in structured cases such as the mentioned 3D grid example, assembly descriptions could be made mush shorter by using generic, parametrised descriptions. Instead of, for example, describing $5 \times 5 \times 5$ grid, it would be much shorter and more reusable to describe a $n \times n \times n$ grid, and specify n when needed.

SpecMOD features hierarchy and genericity (in the form of parametrised types), mechanisms which are useful to describe assemblies in a structured and flexible manner. If they were to be encoded in SpecMOD, DirectMOD concepts could benefit from SpecMOD hierarchy, genericity and specialisation features. Such an approach is a promising solution to address the DirectL2C description problem.

In order both to provide a high-level frontend to DirectMOD and to evaluate SpecMOD on a use case, preliminary work on a SpecMOD-powered high-level DirectMOD-like language has been conducted, with the help of Maverick Chardet during his internship in the team. This section briefly describe the principle of this language, the work accomplished so far, and presents preliminary figures regarding the gains compared to DirectL2C.

Principle We have devised a high-level language to describe component assemblies and transformations. This language is built on the same concepts as DirectMOD but adds genericity and hierarchy. This language is meant to be written directly by the user.

Figure 6.5 presents how SpecMOD can be used to specify the behaviour of this high-level language. First, the high-level code written by the user is encoded into an abstract SpecMOD assembly. This encoding step is a simple translation of the concepts described in the high-level code without any meaningful transformation. For example, a piece of code describing a parametrised component in the high-level language would be translated to an abstract parametrised SpecMOD type. This first step can be thought of as a denotational semantics of the high-level language to a subset of SpecMOD. Second, the SpecMOD encoding undergoes specialisation until it becomes a primitive SpecMOD assembly. Finally, this primitive assembly is decoded into a DirectL2C assembly/transformation description. Once again, this decoding step is nothing but a direct translation; all the meaningful transformations are done during the SpecMOD specialisation phase. The subset of SpecMOD that encodes DirectMOD-style assemblies has to be stable by specialisation for this last step to be possible.

The advantages of this approach is that all the complex transformations are relegated to SpecMOD, and can thus benefit from SpecMOD capabilities regarding assisted specialisation/adaptation.

So far, we have focused on the formal specification of the high-level language and the two necessary encodings.

High-level language A full formal grammar of the language has been devised. The language itself is built on DirectMOD concepts with the following additions:

- composite components;
- parametrisable components and transformations (resembling simplified templates);
- text description of transformations inspired by graph transformation languages such as VIATRA2 [101].

The grammar of the language is not of particular interest in itself, and it would be lengthy to present it in full, but we give a few examples below to illustrate the kind of language we talk about. The code snippet below implements a master-worker composite component composed of a `Master` component and n `Worker` components:

```
composite component MasterWorker <int n> {
        Scheduler sc;
        Master<n> mast;
        Worker wk[n];
        sc.out -> mast.in;

        foreach i in [1,n] {
             mast.out[i] -> wk[i].in;
        }
  }
```

Note the use of template-style syntax to specify the integer parameter. Also note that, to be able to declare the contents of a generic composite, devices such as arrays of components and foreach loops are required, in addition to simple instantiation and connection syntax.

A second example is a transformation which create n new `Worker` components, adds an array of ports to the master, and connect the new workers:

```
transformation add_worker<int n, type T> (Master m1) {
        del m1;
        add Master m2;
        add T worker[n];
        add port m2.output[n];
        foreach (i; 0..n) {
           add m2.output[i] -> worker[i].input;
        }
  } match {
        topo m1 => m2;
        state m1 => m2;
  }
```

The `->` notation denotes a connection between two ports.

Note that instead of fully declaring the origin and destination, the transformation is described in terms of differences between origin and destination (in this case, components and connections are added). While the language also allows explicit declaration of everything, including DirectMOD's topology and state mapping, this approach is more compact and more legible.

Overall, this language is a mix of DirectMOD and SpecMOD, with a bit of syntactic sugar.

Assembly/transformation	DirectL2C	High-level language	Ratio
Master-worker	51	25	2,04
⇨`Insert`	109	46	2.37
AMR 3D $3 \times 3 \times 3$	1167	45	25.93
AMR 3D $5 \times 5 \times 5$	6181	45	137.36
AMR 3D $7 \times 7 \times 7$	17891	45	397.58

Table 6.1: Comparison between DirectL2C assembly/transformation description language and the high-level language, in terms of useful lines of code (i.e., not counting empty lines and very short lines such as loop delimiters).

Encodings In order to implement the transformation for the high-level language to DirectL2C descriptions, two encodings are required. We have devised preliminary versions of both those encodings, although formal work on them is still ongoing. These encodings present two particular challenges.

First, an encoding of DirectMOD-style transformations in SpecMOD is required. One possible approach is to use SpecMOD components as labelled edges to encode relations inside the transformation, in a fashion similar to how connectors were encoded in out first SpecMOD example in Figure 6.2. While this approach technically works, it leads to very complex SpecMOD assemblies. For example, the simple ⇨`Insert` transformation presented in Figure 4.3 in Chapter 4 requires no less than 9 components and 14 endpoints to be encoded in SpecMOD.

Second, it is necessary that the class of SpecMOD assemblies encoding concepts from DirectMOD or the high-level language is stable by specialisation. Indeed, in order to be able to decode a SpecMOD assembly into a DirectL2C description, it is necessary that the SpecMOD assembly actually encodes a DirectMOD assembly. Fortunately, SpecMOD provides a device to prevent unwanted specialisations in the form of the signature constraints. Signature constraints can be used to forbid endpoint configurations that would lead to invalid encodings. However, signature constraints are only local, and it is yet unclear if they are sufficient to provide stable encoding for our purposes. Extra work on this subject is required, and should include a proof of stability.

Figures In order to get a first idea on how useful our high-level language could be compared to DirectL2C *ad hoc* description language, we have worked on a series of examples and compared the sizes of descriptions in the two languages. To do that, we have written a few assembly descriptions using the high-level language and have devised, by hand, the resulting DirectL2C descriptions.

Table 6.1 presents the size of descriptions in both languages for 5 examples: a generic master-worker example similar to the one presented in Figure 6.4, a simple ⇨`Insert` transformation similar to the one presented in Figure 4.3, and a 3D AMR assembly. Without surprise, the high-level descriptions are a lot shorter, which shows how useful such a language could be (without even mentioning the other advantages provided by SpecMOD specialisation phase).

Concluding remarks This ongoing work constitutes an example of a use case for SpecMOD, as well as way to combine two of our contributions into a cohesive whole. While extra work—formal and practical—is required, our preliminary study of the subject is promising.

6.4 Conclusion

We have presented a calculus for step-by-step specialisation of component assemblies. Specifically, we have presented a component assembly model, a type system definition and a set of operations which allow local specialisation. We have also introduced specialisation processes which perform step-by-step specialisation of assemblies until a specific condition is met.

We have seen in Section 6.2 that the calculus proposed in this chapter is capable of encoding common component model features from the literature in a way that is not too complex. This means that our approach is compatible with most existing component models out there.

In Section 6.3.1, we have discussed the existence of specialisation processes in the literature and the usefulness of reusing and composing them. We have determined that although certain limitations were encountered, such as the difficulty to enforce global constraints, most of the examples we considered can be made into specialisation processes.

Finally, in Section 6.3.2, we have presented preliminary work to use SpecMOD to transform a high-level language into a DirectL2C description.

Chapter 7

Conclusion

7.1 Conclusion

This thesis has contributed to the problem of scalable, reconfigurable component models for HPC. While component models are a promising approach to help software engineering in HPC, existing HPC component models do not efficiently support reconfigurable applications, and existing reconfigurable component models are not scalable enough for HPC. The goals of the thesis are to provide models, algorithms and tools to tackle various challenges pertaining to scalable reconfigurable HPC component models.

Context and related works The first part of the thesis, composed of Chapter 2 and Chapter 3, presented the context and state of the art necessary to introduce the problem, and list relevant existing concepts and ideas.

Chapter 2 presented the basics of *High-Performance Computing* (HPC) and *component models*, the two domains at the intersection of which the subject of the thesis is situated. For each of those domains, an introduction to the most common notions was given. In particular, the example of *Adaptive Mesh Refinement* (AMR) was presented. AMR is a computing method used in certain HPC applications which consists in dynamically changing the structure of a data mesh at runtime, in order to have both high precision where it matters and high performance elsewhere. AMR applications are very complex from a developer's perspective, dynamical in nature, and have high performance requirements. For all these reasons, AMR is a motivating and challenging use case for the thesis.

Chapter 3 provides a detailed presentation of the state of the art in application reconfiguration, whether it is component-based or not. A first simple taxonomy of approaches is given, by level of abstraction ranging from *ad hoc* solutions to autonomic loops. Then, a list of reconfiguration issues and sub-problems is presented. This list includes:

- *Reconfiguration specification*, i.e., how a developer can specify application transformations. Among the literature, we have singled out *graph transformation* as an interesting formalism that may be relevant to the thesis' problem.
- *Reconfiguration execution*, i.e., how to execute an reconfiguration at runtime without running into state inconsistencies or deadlocks. This problem is particularly relevant to the HPC use case as it is performance-critical.
- *Application representation at runtime*, i.e., how to maintain of model of a running application in a fashion that ensures consistency and that is scalable. This is relevant to the HPC use case as it has an important impact of scalability.

Existing reconfigurable component models were presented and discussed. As it happens, none of these models is fully suited to our particular use case; problems include: being non-scalable,

imposing a specific control model, and not allowing third-party reuse.

DirectMOD Chapter 4 presents DirectMOD, a component model which addresses the problems of reconfiguration specification and runtime representation. In order to present DirectMOD, a preliminary non-reconfigurable component model is introduced, which is also used in Chapter 5. Then, DirectMOD itself is presented through a series of formal definitions of its syntax and semantics. DirectMOD is built around three core ideas:

- *Transformations* are formal constructs similar to graph transformations which are used to specify assembly reconfiguration. DirectMOD transformation differ from traditional graph transformations in that they provide *state* and *topology* mappings which help fine-tuning transformation side-effects.
- *Transformation adapters* are explicit mapping of transformations to their target in the running assembly. Having these devices mean that there is no need for pattern-matching as is commonly found in graph transformation frameworks, which can be a costly process.
- *Domains* are special components responsible for executing transformations and maintaining assembly representation. Contrary to controllers in hierarchical component models, domains are in no way restricted in the shape of their managed subassembly, allowing arbitrary control of reconfiguration concurrency.

An implementation of DirectMOD, called DirectL2C, was made for the purpose of evaluating DirectMOD. Preliminary experiments on the Grid'5000 platform show that DirectL2C scales up to 128 cores. However, DirectL2C assemblies and transformations are verbose and difficult to write, which is a symptom of the low-level nature of DirectMOD. Also, the locking code inside domain is difficult to write and error-prone.

Locking Chapter 5 addresses the problem of the locking code inside call-stack-based assemblies. Call-stack-based execution model with possibly blocking calls are particularly prone to deadlocks when trying to lock a part of the applications. Locking algorithms in such contexts are usually complex and application-specific, leading to a lot of error-prone work for programmers. While generic tools exist in the literature to help with locking, they rely on information or meta-information which is difficult to provide in a component-based context with minimalist interfaces. In addition, completely stopping components can lead to over-synchronisation, while allowing the locking of individual ports might be more efficient.

We propose a minimalist metadata scheme (meant to be used in conjunction with the preliminary model introduced in Chapter 4), a locking paradigm(i.e., a set of operations which allows to lock individual parts of an assembly) which allows port-wise locking, and a locking algorithm. Provided some hypotheses on the application are met, the algorithm is capable of safely locking subassemblies in call-stack-based component assemblies. While those hypotheses greatly limit the possibilities of application of our proposition, there are useful use cases such as communication subassemblies in stencil applications.

A threefold evaluation of our propositions is performed, using an implementation based on DirectL2C:

- overhead induced by the locking paradigm is measured, and is very low;
- the locking time of our approach is compared to component-wise locking, and is better on all cases, except the ones with very little synchronisation;
- our DirectL2C benchmark is compared to a L2C benchmark implementing the same application and is much shorter and simpler.

SpecMOD Chapter 6 presents SpecMOD, a model for the specialisation of hierarchical component assemblies. SpecMOD is intended both as a generalization of specialisation processes

such as can be found in the literature, and as a mechanism to add genericity and hierarchy to lower-level component models, such as DirectMOD. SpecMOD is strongly inspired by HLCM, a pre-existing component model which used automatic specialisation to transform high-level component assemblies into low-level ones.

SpecMOD takes the form of a formal model, complete with syntax and semantics. Several extensions for it are proposed, to encode useful features such as parametrised types and reversible operations.

The generality of the approach is discussed at length, and preliminary work to transform a high-level language to DirectMOD assemblies using SpecMOD is presented.

Conclusion Overall, this thesis has addressed several sub-problems related to reconfigurable HPC component models, through three contributions. Those contributions range from low-level synchronisation concerns (Chapter 5), to very abstract and formal concerns (Chapter 6). Preliminary work has been towards integrating those contributions into one unified solution.

7.2 Perspectives

While the thesis has dealt with varied sub-problems, a lot of perspectives remain open for HPC reconfigurable component models.

Continue evaluation and implementation Evaluation and implementation of our contributions are not complete at the time of writing. We identify three main ways to improve what has already been done.

First, extensive large-scale experiments should be conducted with our DirectL2C AMR benchmarks. Current scalability figures were obtained on Grid'5000 on 128 cores, which, while a useful result, is still quite far from the parallelism of cutting-edge HPC hardware. Pushing our benchmarks to higher processor counts could reveal new bottlenecks which would need to be addressed. This is a very short-term perspective, as experiments on the Curie supercomputer should be conducted shortly.

Second, SpecMOD (or at least a subset of it) should be implemented to perform a detailed evaluation on a use case. Using HLCM to implement and evaluate SpecMOD is a possibility, as both models have a lot in common, but a separate implementation could be the better approach as SpecMOD is technically more general. In particular, it would be interesting to study the usefulness of SpecMOD as a model to implement automatic static adaptation (and if there is a difference with HLCM). Static adaptation on Fast Fourier Transforms (FFTs) has been studied in the team already [62], and could be a relevant HPC use case.

Third, the work that have been started to use SpecMOD as an engine to compile a high-level component model and transformation language to DirectMOD, should be continued. While implementing such a language would require a working implementation of SpecMOD, formal work remains to be done, such as finishing the specification of the transformation and proving its correction. The end result of such an effort, would be to apply this language on the AMR example, and evaluate precisely just how useful it is.

Extend multi-domain support Medium-term perspectives include building on DirectMOD's domains to better support transformations that cross domain boundaries. Indeed, as was discussed in Chapter 4, multi-domain transformations, while supported by the model are difficult to put in practice (multi-domain locking might cause deadlocks that single-domain would not) and inelegant.

While this formulation with domains is very DirectMOD-specific, it is actually an instance of the more general problem of how to maintain multiple local representations. The approach chosen in DirectMOD was to have strictly disjoint local representations, and to lock all enclosing representations when a transformation occurs. While the disjointness of domains was a useful property for single-domain transformations, it might not be the best approach in the general case.

Possible approaches to this problem include switching to nested domains, or to unrestricted overlapping domains. Either of those approaches poses the problem of the semantics of transformations in a context where an element might appear in several representations. Formalisms dealing with transformations on non-flat assemblies, such as Milner's bigraphs [75], could be useful.

Collective communications Another perspective, would be to study the use of DirectMOD with collective communications, i.e., non-binary component interactions. Collective communications are an important topic in HPC, and are widely used, e.g., in MPI applications.

Collective communications could be modelled in DirectMOD as components, or could be the subject of an extension of the model. While modelling collective communication is probably sufficient to write assemblies, model-level tools (such as our locking algorithm) could benefit from meaningful collective connectors, resembling for example BIP's connectors [12].

High-level runtime model Finally, as a long-term perspective, it would be interesting to have a high-level runtime reconfigurable HPC component model (*runtime* being the important word here). Indeed, while DirectMOD is a runtime model, it does not have high-level features such as hierarchy; and while our preliminary work towards a generic hierarchic language built on SpecMOD might be high-level, it is a purely static approach with no runtime semantics. A high-level runtime reconfigurable model would be a runtime model, with features such as hierarchy, that provides high-level transformations and that is capable of maintaining a high-level representation of the assembly through execution.

In particular, it would be interesting to maintain the duality "high-level easy model" / "low-level efficient model" (such as is advocated, e.g., by HLCM) through execution. This would require a way to keep the consistency between the low-level model and the high-level one, which is not an easy problem when concurrency is involved.

Bibliography

[1] Top500 list, june 2015. `http://www.top500.org/list/2015/06/`. Accessed: 2015-10-15.

[2] Gul Agha and Carl Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In *Foundations of Software Technology and Theoretical Computer Science*, pages 19–41. Springer, 1985.

[3] Reinhart Ahlrichs, Michael Bär, Marco Häser, Hans Horn, and Christoph Kölmel. Electronic structure calculations on workstation computers: The program system turbomole. *Chemical Physics Letters*, 162(3):165–169, 1989.

[4] OSGi Alliance. Osgi service platform core specification v4. 1, 2007.

[5] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[6] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[7] Eshrat Arjomandi, William O'Farrell, and Ivan Kalas. Concurreny support for c++: An overview. Technical report, C++ Report, 1993.

[8] Colin Atkinson. *Component-based product line engineering with UML*. Pearson Education, 2002.

[9] O. Aumage, E. Brunet, N. Furmento, and R. Namyst. New madeleine: a fast communication scheduling engine for high performance networks. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007.

[10] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In IvanI. Ivanov, Marten Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[11] Bruno Barroca, Levi Lúcio, Vasco Amaral, Roberto Félix, and Vasco Sousa. Dsltrans: A turing incomplete transformation language. In *Software Language Engineering*, pages 296–305. Springer, 2011.

[12] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Proceedings of the Fourth IEEE International Conference on*

Software Engineering and Formal Methods, SEFM '06, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[13] Ananda Shankar Basu. *Component-based Modeling of Heterogeneous Real-time Systems in BIP*. PhD thesis, Université Joseph Fourier, 2008.

[14] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. GCM: a grid extension to Fractal for autonomous distributed components. *Annales des Télécommunications*, 64(1-2):5–24, 2009.

[15] Francoise Baude, Denis Caromel, and Matthieu Morel. From distributed objects to hierarchical grid components. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, pages 1226–1242. Springer, 2003.

[16] Siegfried Benkner, Sabri Pllana, Jesper Larsson Träf, Philippas Tsigas, Uwe Dolinsky, Cedric Augonnet, Beverly Bachmayer, Christoph Kessler, David Moloney, and Vitaly Osipov. Peppher: Efficient and productive usage of hybrid computing systems. *IEEE Micro*, 31(5):28–41, 2011.

[17] Boutheina Bennour, Ludovic Henrio, and Marcela Rivera. A reconfiguration framework for distributed components. In *Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution@ runtime*, pages 49–56. ACM, 2009.

[18] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[19] David E Bernholdt, Benjamin A Allan, Robert Armstrong, Felipe Bertrand, Kenneth Chiu, Tamara L Dahlgren, Kostadin Damevski, Wael R Elwasif, Thomas GW Epperly, Madhusudhan Govindaraju, et al. A component architecture for high-performance scientific computing. *International Journal of High Performance Computing Applications*, 20(2):163–202, 2006.

[20] Julien Bigot. *Du support générique d'opérateurs de composition dans les modèles de composants logiciels, application au calcul scientifique*. PhD thesis, INSA de Rennes, December 2010.

[21] Julien Bigot, Zhengxiong Hou, Christian Perez, and Vincent Pichon. A low level component model easing performance portability of hpc applications. *Computing*, pages 1–16, 2013.

[22] Julien Bigot and Christian Pérez. Increasing Reuse in Component Models through Genericity. In *Proceedings of the 11th International Conference on Software Reuse, ICSR '09*, pages 21–30, Falls Church, VA, United States, September 2009. Springer-Verlag.

[23] Julien Bigot and Christian Pérez. High Performance Composition Operators in Component Models. In Ian Foster, Wolfgang Gentzsch, Lucio Grandinetti, Gerhard R. Joubert, editor, *High Performance Computing: From Grids and Clouds to Exascale*, volume 20 of *Advances in Parallel Computing*, pages 182 – 201. IOS Press, 2011.

[24] Juergen Boldt. The Common Object Request Broker: Architecture and Specification. July 1995.

[25] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. From high-level component-based models to distributed implementations. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, page 209–218, New York, NY, USA, 2010. ACM.

[26] Don Box. Essential com. object technology series, 1997.

[27] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM SIGPLAN Notices*, page 211–230. ACM, 2002.

[28] Fabienne Boyer, Olivier Gruber, and Damien Pous. Robust reconfigurations of component assemblies. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 13–22. IEEE Press, 2013.

[29] M. Bozga, M. Jaber, and J. Sifakis. Source-to-Source Architecture Transformation for Performance Optimization in BIP. *Industrial Informatics, IEEE Transactions on*, 6(4):708–718, Nov 2010.

[30] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[31] Jérémy Buisson, Everton Calvacante, Fabien Dagnat, Elena Leroux, and Sébastien Martinez. Coqcots & Pycots: non-stopping components for safe dynamic reconfiguration. In *The 17th International ACM Sigsoft Symposium on Component-Based Software Engineering*, page 1, Lille, France, June 2014.

[32] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, pages 40–48, Aug 2006.

[33] Carsten Burstedde, Omar Ghattas, Michael Gurnis, Tobin Isaac, Georg Stadler, Tim Warburton, and Lucas Wilcox. Extreme-scale amr. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[34] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[35] Sayantan Chakravorty, Celso L Mendes, and Laxmikant V Kalé. Proactive fault tolerance in mpi applications via task migration. In *High Performance Computing-HiPC 2006*, pages 485–496. Springer, 2006.

[36] Manuel Aguilar Cornejo, Hubert Garavel, Radu Mateescu, and Noel De Palma. Specification and verification of a dynamic reconfiguration protocol for agent-based applications. In *DAIS*, volume 198, pages 229–244. Springer, 2001.

[37] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M.R.V. Chaudron. A Classification Framework for Software Component Models. *Software Engineering, IEEE Transactions on*, 37(5):593–615, sept.-oct. 2011.

[38] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, Jan 1998.

[39] Pierre-Charles David, Thomas Ledoux, et al. Safe dynamic reconfigurations of fractal architectures with fscript. In *Proceeding of Fractal CBSE Workshop, ECOOP*, volume 6, 2006.

[40] Pierre-Charles David, Thomas Ledoux, Marc Léger, and Thierry Coupaye. Fpath and fscript: Language support for navigation and reliable reconfiguration of fractal architectures. *annals of telecommunications-annales des télécommunications*, 64(1-2):45–63, 2009.

[41] Bernholdt D.E., Allan B.A., Armstrong R., Bertrand F., Chiu K., Dahlgren T.L., Damevski K., Ewasif W.R., Epperly T.G.W, Govindaraju M., Katz D.S., Kohl J.A., Krishnan M., Kumfert G., Larson J.W., Lefantzi S., Lewis M.J., Malony A.D., McInnes L.C., Nieplocha J., Norris B., Parker S.G., J. Shende Ray, T.L. S. Windus, and S Zhou. A Component Architecture for High Performance Scientific Computing. *International Journal of High Performance Computing Applications*, May 2006.

[42] Maarten de Mol and Arend Rensink. On a graph formalism for ordered edges. *Electronic Communications of the EASST*, 29, 2010.

[43] Noël De Palma, Luc Bellissard, and Michel Riveill. Dynamic reconfiguration of agent-based applications. In *Third European Research Seminar on Advances in Distributed Systems (ERSADS'99)*, 1999.

[44] Travis Desell, Kaoutar El Maghraoui, and Carlos A Varela. Malleable applications for scalable high performance computing. *Cluster Computing*, 10(3):323–337, 2007.

[45] Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino. A survey of parallel programming models and tools in the multi and many-core era. *Parallel and Distributed Systems, IEEE Transactions on*, 23(8):1369–1386, 2012.

[46] Jack Dongarra et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, page 1094342010391989, 2011.

[47] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: distributed shared memory programming*, volume 40. John Wiley & Sons, 2005.

[48] Christian Engelmann, Geoffroy R Vallee, Thomas Naughton, and Stephen L Scott. Proactive fault tolerance using preemptive migration. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 252–257. IEEE, 2009.

[49] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, November 1976.

[50] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0. 09 2012.

[51] Michael R Genesereth and Steven P Ketchpel. Software agents. *Commun. ACM*, 37(7):48–53, 1994.

[52] EE Group et al. Jsr 220: Enterprise javabeanstm, version 3.0 ejb core contracts and requirements version 3.0, final release. *May*, 28:60–66, 2006.

[53] Khronos OpenCL Working Group et al. The opencl specification. *version*, 1(29):8, 2008.

[54] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Bernhard Rumpe, Klaus Müller, and Ina Schaefer. Engineering delta modeling languages. In *Proceedings of the 17th International Software Product Line Conference*, pages 22–31. ACM, 2013.

[55] Ludovic Henrio and Marcela Rivera. Stopping safely hierarchical distributed components: application to gcm. In *CBHPC '08: Proceedings of the 2008 compFrame/HPC-GECO workshop on Component based high performance*, pages 1–11, New York, NY, USA, 2008. ACM.

[56] Berk Hess, Carsten Kutzner, David Van Der Spoel, and Erik Lindahl. Gromacs 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of chemical theory and computation*, 4(3):435–447, 2008.

[57] Michael Hicks, Jonathan T Moore, and Scott Nettles. *Dynamic software updating*, volume 36. ACM, 2001.

[58] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, November 2005.

[59] Richard M Hodur. The naval research laboratory's coupled ocean/atmosphere mesoscale prediction system (coamps). *Monthly Weather Review*, 125(7):1414–1430, 1997.

[60] Tassilo Horn and Jürgen Ebert. The GReTL Transformation Language. In Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations*, volume 6707 of *Lecture Notes in Computer Science*, pages 183–197. Springer Berlin Heidelberg, 2011.

[61] Tobin Isaac, Carsten Burstedde, and Omar Ghattas. Low-cost parallel algorithms for 2: 1 octree balance. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 426–437. IEEE, 2012.

[62] Richard Jérôme, Christian Pérez, and Vincent Lanore. Towards application variability handling with component models: 3d-fft use case study. In *UnConventional High Performance Computing 2015*, page 12. Springer, 2015.

[63] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, page 327–336. ACM, 2010.

[64] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[65] Christoph Kessler and Welf Löwe. Optimized composition of performance-aware parallel components. *Concurrency and Computation: Practice and Experience*, 24(5):481–498, 2012.

[66] Benjamin S Kirk, John W Peterson, Roy H Stogner, and Graham F Carey. libmesh: a c++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22(3-4):237–254, 2006.

[67] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *Software Engineering, IEEE Transactions on*, 16(11):1293–1306, 1990.

[68] A. Langer, J. Lifflander, P. Miller, Kuo-Chuan Pan, L.V. Kale, and P. Ricker. Scalable algorithms for distributed-memory adaptive mesh refinement. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 100–107, Oct 2012.

[69] Vincent Lanore and Christian Pérez. A Reconfigurable Component Model for HPC. In *The 18th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE'2015)*. ACM, 2015.

[70] Kung-Kiu Lau and Zheng Wang. Software Component Models. *Software Engineering, IEEE Transactions on*, 33(10):709–724, 2007.

[71] Jinpil Lee and Mitsuhisa Sato. Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 413–420. IEEE, 2010.

[72] David B Loveman. High performance fortran. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 1(1):25–42, 1993.

[73] Peter MacNeice, Kevin M Olson, Clark Mobarry, Rosalinda de Fainchtein, and Charles Packer. Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer physics communications*, 126(3):330–354, 2000.

[74] M. D. McIlroy. Mass-produced Software Components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.

[75] Robin Milner. Bigraphical reactive systems: basic theory. Technical report, Technical Report 523, Computer Laboratory, University of Cambridge, 2001.

[76] GE Moore. Cramming more components on to integrated circuits. *Electronics, USA*, 1965.

[77] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing.* " O'Reilly Media, Inc.", 1996.

[78] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[79] CUDA Nvidia. Programming guide, 2008.

[80] C Peniguel, I Rupp, H Leroyer, and M Guillaud. Thermal analysis of high level waste geological disposal module with the thermal code syrthes. In *2012 20th International Conference on Nuclear Engineering and the ASME 2012 Power Conference*, pages 283–292. American Society of Mechanical Engineers, 2012.

[81] Nicolas Pessemier, Lionel Seinturier, Thierry Coupaye, and Laurence Duchien. A model for developing component-based and aspect-oriented systems. In *Software Composition*, pages 259–274. Springer, 2006.

[82] Vincent Pichon. *Contribution à la conception à base de composants logiciels d'applications scientifiques parallèles.* PhD thesis, Ecole normale supérieure de lyon - ENS LYON, November 2012.

[83] Ulrike Prange and Hartmut Ehrig. Graph transformation in adhesive hlr categories. In *Pfalzgraf, J.(Hrsg.): Advances in Multiagent Systems, Robotics and Cybernetics: Theory and Practice. Proceedings of Intern. Conf. on Systems Research, Informatics and Cybernetics.* Citeseer, 2005.

[84] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[85] A Ribes, C. Perez, and V. Pichon. On the design of adaptive mesh refinement applications based on software components. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 383–386, Oct 2010.

[86] Marcela Rivera. *Reconfiguration and life-cycle distributed components (asynchrony, coherence and verification)*. Theses, Université de Nice, 2011.

[87] Marko Rosenmüller and Norbert Siegmund. Automating the configuration of multi software product lines. *VaMoS*, 10:123–130, 2010.

[88] Lionel Seinturier, Nicolas Pessemier, Clément Escoffier, and Didier Donsez. Towards a reference model for implementing the fractal specifications for java and the. net platform. In *ECOOP 2006-Workshop Fractal*, 2006.

[89] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *High Performance Computing for Computational Science–VECPAR 2010*, pages 1–25. Springer, 2011.

[90] Alan D Simpson, Mark Bull, and Jon Hill. Identification and categorisation of applications and initial benchmarks suite. *PRACE-PP Public Deliverables, http://www. prace-ri. eu/Public-Deliverables*, 2008.

[91] Alan D Simpson, Mark Bull, and Jon Hill. Identification and categorisation of applications and initial benchmarks suite. *PRACE-PP Public Deliverables, http://www. prace-ri. eu/Public-Deliverables*, 2008.

[92] Borja Sotomayor, Kate Keahey, and Ian Foster. Combining batch execution and leasing using virtual machines. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 87–96. ACM, 2008.

[93] Borja Sotomayor, Rubén S Montero, Ignacio M Llorente, and Ian Foster. Virtual infrastructure management in private and hybrid clouds. *Internet computing, IEEE*, 13(5):14–22, 2009.

[94] Volker Springel, Naoki Yoshida, and Simon DM White. Gadget: a code for collisionless and gasdynamical cosmological simulations. *New Astronomy*, 6(2):79–117, 2001.

[95] Joachim Stadel, James Wadsley, and Derek C Richardson. High performance computational astrophysics with pkdgrav/gasoline. In *High Performance Computing Systems and Applications*, pages 501–523. Springer, 2002.

[96] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. 2002.

[97] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 117–128. ACM, 2011.

[98] Romain Teyssier. Cosmological hydrodynamics with adaptive mesh refinement: a new high resolution code called ramses. *arXiv preprint astro-ph/0111367*, 2001.

[99] François Trahay, Élisabeth Brunet, and Alexandre Denis. An analysis of the impact of multi-threading on communication performance. In *CAC 2009: The 9th Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2009*, Rome, Italy, May 2009. IEEE Computer Society Press.

[100] Sathish S Vadhiyar and Jack J Dongarra. Srs: A framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 13(02):291–312, 2003.

[101] Dániel Varró and András Balogh. The model transformation language of the viatra2 framework. *Science of Computer Programming*, 68(3):214–234, 2007.

[102] Nanbor Wang, Kirthika Parameswaran, Michael Kircher, and Douglas C. Schmidt. Applying Reflective Middleware Techniques to Optimize a QoS-Enabled CORBA Component Model Implementation. In *COMPSAC*, pages 492–499. IEEE Computer Society, 2000.

[103] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M Göschka. On patterns for decentralized control in self-adaptive systems. In *Software Engineering for Self-Adaptive Systems II*, pages 76–107. Springer, 2013.

[104] Jules White, Brian Dougherty, Doulas C Schmidt, and David Benavides. Automated reasoning for multi-step feature model configuration problems. In *Proceedings of the 13th International Software Product Line Conference*, pages 11–20. Carnegie Mellon University, 2009.

[105] Fu Yan. Gscript: a script language that supports both com and corba. In *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, volume 1, pages 558–562 vol.1, May 2000.

[106] Gengbin Zheng, Esteban Meneses, Abhinav Bhatele, and Laxmikant V Kale. Hierarchical load balancing for charm++ applications on large supercomputers. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 436–444. IEEE, 2010.