

Towards reliable self-integrative IT systems

Dirk Niebuhr
TU Kaiserslautern
Gottlieb-Daimler-Str.
D-67653 Kaiserslautern
+49-631-205-3366
niebuhr@informatik.uni-kl.de

Christian Peper
Fraunhofer IESE
Sauerwiesen 6
D-67661 Kaiserslautern
+49-631-707-219
peper@iese.fhg.de

Andreas Rausch
TU Kaiserslautern
Gottlieb-Daimler-Str.
D-67653 Kaiserslautern
+49-631-205-3360
rausch@informatik.uni-kl.de

ABSTRACT

IT components with information processing and communication capabilities are embedded in almost all commonplace and industrial objects. In our daily life we depend more and more on these IT components. On the other hand, users will increasingly expect that those IT components collaborate autonomously and thus provide emergent properties.

Hence we have to guarantee their reliability, even though the resulting organically grown IT systems are never developed and tested in advance: If, for example, a novel type of child safety seat with integrated child detection is used on the front passenger seat, the passenger side airbag could be deactivated automatically and the child safety lock could be activated. However, if the safety seat is located on the rear passenger bench, the passenger side airbag must not be deactivated!

This paper suggests a framework for the development of reliable so called self-integrative systems. Thereby IT components are integrated (resp. removed) at execution time. The reliability of the resulting IT system is checked with regard to the specified functional, non-functional, and emergent properties based on an underlying formal model.

Categories and Subject Descriptors

D.2, D.2.1, D.2.2, D.2.4, D.2.6, D.2.11, D.2.12, D.3.1, F.3.1.

General Terms

Design, Reliability, Languages, Theory, Verification.

Keywords

System development, Development Techniques, Adaptive systems, Self-integrative systems

1. INTRODUCTION

In these days the trend "everything, every time, everywhere" becomes more and more obvious: for example the internet can be accessed using mobile phones. Furthermore electronic assistants, so called "information appliances", like network-enabled PDAs, WAP-capable cell-phones and electronic books or tourist guides are available in stores all over the country.

The continuing progress of all IT branches towards "smaller, cheaper, more powerful" mainly enables this trend. Moreover new developments in the field of materials science like midjet-sensors, organic light emitting devices or electronic ink and the evolution in communications technology, especially in the wireless sector, contribute to embedding IT components in nearly every industrial or everyday life object.

These IT components, such as a vehicle's navigation system or a child detection system integrated in a child safety seat, can have different capabilities and size. Nevertheless, they share the following characteristics: *IT components*

- are independent IT systems consisting of hardware and software
- provide specific functionality with appropriate non-functional properties
- are capable of processing information and have the ability to communicate
- have strict demands for self-integration within an organic environment

Nowadays, these IT components are being more and more used within an organically grown, inhomogeneous, and dynamic environment. Users expect those IT components to collaborate autonomously (*self-integrative*) and thus provide a real added value to the user (*emergent properties*). On the other hand, we depend more and more on those organically grown IT systems. Hence their correctness has to be guaranteed (*reliability*) even though these organically grown IT systems are never developed and tested in advance. Providing those kinds of reliable self-integrative IT systems with emergent properties is still a great challenge in software engineering [13], [14].

We understand *reliable self-integrative IT systems* as distributed application systems consisting of an organically grown set of collaborating IT components. Additional IT components can be integrated into the system at any time. Integrated IT components can be removed from the system, respectively they can fail as the result of a defect. Integration (resp. removal) of IT components into (resp. from) a self-integrative IT system has to be performed at *execution time*.

A *reliable self-integrative IT system* has to guarantee that only *valid configurations* of the system can be executed. In order to achieve this, the *correctness* of a system has to be evaluated considering the integrated (resp. removed) IT components. In this context, *correctness* means that the system under consideration is consistent with its specification following the well-known notion of Hoare triples [15]. If the resulting system is not correct, the previous valid system configuration has to be executed or the system has to be halted for safety reasons.

A correct and safe system configuration should only be executed if it is better than the one executed before. A *reliable self-integrative IT system with emergent properties* has the ability to rate different system configurations with respect to the added user value and select the reliable and optimal system configuration.

Approaches supporting the development of such IT components have been developed and successfully deployed over the past years like for example the Kobra-method for component-based software development [1]. Systems are no longer redeveloped from the scratch but composed of existing components ([2], [3]). Moreover a couple of various technologies exist to support self-integration of these components during execution time, e.g. Universal Plug-and-Play (UPnP) [4], Salutation-Framework [5], Jini Framework [6], Ronin-Agent-Framework [7] and CORBA Trading Service [8].

However, these “classical” development approaches and implementation technologies mainly focus on the specific requirements of the IT component under consideration (cf. [9]). These development approaches have to be enhanced towards a sophisticated development approach for those kinds of reliable self-integrative IT systems with emergent properties.

The paper is structured as follows: in Section 2 the sample application is introduced and the research topics, which are motivated by this application, are explained. Section 3 deals with the approach, we introduce in order to provide solutions to those research questions and finally Section 4 gives some information what still can be done in the future after establishing the development approach.

2. SAMPLE APPLICATION

The sample application we use to motivate and illustrate the proposed development approach for reliable self-integrative IT systems is a vehicle. For our sample we assume that after development and manufacturing of the vehicle a new type of child safety seat has been developed. This child safety seat has an integrated IT component to be integrated into the car system dynamically, although the seat was not known when the vehicle was manufactured. According to the presence respectively position of the child safety seat, the airbag and the window lift in the front respectively in the rear have to be deactivated. Furthermore the child lock has to be activated if the seat is placed in the rear. This behavior of the system can be seen in Figure 1 for the front and rear position of the child safety seat. Green color means, that the system is activated while orange color means, that the system must not be activated. In case of the child lock, the orange color means that the door is locked.

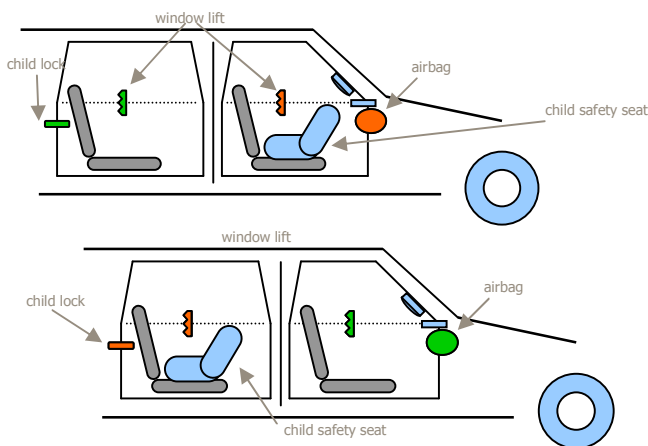


Figure 1. Behavior of the system in two different cases

The following research topics can be derived from this sample application:

- *Integration and deactivation of IT components:* If a navigation system is integrated into a vehicle, the voice output of the navigation system could be used for additional acoustical output of critical system states like pressure drop in a tire or inflated temperature of the engine. Therefore, the IT component that is controlling the display has to be exchanged. Integration and exchange of an IT component nowadays require a complete update of the vehicle’s software. Because of the diversity of variants, only some selected configurations of a vehicle are tested and released. That is the reason why it may even be necessary to exchange parts of the system that are not affected by the change, like the control of the display in this example. This situation is not acceptable at all. Self-integrative IT systems have to support the integration and deactivation of IT components in an efficient way.
- *Reliability of self-integrative IT systems:* If a new kind of child safety seat with an integrated chip for child detection is placed on the front passenger seat, then the child safety seat could register itself at the on-board computer and cause the deactivation of the airbag and the window lift and, furthermore, the activation of the child safety lock. Nevertheless, according to the requirements specification, it still has to be possible to deactivate the child lock using a central switch (correctness of functional properties). Moreover, the integration of the child safety seat requires a check of non-functional properties such as safety properties, too. The airbag for the front-seat passenger must not be deactivated if the child safety seat is placed in the rear of the vehicle, for example (correctness of non-functional properties).
- *Design of an open domain architecture framework with emergent properties:* A co-worker arranged a meeting with a customer using his brand new UMTS-based PDA with an included visual telephone. He has never used this PDA in his car before. Although the PDA had not yet been developed at the time the car was manufactured, the PDA is recognized by the car and integrated into the system of the vehicle. This happens instantly when the person enters the car in order to drive to the meeting. The self-integrative PDA transmits the address of the customer to the navigation system of the vehicle. The navigation system can now calculate the best route and guide the driver. The self-integration of new IT components, which were not known at the development time of the system, provides the possibility to improve the value of the whole system over the years – emergent properties. Self-integration can only be realized based on a shared and open domain architecture framework. This framework has to be designed from domain requirements similar to a framework of a product line. But product lines are not sufficient enough. The number of possible valid configurations is uncountable as extensions have to be provided to support unpredictable variability and added value.

3. DEVELOPMENT OF RELIABLE SELF-INTEGRATIVE IT SYSTEMS

The main research is how to correctly compose an executable system based on temporarily available physical components with respect to their different functionalities. Such a system implements at least a single predetermined logical architecture in order to satisfy certain user requirements. For this reason, the set of all applicable logical domain architectures is described in an *open domain architecture framework* as shown in Figure 2.

The derivation of such a framework specification from the user requirements specification is not considered within this paper. We rather concentrate on the selection and adaptation of suitable specification techniques for the domain architecture framework and on the automatic realization of the framework with dynamically changing components. Note that the determination of a framework specification technique is an important prerequisite for the presented approach.

The *domain architecture framework* describes a potentially infinite set of valid *domain architectures*. A *domain architecture* is a set of collaborating *logical IT components*. For each IT component, *syntax and behavior* of the component's interface is specified (cf. [16], [17], [18], [19]). Therefore the number of all IT components in the framework is finite.

In reliable self-integrative IT systems, the availability of *physical IT components* can arbitrarily change during execution time: integrated components can fail and new components can come along, as illustrated by D1 in Figure 2. Whenever the available physical components change or fail during execution, the system has to be reconfigured.

To this purpose, a *realization relation* is dynamically derived (D2 in Figure 2) using the specifications of the architecture framework and those of the available physical components. Each physical IT component as well as each logical IT component provides a specification of their syntactical interfaces. The realization relation maps a physical IT component to a corresponding logical IT component in the domain architecture framework, iff their syntactical interfaces are identical. Note that the realization relation is finite, because the number of physical components and the number of logical components are both limited.

If the resulting mapping is complete, i.e., all logical IT components with a domain architecture could be mapped to available physical IT components, the mapping is called a *system configuration*. However, this system configuration is only a *valid system configuration*, iff each physical IT component is a *correct* implementation of the specification provided by the corresponding IT component in the domain architecture. Whether a physical IT component is a correct refinement of the specified IT component cannot be automatically proven during execution time, as for a complete mathematical proof a theorem prover with user interaction is required.

For that reason, we propose a runtime verification approach: the related domain architecture specification contains a set of predicates for each IT component, similar to the predicates used in Hoare triples [15]). Obviously, Hoare triples are not powerful enough. We use temporal logical expressions with free variables

to cover timing constraints. Additionally, we model not only the component but also its abstract environment to reason on an abstract global system state (cf. [18], [19]).

These predicates are added to the physical IT components using techniques like metadata annotations in Java (cf. [20]) before they are executed (see D4 in Figure 2). During their execution a runtime verification tool (like [21]) checks the predicates. As a complete correctness check may take too long, we only check each single computation step in advance.

It must be guaranteed that the execution of the physical IT components does not lead into an IT system state where the predicates do no longer hold. Hence these physical IT components must be executed in a safe environment (“*SANDBOX*” in Figure 2) this has to be ensured by the *execution environment* for reliable self-integrative IT systems. Whenever the execution environment detects that a further execution will lead to an invalid system state, the system configuration is outdated and has to be recalculated. The determination of the realization mapping as well as the annotation and monitored execution of the physical IT components is also automatically performed with each change in the physical constellation of IT components.

In general, the same scenario of physical IT components will realize multiple domain architectures of the framework. To evaluate the realized architectures with respect to emergent properties, an *emergence evaluation function* (EEF) has to be defined. This function describes how to traverse the set of domain architectures when looking for a “good” realization. The EEF is expected to change dynamically, e.g., if a user changes his or her preferences. Approaches for the derivation of the EEF – e.g., by learning from the user behavior – have to be further considered.

The emergence properties of the IT system are exploited by searching the set of valid domain architectures for a local or even global maximum. The search uses the EEF, which could be given by a partial ordering of all domain architectures in the framework. This ordering is expected to have at least one minimum – representing minimal valid domain architecture without any added value. The best detected domain architecture is chosen for annotation and instantiation as described above. Suitable techniques for the efficient management and application of the EEF have to be developed.

First experiences in [22] have shown that it is important to integrate location information into the involved specifications: if a display and an input device shall be used to realize a user interface, they should be close to each other. This can be specified in the corresponding domain architecture by putting them into the same location group (cf. [23]). A physical component scenario may only realize this architecture if it can offer a component constellation fulfilling this restriction. It is expected that further similar property classes can be identified during the project.

Using this approach, it can be guaranteed that reliable self-integrative IT systems satisfy the functional, non-functional and emergent properties of the domain architecture framework specification. Emergence properties are explicitly designed and incorporated into the framework to exploit the potentials of dynamic IT systems.

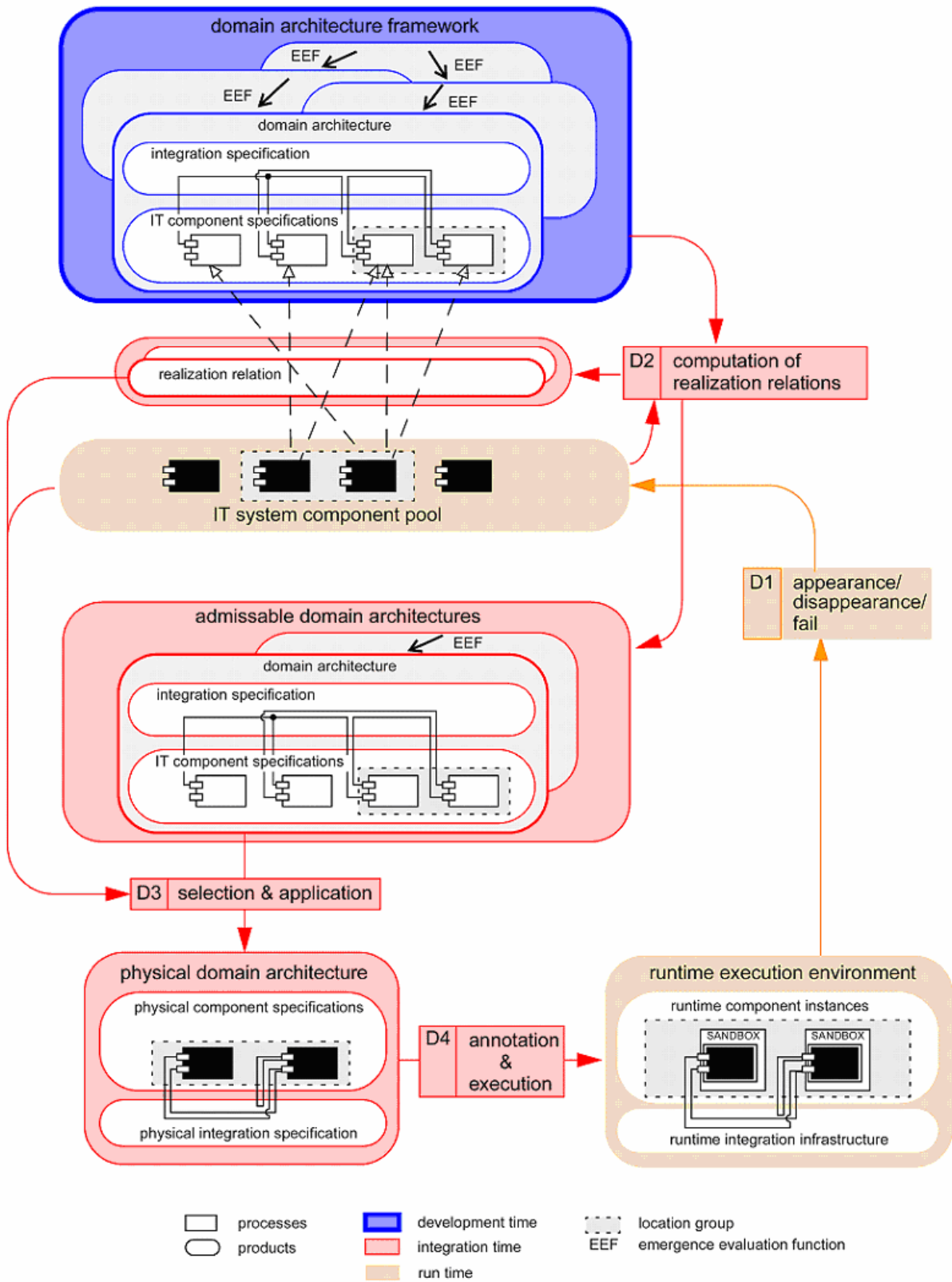


Figure 2. Development approach for reliable self-integrative IT systems

4. FURTHER WORK

In this paper we have motivated and presented a rough sketch for a development approach for reliable self-integrative IT systems with emergent properties. It provides techniques to dynamically integrate and remove IT components at execution time. The reliability of the resulting IT system is checked with regard to the specified functional, non-functional, and emergent properties based on an underlying formal model.

The described specification techniques and formal foundations have to be further designed and elaborated. A first step towards the needed specification techniques for the proposed IT components has already been developed and applied (cf. [12], [18], [19]).

Based on these experiences the proposed integration and verification techniques can be developed. An algorithm for the composition of the IT components and a mechanism for verification of the system have to be developed and realized within a development and execution platform for self-integrative systems.

Such a platform is providing the central services used by all IT components, such as a lookup-service or an integration service. We experimented with Jini which is such a platform and developed a prototype capable of dynamic integration of heterogeneous devices [24]. This prototype might be extended with the concepts of run time verification sketched in this paper. Note that some basic requirements have to be matched by all components participating in this platform, e.g., compatibility of communication hardware and protocols, or export of the offered functionality in a standardized format. Tool support for the development of self-integrative software systems has also to be provided within the platform, because the involved specifications are automatically processed, e.g., in consistency checks or code generation.

5. REFERENCES

- [1] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. Component-Based Product Line Engineering with UML. The Component Software Series. Addison-Wesley, 2001. ISBN 0-201-73791-4. 2001.
- [2] Clemens Szyperski. Component Software: Beyond Object-Oriented Programming. Addison Wesley Publishing Company. 1997.
- [3] Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig. Putting the Parts Together – Concepts, Description Techniques, and Development Process for Componentware. Proceedings of the 33th Annual Hawaii International Conference on System Sciences, IEEE Computer Society. 2000.
- [4] Universal Plug and Play Device Architecture Reference Specification. Microsoft Corporation. 2004.
- [5] Salutation Architecture Specification (Part 1), version 2.1. The Salutation Consortium Inc. 1999.
- [6] John Rekish. UPnP, Jini and Salutation - A look at some popular coordination framework for future network devices. Technical Report, California Software Lab. <http://www.cswl.com/whiteppr/tech/upnp.html>. 1999.
- [7] Chen Harry. Ronin Agent Framework. <http://gentoo.cs.umbc.edu/ronin/>. 2004.
- [8] OMG. CORBA 3.0 – OMG Specification. 2002.
- [9] D. Chakraborty, A. Joshi, Y. Yesha, T. Finin. Service Composition for Mobile Environments. Journal on Mobile Networking and Applications (MONET), Special issue on Mobile Services. 2004.
- [10] Leslie Lamport. Proving the Correctness of Multiprocess Programs. IEEE Transactions on Software Engineering, Vol.3 No.2. 1977.
- [11] Hermann Kopetz, Günther Bauer. The Time-Triggered Architecture. Proceedings of the IEEE, Vol.91 No.1. 2003.
- [12] Dominic Hillenbrand. Spezifikation und Simulation eingebetteter Komponenten. Diploma Thesis, Dept. of comp. science, University of Kaiserslautern, 2004.
- [13] Positionspapier der Gesellschaft für Informatik (GI) und der Informationstechnischen Gesellschaft im VDE (ITG). Organic Computing, Computer- und Systemarchitektur im Jahr 2010. 2003.
- [14] Dirk Niebuhr, Christian Peper, Andreas Rausch. Towards a development approach for dynamic-integrative systems. In: Proceedings of the Workshop for Building Software for Pervasive Computing, 19th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA). November 2004.
- [15] Charles A. Hoare. Communicating Sequential Processes. Prentice Hall. 1985.
- [16] Andreas Rausch. Software Evolution in Componentware Using Requirements/Assurances Contracts. In Proceedings of the 22th International Conference on Software Engineering, IEEE Computer Society. 2000.
- [17] Andreas Rausch. Componentware: Methodik des evolutionären Architektorentwurfs. PhD Thesis, Technische Universität München. Herbert Utz Verlag. 2001.
- [18] Andreas Rausch. "Design by Contract" + "Componentware" = "Design by Signed Contract". In Journal of Object Technology, Vol. 1, no. 3. 2002.
- [19] Andreas Rausch, Manfred Broy. Evolutionary Development of Software Architectures. In: Technology for Evolutionary Software Development, a Symposium organised by NATO's Research & Technology Organization (RTO). 2002.
- [20] JSR175. Java specification request for metadata annotations (JSR175). <http://77jcp.org/en/jsr/detail?id=175>.
- [21] Eric Bodden. A lightweight LTL runtime verification tool for Java. In proceedings of the OOPSLA'04 ACM Conference on Object-Oriented Systems, Languages and Applications (Companion), Vancouver, Canada, October 2004.
- [22] Thomas Fischer, Dirk Niebuhr, Marcus Trapp. AmI dynamic integration demonstrator - assistant training application scenario. Demonstration and presentation at the German-Hungarian Ambient Intelligence workshop, Budapest, October 2004.
- [23] Klaus Bergner, Radu Grosu, Andreas Rausch, Alexander Schmidt, Peter Scholz, Manfred Broy. Focusing on Mobility. In: HICSS 32, Proceedings of the Thirty-Second Annual Hawaii International Conference on System Sciences, Ralph H. Sprague, Jr., IEEE Computer Society. 1999.
- [24] Christian Bartelt, Thomas Fischer, Dirk Niebuhr, Andreas Rausch, Franz Seidl, Marcus Trapp. Dynamic Integration of Heterogeneous Mobile Devices. In Proceedings of the workshop DEAS 2005 (ICSE 2005), 2005.