Abstract—We propose a framework for top-down Web Service interoperation based on an aggressive version of model-driven development (AMDD). The point here is to govern the construction and customization of complex Web Applications at the model level in a framework that allows application experts to directly formulate their desires in an adequate way. Adequate means in this context that applications can be automatically validated, executed, tested, and deployed by the application experts, inside a framework that takes care also of second-order concerns.

Our approach, which focuses on functionalities as the basic entities of the design space is tailored to make second-order issues like interoperation, distribution, and compatibility simple for the many, difficult for the few: simple for the many, as the advocated approach hides most of the intricate second-order issues from the application designer, and difficult for the few, as these issues must be dealt with by means of complex compilation, synthesis or technology mappings. Our experience indicates that this approach has the potential to cover and thereby drastically simplify the bulk of modern Web application development and customization.

I. MOTIVATION

The Semantic Web, as sketched in Tim Berners-Lee’s visionary paper [4], proactively proposes collaborative solutions to heterogeneous, situative problems, like in ubiquitous computing scenarios. Accordingly, future (Web) applications will be highly heterogeneous, they will comprise special purpose code, perhaps written in different programming languages, integrate legacy components, glue code, and adapters combining different technologies, which may run distributed on different hardware platforms, on powerful servers or at (thin and ultra-thin) client sites.

Already today’s (Web) applications require an unacceptable effort until deployment, which is typically caused by incompatibilities, feature interactions, and the sometimes catastrophic behavior of component upgrades, which no longer behave as expected. As explained in more detail in Section II, also recent approaches, like the one based in BPEL4WS, are too low level and intricate to be mastered by application experts who are not programmers.

In this paper we propose a high-level framework for the programming-free Web application design, which is based on

- a simple functional characterization of the components in terms of taxonomies (or, more generally, ontologies),
- a graphical application-composition environment that works with ontologies and, most importantly,
- two methods for service choreography and orchestration, i.e. composition of services that respects static and dynamic compatibilities:
  - a method based on constraints written in temporal logic [30], [22] which characterize the goal of the intended application. Specifications of this kind can automatically be used to synthesize executable specifications in terms of Service Logic Graphs (see next item), which themselves can be automatically deployed.
  - Specifications in terms of Service Logic Graphs (SLGs) [21], [23]. SLGs are coarse granular operational models for the intended process flow of the application. Their structure is tailored to establish a bridge between an application viewpoint and a realization in software.

Both of these methods can be applied statically (choreography), with a subsequent compilation step, or dynamically (orchestration) in an interpreter-based environment, which is particularly suited for prototyping.

Our AMDD approach [24] yields the conceptual backbone for the realization of our high-level framework for the programming-free Web application design. It identifies adequate modelling levels, as well as the corresponding need for compilation and synthesis, and it organizes the required distribution of labor. The Application Building Center (ABC), a successor of the METAFrame environment [21], is a first corresponding implementation, which already addresses all the issues raised above.

This exposition focusses on the component management and compatibility issues of AMDD as realized in terms of SLGs, as this level establishes a rather transparent link between the state of the art modelling of Web Services, and typical description at the application expert level.

In the following, Section II will sketch the recent development in the Web Services community, Section III provides an overview of the AMMD approach, while Section IV presents our Application Building Center (ABC), a realization of the AMDD approach. Finally, Section V
discusses three example applications, which have been realized using the ABC, and Section VI presents our conclusions and future perspectives.

II. BACKGROUND

An ongoing significant effort is being spent in the definition, standardization, and improvement of a Business Process Execution Language for Web Services (BPEL4WS) [5], a language aimed at describing business processes that include multiple Web Services and standardized message exchange internally and between partners [16]. Establishing a standard for describing the flow of tasks, the order in which they need to be performed, the type of data shared and how other partners are involved would greatly simplify the definition of very complex processes and workflows, and thus be beneficial for the specification of complex Web applications.

Currently, however, BPEL4WS does not yet satisfy our needs in a few respects that concern both its conceptual maturity and the acceptance by the potential users. Concerning acceptance, BPEL4WS is still quite close to a programming language, while we want to address domain experts who are not themselves programmers. Initial attempts to provide a graphical version may be promising in the long term but are currently no solution, as we cannot require our users to install packages at their sites.

Concerning its focus, BPEL4WS aims at the definition of workflows at a level which is conceptually too complicated to be mastered by domain experts. In fact, BPEL4WS allows the definition of both executable and abstract processes. Thereby

- the executable processes allow a wealth of composition operators, including problematic features like e.g. join conditions with negative occurrences (XOR-joins), whose semantics is inherently non-local. This is not unique to BPEL4WS: also in the case of widespread workflow definition languages like the EPCs (Event-driven Process Chains), the semantics of these constructs refers to itself, and in the scope of a negation, producing a vicious circle. This is a serious problem if one looks for an easy and clean semantics, a problem felt both in the EPC world [14] and in the Semantic Web context [17]. Relying on the 80/20-tradeoff (one can cover 80% of the applications with only 20% of the conceptual effort), we therefore prefer to deal with a very simple and well-established subset of operators, not being afraid of paying the price of limiting the scope of our setting.
- The abstract processes allow the descriptions of business protocols, which is very useful to define the interaction of processes that one already knows. Using the Web, one typically does not know what is in the repository. Thus loose logical specifications are required, which support the domain expert in his use of the available components functionalities. Our synthesis facility [30], [22] is able to automatically generate executable workflows using loosely specified components functionalities.

Finally, the 80/20-approach also eliminates most of the problems one faces when using the Web Service Choreography Interface (WSCI) [31], the current W3C proposal for the description of dynamic behaviors on top of WSDL, as our modelling level avoids to explicitly deal with details of message exchange between components.

III. AMDD: AGGRESSIVE MODEL DRIVEN DESIGN

Even though high level programming languages and even model driven development are used for component development, the system-level point of view is typically not adequately supported. In fact, in particular a significant part of the deployment of a heterogeneous application is still a matter of assembly-level search for the reasons of incompatibility, which may be due to minimal version changes, slight hardware incompatibilities, or simply to hideous bugs, which come to surface only in a new, collaborative context of application. Integration testing and the quest for 'true' interoperability are indeed major cost factors and major risks for system implementation and deployment.

AMDD is a framework for top-down organization of (Web) Service interoperation, which moves most of the recurring problems of compatibility and consistency of software (mass) construction and customization from the coding and integration level to the modelling level. Rather than using component models - as usual in today's Component Based Development paradigm - just as a means of specification, which

- need to be compiled to become a 'real thing' (e.g., a component of a software library),
- must be updated (but typically are not), whenever the real thing changes
- typically only provide a local view of a portion or an aspect of a system,

application models are the center of the design activity, becoming the first class entities of the global system design process. In such an approach, as shown in Sect. IV,

- libraries are established at the modelling level: building blocks are (elementary) models, rather than software components,
- applications are specified by model combinations (composition, configuration, superposition, conjunction...), viewed as a set of constraints that the implementation needs to satisfy,
- global model combinations are compiled (synthesized, e.g. by solving all the imposed constraints) into a homogeneous solution for a desired platform,
- application changes (upgrades, customer-specific adaptations, new versions, etc.) happen primarily (ideally, only) at the modelling level, with a subsequent global recompilation (re-synthesis).
This aggressive style of model-driven development [24] strictly separates compatibility and migration issues from model/functionality composition and heavily relies on compilable and synthesizable models. This way it would be possible to better control or even in some cases overcome the problem of incompatibility between (global) models, (global) implementations, and their components, since compatibility is established and guaranteed at development time, and later-on maintained by (semi-) automatic compilation and synthesis.

In fact, AMDD, when applicable, has the potential to drastically reduce the long-term costs due to version incompatibility, system migration and upgrading. Thus it helps protecting the investment in the software infrastructure. We are therefore convinced that this aggressive style of model-driven development will become the development style at least for mass customized software in the future.

A. The Scope of AMDD

Of course, AMDD will never replace genuine application development, as it assumes techniques to be able to solve problems (like synthesis or technology mapping) which are undecidable in general. On the other hand, more than 90% of the application development costs arise worldwide at a rather primitive development level, during routine application programming or software update, where there are no technological or design challenges. There, the major problem faced is software quantity rather than achievement of very high quality, and automation should be largely possible. AMDD is intended to address (a significant part of) this 90% ‘niche’.

In other words, AMDD aims at making things that inherently are simple as simple as they should be. In particular this means that AMDD is (at least in the beginning) characterized by abstractions, neglecting interesting, but at a certain level of development unnecessary, details, like e.g. distribution of computation, methods of communication, synchronization, real time. General application development practices can be replaced here by a model and pattern-based approach, adequately restricted to make AMDD effective.

The bulk of Web services development has this profile, and being a fast-moving area, the benefit of moving significant parts of the application development to the application expert, which eliminates communication hurdles and drastically reduces time to market, is particularly high here.

IV. AN AMDD-REALIZATION

The Application Building Center (ABC) developed at METAFrame Technologies in cooperation with the University of Dortmund is intended to promote the AMDD-style of development in order to move the application development for certain classes of applications towards the application expert. The ABC has been successfully used in the design and customization of Web Services. Even though the ABC should only be regarded as a first step of AMDD development, it comprises the essential points of AMDD, that concern dealing with a heterogeneous landscape of models, supporting a number of formal methods and providing tools that enforce formal-methods based validation and verification, and providing automatic deployment and maintenance support, as shown in Fig.1.

A. Heterogeneous landscape of models

The central model structure of the ABC are hierarchical Service Logic Graphs (SLGs)[21], [23] (Fig. 2). SLGs are flow chart-like graphs. They model the application behavior in terms of the intended process flows, based on coarse granular building blocks called SIBs (Service-Independent Building blocks) which are intended to be understood directly by the application experts [21] – independently of the structure of the underlying code, which, in our case, is typically written in Java/C/C++. The component models, the feature-based service models called Feature Logic Graphs
Here, as also detailed in [12] for a specific application,
- the elementary (Web-)Services are modelled as SIBs,
- application models are realized as SLGs, and
- SLGs are themselves constructed directly as a combina-
tion of basic services, or more often by reuse of specific
subservices in terms of (inhouse or external) features,
modelled as Feature Logic Graphs.

Web applications are composed by drag and drop in a
simple, graphical service definition environment, at the model
level.

Additionally, as shown in (Fig. 3) the ABC supports model
specifications and consistency specifications in terms of
1) two modal logics, to abstractly and loosely characterize
valid behaviors (see also [10]),
2) a classification scheme for building blocks (SIBs or
hierarchical sub-services called Macros, Features, or
services, depending on their role in the design) and
types in terms of ontologies, and
3) high level type specifications, used to specify compat-
ibility between the building blocks of the SLGs.

The granularity of the building blocks is essential here as
it determines the level of abstraction of the whole reasoning:
the verification tools directly consider the SLGs as formal
models, the names of the (parameterized) building blocks as
(parameterized) events, and the branching conditions as
(atomic) propositions. Thus the ABC focusses on the level of
component composition rather then on component construc-
tion: its compatibility, its type correctness, and its behavioral
correctness are under formal methods control [23].

B. Formal methods and tools

The ABC comprises a high-level type checker, two model
checkers, a model synthesizer, a compiler for SLGs, an
interpreter, and a view generator. The model checkers and the
type checker take care of the consistency and compatibility
conditions expressed by the four kinds of constraints/models
mentioned above, and the model synthesizer allows one to
automatically generate executable models from constraint-
based requirements specifications.

The central difference with respect to WSCI is the intro-
duction of a declarative level of choreography: by means of
constraints, we can formulate which properties the composi-
tion or interaction between services should exhibit, and we
can check that the behaviours that we require to cooperate
indeed comply with those requirements. WSCI resorts instead
to a BPEL4WS-like programming-style restriction of the
global behaviour, which is syntactically inadequate (since
it requires programming skills) for the kind of application
designers we address here, and conceptually inadequate,
since it treats the management and control layer essentially
in the same way as the managed or controlled processes.

The capability of dealing with abstract concepts and ab-
stract functionalities via ontologies simplifies dealing with
generalizations in the constraints. The fact that we use doubly
labelled semantic models for interpreting the logics (Kripke
Transition Systems, which label both the states and the
transitions) allows us to naturally deal with data and with
functionalities.

C. Automatic deployment and maintenance support

An automated deployment process, system-level testing
[25], regression testing, version control, and online moni-
Fig. 3. AMDD correctness and Consistency Mechanisms in the ABC

V. AMDD EXAMPLES BASED ON THE ABC

This section presents three practically relevant (Web) applications, ranging from an enhancement of the workflow capabilities of a content management system for non-experts, over our Integrated Test Environment, which enables test experts without deep programming knowledge to create and modify their testing scenarios, to the Online Conference Service, a rather complex distributed, role-based, and personalized Web service.

A. Enhancing Workflows of a Content Management System

In this project, we used the restrictive workflow management functionalities of a commercial content management system (CMS) as a basis for a component library in the ABC, added some more global features, like e.g. a version management functionality, and taxonomically classified the resulting library, see Fig. 4. This directly enabled us to graphically design workflows far beyond the capabilities of the original CMS.

Besides increasing the modelling power and the range of applicability, using the ABC also allowed us to check the consistency of the workflows. A simple but important constraint we checked was that 'a new page will never be published before it is approved'. After a simple translation into temporal logic, this constraint can now automatically be model checked for any of the workflows within a small fraction of a second. It turned out that already this constraint did not hold for quite some standard workflows of the CMS.

Thus, using the model checking feature of the ABC, it was straightforward to enhance the CMS environment to avoid such mistakes once and forever, and to combine the CMS features for an editorial workflow with additional features like version control, automated update cycles, and features for fault tolerance, e.g. for taking care of holidays or sickness during the distribution of labor.

B. ITE: The Integrated Test Environment

A completely different application is the Integrated Testing Environment (ITE) [7], [20], [8] developed in a project with Siemens ICN in Witten (Germany). The core of the ITE is the test coordinator, an independent system that drives the generation, execution, evaluation and management of the system-level tests. In general, it has access to all the involved subsystems and can manage the test execution through a coordination of different, heterogeneous test tools. These test tools, which locally monitor and steer the behavior of the software on the different clients/servers, are technically treated just as additional units under test. The ITE has been successfully applied along real-life examples of IP-based and telecommunication-based solutions: the test of a web-based application (the Online Conference Service described below) and the test of an IP-based telephony scenario (e.g. Siemens’...
testing of the Deutsche Telekom’s Personal Call Manager application [10], which supports among other features the role based, web-based reconfiguration of virtual switches).

In this project we practiced the AMDD approach at two levels:

- the modelling of the test environment itself, and
- the modelling of test cases.

The benefits of the AMDD approach became apparent once a drastic change of the requirements of the test scenario in the telephony application occurred, which meant a new quality of complexity along three dimensions ([20]):

- testing over the internet,
- testing virtual clusters, and
- testing a controlling system in a non-steady state (during reconfiguration).

We could inherit a lot of the conceptual structure of the ‘old’ ITE for the new version of the test environment. Even more striking was the fact that the test cases hardly needed any adaption, except for some specific changes directly related to the functionality changes. Thus a change that Siemens considered to be ‘close to impossible’ became a matter of a few weeks.

C. OCS: The Online Conference Service

The OCS (Online Conference Service), see [18], [15], [19] for a description of the service and of its method of development, is a server-based Java application that customizes a strongly workflow-oriented application built with the ABC. It proactively helps authors, Program Committee chairs, Program Committee members, and reviewers to cooperate efficiently during their collaborative handling of the composition of a conference program. The service provides a timely, transparent, and secure handling of the papers and of the related tasks for submission, review, report and decision management. Several security and confidentiality precautions have been taken, in order to ensure proper handling of privacy and of intellectual property sensitive information. In particular,
- the service can be accessed only by registered users,
- users can freely register only for the role Author,
- the roles Reviewer, PC Member, and PC Chair are sensitive and conferred to users by the administrator only,
- users in sensitive roles are granted well-defined access rights to paper information,
- users in sensitive roles agree to treat all data they access within the service as confidential.

The service has been successfully used for over 50 computer science conferences, including many ETAPS Conferences. This year it has served e.g. the full ETAPS, with 5 instances of the OCS running in parallel.

The Online Conference Service allows fully customizable, role-based business-process definitions, it is tailored for personalized support of each participant in the course of the operations of a virtual Program Committee meeting, and it is customizable and flexibly reconfigurable online at any time for each role, for each conference, and for each user [12].

The AMDD approach drastically simplified the realization and organization of the steady evolution of the OCS, which was guided by the growing demands of the users. It allowed to completely separate the issues of functionality implementation from the workflow (process) modelling (in term of SLG’s), to reuse in particular the constraints for the permission handling. In fact, this property remained even true when developing an Online Journal Service (OJS), which required to change most of the workflows, and to add new functionality.

VI. CONCLUSIONS AND PERSPECTIVES

We have proposed a framework for top-down Web Service interoperation based on an aggressive version of model-driven development (AMDD). The point here is to govern the construction and customization of complex Web Applications at the model level in a framework allowing application experts to directly formulate their desires in a way which can then automatically be deployed. Key characteristics are (1) a simple functional characterization of the components in terms of ontologies, together with a fast mechanism for component discovery, (2) a graphical application-composition environment that works with ontologies and, most importantly, (3) two methods for service choreography and orchestration. The first of these methods allows one to synthesis executable models from goal-oriented, temporal specifications and the second to graphically design operational models of complex workflows. Both methods can be applied statically (at design time of a Web Service) or dynamically for modifying a Web Service at runtime. The latter option only works in an interpreter-based scenario.

Our approach focuses on functionalities as the basic entities of the design space of Web Services. It is tailored to make second order issues like interoperation, distribution, and compatibility simple for the many (by hiding most of the intricate second-order issues form the application designer), and difficult for the few, who have to provide the required methods for compilation, synthesis or technology mapping. Our experience indicates that this approach has the potential to cover and thereby drastically simplify the bulk of
modern Web application development and customization, in particular, as it eliminates communication hurdles (between the domain expert and the developer).

Our current implementation is based on taxonomical descriptions for the components and temporal logic specifications for the goals of the overall application. We are currently investigating the impact of generalizing this setting to a setting using various versions of Description Logics [3]. Obviously, this generalization will lead to more concise specifications, however, in the past, we did not feel the need for the additional expressive power, which has a drastic effect on the decision complexity.

Acknowledgement

We would like to thank Ralf Nagel for his contributions concerning the content management system.

VII. REFERENCES
