Requirements as First-Class Citizens: Integrating Requirements Directly with Implementation Artifacts

Markus Voelter[1], Daniel Ratiu[2], and Federico Tomassetti[3]

[1] independent/itemis, Stuttgart, Germany, voelter@acm.org
[2] fortiss gGmbH, Muenchen, Germany, ratiu@fortiss.org
[3] Politecnico di Torino, Torino, Italy, federico.tomassetti@polito.it

Abstract. Requirements often play second fiddle in software development projects. The tools for managing requirements are only loosely integrated with the tools used for implementing the system. Furthermore, while implementation tools are based on a rich syntax and well-understood semantics (the programing language itself), requirements tools are often only aware of weakly structured text. This leads to accidental complexity in integrating requirements with each other and with implementation artifacts. In this paper we describe an approach based on language engineering technologies that results in integrated development environments where both requirements and the code are treated as first class entities. Parts of requirements can be used directly as the implementation, and they are managed with the same tools that are used for the implementation. The approach is illustrated by an extension of the mbeddr system, a comprehensive IDE for embedded software development, with functionality for managing requirements.

1 Introduction

Collecting, organizing and managing requirements is essential in the embedded software development. Complete configuration management and traceability from the code to requirements is required by safety standards such as IEC61508. Still, it is often a cumbersome activity which either is relentlessly executed with major effort (typically if the process requires it) or it is mainly overlooked, leading to poorly structured and maintained requirements.

One major problem with the traditional ways of collecting and maintaining requirements is the inadequacy of the supporting tools [12]. Requirements are often collected and managed using MS Office documents [9] or tools like Rational DOORS, which basically manage numbered paragraphs of text with very limited additional structure and weak modeling capabilities [3]. The relation between requirements and implementation code or tests is collected in other documents, or expressed with comments in the code. This requires manual synchronization with the actual system implementation [4]. As emphasized in [2], a deeper, more seamless integration of requirements with the other development activities would increase their value to all stakeholders.

The study in [7] identifies element-level tracing as a major challenge. It is not surprising that, when possible, practitioners try to escape the need for systematic

requirements management and tracing, using an "agile" process as an excuse. However, while agile approaches to requirements engineering limit the burden of managing them, they often do not provide a maintenance strategy; instead, requirements are considered transient artifacts. This is not acceptable in many embedded systems projects, where quality standards require a more structured approach to requirements management.

In this paper we present an approach to use language engineering and projectional editors to deeply integrate requirements into software development tools. Requirements are captured using an extensible language specific to the "requirements domain", just as other artifacts are expressed with languages specific to their particular domains (C code, state machines, data flow blocks). We have implemented this approach based on the JetBrains MPS language workbench and the mbeddr stack for implementing embedded software (discussed in the next section). The approach leads to four main benefits:

- *Rich and extensible requirements language:* By using the language modularization and composition features supported by language workbenches, requirements can have a rich structure, domain-specific extensions to (generic) requirements can be defined, and custom traces or consistency checks can be seamlessly integrated. The need to extend requirements engineering tools in order to to properly express requirements was pointed out already almost forty years ago in [1]. Today we can benefit from language workbenches that support extensibility with very limited effort, supporting end user-friendly notation such as tables or mathematical formulas.
- *Deep integration of requirements with other artifacts:* Various ways of integrating implementation artifacts and requirements become technically simple. For example, traces to requirements can be attached to arbitrary elements in arbitrary implementation artifacts.
- *Ability to query the requirements model:* Relationships between requirements and artifacts can be queried to find out, for example, which requirements are related to failing tests or are not connected to implementing artifacts.
- *Reduction of friction loss between tools:* Since requirements are first class citizens in the development tool, the same tooling is reused both for editing and managing of requirements and implementation artifacts. This includes version control and diff/merge.

2 mbeddr and MPS at a Glance

mbeddr[4] is an open source project supporting embedded software development based on incremental, modular domain-specific extension of C. It also supports other languages, which is what we exploit in this paper. Fig. 1 shows an overview, details are discussed in [10] and [11]. mbeddr builds on the JetBrains MPS language workbench[5], a tool that supports the definition, composition and integrated use of general purpose or domain-specific languages. MPS uses a projectional editor, which means that, although a syntax may look textual, it is not

[4] `http://mbeddr.com`
[5] `http://jetbrains.com/mps`

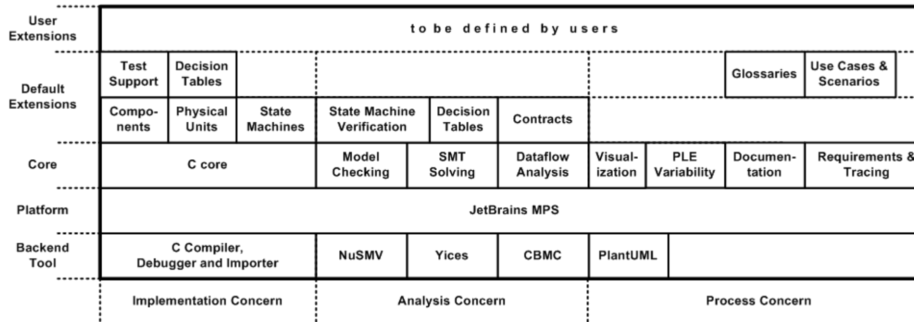

Fig. 1. The mbeddr stack rests on the MPS language workbench. The first language layer contains an extensible version of C plus special support for logging/error reporting and build system integration. On top of that, mbeddr introduces default C extensions.

represented as a sequence of characters which are transformed into an abstract syntax tree (AST) by a parser. Instead, a user's editing actions lead *directly* to changes in the AST. Projection rules render a concrete syntax *from* the AST. Consequently, MPS supports non-textual notations such as tables or mathematical symbols, and it also supports wide-ranging language composition and extension – no parser ambiguities can ever occur when combining languages.

mbeddr comes with an extensible implementation of the C99 programming language. On top of that, mbeddr ships with a library of reusable extensions relevant to embedded software. As a user writes a program, he can import language extensions from the library and use them in his program. The main extensions include test cases, interfaces and components, state machines, decision tables and data types with physical units. For many of these extensions, mbeddr provides an integration with static verification tools [8]. mbeddr also supports several important aspects of the software engineering process: documentation, requirements and product line variability. These are implemented in a generic way to make them reusable with any mbeddr-based language (we discuss aspects of the requirements support in detail in the remainder of this paper). Finally, users can build extensions to any of the existing languages or integrate additional DSLs.

3 Challenges and Solutions

In this section we describe a set of challenges in requirements engineering as well as our approach to solving them in mbeddr. While not all of the solutions are unique to mbeddr, the combination of features and their flexible extensibility is.

3.1 Requirements Versioned with Code

Challenge Traditionally, requirements are stored in a tool-specific database. Other development artifacts are instead typically stored in version control systems (VCS) such as git, SVN or ClearCase. This leads to problems when trying to keep requirements in sync with the implementation.

Solution In mbeddr, requirements are treated like any other artifact and stored in MPS models (represented as XML files on the files system). This

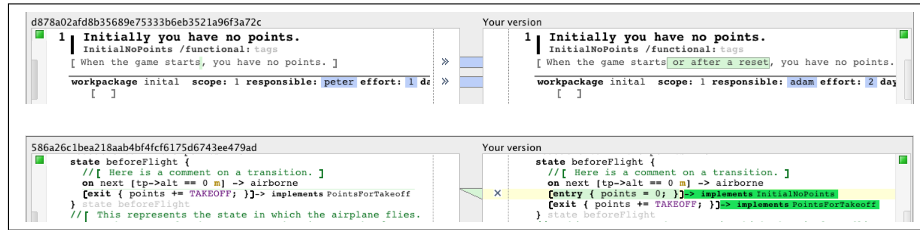

Fig. 2. Top: Code and requirements are versioned in the same way, with the same tools. Since the requirements have a rich structure, the diff shows the requirements model elements that were changed, namely, the text, the responsible and effort. **Bottom:** Example code implementing this requirement, plus a corresponding trace and the diff.

way, requirements and implementation artifacts are stored in the same VCS. Diff/merge is supported for requirements in the same way as for any other artifact (see Fig. 2). mbeddr is not the only tool that uses this approach: Requality[6] and Yakindu[7] use this approach as well.

To represent requirements, mbeddr provides a special language. Each requirement has an ID, a short description, an optional longer prose and additional attributes. Requirements can also be nested. An example is shown in Fig. 4.

3.2 Traceability into Code

Challenge Tracing can be used to express the relationship between implementation artifacts and requirements: a program element has a pointer to one or more requirements, expressing that this particular element is somehow related to a requirement. By using different trace kinds, the nature of "somehow related" can be qualified. Trace kinds typically include `implements` or `tests`. Tracing is supported by several requirements tools. For example, Reqtify[8] supports tracing into SCADE models and Yakindu supports tracing into various Eclipse-based artifacts, requiring specific tool adapters for each of them. Consistent element-level tracing becomes a challenge, however, when working with different implementation languages, techniques and tools. The empirical study in [7] identifies this as one of the major challenges in today's use of model-driven engineering tools.

Solution In the context of mbeddr, a *program* is anything expressed with any MPS-based (programming or modeling) language. This includes C and all of mbeddr's C extensions. Fig. 3 shows a piece of mbeddr program code. The root element is a `module`, and it has an annotation that specifies to which requirements modules we may want to trace from within that module. We can then add a trace to any program element in that module (expressed in C and the state machines extension), tracing to any requirement in the referenced requirements module. There are four important characteristics of this implementation:

[6] `http://requality.ru`
[7] `http://www.yakindu.de/requirements/`
[8] `http://www.3ds.com/products-services/catia/portfolio/geensoft/reqtify/`

```
requirements modules: FlightJudgementRules
module StateMachines imports DataStructures, stdlib_stub, stdio_stub {

  [#define TAKEOFF = 100;]-> implements PointsForTakeoff
  [#define HIGH_SPEED = 10;]-> implements FasterThan100
  [#define VERY_HIGH_SPEED = 20;]-> implements FasterThan200

  statemachine FlightAnalyzer initial = beforeFlight {
    in next(Trackpoint* tp) <no binding>
    in reset() <no binding>
```

Fig. 3. A C module with a set of constants, each with a trace to a single requirement. Traces can be added to any program element expressed in any language.

```
3 | Points you get for each trackpoint
  | InFlightPoints /functional: tags
  Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent feugiat enim arcu, ut egestas
  velit. Suspendisse potenti. Etiam risus ante, bibendum ut mattis eget, convallis sit amet nunc.
  It uses §req(PointsFactor) to calculate the total points.
  ------------------------------------------------------------------------------------------------
  calculation PointForATrackpoint (int8): Points for each Trackpoints
      params uint16 alt: current altitude of the trackpoint
             int16 speed: current speed of the trackpoint
    = BASEPOINTS * 0 |             | alt > 2000 | alt > 1000 |
                     | speed > 180 | 30         | 15         |
                     | speed > 130 | 10         | 20         |
    tests: PointForATrackpoint(500, 100) == 0
           PointForATrackpoint(500, 1200) == 0
           PointForATrackpoint(1100, 140) == 200
```

Fig. 4. A calculation is a function embedded into a requirement. They include test cases that allow "business people" to play with the calculations. An interpreter evaluates tests directly in the IDE for quick turnaround.

1. The trace is not an independent program element that is just "geographically close" to the program element it traces. Instead, the trace is a child of the traced element. This means that, if you move, copy, cut or paste the element, the trace moves with it.
2. Since MPS is a projectional editor, the program can also be shown *without* the traces, if the user so desires.
3. The requirements trace is a well-typed program element that can be used in analyses. For example, it is possible to find all program elements that have traces to a particular requirement using MPS' generic Find Usages facility.
4. The tracing facility is *completely independent* of the traced language. Program elements defined in any (MPS-based) language can be traced. User-defined languages with automatically work with the tracing mechanism.

Our tracing framework still requires users to manually establish and maintain the traces according to the actual relationship between programs and requirements (except in cases where artifacts are generated from higher-level artifacts, in which case traces can be added automatically). However, the approach does solve the technical challenges associated with ubiquitous tracing support. In particular, the fact that referential integrity is automatically checked and that arbitrary analyses can be built on top of the program/requirement/trace data, eases the work of the developer.

3.3 Formal Business Logic in Requirements

Challenge In today's practice, there is a big gap between requirements and implementation artifacts: requirements are prose text, and developers are expected to understand the text and write implementation code that is faithful to the requirements. This is inefficient, tedious and error prone. In many cases, aspects of a system's requirements are known to domain experts and requirements engineers in a structured/formal way. Examples include price calculations, legal rules or control algorithms. It would be useful a tool were available that is able to incorporate such structured/formal aspects into requirements – with IDE support for the languages used to express these structured/formal aspects. With today's tools, this is hard to achieve, and the study in [7] identifies this as another major challenge in today's practice.

Solution mbeddr supports embedding structured/formal parts of the business logic into requirements, and then use these parts directly in the implementation. Fig. 4 shows two requirements. The first one defines a constant `BASE_POINTS` with the type `int8` and the value `10`. The second requirement defines a calculation `PointsForATrackpoint`. A calculation has a name, parameters, and a result expression, which, in this case, uses a decision table (a representation of nested `if`-statements). The calculation also references the `BASE_POINTS` constant. Using constants and calculations, business users can formally specify some important business data and rules, while not having to deal with the actual implementation of the overall system. To help with getting these data and rules correct, calculations also include test cases. These are evaluated directly in the IDE, using an interpreter: users can directly "play" with the calculations.

To increase the usefulness of the constants and calculations specified by business users in requirements, these calculations should make their way into the code directly. Fig. 5 shows a component, expressed in mbeddr's component extension to C that invokes a calculation (the green code). When this code is translated to C, the expression in the calculation is translated into C and inlined.

The constants and the calculations are just examples of possible "plug in" languages into mbeddr's requirements system. Any DSL, using a wide range of business user-friendly notations, can be plugged into requirements.

```
exported component Judge2 extends nothing {
  provides FlightJudger judger
  int16 points = 0;
  void judger_reset() ← op judger.reset {
    points = 0;
  } runnable judger_reset
  void judger_addTrackpoint(Trackpoint* tp) ← op judger.addTrackpoint {
    points += PointForATrackpoint(stripunit[tp->alt], stripunit[tp->speed]);
  } runnable judger_addTrackpoint
  int16 judger_getResult() ← op judger.getResult {
    return points;
} } runnable judger_getResult
```

Fig. 5. Implementation code can directly call calculation functions defined in requirements. In this case, a calculation is called from a component, expressed in the mbeddr's components C extension.

3.4 Verification and Validation

Challenge For each functional requirement there should be a suite of tests to verify it. The functional coverage represents the degree to which all functional requirements are covered through verification activities. Making sure that all requirements are verified in the code is usually a manual process.

Solution The solution to this challenge combines language extensibility and tracing. Language extensions that express test cases and domain-specific formal verifications are available, and traces (with suitable `kinds`) are used to tie tests and verifications back to the tested and verified requirements (Fig. 6, Fig. 7). The deep integration of requirements and code also helps to identify inconsistencies or underspecifications in requirements. This way, requirements can be implemented, tested and verified iteratively, checked automatically and continuously.

```
exported test case testFlightAnalyzer {
  FlightAnalyzer f;
  sminit(f);
 [assert(0) smIsInState(f, beforeFlight);] -> test InitialNoPoints
 [assert(1) f.points == 0;              ]

 [smtrigger(f, next(makeTP(0, 20)));                        ] -> test PointsForTakeoff
 [assert(2) smIsInState(f, airborne) && f.points == 100;    ]
} testFlightAnalyzer(test case)
```

Fig. 6. mbeddr supports the definition of test-cases as first class constructs. Entire test cases, or parts thereof, can be traced to the requirements they cover.

```
int16 step = 0;
while ( step < 5 ) {
  harness module: {
    assign var tp nondeterministicaly
    nondeterministic_choice: {
      choice: smtrigger(flightAnalyzer, reset);
      choice: smtrigger(flightAnalyzer, next(&tp));
    }
    [until smIsInState(flightAnalyzer, airborne)] -> verify InitialNoPoints
    [  must flightAnalyzer.points == 0          ]
    [after smIsInState(flightAnalyzer, airborne)] -> verify PointsForTakeoff
    [  then flightAnalyzer.points >= 100        ]
  }
} while
```

Fig. 7. Similar to test cases, mbeddr supports the definition of verifications that model the environment of the system and contain verification conditions. In this example, the verification considers five steps, at each step the value of the `event` parameter is defined nondeterministically and the event sent to the state machine is chosen nondeterministically. The two verification conditions are linked to the corresponding requirements.

3.5 Requirements Assessment

Challenge To answer questions like "how many requirements were implemented", "what are the quality assurance tasks for a certain requirement", or "how much effort has been spent on each requirement" one needs today to use various different tools, because answering these questions often requires access to the requirements, the traces as well as the implementation artifacts. This makes answering these questions expensive, preventing team members from getting an overview over the state of the project.

Solution mbeddr's assessments provide a flexible way of answering questions like the ones mentioned above. An assessment is a query over the integrated model. The results are presented in a query-specific way. Users can navigate from the result entries to various linked elements. A color code highlights results that have been added in the most recent update. Fig. 8 shows an example.

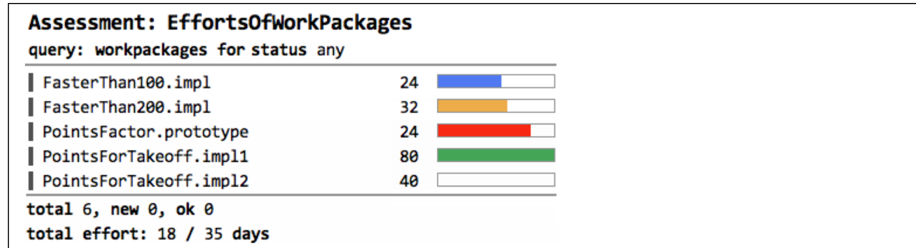

Fig. 8. An assessment that shows information about work packages. Each row in the result shows a work package, the total amount of hours planned for it, as well as the implementation progress: the bar shows percent completed, and the color shows whether the effort is below what has been planned, whether it is on a bad trend ($\%_{effort} > \%_{completion}$) or whether the effort is above the budget.

4 Related Work

As mentioned in the introduction, Winkler and von Pilgrim [12] performed a literature review on traceability. They conclude that tracing is rarely used in practice and the most prominent problem leading to this is the lack of proper tool support. Our approach provides a possible solution to this dilemma and could therefore contribute to helping practitioners in adopting requirements traceability, particularly, in contexts where the process requires it.

In [7], Kuhn et al. present an empirical study on the problems that currently plague model-driven development in the industry. They have interviewed a number of developers from General Motors who use mainstream model-driven development tools. Traceability between implementation artifacts (model, code) and requirements is very important to all engineers interviewed in the study. Their current tool-chain only provides document-to-document traceability, which is not granular enough. Traceability is required on the level of model or program elements, for any level of abstraction. mbeddr's approach supports this issue. The study also found that diffing between various versions of the same model is insufficiently supported; mbeddr solves this problem as well. Finally, the study finds that developers miss the ability to use problem-appropriate abstractions and notations when describing systems. mbeddr's extensibility solves this problem as well – for requirements, and for any other language in mbeddr.

Favaro et al. [5] present an approach to requirements engineering that has some commonalities with ours. Like us, they have the goal to introduce structured, model-based requirements. Their approach relies on the use of a wiki enriched by semantic links supporting navigation from the requirement to the artifact (but not vice versa). They emphasize two points with which we strongly

agree: a) the importance of having an adaptable mechanism for requirements, depending not only on the nature of the project but also on the kind of the requirement, with a lighter process for ”non-technical” requirements; b) the fact that requirements and implementation artifacts are intrinsically integrated. mbeddr's approach provides tight integration between requirements and artifacts, as well as the possibility to have both a flexible approach and specific IDE support for any particular kind of formal language embedded into the requirements.

5 Discussion, Conclusions and Future Work

Discussion The tooling described in this paper solves some important challenges in requirements engineering. However, it is assumed that all artifacts reside in MPS, which limits the applicability of the approach (an import/export facility for requirements is provided, though). On the other hand, mbeddr demonstrates the benefits of building tool suites on top of a language workbench like MPS. In our opinion, this trade-off is worthwhile.

mbeddr allows us to scale the level of sophistication of the language used to express requirements to the needs of the project. This way, small projects can use a basic version of the language to express requirements in a very lightweight form. For more demanding contexts, for example the development of complex embedded systems, the tooling permits to plug-in domain-specific extensions of the requirements language. The efforts of extending the requirements language is limited (language extension efforts are discussed in [10]). Also, we have built a set of extensions of the requirements language specific to a concrete development project on the fly, as part of the project. The efforts were a few hours.

Since the mbeddr requirements tool looks like a ”programming IDE”, it may appear to appeal mostly to programmers. However, in a recent project, the product manager has expressed that he is very happy using it. Also, we are currently in the process of introducing a similar tool to business users in the insurance domain. In both cases, the ability to use domain-specific abstractions and notations such as tables or formulas were cited as a major advantage.

Conclusion mbeddr's core idea is discussed in [10]: building domain-specific tools is not just about adapting a *tool* to a particular domain (e.g. automotive, medical devices, user interfaces). It is rather more important to adapt to the domain the languages, formalisms and data formats that underlie the tool. If this is done based on a language workbench, the necessary efforts are limited, and you get the *tool* adaptation essentially for free. This is because the actual tool, JetBrains MPS, is essentially a very powerful editor for any kind of language. In this paper, we have demonstrated this idea for requirements management. All the benefits discussed in this paper involve only *language* engineering. No *tool* aspects have been customized. End user feedback for this approach is very encouraging.

Future Work There are two main areas for future work. First, we will add reporting functionality, targeting requirements documents in HTML and Latex. The reports will include the diagrams, as well as trace reports. Second, a

colleague of ours is currently working on an MPS editor component that supports mixing free text (with text-like editing support) and instances of language concepts. Integrating this editor with the requirements management tooling discussed in this paper will be extremely useful: for example, one could reference other requirements from within the prose description of a particular requirement, while making sure that this reference would take part in refactorings. In addition we may experiment with integrating information retrieval techniques to suggest possible traces between requirements and development artifacts, as suggested by Hayes et al. in [6]. This would reduce the effort of establishing and maintaining the traces.

Acknowledgements We thank the mbeddr and MPS teams for creating a powerful platform that can accommodate the things described in this paper.

References

1. T. Bell, D. Bixler, and M. Dyer. An extendable approach to computer-aided software requirements engineering. *Software Engineering, IEEE Transactions on*, SE-3(1):49–60, 1977.
2. B. H. C. Cheng and J. M. Atlee. Research directions in requirements engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 285–303, Washington, DC, USA, 2007. IEEE Computer Society.
3. J. M. C. de Gea, J. Nicols, J. L. F. Alemn, A. Toval, C. Ebert, and A. Vizcano. Requirements engineering tools: Capabilities, survey and assessment. *Information and Software Technology*, 2012.
4. B. Draxler. Bidirectional tracing of requirements in embedded software development. Technical report, University of Satzburg, 2006.
5. J. Favaro, H.-P. de Koning, R. Schreiner, and X. Olive. Next generation requirements engineering. In *Proc. 22nd Annual INCOSE International Symposium, 2012.*
6. J. Hayes, A. Dekhtyar, and J. Osborne. Improving requirements tracing via information retrieval. In *Requirements Engineering Conference, 2003. Proceedings. 11th IEEE International*, pages 138–147, 2003.
7. A. Kuhn, G. Murphy, and C. Thompson. An exploratory study of forces and frictions affecting large-scale model-driven development. In R. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*. Springer, 2012.
8. D. Ratiu, M. Voelter, B. Schaetz, and B. Kolb. Language Engineering as Enabler for Incrementally Defined Formal Analyses. In *FORMSERA'12*, 2012.
9. A. Talbot. An investigation into requirements engineering current practice and capability in small and medium software development enterprises. Master's thesis, Auckland University of Technology, 2011.
10. M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz. mbeddr: Instantiating a language workbench in the embedded software domain. *Journal of Automated Software Engineering*, 2013.
11. M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems. In *Proc. of the 3rd conf. on Systems, programming, and applications: software for humanity*, SPLASH '12.
12. S. Winkler and J. Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling*, 9:529–565, 2010.