Runtime Variability Management for Energy-Efficient Software by Contract Negotiation

Sebastian Götz, Claas Wilke, Sebastian Cech, and Uwe Aßmann

Technische Universität Dresden
Institut für Software- und Multimediatechnik
D-01062, Dresden, Germany
sebastian.goetz@acm.org,
{claas.wilke, sebastian.cech, uwe.assmann}@tu-dresden.de

Abstract. Improving the energy efficiency of software systems requires runtime adjustments and explicit knowledge about the system's variability. Component-based software has inherent variability in terms of multiple implementations for components. These implementations utilize hardware resources, which are direct energy consumers, leading to a further dimension of variability: the mapping of implementations to resources. The performance modes of hardware resources span a third dimension of variability. Hence, to realize energy-efficient software systems the central question is: which implementations should run on and utilize which resources in which performance mode to serve the user's demands? This question can only be answered at runtime, as it relies on the runtime state of the system. In this paper, we show how combined hard- and software models can be utilized at runtime to determine valid system configurations and to identify the optimal one.

1 Introduction

Component-Based Software Development (CBSD) [15] has become a major development approach for software systems. Although both functional and non-functional properties of component-based software systems have been considered, only few approaches focus on energy consumption as a non-functional requirement. Within the research projects CoolSoftware[1] and QualiTune[2] we are developing a *model-driven* CBSD approach for software systems that can be optimized w.r.t. their provided quality and energy consumption at runtime. We call this approach *Energy Auto Tuning (EAT)* [8]. Since the energy consumption of software components depends on the utilized hardware, our component model requires a modeling of both: software components having multiple implementations and hardware components (for simplicity called *resources*). Notably,

[1] http://www.cool-software.org/
[2] http://www.qualitune.org/

components do not only impose energy consumption by being executed on hardware. For example, energy consumption due to sending data using a network device strongly depends on the kind of network device utilized. While resources can vary in their energy consumption and provided qualities (e.g., CPU speed or memory size), software components can vary in their required hardware resources (e.g., different sort algorithms require different amounts of memory) and their required and provided qualities (e.g., a software component can require another component's service of a certain quality and/or can provide services in different qualities to other components).

As models comprising only few components, can already describe different component implementations and deployment locations (e.g., servers), they span a solution space that describes all possible configurations of modeled software systems on modeled hardware landscapes. The problem we address in this paper is, how to select the most appropriate configuration from this solution space specified by models w.r.t. minimum energy consumption at runtime. Thus, we first have to compute all configurations that are valid (i.e., which fulfill requirements of involved components such as dependencies, required resources and provided qualities). Second, we have to select the variant requiring the minimum energy consumption whilst still serving the user's non-functional requirements. We use contracts to model quality dependencies between components and call the determination of the optimal configuration *contract negotiation*, following the definition of Quality-of-Service-level contracts by Beugnard et al. [2] along with Meyer's *design by contract* principle [10].

The process of contract negotiation has to take place at runtime, which imposes the need to utilize our aforementioned models at runtime, too. E.g., the available hardware infrastructure is represented by a model describing each existing resource along with its properties. This model changes over time, e.g., due to failing or added resources. The same holds for our software models, which need to be kept up to date, e.g., due to added components. The runtime data represented by these models is collected by resource or energy managers. Both processes, hence, work on runtime models.

The remainder of this paper is structured as follows: We introduce the Cool Component Model (CCM) which is the basis for our CBSD architecture in Sect. 2 using a simple video server example. Furthermore, we present the Energy Contract Language (ECL), that describes provided and required qualities of software and hardware components. We describe our *contract negotiation* approach in Sect. 3. Afterwards, in Sect. 4 we present and demarcate from related work. Finally, in Sect. 5 we discuss future work and conclude this paper.

2 Background / Context

To capture EAT systems, we developed the energy-aware component model CCM and the contract language ECL. The CCM provides concepts to model hierarchical system architectures and covers both software components and hardware resources, because software consumes energy only in an indirect manner (i.e.,

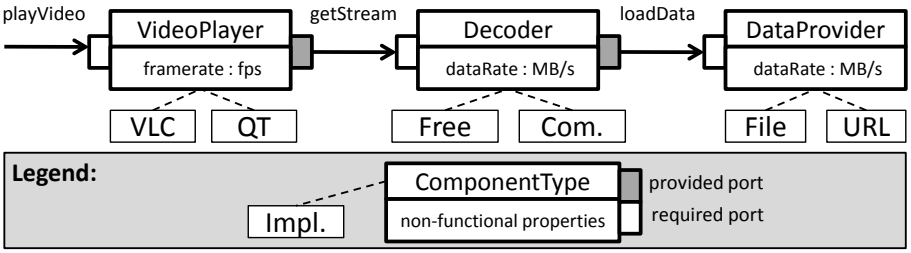


Fig. 1. VideoPlayer SW-Component Types and Implementations.

the energy is consumed by physical resources which are utilized by the software). ECL provides concepts to express dependencies between CCM components based on non-functional properties. This implies dependencies between software components as well as software and hardware components. In this section we introduce CCM and ECL, by means of a video application scenario. In Sect. 3 we use the scenario to explain our contract negotiation approach.

2.1 Capturing Software Components and Resources

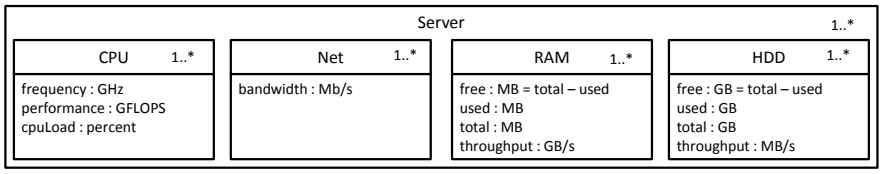
The CCM distinguishes between modeling of the system structure of hardware resources, software components and variants of both. In our project's scope, variants are concrete hardware resources as well as software component implementations. The system structure defines how a system may look like and, thus, represents type declarations for specific variants. For instance, consider the upper part of Figure 1 that shows the types of a video application. It consists of several software component types, namely a `VideoPlayer`, a `Decoder` and a `DataProvider`[3]. Each type may have one or more port types representing an interface of the component. Port types can be used to connect different components. A set of connected components describes the software part of a system.

Concrete implementations (i.e., variants) of a software component (shown in the lower part of Figure 1) have to correspond to the component's type. In our example there are two variants of the type `Player`, the *VLC* and Quicktime (*QT*) implementation. For the `Decoder` type a free (*Free*) and a commercial (*Com.*) implementation are available. Finally, the `DataProvider` is implemented as a local file reader (*File*) and a remote URL reader (*URL*).

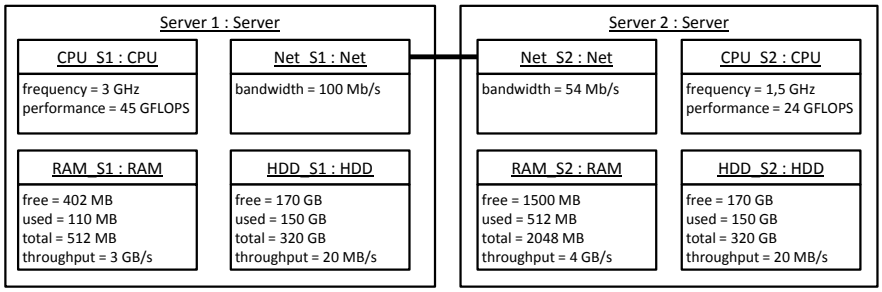
To capture types available in the hardware landscape, resource types have to be specified. Figure 2(a) defines resource types of a hardware landscape on which our video application shall be executed. The `Infrastructure` consists of one or more `Server`s, whereas each server contains one or more `CPU`s, network interfaces (`Net`), `RAM` chips and hard disks (`HDD`). For reasons of simplicity, we omit port types of resource types in the given example.

For each component type (software and hardware) non-functional properties can be defined. For instance, the software component type `Player` defines a

[3] We denote component types using typewriter and variants of them using italic font.



(a) CCM Structure Model for Hardware Landscapes.



(b) CCM Variant Model of a Hardware Landscape Comprised of 2 Servers.

Fig. 2. CCM Hardware Structure and Variant Model.

property `framerate` in frames per second (fps) whereas the resource type `HDD` defines a property `used` (disk space) in GB. Such properties play an important role for specifying ECL contracts and are the basis for contract negotiation.

Figure 2(b) shows a concrete hardware landscape of the resource type system mentioned above. It consists of two servers with specific resources according to the definitions at the type level. The servers are connected by their network devices as depicted by the solid line between *Net_S1* and *Net_S2*. Consider that properties defined at type level are available at variant level with concrete values. Furthermore, each hardware resource variant has to provide a behavior model that defines its energy consumption w.r.t. its utilization. These behavior models have to be provided as templates for each resource type and are instantiated for each concrete resource using the values determined by our resource managers at installation time (i.e., the first time the resource is registered at the runtime environment). We derive the implied energy consumption for a given system configuration (i.e., the distribution of SW components in the infrastructure) and user request by simulating these models. Since the energy consumption computation is not part of contract negotiation, these details are omitted here. The general idea is described in [7].

Notably, variant models are not defined by the developer, but generated at runtime by our Three Layer Energy Auto Tuning Runtime Environment (THEATRE) in accordance to the structural models for HW and SW. THEATRE consists of three layers: the user-, software- and resource layer. Each layer is controlled by a global manager; the Global User Manager (GUM), Global Energy Manager (GEM) and Global Resource Manager (GRM). The GEM has the central role of retrieving information from the GUM and GRM to initiate the

process of contract negotiation and, based on the result, to perform a system reconfiguration. The GUM knows about the details of user requests and associated non-functional requirements of the respective users. Finally, the GRM knows the details about the currently available hardware by monitoring it. These managers generate the respective variant models and keep them up-to-date. Distributed servers are handled by local managers for each layer. E.g., in a 2-server scenario, the first server takes the lead by hosting the global managers, whereas the second server hosts local managers only. Each server, which is part of the system, runs a Local Resource Manager, which registers the server and all its resources at the GRM and sends notifications whenever a property (i.e., the frequency of a CPU) changes. It is important to note that we include subsymbolic information into our variant models (i.e., concrete numbers) and postpone symbolization until contract negotiation. The information required to derive symbols from the values of non-functional properties is encapsulated in our contracts, which are described in the following subsection.

2.2 Specification of ECL Contracts

ECL is used to define dependencies between CCM components using contracts, which are specified for each variant. Therefore, an ECL contract represents a specific view of a variant regarding its dependencies to other types. A contract may define one or more `mode`s, whereas each `mode` defines dependencies to other components. Software components can depend on other software components as well as hardware resources, whereas hardware resources can depend on other hardware resources only. Each dependency relates to a component type and defines bounds for required values of properties at runtime. In addition to constraints expressing required properties, provided properties are specified as well.

Listing 1 shows a contract for the *VLC* video player as a concrete implementation of the `VideoPlayer` component. It defines that the player can be used in two modes: `high-` and `lowQuality`. For `highQuality` the contract specifies that a `CPU` and a `Net` device are required. The `CPU` needs to be utilized at most to 50% and needs to have a frequency of 2 GHz at least.[4] The `Net` device has to offer at least a 10 MBit/s bandwidth. Furthermore, a `Decoder` component is required. Any implementation of that software component type, which is able to provide a data rate of at least 50 KB/s can be used. Finally, the contract defines that in the `highQuality` mode a minimum framerate of 25 fps and a resolution of 1024x768 pixels is provided. To determine the hardware requirements, micro-benchmarks written by the component developer, which address the non-functional properties of interest, are used. As each software component variant is defined by one contract, there exist six contracts specifying SW/HW dependencies in total. The remaining contracts not shown in Listing 1 are similarly structured.

[4] To ease comprehensibility, we use frequency instead of a performance property measured in instructions per second.

```
1  contract VLC implements VideoPlayer {
2    mode highQuality {
3      //required resources
4      requires resource CPU {
5        max cpuLoad = 50 percent
6        min frequency = 2 GHz
7      }
8      requires resource Net {
9        min bandwidth = 10 MBit/s
10     }
11     //dependencies on other SW components
12     requires component Decoder {
13       min dataRate = 50 KB/s
14     }
15     //what is provided in turn
16     provides min frameRate 25 Frame/s
17     provides min imageWidth 1024 Pixel
18     provides min imageHeight 768 Pixel
19   }
20   mode lowQuality { ... }
21 }
```

Listing 1. Example Contract for VLC Video Player.

```
1  contract StandardRAM implements resource RAM {
2    mode low {
3      provides max free: 0.1*total MB
4    }
5    ...
6  }
```

Listing 2. Example Contract for Memory Resource.

In addition to contracts for software components, we allow to define contracts for resources, too. Such contracts suitably illustrate, that the modes of contracts are symbols, which are derived from the resources' properties. E.g., a contract for the resource `RAM` could define the modes *HIGH*, *MEDIUM* and *LOW* as indicated in Listing 2.

In summary, a system modeled with CCM and ECL is highly variable in terms of multiple implementations of component types, multiple quality modes of each implementation and, according to resource requirements of each quality mode, multiple possible mappings of implementations to hardware resources.

3 Contract Negotiation

There are three different kinds of variability captured by CCM/ECL. First, multiple implementations may exist for each software component type. Second, in

an IT infrastructure with more than one server, variability exists in the decision on how to distribute the software components on this IT infrastructure. Finally, the different quality modes specified in ECL denote the third kind of variability. To utilize this variability at runtime for increasing energy efficiency, an approach to determine the optimal system configuration in this regard is required.

This determination can be seen as a special kind of constraint solving optimization problem (CSOP). The goal is to identify the configuration, which implies the lowest energy consumption whilst still serving the user's request and demands. Thus, resource employment (cost) has to be negotiated against the gained utility, where the connection between cost and utility can be expressed as constraints. Hence, constraint programming in general can be applied to solve the CSOP. Because our particular problem is a tradeoff negotiation, expressible using linear constraints, we can use linear programming, which allows the employment of more efficient solving algorithms. To express mappings of components to resources, we need to constrain the domain of this kind of variables to be of Boolean type, i.e., only the integer values 0 and 1 are permitted. Thus, our problem can be classified as a mixed integer linear program (MILP).

An ILP consists of an objective function, a set of constraints and variables being used in both [12]. The objective function is either maximizing or minimizing. In our special case, the objective is to minimize the energy-rated resource usage. Energy-rated means, that there is a factor, which translates between resource usage and energy consumption. This normalizes the different resource usage domains, which else would not be comparable (e.g., size of RAM versus frequency of CPU). A naive approach to determine these factors is to use the standard energy consumption rate, which usually can be found in the resource specification. A more sophisticated approach takes energy-saving and performance modes of resources into account. In this case, factors can be computed using profiling approaches, like Süttner presented in [14].

Our ILP comprises four kinds of variables. Variables expressing resource usage are the first kind, e.g., $usage\#Server_1\#RAM_{S1}\#size$. This variable denotes the size used of the main memory on Server 1. The second kind of variable expresses the mapping selection. E.g., $b\#FreeDecoder\#fast\#Server_2$ denotes whether or not the `FreeDecoder` implementation in `fast` mode has to be mapped to Server 2 (the initial `b` is meant to indicate that this variable is of Boolean type). Software-related properties, like $framerate$, form the third kind of variable. Finally, the server baseload consumption forms the forth kind of variable.

The objective function of our ILP is shown in Equation 1. The goal is to minimize the sum of all resource usage variables and the server baseload consumption. The resource usage variables are normalized by a respective *factor*, which translates resource usage into power consumption and is determined by our resource managers the first time a resource registers at the system.

$$min \sum (factor_{xyz} \times usage\#server_x\#resource_y\#property_z) + \sum baseload\#server_i \tag{1}$$

The constraints of the ILP can be divided into four classes: (i) selection criterias, (ii) resource usage and server baseload, (iii) implied values for non-functional properties and (iv) user demands. The first class corresponds to the information present in the structural model, that is, which (and how many) components are required. In our example, one implementation of each software component is required. The requirement, that exactly one implementation of a component type t has to be chosen, can be expressed by constraints of the following form.

$$\sum_{x,y,z} (b\#impl_{x,t}\#mode_y\#server_z) = 1.0; \tag{2}$$

The second class of constraints describes the boundaries of resource usage variables and how they correlate to the mapping of implementations to resources. For each resource usage variable a constraint for the upper bound (3) and the lower bound (4) is introduced. The values for these boundaries are extracted from the hardware variant model.

$$usage\#Server_1\#RAM_{s1}\#size <= 512.0; \tag{3}$$

$$usage\#Server_1\#RAM_{s1}\#size >= 0.0; \tag{4}$$

The baseload of servers (determined by the local resource managers) is reflected by constraints for each mapping variable as depicted in Equation 5.

$$baseload\#server_i = b\#impl_x\#mode_y\#server_i \tag{5}$$

The impact of mapping an implementation to a resource can be extracted from ECL contracts and can be represented in constraints like exemplary depicted in Equation 6. Here, e.g., the first addend of the sum, states that the `FreeDecoder` implementation requires 512 MB of RAM to operate in `fast` mode. The same statement holds for any other server, but the example equation refers to RAM usage of Server 1 only.

$$\begin{aligned} usage\#Server_1\#RAM_{s1}\#size = \\ 512.0 * b\#FreeDecoder\#fast\#Server_1 + \\ 256.0 * b\#FreeDecoder\#slow\#Server_1 + \\ 128.0 * b\#CommercialDecoder\#slow\#Server_1 + \\ 512.0 * b\#CommercialDecoder\#fast\#Server_1 + \\ 1536.0 * b\#CommercialDecoder\#ultrafast\#Server_1 \end{aligned} \tag{6}$$

The same principle is applied for software-related non-functional properties, which are the third class of constraints as shown exemplary in Equation 7. It states, among others, that the *throughput* will be five bit per second, if the `URLReader` is mapped to Server 1 and configured to run in `URL` mode.

$$
\begin{aligned}
throughput = 5.0 * b\#URLReader\#url\#Server_1 + \\
20.0 * b\#FileReader\#file\#Server_2 + \\
5.0 * b\#URLReader\#url\#Server_2 + \\
20.0 * b\#FileReader\#file\#Server_1
\end{aligned} \tag{7}
$$

Finally, the user demands have to be integrated as a constraint, too. Such a user request, like playing a video with a framerate of at least 20 frames per second, can be integrated as a constraint in a straightforward way:

$$framerate >= 20.0; \tag{8}$$

The ILP as a whole is generated at runtime, whereby the required information is extracted from the runtime model of the current hardware infrastructure and software configuration (software variant model), as well as from the ECL contracts. To solve the ILP a variety of free-to-use solvers exists. For our prototype we have chosen LP_Solve 5.5[5]—one of the mature, stable solvers. LP_Solve allows to solve linear programs (LP), too.

The key difference between LP and ILP is that an ILP restricts its variables to be integers instead of floating reals. In our scenario we need floating reals for the resource usage and property variables, but integers for our mapping selection variables. In consequence, the ILP presented above is a mixed integer linear program (MILP). The need for the integer restriction can be easily illustrated: if the variables $b\#VLC\#highQuality\#Server_1$ and $b\#VLC\#highQuality\#Server_2$ are allowed to be floating reals the LP's solution could be, to map 33% of the `VLC` to Server 1 and the remaining 67% to Server 2, which is obviously not possible.

The most commonly used algorithm to solve a MILP is the simplex algorithm [12]. The major drawback of simplex is its exponential runtime. But, importantly for our scenario, it is an iterative approach. That is, once an MILP has been solved, slight changes to it do not require to perform the whole computation again, but only parts of it. Thus, unless the system significantly changes, our optimization approach will benefit from this property of the algorithm. We plan to measure the energy consumption of solving our ILPs to derive a model for the prediction of the energy required to compute the optimal configuration.

The solution of the example introduced throughout the paper is that the *VLC* implementation should run in *highQuality* mode on Server 1, the *CommercialDecoder* should run in *slow* mode on Server 2 and the *URLReader* in *URL* mode on Server 1. The `Decoder` implementation is mapped to Server 2 instead of Server 1, due to the CPU performance requirements. If all implementations run on Server 1, the resulting energy consumption will be lower, but the framerate of at least 20 fps cannot be ensured. Furthermore, the solution of the ILP tells us amongst other details, that we need the CPU of Server 1 to operate at 1.5GHz and the CPU of Server 2 at 800 MHz. Thus, we could force the CPUs

[5] http://lpsolve.sourceforge.net/5.5/

to operate slower than usual by exploiting the application knowledge in terms of the ILP to save energy whilst ensuring the requested user utility.

Notably, if a new server, with a more powerful CPU, is added to the infrastructure, the ILPs solution is to map all three implementations to that new server, which requires a system reconfiguration: all implementations have to be migrated from Server 1 or 2 to Server 3. In general, changes in the infrastructure as well as to (the available) component implementations are propagated to our variant models at runtime, which are then used to generate an ILP to derive the optimal system configuration. Finally, the system has to perform a reconfiguration, which is a sequence of migration steps.

4 Related Work

Within the research project COMQUAD, a component model was developed that separated components into their specifications and implementations [6], similar to the component types and component implementations of the CCM. Additionally, the contract language CQML+ [13] was developed to describe required and provided non-functional properties of software components. The enhancement of our approach is the more detailed modeling and monitoring of resources.

During the research project SPEEDS and its successor CESAR, the HRC metamodel [16] was developed. It allows for component-based development of embedded systems, which includes capabilities to describe hard- and software components and their behavior. CESAR focuses on a multi-viewpoint, multi-level development process for embedded systems. Contracts are a central concept in HRC models which are organized in behavior, safety and real-time viewpoints. Contracts capture functional and non-functional assumptions and promises of HRC components. They are used to reason about the consistency of a given HRC model. In contrast to our approach, SPEEDS and CESAR focus on contract negotiation at development time and not at runtime. Thus, the HRC metamodel and its successor CSM support variability at development time, whereas the CCM focuses on runtime. In addition, neither SPEEDS nor CESAR explicitly consider energy consumption or EAT.

In the MADAM research project and its successor MUSIC, a component model for self-adaptive applications on mobile devices has been developed [5]. It supports modeling of non-functional properties and implementation variants. Although, energy optimization is possible in general, in contrast to our approach, MADAM/MUSIC do not focus on complete hardware landscapes.

The DIVA research project focused on the management of dynamic adaptive systems with special focus on the problem of exponential growth of potential system configurations by combining methods from aspect-oriented programming/modeling [9] and Model-Driven Software Development (MDSD) [11]. The DIVA approach allows to automatically adapt a system at runtime supporting goal-based optimization of non-functional properties as well as rule-based reconfiguration of the system [4]. The major difference to our approach is the level of abstraction regarding the values of non-functional properties. DIVA symbol-

izes the impact of implementations on non-functional properties (i.e., the free size of memory is represented by symbols like *LOW*, *MEDIUM* and *HIGH* and the impact of an implementation can only be expressed as being low, medium and so on, too). Our approach allows to consider subsymbolic information in addition (i.e., the actual value of free size of memory in MB). We encapsulate symbolization in our contracts, as was shown in Sect. 2.2. Though, reasoning on subsymbolic information is less efficient, due to the raised complexity, it allows to derive finer-grained configurations. E.g., a configuration could include not just the information which CPU to use, but the (optimal) frequency this CPU should have. Such fine-grained information allows to reduce energy consumption in addition to the coarse-grain decision of which resources to use. Current hardware is usually far from being energy-proportional [1], which is reflected by a very high baseload electricity and a narrow working area. Imagine, for example, a server consuming 100W being idle and 120W at full load. In consequence, energy savings can mostly be achieved by selecting or turning off the right resources. In such a scenario symbolic reasoning, as in DIVA, is feasible. But especially for the next generation of hardware, which is supposed to be more and more energy-proportional [3] there is a need for finer-grained energy optimizations.

5 Conclusion

In this paper we introduced our contract negotiation approach, which allows to identify the most energy-efficient mapping of software component implementations to resources serving a user's request and demands.

We showed how to formulate this optimization problem as an integer linear program (ILP) and presented a mechanism using models at runtime to generate the ILPs in accordance to the current hard- and software as well as the current users requests and demands. This allows to optimize even dynamic systems, whose hard- and software entities can supervene, disappear or change over time.

Our approach allows for runtime subsymbolic reasoning, which demarcates us from existing approaches. Notably, energy efficiency is just one quality which benefits from subsymbolic reasoning. E.g., optimizations in systems integrating the physical and virtual world (like robot swarms) require the support for floating point numbers, too.

In the future we plan to improve the approach to consider more complex relations between resource usage and energy consumption and will evaluate our approach in a real world scenario. Furthermore, we plan to investigate how our approach can be applied to systems integrating the physical and virtual world.

Acknowledgement

This research has been funded by the European Social Fund and Federal State of Saxony within the project ZESSY #080951806, by the Federal Ministry of Education and Research within the project CoolSoftware #FKZ13N10782, part of the Leading-Edge Cluster "Cool Silicon" within the scope of its Leading-Edge Cluster Competition and by the collaborative research center 912 (HAEC), funded by the DFG.

References

1. L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, 2007.
2. A. Beugnard, J.-M. Jézéquel, and N. Plouzeau. Contract aware components, 10 years after. In *Electronic proceedings in theoretical computer science*, number 37, pages 1–11, 2010.
3. S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54:67–77, May 2011.
4. F. Fleurey and A. Solberg. A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 606–621, Berlin, Heidelberg, 2009. Springer-Verlag.
5. K. Geihs, M. U. Khan, R. Reichle, A. Solberg, S. Hallsteinsen, and S. Merral. Modeling of component-based adaptive distributed applications. In *SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 718–722, New York, NY, USA, 2006. ACM.
6. S. Göbel, C. Pohl, S. Röttger, and S. Zschaler. The COMQUAD component model - enabling dynamic selection of implementations by weaving non-functional aspects. In *Proceedings of the 3rd international conference on aspect-oriented software development, Lancaster, UK, March 22 - 24, 2004*, volume 3, pages 74–82, New York, NY, USA, March 2004. ACM Press.
7. S. Götz, C. Wilke, M. Schmidt, and S. Cech. THEATRE resource manager interface specification. Technical Report TUD-FI10-0X, Technische Universität Dreden, Dresden, Germany, 2010.
8. S. Götz, C. Wilke, M. Schmidt, S. Cech, and U. Aßmann. Towards energy auto tuning. In *Proceedings of First Annual International Conference on Green Information Technology (GREEN IT)*, pages 122–129. GSTF, 2010.
9. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Berlin / Heidelberg, 1997.
10. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
11. B. Morin, O. Barais, G. Nain, and J.-M. Jézéquel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 122–132, Washington, DC, USA, 2009. IEEE Computer Society.
12. G. L. Nemhauser and L. A. Wolsey. *Integer and combinatorial optimization*. Wiley-Interscience, New York, NY, USA, 1988.
13. S. Röttger and S. Zschaler. CQML+: Enhancements to CQML. In *Proceedings of the 1st International Workshop on Quality of Service in Component-Based Software Engineering*, pages 43–56, Toulouse, France, 2003. Cépaduès-Éditions.
14. P. Süttner. Abstract behavior description of CCM software components (Abstrakte Verhaltensbeschreibung von CCM Softwarekomponenten). Master's thesis, Technische Universität Dresden, Mar. 2011.
15. C. Szyperski, D. Gruntz, and S. Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley and ACM Press, 1999.
16. The SPEEDS Consortium. D.2.1.5 SPEEDS L-1 Meta-Model. http://speeds.eu.com/downloads/SPEEDS_Meta-Model.pdf, May 2009.