Representing Formal Component Models in OSGi*

Marco Müller, Moritz Balz, Michael Goedicke
Specification of Software Systems
Institute of Computer Science and Business Information Systems
University of Duisburg-Essen, Campus Essen, Essen, Germany
{marco.mueller, moritz.balz, michael.goedicke}@s3.uni-due.de

Abstract: Formal component models have been subject to research for decades, but current component frameworks hardly reflect their capabilities with respect to composition, dependency management and interaction modeling. Thus the frameworks don't exploit the benefits of formal component models like understandability and ease of maintenance, which are enabled when software is composed of hierarchical and reusable components that are loosely coupled, self-describing and self-contained. In this contribution, we try to examine the discrepancies between the state of research and the capabilities of an existing module framework, the widely-used OSGi bundle management framework for the Java platform. Based on this we propose modifications and enhancements to the OSGi framework that allow to exploit the benefits of formal component models in OSGi-based applications.

1 Motivation

When software becomes larger and more complex, modularity is needed. The breakdown of large systems into small, independent and manageable pieces promises to let systems be easier to understand, maintain and change. For this reason, component models are subject to research in information technology for almost forty years [Par72]. The component models that have been developed as a result do not only facilitate a structured creation of modular systems, but are in many cases also formally founded [AG97, CS01, CFGGR91, MDEK95, SG94]. The focus of such component models can be classified roughly into six areas: (1) Composition of components; (2) provision of functionality of a component and its appropriate description; (3) management of dependencies and the definition of required components; (4) instantiation of components; (5) modeling of interaction between components; (6) creation of executable systems by connecting component instances at deployment time. A formal foundation for these definitions allows for detailed specification of modules and their interaction, and also enables developers to verify desired characteristics of modular software systems.

However, while the research on this topic is thus very advanced, implementations of the related concepts are hard to find in current programming languages, platforms and frame-

*Part of the work reported herein was funded by the German Federal Ministry of Education and Research under the grant number 03FPB00320.

works. When modern object-oriented programming languages like Java or C# are considered, module definitions are optional at all and only provided by external frameworks. In addition, the features provided by these frameworks lack important parts of the expressiveness formal component models proposed long ago. Since the ability to partition large software systems into manageable pieces is still desirable, we will consider the widely-used OSGi [OSG05] framework, which is based on the Java programming language and platform, in this contribution. OSGi does not focus on components, but manages so-called *bundles* containing program code and libraries as well as their dependencies. On top of this, services can be defined, which provide implementations of interfaces across bundles and enable loose coupling.

We examine how far OSGi is appropriate to create module-based systems with respect to the features defined by formal component models. Then we will propose modifications and enhancements to OSGi, that must be made in order to reach this objective. First we summarize the features of formal component models in section 2. Based on this, we examine OSGi with respect to these features in section 3, leading to a comparison of desirable and existing functionality in OSGi. A proposal of changes to support the objectives of formal component models in OSGi is made in section 4, before we present an overview of related work in section 5 and conclude in section 6.

2 Formal Component Models

Different formal component models exist that focus on various aspects, including component interconnection [AG97], message flow [CS01], data abstraction and concurrency [CFGGR91], dynamic architectures [MDEK95, BHP06], modeling and prediction of non-functional attributes [GMRS08, RBH+07, Inf03]. In this section we will try to summarize the features of established component models to allow for an evaluation of OSGi in section 3. Considering these models, which all have different focuses, we can categorize their features into the following six areas mentioned in the introduction.

2.1 Composition

Composition of components considers the fact that modular software systems may be structured hierarchically. This is for example reflected in the component model proposed by Cox & Song [CS01] that distinguishes between simple and composite components. A simple component consists of *inports* and *outports* being sets of attributes defining provided and required functionality, a *function* describing the actual computations of the component and events that *trigger* the component to act. Composite components have no functions or triggers, but instead consist of *embedded component*s and a set of *connections* defining the interconnection of embedded components and the in- and outports of the composite component with the corresponding counterpart of an embedded component.

Enabling architects to construct composite components is stated to be the primary purpose

of the configuration language Darwin [MDEK95]. Simple components are essentially described by their provided and required services. Composite components contain instances of other components and specify the bindings between them. Internal components and their dependencies are hidden from external components, so that the compositions form a hierarchical abstraction. In general, components can be differentiated by their compositionality. Primitive components directly implement the functionality they provide, while composite components consist of components themselves, thus hiding the context from their enclosed components and vice versa.

2.2 Provided Interfaces

In larger systems, components offer services that can be used by other components. Since the context is not known beforehand and maybe not intended by the component developers, the provided interfaces must be described thoroughly. This is considered by Allen & Garlan who define component interconnection with so-called *connectors* [AG97] providing a formal basis to describe their behavior. In this case components define so-called *ports*, which are essentially state charts with messages, to describe exported behaviour, called *processes*. The messages between those processes contain untyped data.

Similarly, the Π language [CFGGR91] defines the so-called *export* of a component in three views: the *type view* describing exported data types, the *imperative view* describing the exported behaviour in an imperative manner, and the *concurrency view* describing concurrency constraints for the execution of the imperative operations. In summary, the export of a component is commonly described by a set of operations which can be called by the context of the component. However, the definitions vary in the degree of detail, especially regarding data types.

2.3 Dependencies

The provision of services introduces dependencies which must be managed during development and assembly of a component-based software. Since dependencies may be based not only on direct connections, but also on functional requirements that can be satisfied by different components, the requirements must be described precisely. An example for this is Darwin describing required services with an interface name, a communication mechanism and a data type. Data type and communication mechanism are not specific to the language, but interpretation of these is left to the underlying platform. Thus the dependencies are essentially the name of an interface. The implementation of a component can use the required service description without knowing the name of the component bound to this dependency at run time. The components are thus context-independent.

The dependency of a component in Π is called *import* and also defined in three views, similar to the export. The *type view* describes the imported data types with their operations, the *imperative view* describes the imported behaviour in an imperative manner, and the

concurrency view describes constraints for parallel execution of the expressed behaviour. The import is also context-independent, i.e. the implementation of the component can use the import without knowledge about the context of the component at run time.

Palladio [RBH+07], SOFA 2 [BHP06], and ROBOCOP [Inf03] use a different approach: Dependencies are also well-defined interfaces, but the interfaces are first-class entities, which are shared by the consuming and the providing component. Thus the components are not context-independent at design time, because the shared interface must be known.

Since dependencies require certain functionality, components must provide a description of the functionality they expect. Differences exist regarding the degree of details: The data types may be completely defined as in Π or be completely left to a surrounding platform as in Darwin. Additional properties like concurrency constraints are also supported by some models. For context independence, the requirements must be described locally to the component and thus without knowledge of its future context.

2.4 Instantiation

At run time multiple instances of components can exist. For this reason, in SOFA 2 a system is described as a component architecture. The components are instantiated and provided with a unique name for each component. The component instances can then be referenced in the architecture by connections. This expressiveness cannot be taken for granted, as it is for example not available in Cox & Song's model, where components are executed without considering instances. Nevertheless, in this case a component equivalence relation is defined to compare two components with each other. Components are divided into component classes where the members of a component class are syntactically identical and differ only by their identification. While the approaches are different, the need is clear to consider instances at least, if only in the case that singleton instances are equipped with identification mechanisms.

2.5 Interactions

In a system consisting of dependent components, the components must be able to communicate. This dynamic aspect of component interaction is considered by Allen & Garlan in the form that components contain processes that are subject to communication events. The communication events are locally defined in each component and connector specification. These context-independent specifications are glued together by the connector instances mapping the events and data parameters of one role to the counterparts of the second role.

The interfaces in Palladio optionally include a protocol definition, e.g. declared as a finite state machine or a regular expression. The protocol defines the sequence of operation calls, a client may call on a provided interface. In case of a required interface, the protocol describes how the client uses the required interface. Components in Π use a likewise approach for defining a sequence and concurrency constraints for operation calls for re-

quired and provided behaviour and types using so-called *path expressions* over operation names. In summary, since component communication is inevitable for a modular system, the related interaction is subject to specification by the formal component models.

2.6 Assembly

An executable system in the model of Allen & Garlan is constructed by the definition and instantiation of components and connectors as well as the definition of the interconnection between those elements. In these interconnections the ports of the component instances are bound to the roles of the connector instances. Darwin describes the system with a component definition instantiating embedded components and connecting their provided and required services directly. Commonly, a system is constructed by instantiating the desired components and interconnecting their provided services and dependencies using a separate configuration. This assembly must be considered a separate stage in the development process which can be taken only if the components themselves are finished, but before the system is being executed.

3 Components in OSGi

OSGi is a component system for the Java platform. It is widely-used, from embedded systems to server-based enterprise applications, since no adequate functionality is provided by the language or the platform. The basic functionality of OSGi is the management of so-called *bundles* which are technically Java libraries (Java Archive; JAR) containing compiled classes and binary resources. Bundles are configured in the descriptor file of their JAR file (`MANIFEST.MF`) with name-value pairs. The meta data include information about the ID, name, and version of a bundle. It also describes the relations to other bundles, either with a specification of bundle IDs, or with a definition of packages provided by other bundles. At run time, the OSGi framework is responsible for loading the bundles, resolving dependencies, and controlling the access to provided packages.

While this simple structure is an appropriate solution for dependency management and access control, it does not describe provided functionality of bundles in detail and forces a tight integration between bundles at the same time. Addressing these issues, an additional layer has been added to OSGi that supplements the bundle concept: The *Service Layer* is responsible for managing so-called *service components*. They encapsulate the functionality provided by a bundle by offering a named service, which is published in a registry and can be retrieved and accessed by any class running inside the OSGi framework. Their interface is described with a Java interface, thus using the Java semantics for method signatures and data types. Considering both, bundles and services, we will now relate the features of OSGi to that of the formal component models introduced above.

3.1 Composition

OSGi does not support composition of components. Bundles are only connected by their dependencies and must thus be considered simple components. The Service Layer does not add any additional semantics regarding composition since it only describes interfaces of bundles, but does not provide another view on bundles and their dependencies.

3.2 Provided Interfaces

Since OSGi allows to modularize applications, single bundles are likely to provide functionality to other bundles. They *export* packages containing Java types to make them available for use by other bundles. With exported and non-exported packages information hiding can be realized. Services allow for more encapsulation by offering just an interface description being published under a certain name, with the bundle itself instantiating and managing the underlying objects. This means that the description of provided functionality is possible without causing the need to reveal internal functionality of the bundle.

However, the semantics of interfaces and data types in use are simply those of Java. This causes a tight integration into the underlying Java platform. In addition, the bundles themselves are tightly integrated, since all type definitions being exported must be available to all using bundles in a shared bundle. This enforcement of direct dependencies contradicts on the one hand the principles of the formal component models which assume that components can describe their services completely independent, as for example defined in Π. On the other hand, the Java platform and a network of dependent bundles can provide a rich ecosystem of types which are hard to describe from scratch for each provided interface.

3.3 Dependencies

OSGi allows to specify dependencies between bundles in three ways: (1) By referencing the bundle name; in this case, all exported package of the dependency are available in the bundle. (2) By referencing package names; when bundles are available that export these packages, they are accessed. (3) By referencing service component descriptors; the reference is described by the Java interface, the cardinality, and whether the reference is optional. In all cases, the dependencies are resolved at run time when the bundles started, thus causing errors if dependencies cannot be satisfied.

This contradicts the specifications of formal component models since it leads to a tight integration between bundles. As described for the interface provision above, all types in use must be provided by bundles that are shared between all using bundles. When the bundle providing a service also exports the related types, the depending bundle is statically bound to it as illustrated at the left hand of figure 1. Since the service interface must be available for consumers at compile time, the interface has to be copied to implement different providing bundles for one service. A change in the service interface results in

incompatible services so that each consumer must be updated and recompiled.

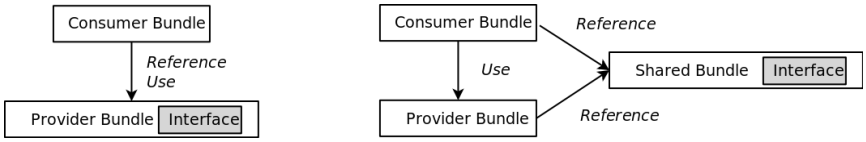

Figure 1: Independence of service components. At the left hand, the interface is packaged with the providing bundle, so that the consumer references the service bundle directly. At the right hand, the bundles are decoupled, allowing services to be provided with the same interface by multiple bundles.

These problems can be circumvented in OSGi by providing the service interface in a separate bundle which is referenced by all providers and consumers as shown at the right hand in figure 1. This has the advantage that consumers can be implemented without knowledge about the actual service provider and thus without a static reference to it. Different services can also be provided under the same interface, since only the shared type definitions must be known before the application is assembled. However, the interface still needs to be defined before the service consumer can be developed and each change on the interface results in the requirement to recompile the affected consumers.

In contrast, formal component models propose to define dependencies by not using shared types, but instead defining the functionality to import locally. This means that components describe all interfaces and related data types completely by themselves and are thus independent from external descriptions. At assembly time, these descriptions are sophisticated enough to to determine if the provided interface of any available component matches the requirements of the depending component. By this means, components are defined completely context-independently, which cannot be realized with OSGi.

3.4 Instantiation

OSGi bundles cannot be instantiated since they are simply collections of classes. Service components are instantiated by their owning bundles and registered at the service registry using a unique name, so that only one instance exists. More than one instance of a service component can only exist when different names for the instances are used, which could be considered confusing. The types that are exported in packages can be instantiated by consumers without restriction and independently from the OSGi framework. In contrast to the instantiation features of formal component models, OSGi does therefore not provide components with identification mechanisms but only with names for the single instances.

3.5 Interactions

As bundles mainly manage visibility of packages, the interaction between bundles is effectively arbitrary communication between Java types and thus not under control of the

framework. In the Service Layer, creation of a reference to service objects is controlled by the framework during lookup. However, the communication itself is not surveyed or intercepted by the OSGi framework. The features provided by some formal component models for specifying constraints, for example with respect to data types, value ranges, or concurrent access for component interaction, like path expressions in Π, are not realized.

3.6 Assembly

A software system consisting of OSGi bundles is assembled at run time. The wiring of services is controlled programmatically or descriptively in service component definitions. However, semantic validation is not possible before the system is started. The reason is that dependencies are described with names and Java types rather than independent interface and interaction descriptions, as are provided by formal component models. Thus required types and services must be available when a bundle is started, otherwise errors occur. A validation before bundles are started is not supported by OSGi since the availability of requirements cannot always be determined completely without activating the bundles.

4 Proposal for Improving of the OSGi Service Layer

We have seen that some OSGi features can be related to concepts of formal component models, especially on the Service Layer. We now propose a set of changes to OSGi in order to make use of formal component specifications. First we change the definition of dependencies and refine the description of provided services to create context-independent components. Second we introduce composite components to the Service Layer to enable component hierarchies. At last we introduce modeling of component interaction with path expressions and type conditions.

4.1 Dependencies

OSGi relies on the semantics of Java interfaces for services and on the Java platform for related data types. There are three ways to define dependencies and provision of data types that are not offered by the platform: First, shared data types can be defined in an own bundle which is accessed by all components participating in the communication. This is essentially extending the platform with the needed data types. Second, data types are defined by Java interfaces by the providing component. Their implementations are instantiated and managed by the provider and only accessed by the consumer.

Third, service interface and data types are completely specified by provider and consumer. The OSGi framework is responsible for resolving dependencies by matching both interfaces and data types at assembly time. At run time, the framework must map any data that is exchanged between both representations. This approach makes bundles and services

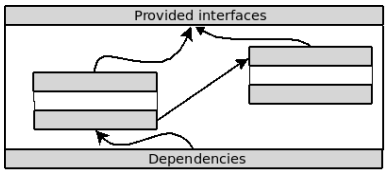

Figure 2: A composite component declaring composite provided resources and dependencies. The embedded components are hidden to the context.

context-independent since they are self-contained and do not rely on external specifications of data types. Tools can be developed that support assembly time validation as well as the run time connection and thus support developing context-independent components.

4.2 Composition

Composite components in formal component models hide subcomponents and cannot be distinguished from simple components from outside. Internally, their functionality relies on dependencies between components as illustrated in figure 2. To enable this behaviour in OSGi, the framework must consider libraries embedded in a bundle as bundles too, i.e. scan for component descriptors and activate bundles and services. Component descriptors of composite bundles are responsible for interconnecting subcomponents, defining composite export and import, and mapping them to exports and imports of subcomponents.

4.3 Interaction

Formal component models like Π and Palladio describe component interaction and take concurrency into account. We propose to add constraints regarding data types, value ranges and sequences of calls to exported and imported services.

Data types described in OSGi are limited to Java semantics. While the static type system is already expressive, it does e.g. not allow to specify constraints to method parameters apart from the built-in data type constraints. If, for example, a parameter of an operation declared in a component's exported behaviour must not be null, this requirement could only be stated in a documentation. To describe value ranges for data types, we propose to consider approaches like the Java Modeling Language [BHS07] and use the related concepts in service component definitions. Data types and behaviour are annotated with pre- and postconditions using XML descriptor files or Java annotations. The platform that controls communication between bundles and services can evaluate whether the conditions are met at run time, which is illustrated in figure 3, and prevent any invalid invocations.

The sequence of calls of exported and imported services can be described statically using Π's path expressions. These could e.g. be used to enforce the construction of a type before

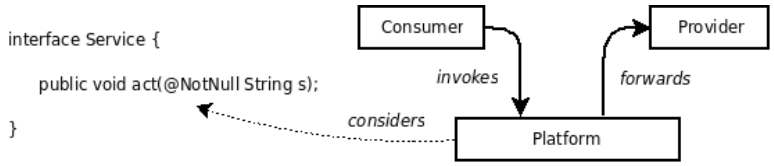

Figure 3: The proposal to use type constraints when the OSGi platform forwards requests between services. For example, meta data in the interface program code can force parameters to be not null, which is validated by the platform during requests.

it is used. Path expressions in this context consist of operation names and operators describing the permitted call sequence. For example, the sequence `create;(*[{read}|write]*);destroy` describes that first the operation *create* must be called. Afterwards optionally and repeatedly either *concurrent* `read` calls or a single *write* call can be executed. At last the operation *destroy* must be called. The framework has to verify that the communication complies with the path expressions and prevent it if the constraints are not met. In the future, one can imagine tools that use static code analysis to verify call sequences already at development time based on these interface descriptions.

4.4 Assembly

When independently-developed components are assembled in order to create an executable system, their requirements regarding dependencies, services and data types must be verified. In OSGi this currently happens at run time when bundles are activated. We propose to create tools that assemble the software before it is started and at that time consider all descriptions of imports, exports, and communication constraints introduced above. This stage would allow for a more thorough verification of the interconnections between components, which is important since the context of component-based systems is not always known by component developers beforehand. Based on the same information, additional tools could monitor the current state and the interactions of the running system afterwards.

5 Related Work

Several approaches aim at providing module concepts inside Java. In the Spring framework so-called *Spring Beans* can be defined including dependencies on other beans. Spring Dynamic Modules (Spring DM) [CHLP08] enables developers to use Spring Beans as OSGi Services and vice versa. Spring DM provides these beans to the system using Java interfaces. However, components in this context do not differ from the OSGi Service Layer. The Java Enterprise Edition allows to write distributed server-side applications which also consist of modules. In this context, modules are called *Session Beans* [Sun08] and are able to define dependencies using Java interfaces and annotations. However, they are tightly coupled to their type definitions, so that context independence is not possi-

ble. Modeling of interfaces and interactions also does not exceed the functionality of the Java language. In the CORBA middleware, the CORBA Component Model [Obj06] uses loosely-coupled components with parameterized events as interaction mechanism. Interfaces and data types are defined independently from implementation languages in an Interface Definition Language providing more detailed features regarding value ranges and parameter constraints than Java. However, this is not well integrated in Java because no formal mapping between the language elements exists and generated helper classes are used for data conversion. Beanome [CH02] manages components in OSGi bundles that are described by XML files. However, this approach uses shared interfaces, rendering the components context-dependent. Beanome was developed for the outdated OSGi 2 and is not maintained anymore.

6 Conclusion

In this contribution we examined a selection of formal component models and a widely-used practically driven component framework in Java, based on the observation that concepts of modularity are widely accepted, but hardly taken to full advantage in current object-oriented programming languages and frameworks. We compared the features of the formal component models with the component framework OSGi. Dependencies between bundles and services in OSGi can be related to component concepts, but do not allow a loose coupling. Composite components, descriptions of exported interfaces apart from Java interfaces, and modeling of interactions are not supported in OSGi at all.

Based on this, we proposed enhancements to OSGi with respect to the description of provided interfaces and services, loose coupling by means of more precise definitions of required interfaces, and application of path expressions to OSGi for describing interactions. In future work we plan to build a prototype of the enhanced OSGi framework, and tools that support the development of single components as well as the assembly of complete software systems from single components using the proposed mechanisms. We also plan to consider more formal component models and more frameworks for comparison, to find out how practically driven frameworks can be used for differently focused architectural views. With an implementation of these concepts we hope to reduce the gap between research and practice by introducing advanced concepts into the widely-accepted OSGi framework and thus supporting the idea of component-based development.

References

[AG97] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[BHP06] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software. The KeY Approach*. Springer-Verlag New York, Inc., 2007.

[CFGGR91] Joachim Cramer, Werner Fey, Michael Goedicke, and Martin Große-Rhode. Towards a Formally Based Component Description Language. In *TAPSOFT '91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2: Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Developmemnt (CCPSD)*, pages 358–378, London, UK, 1991. Springer-Verlag.

[CH02] Humberto Cervantes and Richard S. Hall. Beanome: A Component Model for the OSGi Framework. In *Software Infrastructures for Component-Based Applications on Consumer Devices*, 2002.

[CHLP08] Adrian M Colyer, Hal Hildebrand, Costin Leau, and Andy Piper. Spring Dynamic Modules Reference Guide, 1.2.0, 2008. `http://static.springframework.org/osgi/docs/1.2.0/reference/pdf/spring-dm-reference.pdf`.

[CS01] Philip T. Cox and Baoming Song. A Formal Model for Component-Based Software. In *Proc. IEEE Symposia on Human Centric Computing Languages and Environments*, 2001.

[GMRS08] Vincenzo Grassi, Raffaela Mirandola, Enrico Randazzo, and Antonino Sabetta. KLAPER: An Intermediate Language for Model-Driven Predictive Analysis of Performance and Reliability. In *The Common Component Modeling Example: Comparing Software Component Models*, pages 327–356, Berlin, Heidelberg, 2008. Springer-Verlag.

[Inf03] Information Technology for European Advancement. ROBOCOP: Robust Open Component Based Software Architecture for Configurable Devices Project, Deliverable 1.5 - Revised specification of framework and models, July 2003. http://www.hitech-projects.com/euprojects/robocop/deliverables.htm, visited at 2009-12-02.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. *Lecture Notes in Computer Science*, 989:137–153, 1995.

[Obj06] Object Management Group, Inc. CORBA Component Model Specification, Version 4, April 2006. `http://www.omg.org/cgi-bin/doc?formal/06-04-01`.

[OSG05] OSGi Alliance. *OSGi Service Platform, Core Specification, Release 4, Version 4.1*. IOS Press, Inc., 2005.

[Par72] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[RBH+07] Ralf Reussner, Steffen Becker, Jens Happe, Heiko Koziolek, Klaus Krogmann, and Michael Kuperberg. The Palladio Component Model. Technical report, Chair for Software Design & Quality (SDQ), University of Karlsruhe (TH), Germany, May 2007.

[SG94] Harald Schumann and Michael Goedicke. Component-Oriented Software Development with PI. Technical Report 1/94, Specification of Software Systems, Department of Mathematics and Computer Science, University of Essen, 1994.

[Sun08] Sun Microsystems, Inc. JSR 318: Enterprise JavaBeans™3.1 - Proposed Final Draft, March 2008. `http://jcp.org/en/jsr/detail?id=318`.