An Architectural Meta-Application Model for Coarse Grained Metacomputing

Stephan Kindermann[1] and Torsten Fink[2]

[1] University of Erlangen-Nuremberg, Germany
snkinder@informatik.uni-erlangen.de
[2] Free University of Berlin, Germany
tnfink@computer.org

Abstract The emerging infrastructures supporting transparent use of heterogeneous distributed resources enable the design of a new class of applications. These meta-applications are composed of distributed software components. In this paper we describe a new model for component based meta-application design based on a formal architectural description of the gross organization of an application. This structural description is enriched by a formal process algebraic characterization of component behavior. Using this behavioral model we can formally check meta-applications in an early development phase. We present simple architectural styles developed to support data-flow and control-flow driven meta-application design on top of the Amica metacomputing infrastructure.

1 Introduction

There is a growing interest in defining and constructing an infrastructure which gives users the illusion that distributed, heterogeneous (computing and storage) resources constitute one giant transparent environment, a metacomputer [10]. It enables the development of a new class of applications: meta-applications. They are composed of multiple (partly reusable) components. Looking at the current practice of meta-application development there is no agreement on a common programming model for component based application design. Such a model should provide an abstract and intuitive description of components with their interfaces and should define a basic set of composition mechanisms based on a concise formal semantics.

In this paper we describe an abstract and extendable programming model defined to develop meta-applications on top of our metacomputing environment Amica[1] [4]. We describe the overall coarse organization of meta-applications in a formal architecture description language (ADL) (see e.g. [8]). Different architectural styles [17, 2] are defined to build up a basic description vocabulary. This can be refined to define domain specific extensions. For every element of the vocabulary its behavior is defined by a process algebra term. The overall behavioral model of an application is automatically derived through the appropriate

[1] Abstract Metacomputing Infrastructure for Coarse Grained Applications

composition of the behavioral description of its components. This model can then be checked against a set of formal requirements (e.g. liveness and progress properties). This allows formal correctness checks on Amica meta-applications in an early development phase.

The remainder of this article is as follows. In Sec. 2 we introduce shortly the basic concepts and services of Amica. In Sec. 3 we introduce our new programming model. In Sec. 4 we describe firstly how applications given in our programming model are executed using Amica and secondly how a behavioral model of the application is generated automatically which can be used as input to formal analysis tools. In Sec. 5 we give an overview of related work and discuss the advantages of our approach. Finally, an outlook to future work is given.

2 The Amica Metacomputing Infrastructure

Amica has been designed as a prototypical middleware foundation to investigate the composition of metacomputing applications from reusable components (e.g. legacy systems). Amica provides abstraction of heterogeneous distributed data storage by *data objects* and of computing facilities by *metabricks*. Data objects (possibly replicated) are related dynamically to real storage resources (data store objects) interconnected by link objects which provide abstraction of the networking infrastructure of the metacomputer. Metabricks are related to the basic computational services (bricks) based on a broker mechanism which looks for appropriate brick factories. The concise meaning of 'appropriate' is given by a cost function which takes into account the current load of computing resources provided by special objects named computation units. Essentially, Amica provides an infrastructure to carry out the instantiation of abstract data storage and computing service requests transparently taking into account the current load within the distributed system. The implementation relies on the standard middleware foundation CORBA and uses in the current version an interpreter based instantiation approach. For a more detailed description of the Amica infrastructure see [4].

3 The Amica Programming Model

A metacomputing application on top of Amica is a component based piece of software. Basic data storage and computation components are directly related to the abstract data objects and metabricks of Amica. Other components can include additional application specific code (named *user bricks* in Amica). Additionally to these basic building blocks a set of connectors is necessary to wire the components according to control flow and data flow.

In the following we apply a formal architecture description language to abstractly describe the organization of meta-applications as a hierarchical collection of interacting components with well defined properties and interfaces. This structural description is combined with a formal characterization of component behavior based on a process algebra. On this combination we build up a basic vocabulary for meta-application description which can later be flexibly extended to deal with specific problem domains. The use of process algebras as semantic foundation with their compositionality property allows us to derive flexibly the overall abstract meta-application behavior from the behaviors of its components. Thus, error situations resulting from incorrect component compositions can be detected in an early phase of application development.

3.1 Architecture Description

Despite the variety of existing software architecture description languages (see e.g. [1]) there is a considerable agreement about the role of structure in architecture description. Our description of metacomputing applications is based on ACME [7] which emerged from a joint effort of the architecture research community to provide a common intermediate representation for different ADLs.

Structural Description
Meta-applications are given as collections of components interconnected by connectors in a structural meta-application graph:

Definition 1. A structural meta-application graph (SMG) is given as a bipartite graph, which is characterized by a triple $G = (Nodes, I, E)$.

- The $Nodes$ of the graph consist of a set of components $Comp$ and a disjoint set of connectors $Conn$: $Nodes = Comp \cup Conn, Comp \cap Conn = \{\}$.
- The function I associates each node with its set of interconnection points ($\in IP$). Interconnection points are subdivided into ports (for components) and roles (for connectors). ($IP = Ports \cup Roles$).

$$I : Nodes \rightarrow 2^{IP}; c \in Comp \Rightarrow I(c) \subseteq 2^{Ports}; c \in Conn \Rightarrow I(c) \subseteq 2^{Roles}$$

- The interconnection of components and connectors via their ports and roles is given by the relation $E \subseteq (Comp \times Ports) \times (Conn \times Roles)$.

Behavioral Description
To be able to statically analyze behavioral aspects of Amica based meta-applications we enriched the ACME based architecture description by a formal characterization of component and connector behavior using labeled transition systems. Because of its support for compositional modeling we use the process algebraic description language Lotos [13].

For a node c in a meta-application graph with an interconnection point $p1$ and an associated set of actions $\{a1, .., an\}$ (e.g. services needed or provided or events emitted at the interconnection point) the associated process algebra term $Beh(c)$ includes the parameter list $[p1\_a1, .., p1\_an]$. This list is extended accordingly if multiple interconnection points are defined for a node. The actions in this list correspond to the actions which are externally observable and through which the (node-)process can interact (synchronize, communicate via rendezvous or message passing) with other processes:

Definition 2. A behavioral meta-application graph BMG is given as a quintuple $(Nodes, I, E, Beh)$ where

- $(Nodes, I, E)$ defines a structural meta-application graph and
- the function $Beh : Nodes \to LotosTerm$ associates each node to its behavioral description in form of a Lotos process algebra term.

In the sequel the behavioral description of nodes is given graphically in the more intuitive form of labeled transition graphs where possible and the Lotos description is omitted. In Fig. 1 the behavior of basic control flow components and connectors is illustrated.

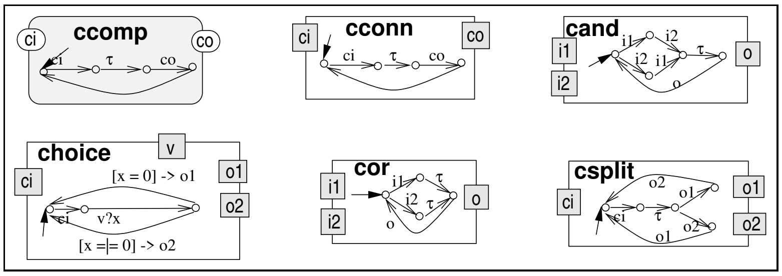


Figure1. Behavioral specification of control flow components and connectors

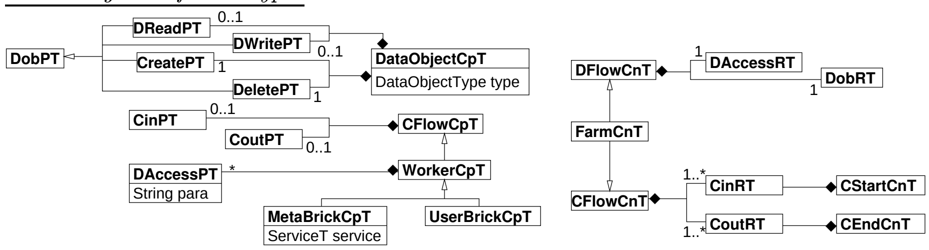
3.2 An Architectural Style for Amica Meta-Applications

Each node and each interconnection point in a (behavioral) meta-application graph is an instance of a type from a set of predefined type definitions. These types are used to build up a basic vocabulary *Voc* to describe the architecture of a meta-application. This vocabulary along with a set of constraints is often called an architecture style [17, 2].

Definition 3. A meta-application vocabulary Voc to build up (behavioral) meta-application graphs is given as a quintuple (NT, IPT, SC, BC) where NT is a set of node (component or connector) types, IPT is a set of interconnection point (port or role) types SC is a set of structural constraints and BC is a set of behavioral constraints.

The description of metacomputing applications on top of Amica is currently based on a simple architectural vocabulary which is illustrated in Fig. 2. It contains component and connector types to characterize the flow of control and the flow of data, to describe farm parallelism, and to provide access to data storage and computing resources (called metabricks in Amica). The structural constraints are given by the associations in the class diagramm and by constraints on the relation E. Semantic constraints can be given in an action based temporal logic using XTL [15] macros.

Class diagram of node types



Constraints on E

$$\begin{aligned}E \subseteq\ & (\mathrm{CFlowCpT} \times \mathrm{CinPT}) \times (\mathrm{CFlowCnT} \times \mathrm{CoutRT}) \cup \\ & (\mathrm{CFlowCpT} \times \mathrm{CoutPT}) \times (\mathrm{CFlowCnT} \times \mathrm{CinRT}) \cup \\ & (\mathrm{WorkerCpT} \times \mathrm{DAccessPT}) \times (\mathrm{DFlowCnT} \times \mathrm{DAccessRT}) \cup \\ & (\mathrm{DataObjectCpT} \times \mathrm{DobPT}) \times (\mathrm{DFlowCnT} \times \mathrm{DobRT})\end{aligned}$$

Figure2. Basic meta-application vocabulary

Data Flow

Data is stored in instances of the type `DataObjectCpT`. Its definition includes two interface types `DReadPT` and `DWritePT` for read-write access. Also two interface types to create and delete object instances are provided (`CreatePT` and `DeletePT` in Fig. 3). The access to data object components is done via connectors of the type `DFlowCnT`. It provides roles for connection to data objects and to objects needing data access (`DobRT`,`DAccessRT`).

In Fig. 3 a graphical representation of the basic data flow components and connectors is given along with their abstract formal behavioral characterization. Data flow on connectors is based on a simple request–confirm protocol.

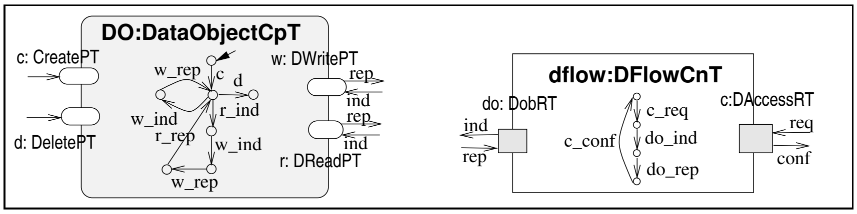
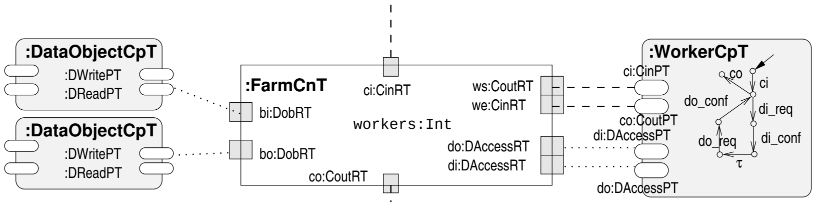


Figure3. Data object components and data flow connectors

Control Flow

Each component which is wired in control flow is an instance of type `ControlCpT`. This type defines two interconnection points of type `CinPT`,`CoutPT`, characterizing incoming and outgoing control flow. Control flow components are connected to control flow connectors (type `CFlowCnT` with roles of type `CinRT` and `CoutRT`). Different subtypes characterize connectors which split and combine control flow (see Fig. 1 for examples).

Combined Data and Control Flow
Instances of the special connector type `FarmCnT` are used to express and control "bag of task" like parallel computations. Farm connectors are used only in a well defined cooperation with data object and worker components, see Fig. 4. After started the farm reads in a bag containing the tasks to be distributed (using `bi:DobRT`) and starts a number of workers over `ws:CoutRT` (this number is given as a property value of the connector). Then, the tasks are distributed to the workers using role `do:DAccessRT`. Thereafter the results are collected over `di:DAccessRT` and stored in a result bag data object. This description of the behavior corresponds directly to the Lotos description given as skeleton in Fig. 4.



Beh(:FarmCnT) =

```
process farm[ci,co,ws,we,bi_ind,bi_rep,bo_ind,bo_rep,di_req,di_conf,do_req,do_conf]:exit :=
   ci; bi_ind;bi_rep?n:Nat;   (  WorkerStart[ws](workers) >>
          ( DistributeTasks[do_req,do_conf](n) ||| CollectResults[di_req,di_conf](n) ) >>
           WorkerEnd[we](workers) >> (bo_ind!n;bo_rep;co;exit) )
        |[ws,we,do_req,do_conf,di_req,di_conf]|
            WorkerGroup[ws,we,do_req,do_conf,di_req,di_conf]
 endproc
         where
process WorkerGroup[ci,co,di_req,di_conf,do_req,do_conf]:exit :=
 (worker[ci,co,di_req,di_conf,do_req,do_conf] |||...||| worker[ci,co,di_req,di_conf,do_req,do_conf] )
endproc
```

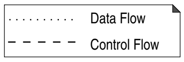


Figure4. Bag of task like parallelism based on farm connector and worker component

3.3 A Small Example

In Fig. 5 a simple examplary composition based on our vocabulary defining a ray tracing meta-application is given. Two data objects are involved, one to store the three dimensional scenario (`scen`) and one to store the generated picture (`pic`).

In `init` these data objects are created and initialised. Then, the remote computation is started by a metabrick. In parallel the user can monitor the current state of the picture by a specialized user brick. Using the graphical

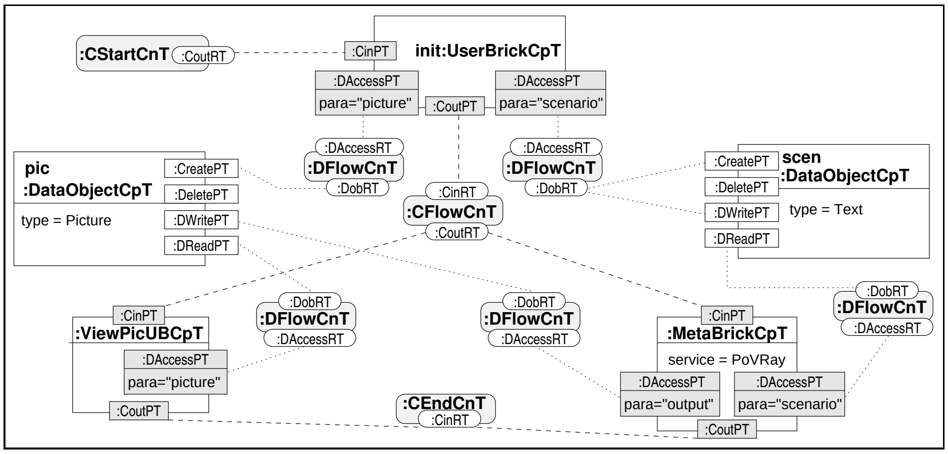


Figure5. An examplary application

front-end for ACME this application can be intuitively defined by drag-and-drop nodes out of the meta-application vocabulary. Another simple application to parallel simulation of mobile communication systems is described in [4].

4 Meta-Application Execution and Formal Analysis

As illustrated in Fig. 6 a meta-application graph is composed using a vocabulary of connector and component types. This composition is then dynamically interpreted and mapped to the Amica metacomputer. Also automatically a behavioral model of the meta-application can be generated, which can be analyzed and checked for design errors (e.g. resulting from component composition mismatches). These two steps, namely interpretation and behavioral model generation are discussed in the following section in more detail.

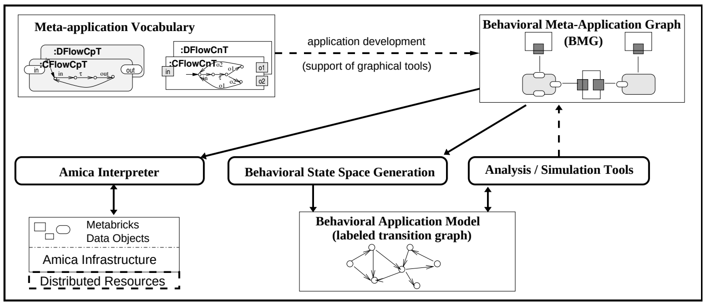


Figure6. Execution and analysis of meta-application graphs

Execution
There is an intuitive mapping of our basic component and connector vocabulary to the services provided by Amica: Components of type `DataObjectCpT` and `MetaBrickCpT` are directly correlated to the data object and metabrick abstractions of basic data storage and computing services provided by Amica. The data store network and the broker mechanism of Amica relate these dynamically to the distributed data objects and bricks (created by brick factories). Data flow connectors are related to link objects and the control flow connectors are interpreted to control component instantiation. In the special case of a farm connector (type `FarmCnT`) a specified number of workers is instantiated, the tasks contained in a bag are distributed and the results are collected.

Analysis
The structure information given in the meta-application graph is used to compose the individual node behaviors to the combined system behavior. Composition is based on appropriate synchronization of component and connector actions. Therefore we first have to define an action renaming operator $\backslash\{(old_1\backslash new_1),..,(old_n\backslash new_n)\}$ which replaces actions over interconnection point names old_i in the general form old_i_post to new_i_post.

Given a behavioral meta-application graph $G = (Nodes, I, E, Beh)$ with $Nodes = Comp \cup Conn = \{CP_1,..,CP_N\} \cup \{CN_1,..CN_M\}$ the associated overall system behavior is given by the parallel (full interleaving) composition of all component instances and also all connector instances. These two groups synchronize over all actions of their interconnection points. Thus the general structure of the overall system behavior description is given as the following process term scheme:

$$(NCP_1 \mid\mid\mid ... \mid\mid\mid NCP_N) \;\mid\mid\; (NCN_1 \mid\mid\mid ... \mid\mid\mid NCN_M)$$

where

$$\begin{array}{l} NCP_i = Beh(CP_i)\backslash\{(old\backslash new) \mid old \in I(CP_i) \wedge \; new = CN_j\_newp \\ \wedge \; \exists \; CN_j \in Conn \mid ((CP_i, old), (CN_j, newp)) \in E\} \end{array}$$

and

$$NCN_i = Beh(CN_i)\backslash\{(old\backslash new) \mid old \in I(CN_i) \wedge new = CN_i\_old\}$$

This simple scheme requires a one–to–one attachment of ports and roles. Attachments of multiple roles to a port are disallowed. In the case of multiple attachments of ports to a role, the above scheme implies an OR semantics (the role-action synchronizes with an action associated with one of the attached ports). We have extended the above scheme to generally allow n-port to one role attachments with different semantics (e.g. and) given by the type of the role.

This scheme generation was implemented in Java and we apply powerful formal analysis tools (the CADP/Aldebaran tool set [6] and the model checker XTL [15]) for abstract meta-application property checks.

5 Related Work

There is no general programming model available for component based programming on top of a metacomputing infrastructure (middleware) available.
Different approaches to build component based meta-applications are used in literature, ranging from simple data-flow models to general component frameworks. In WebFlow [11] and [19] (restricted) data flow graph models are proposed for component composition. Combined data flow and control flow is used in [12] to realize so called control patterns for component composition. In [18] a distributed component architecture toolkit is described for meta-application design on top of the Globus metacomputing infrastructure [5]. Also scripting languages can be used for meta-application description, see e.g. [16].

In contrast to these approaches having no or only a very restricted semantic foundation we use a more flexible and general formal process algebraic model combined with an architectural description of meta-applications using a well defined (and extendable) set of components and connectors.

Other approaches promoting the general idea of combining an architectural description in the more general context of distributed software design are [9] where labeled transition systems are used as a behavioral model for software architecture descriptions in the ADL Darwin [14] and [3] where CSP is used within the ADL Wright.

6 Conclusion

We have described a formal model supporting component based meta-application development. A formal architecture description language was combined with a process algebraic behavioral description to define a basic extendable vocabulary of meta-application components. This vocabulary has been implemented on top of the Amica metacomputing infrastructure. Component compositions can automatically checked based on well known state space analysis methods (e.g. model checking). Next steps include an extension of our vocabulary (including e.g. event propagation and handling) and an application to a more complex problem than the parallel simulation scenario presented in [4]. Also an extension of the formal underpinning is planed. First we want to include the abstract behavior of the Amica infrastructure (now regarded as black box) into the formal analysis (e.g. the broker process for computational resources). Second we want to check not only functional but also performance properties of meta-applications. For this purpose a behavioral description of components using a stochastic process algebra is planed.

References

1. *A Survey of Architecture Description Languages*, Eighth International Workshop on Software Specification and Design, Germany, March 1996.

2. *A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems*, Proceedings COMPSAC, 1997.
3. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, July 1997.
4. T. Fink and S. Kindermann. First steps in metacomputing with Amica. In *Euromicro-PDP 2000*, pages 197–204, 2000.
5. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.
6. H. Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In Bernhard Steffen, editor, *TACAS'98 (Lisbon, Portugal)*, volume 1384 of *LNCS*, pages 68–84, 1998.
7. D. Garlan, R.T. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON '97*, November 1997.
8. D. Garlan and M. Shaw. *Software Architecture: Perspectives on an emerging Discipline*. Prentice Hall, September 1994.
9. D. Giannakopoulou, J. Kramer, and S.C. Cheung. Analysing the behaviour of distributed systems using Tracta. *Journal of Automated Software Engineering*, 6(1):7–35, January 1999. R. Cleaveland and D. Jackson, Eds.
10. A. Grimshaw, A. Ferrari, G. Lindahl, and K. Holcomb. Metasystems. *Communications of the ACM*, 41(11), 1998.
11. Tomasz Haupt, Erol Akarsu, and Geoffrey Fox. Webflow: a framework for web based metacomputing. In *HPCN Europe '99*, April 1999.
12. Robert G. Babb II. Control patterns(tm) - a new code re-use paradigm. White paper, Seki Systems Company, 1999.
13. ISO/IEC. Lotos — a formal description technique based on the temporal ordering of observational behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.
14. Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 3–14, 1996.
15. Radu Mateescu and Hubert Garavel. Xtl: A meta-language and tool for temporal logic model-checking. In Tiziana Margaria, editor, *STTT'98 (Aalborg, Denmark)*, July 1998.
16. Ryan McCormack, John Koontz, and Judith Devaney. Seamless computing with WebSubmit. *Concurrency: Practice and Experience*, in press, 2000.
17. Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan. Architectural styles, design patterns, and objects. *IEEE Software*, pages 43–52, January 1997.
18. J. Villacis, M. Govindaraju, D. Stern, A. Withaker, F. Berg, P. Deuskar, T. Benjamin, D. Gannon, and R. Bramley. Cat: A high performance, distributed component architecture toolkit for the grid. In *Proceedings of the High Performance Distributed Computing Conference*, 1999.
19. D. Webb, A. Wendelborn, and K. Maciunas. Process Networks as a High-Level Notation for Metacomputing. In *IPPS'99, workshop on Java for Distributed Computing, Puerto Rico*, April 1999.