How to Implement a Theory of Correctness in the Area of Business Processes and Services

Niels Lohmann and Karsten Wolf

Universität Rostock, Institut für Informatik, 18051 Rostock, Germany
{niels.lohmann, karsten.wolf}@uni-rostock.de

Abstract. During the previous years, we presented several results concerned with various issues related to the correctness of models for business processes and services (i. e., interorganizational business processes). For most of the results, we presented tools and experimental evidence for the computational capabilities of our approaches. Over the time, the implementations grew to a consistent and interoperable family of tools, which we call SERVICE-TECHNOLOGY.ORG.
This paper aims at presenting this tool family SERVICE-TECHNOLOGY.ORG as a whole. We briefly sketch the underlying formalisms and covered problem settings and describe the functionality of the participating tools. Furthermore, we discuss several lessons that we learned from the development and use of this tool family. We believe that the lessons are interesting for other academic tool development.

1 Introduction

It is a phenomenon which is common to several technologies based on formal methods that they suffer from a devastating worst-case complexity, but perform surprisingly well on real-world instances. Most prominent examples are *model checking* [10] and *SAT checking* [48]. Consequently, any approach based on such technologies needs to be complemented with a prototypical implementation, which provides evidence for the difference between theoretical complexity and actual observed run time. At the same time, such implementations are an important milestone in the technology transfer between academia and industry as they can be tried out and evaluated in realistic case studies such as [42,19].

In this paper, we aim at introducing and discussing a family of tools, SERVICE-TECHNOLOGY.ORG, which has been developed over the previous couple of years and is, in different aspects, related to correctness of business processes and such services which internally run nontrivial processes as well. Whereas various members of the family have already been used to provide experimental data in several articles, for instance [26,36,34,31,60,63,19,30,39], this is the first occasion to introduce and discuss the common principles and lessons learned from SERVICE-TECHNOLOGY.ORG as a whole.

Central artifacts in the tool family are formal models for services and business processes (based on Petri nets and automata-based formalisms) and formal models for *sets of* services (based on annotated automata). The member tools of SERVICE-TECHNOLOGY.ORG[1] are concerned with importing and exporting models from and to more established

[1] Available for download at `http://service-technology.org/tools`.

languages, with analyzing artifacts, and with synthesizing artifacts. The distinguishing feature of the tool family is that each basic functionality is provided by its own command-line based tool. Additional functionality can be derived by combining tools by connecting their input and output data streams. This way, our tool family is similar to the family of basic UNIX tools such as grep, awk, sed, and the like.

Many tools in the tool family implement algorithms that involve exponential worst-case complexity and complex data structures. To this end, *an efficient implementation is the key factor to apply these algorithms to realistic case studies.*

In Sect. 2, we discuss related tools and approaches which as well aim at demonstrating practical applicability of formal methods. Then, in Sect. 3, we give a walk-through introducing the current members of the tool family and the recurring concepts. In the remaining sections, we discuss several observations that we made concerning the integration of the tool family into its environment (Sect. 4), the formal models used (Sect. 5), and the chosen architecture (Sect. 6). Section 7 summarizes the lessons we learned during the development which may as well be applicable to other domains.

2 Related work and tools for process correctness

Errors in business process models obstruct correct simulation, code generation, and execution of these models. Consequently, business process verification techniques received much attention from industry and academia which is reflected by a constant and large number of tools and research papers on this topic. Interestingly, general-purpose model checking tools are hardly used. This could be explained, on the one hand, by the lacking tool support and efficiency in the early days of BPM research. On the other hand, the properties a business process needs to satisfy are typically much simpler than what temporal logics such as CTL or LTL offer. For over one decade, *soundness* [1] is now the established correctness notion. Soundness ensures proper termination (i. e., the absence of deadlocks and livelocks in the control flow) while excluding dead code (i. e., tasks that can never be executed).

Soundness can be naturally expressed in terms of simple Petri net properties, and tools such as *Woflan* [58] exploit efficient Petri net algorithms to verify soundness. Because of a close relationship [37] of many business process modeling languages to Petri nets, soundness analysis techniques are effortlessly applicable to other formalisms. For instance, Woflan is embedded as plugin into the *ProM framework* [2], and with *WofYAWL* [57], there also exists an extension to verify YAWL [3] models.

A common assumption for business process is that they can be modeled by *workflow nets* [1]; that is, models with a single distinguished initial and final state, respectively. This structural restriction is heavily exploited by state-of-the-art tools such as Woflan and may help to avoid expensive state space exploration whenever possible. The *SESE decomposition approach* [56] which is used by the *IBM Websphere Business Modeler* to decompositionally check soundness further assumes the free-choice property [15]. Similarly, the *ADEPT framework* [12] employs a block-structured language and offers efficient algorithms to ensure soundness in adaptive systems [49].

Structural techniques such as Woflan's heuristics or the SESE decomposition are state of the art as a recent case study [19] reports. However, not all features of current languages, such as BPMN or BPEL, can be expressed in terms of the mentioned structural

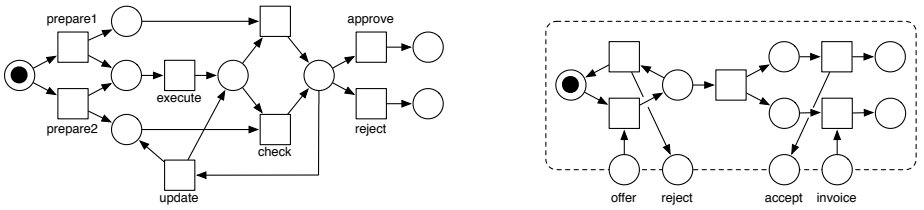

Fig. 1. Petri net models for processes (left) and services (right).

restrictions. At the same time, the case study [19] reveals that state space verification became likewise efficient. Consequently, only state space verification tools offer the necessary flexibility to verify business processes with complex structures.

3 Overview of the tool family

The tools available in SERVICE-TECHNOLOGY.ORG are all concerned with control flow models of business processes or services. These models are typically given as Petri nets. The only distinction between processes and services is the absence (or presence) of distinguished interface places and transitions; see Fig. 1 for simple examples. We use a simple file format for Petri nets and have compilers available which manage the import and export into standard formats such as PNML [8].

Most tools in our tool family read or write such Petri net models; Figure 2 provides an overview of the whole tool family. To avoid repetition of programming efforts, we created a Petri net API which provides an internal representation of all basic Petri net elements and provides standard access methods (e. g., iterating on the set of nodes).

One way of creating a Petri net model to use a plain text editor or to import it from modeling tools that offer PNML export such as *Oryx* [14] or *Yasper* [25]. For obtaining realistic Petri net models, we use two different approaches. First, we have several compilers which translate specifications of practically relevant languages into Petri net models:

- *BPEL2oWFN* [32] for translating WS-BPEL [4] specifications into Petri nets;
- *UML2oWFN* [18] for translating business process models from the IBM Websphere Business Modeler into Petri nets.

From UML2oWFN, we obtain a business process model. For BPEL2oWFN, the user may chose whether to create a business process model from a single WS-BPEL process specification (its internal control flow) or the control flow of a service collaboration specified in BPEL4Chor [13,34], or whether to obtain a service model from a WS-BPEL process specification.

More models can be obtained using distinguished SERVICE-TECHNOLOGY.ORG synthesis tools (see below). These tools typically produce automata-like models for the control flow of a business process or a service. Such a model can, however, be translated into a Petri net model by either using a brute-force translation where each state of the automaton is translated into a separate place of the Petri net, or by using region theory [5] which leads to much more compact Petri net models. Both methods are available in our Petri net API.

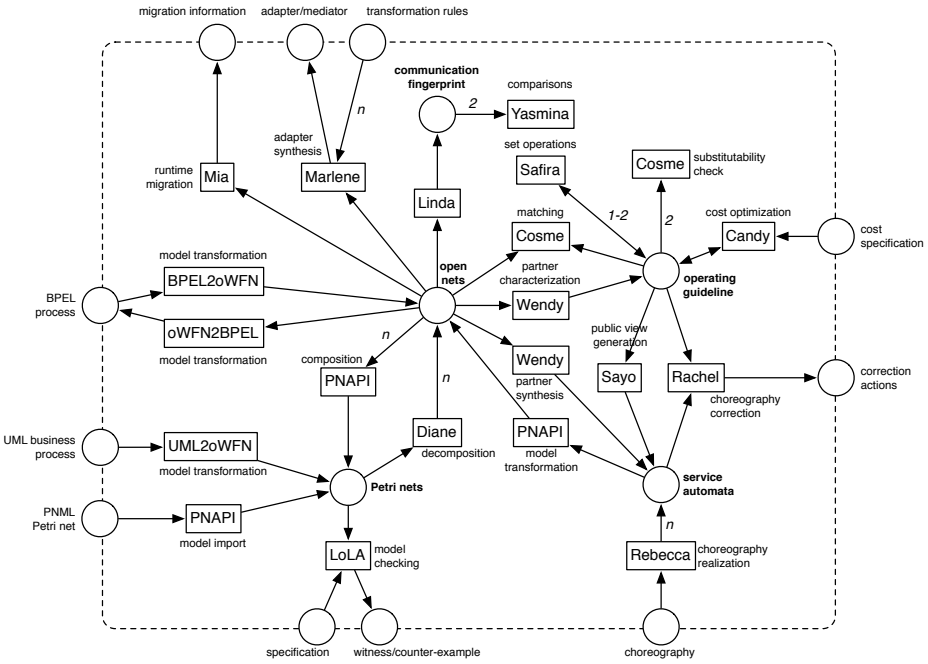

Fig. 2. Illustration of the interdependencies of the SERVICE-TECHNOLOGY.ORG tools.

For the latter method, the API calls the external tools *Petrify* [11] or *Genet* [9] which, consequently, interoperate seamlessly with our tool family.

Our tool family can as well be plugged into existing tools and frameworks. We would like to mention ongoing integration efforts into the ProM framework [2], Oryx [14], and the *YAWL editor* [3]. In these cases, we chose to build plugins to these tools which translate the native modeling language into a representation of our Petri net API. It is then close to trivial to incorporate any subset of the tools mentioned next.

A Petri net model of a business process can be verified. Within the tool family, the Petri net-based model checker *LoLA* [61] is available which has proven to be powerful enough for the investigation of business process models [26,19] or service collaborations [34]. LoLA verifies properties by explicitly investigating the state space of a Petri net using several of state-of-the-art state space reduction techniques.

For a model of a single service, we can not only verify its internal control flow. With the concept of *controllability* ("does the service have at least one correctly interacting partner?") [62], verification covers the interaction of the service with its environment. We decide the controllability problem by trying to construct a canonical ("most permissive") partner. This is one of the tasks of our tool *Wendy* [39]. For constructing the partner service, Wendy explores the state space of the given service. The distinguishing feature of Wendy with respect to a prior implementation is that it employs the state space exploration power of LoLA rather than traversing the state space using its own implementation. Surprisingly, the gain of using the sophisticated data structures and algorithms in LoLA outweighs the loss caused by transferring Petri net models and ASCII representations of computed state spaces between LoLA and Wendy.

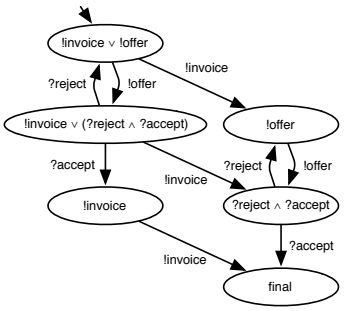

Fig. 3. Operating guideline [36].

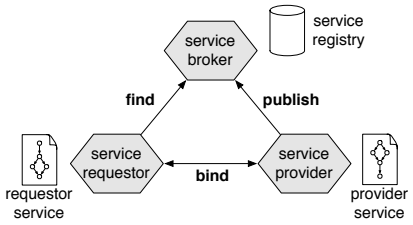

Fig. 4. The SOA triangle [23].

A partner as constructed by Wendy is useful for a several applications. It can be used as a communication skeleton for actually programming such a partner. To this end, our compiler *oWFN2BPEL* [33] is quite useful which maps a Petri net to an abstract WS-BPEL process thus closing the loop between our formal realm and reality.

A related application for the partner created by Wendy is the synthesis of an *adapter* [16] service that mediates between otherwise incorrectly interacting services. This is the task of the tool *Marlene*. Marlene reads service models and a specification of permitted adapter activities (which messages can be created, transformed, deleted by the adapter?) and produces an adapter. As the adapter can be roughly seen as a correctly interacting partner to the given services, it is no surprise that large parts of the synthesis task are left to a call of Wendy.

Instead of inserting an additional component (the adapter) into the service collaboration, it is also possible to replace one of the participating services. This task is supported by the tool *Rachel* [31]. Given a service collaboration which does not interact correctly, Rachel can replace one of its participants (selected by the user) by another service which interacts as similar as possible but establishes correct interaction.

The second central artifact in the SERVICE-TECHNOLOGY.ORG tool family is a finite representation of possibly infinite *sets* of service models. Syntactically, we use *annotated automata* [36]. An annotated automaton A represents the set of all those service models which can establish a simulation relation [44] with the automaton A such that the constraints represented in the annotations of A get satisfied. The most prominent example for a useful set of services is the set of all correctly interacting partners of a given service that we called *operating guidelines* in several publications [35,36,52]. Figure 3 shows an example of such an operating guideline.

Consequently, the transformation of a service model into one or all correctly interacting partner services is a core element of our theory. This is a nontrivial task as obvious ideas like just copying the control flow and reversing the direction of message transfer do not work in general. Our solution [36] annotates the most permissive partner. The corresponding implementation is thus done within Wendy.

In several settings, it is also desirable to replace a service with a new one such that the new one interacts correctly in any collaboration where the old one was participating (be it for technical update or changes in the context like new legal requirements). In this case, the tool *Cosme* can verify the underlying *substitutability* property [52]. Substitutability

means that every correct partner of the old service interacts correctly with the new one. The key to deciding this inclusion on (typically infinite) sets of services is their finite representation as operating guidelines.

For collaborations which are already running, *Mia* [30] suggests transitions from states of the old service to states of the new service such that the old service can be migrated to the new service and the latter can take over the collaboration at run time.

One of the core concepts in the way services are expected to collaborate is the *service-oriented architecture* (SOA) [23] with its eye-catching SOA triangle (see Fig. 4). Some of the SERVICE-TECHNOLOGY.ORG members aim at supporting the procedures visualized in the triangle. The tool *Linda* turns a service model S into what we call a *communication fingerprint* [54]. This is an abstract pattern covering the correct interactions that S is able to produce. In the current version Linda provides lower and upper bounds for the occurrence of messages in correctly terminating runs. The idea of fingerprints is that it is much easier to select a correctly interacting partner from a repository if most entries in the repository can be ruled out by just finding incompatibilities in the communication fingerprint than by performing model checking on the complete service models. The task of comparing communication fingerprints is implemented in the tool *Yasmina*.

Another opportunity for organizing a service repository is to store operating guidelines of services instead of the services themselves. In this case, a SOA requester needs to be compared (matched) with the operating guidelines of registered services. Whereas the generation of operating guidelines is done by Wendy, it is Cosme to perform the actual matching. With the tool *Candy*, nonfunctional properties such as costs can be added to service models to refine the analysis.

In [28], we argued that it would be beneficial to aggregate operating guidelines for speeding up the selection of registered services. At the same time, more abstract *find* requests can be translated into annotated automata; that is, the formalism used for representing operating guidelines. Basic operations to be performed in a registry then boil down to basic set operations (union, intersection, complement, membership test, emptiness test). Our tool *Safira* [27] implements many of these operations by manipulating annotated automata.

If, for some reason, operating guidelines are not suitable for some purpose, our tool *Sayo* transforms the operating guideline of a given service S into some service model S' — a *public view* of S — which is behaviorally equivalent to S. We believe that this procedure could be used as an obfuscator for S; that is, for publishing all relevant information about S without disclosing its control flow. Unfortunately, we still lack some information-theoretic foundation of this obfuscation property.

Beyond SOA, *choreographies* are a promising technology for creating service collaborations. For choreographies, *realizability* [21] is a fundamental correctness property comparable to soundness of workflows. We observed that the realizability problem is conceptually very close to a certain variant of the controllability problem for services [40]. As a result, the tool *Rebecca* which implements the algorithm to check realizability for a choreography and to synthesize realizing services can also be used by Wendy to investigate that variant of controllability.

Finally, we would like to mention our tool *Diane* [43] which provides another connection between business processes and services. Diane takes a Petri net model of a

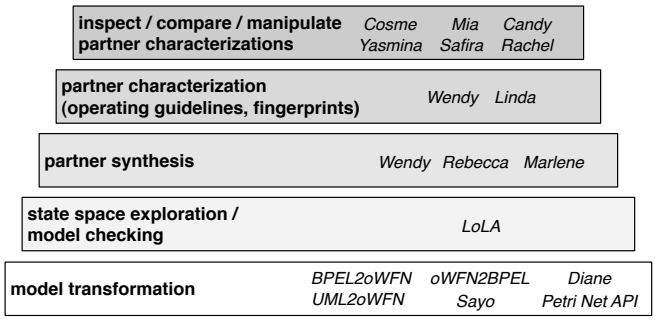

Fig. 5. Our technology stack.

business process N (in fact an arbitrary Petri net) and produces a set of service models $S_1, \ldots, S_n$ such that their composition is equivalent to N. This procedure can be used for divide-and-conquer approaches for the verification of Petri nets in general [46]. We are also working on an extension of this idea leading to suggestions for spinning off services from existing business process models.

To summarize, we have a large set of single purpose tools that exchange information by means of files or data streams. Single-purposeness is the main distinction between SERVICE-TECHNOLOGY.ORG and previous implementations in our group. The tools can be grouped into compilers (BPEL2oWFN, oWFN2BPEL, UML2oWFN), the two core technology providers LoLA and Wendy, a large set of application-oriented tools (Marlene, Rachel, Safira, Rebecca, Linda, Yasmina, Cosme, Candy), and a few model manipulation tools (Sayo, Diane, Petri Net API). Virtually, all tools use the Petri net API for the internal representation of Petri nets. Some external tools are very useful in our family. Apart from the already mentioned tools Petrify and Genet, we rely on powerful libraries for solving linear constraint problems (*lpsolve* [20]), for manipulating Boolean functions using Binary Decision Diagrams (*CUDD* [51]), and for solving the satisfiability problem for propositional Boolean formulae (*MiniSat* [17]).

The application-oriented tools import critical calculations (such as state space exploration or partner synthesis) from the core technology providers LoLA and Wendy. Hence, optimizations done to the data structures and algorithms in these tools directly transfer to the performance of the application-oriented tools. This led us to view our tool family in terms of a technology stack (see Fig. 5).

We already discussed the model transformations which realize the import and export of models, as well as conversions between different file formats. On top of this layer, the bottom technology is state space exploration as provided in LoLA. The next two layers use state space exploration to produce a partner service to a given service which in turn is used as a central element in operating guidelines. Most other applications inherit their computational power at least partly from LoLA or Wendy.

In the remainder of this article, we discuss several lessons we learned from the availability of the tools listed so far and from the way we implemented them. We discuss the lessons in the light of academic tool development which differs in several respects from industrial tool development.

4 Link to reality

This section is devoted to lessons related to the link between our family of tools and reality. The following lessons are not necessarily surprising. We believe, however, that our experience with the SERVICE-TECHNOLOGY.ORG tool family adds some evidence to these lessons.

One of the most apparent purposes for implementing theoretical approaches is to prove applicability in nontrivial context. This is necessary as theoretical results on worst-case complexity typically do not clearly distinguish between working and nonworking approaches. In [19], for example, we demonstrated that — different to common belief — state space methods can very well be competitive for the formal verification of business processes. Only because of the convincing results in this study we have been to raise the issue of using verification even as part of actual modeling process ("verify each time you save or load models").

Lesson 1. *An actual prototype implementation propels transfer of technology from theory to practice.*

Another advantage of the prototype implementation is that experimental data presented in papers become more serious. Over the previous few years, we chose to make tools and experimental input data available over the Internet. The page[2] is an example related to the paper [31]. Because of the simple architecture of our tool family, it is possible to provide just the relevant part of our tool family, and to freeze the state of implementation at the point in time where the experiments have actually been carried out. By making experimental results repeatable by independent instances and by disclosing the source code of the tools, we address recent discussions on fraud and bad scientific practice.

Lesson 2. *Prototype implementations help in making experimental evidence transparent.*

Our implementations also provided valuable feedback to the improvement of tools. In the case study [19], we compared various combinations of soundness checking techniques. Among them were preprocessing approaches based on Petri net structural reduction rules [7,45]. Such a rule locally replaces a Petri net pattern with a simpler one yet preserving given properties (such as soundness). By the experiments on a large sample set, we found that the effect of applying these reductions is marginal if the subsequent state space verification is using the *partial order reduction*, one of the most effective state space reduction methods. Consequently, we were able to shift efforts from further implementation of structural reduction to other, more beneficial tasks.

Lesson 3. *Large case studies help to detect bottlenecks early.*

We observed anecdotal evidence for this during the evaluation of the results of a case study. With our earlier tool Fiona, we calculated the operating guidelines of several industrial service orchestrators we translated from WS-BPEL processes. Both runtime and memory consumption were larger than we expected, compared to the relatively small sizes of the resulting operating guidelines. Fiona faithfully implemented the construction algorithm from [36] and first calculated an overapproximation of the final annotated

[2] `http://service-technology.org/publications/lohmann_2008_bpm`

automaton from which it iteratively removed states that could not belong to the final result. For the concrete examples, however, it turned out that over 90 % of the generated states were eventually removed. We did not observe such a large overhead during the analysis of academic examples. Consequently, any previous optimization effort was put into an *efficient generation* of this overapproximation. With the experience of this case study, we reimplemented the algorithm in the new tool Wendy and implemented a static analysis of the model to *avoid the generation* of spurious states in the first place. Even though this preprocessing employs assumed expensive state space verification, it allowed for dramatic speedups. Wendy is typically 10 to 100 times faster than Fiona, because we could reduce the number of spurious states to less than 1 %.

5 Choice of formalism

In this section, we report on observations that we made concerning the choice of formalism. During the development of some of the tools, we discussed several variations, most notably the use of a workflow net structure [1]. In the end, we found that we would benefit most from the most liberal formalism that is still analyzable: bounded (finite state) Petri nets and finite automata.

Again, the study [19] yields an excellent example. For comparing tools, we translated various business processes not only into the LoLA format, but also into Woflan format which uses the aforementioned workflow nets both in modeling and verification. Sticking to this workflow structure has a couple of advantages in verification, for instance through a simple connection between soundness on one hand and well investigated Petri net properties [1] on the other hand. By the workflow net structure, each model has a single distinguished end state. Unfortunately this was not the case for our models stemming from the IBM Websphere Business Modeler. These models inherently have multiple end states. Using the algorithm of Kiepuszewski et al. [29], we were able to extend the models such that all end states merged into one. This construction, however, is only applicable to nets with *free choice* structure [15]. For free choice nets, many efficient verification techniques are available which do not work at all, or at least not efficiently, for arbitrary Petri nets. So whereas Woflan forced us into a workflow net structure for reasons of efficiency in one regard, it forced us into less structured models, with more serious runtime penalties in other regards.

Lesson 4. *The input formalism of a verification tool should not contain any structural restrictions beyond those required by the implemented technique.*

With a liberal formalism, the resolution of the tradeoff between different structural features can be left to the generation of the model. Whether a model satisfies a certain structural constraint (such as workflow net structure or free choice nature) can typically be easily decided.

Admittedly, the situation in our tool family is easier than in Woflan. Our models are typically generated by the compilers of our tool family, whereas Woflan models can be directly drawn using a graphical editor. Not forcing any structural restrictions in Woflan would mean that users tend to produce spaghetti-like models which are then rather error-prone and hard to verify. Hence, the previous lesson only works in combination with another one.

Lesson 5. *It is beneficial to separate the formalism used for modeling from the one used for verification.*

Then, the generation of a "verification-friendly" model is enforced in the translation between the formalisms. Decoupling the modeling from verification has another advantage. Whereas modeling tools inherently need to address domain-specific notations, structuring mechanisms, and procedures, this is not necessarily the case for verification technology. Actually, when we tuned our tool LoLA for the verification of business processes, we ended up with more or less the same combination of reduction techniques which were also found optimal in the verification of asynchronous hardware circuits [53] or even biochemical reaction chains [55]. In turn, having implemented a verification tool for some purpose, a domain-unspecific formalism helps to conquer new application areas and to transfer experience from one domain to another. A further good example in this regard is the tool Petrify. Although its designers focused on applications in hardware synthesis, they kept their basic formalism sufficiently general and hence allowed us to use their tool in a services context. According to our experience, the ability to reach out for new application domains is much more significant than potential performance gains by domain-specific heuristics. Last but not least, domain-unspecific formalisms tend to be much simpler with respect to basic concepts than domain-specific ones (compare Petri nets or automata with notations such as EPC or BPMN) thus simplifying data structures and algorithms in the tools.

Lesson 6. *Keep verification technology domain-unspecific.*

As our internal representation of business processes and services did not reflect any restrictions from the modeling domain, the only tool affected by the shift from the Petri net semantics for BPEL4WS 1.1 [26] to the one for WS-BPEL 2.0 [32] was the corresponding compiler. In both semantics, we were able to represent complex mechanisms such as fault handlers, compensation handlers, or termination handlers. Likewise, the link to new languages (such as BPMN 2.0) will only involve building a new compiler whereas we will not need to touch any technology provider or application-oriented tool in our family.

As a side-effect of domain-unspecific technology, it is easy to integrate our tools into other frameworks. LoLA is already integrated into several Petri net frameworks including *CPN-AMI* [24], the *Petri Net Kernel* [59], the *Model Checking Kit* [50], and the *Pathway Logic Assistant* [55]. Integrations into other frameworks are progressing, such as into ProM, Oryx, or YAWL. This way, we benefit from functionality of the mentioned tools. In particular, there is no reason for us to invest resources into graphical user interfaces or simulation engines. Instead, we can focus all available resources on our core competencies which are verification techniques.

Providing domain-unspecific verification technology does not necessarily mean that, through a sufficiently general formalism used for verification, the development of compilers into the formalism can be done completely independent of the subsequent technology providers. Compare, for example, the different Petri net semantics given to WS-BPEL [32,47]. The proposal of [47] employs *cancellation regions*, a feature which is supported by YAWL. These cancellation regions can be transformed to ordinary Petri nets. This translation, however, introduces several global status places which introduce dependencies between Petri net transitions. These dependencies, in turn, spoil the partial order reduction, LoLA's most powerful state space reduction technique. Consequently,

the translation [32] carefully avoids the introduction of global status places. A detailed discussion of the different modeling approaches is provided in [38].

Lesson 7. *Despite a domain-unspecific formalism, compilers need to fit to the subsequent technology providing tools.*

6 Architecture

Several lessons can be learned from the architecture of SERVICE-TECHNOLOGY.ORG. Two years ago, most of our results were implemented within a single tool, *Fiona* [41]. As more and more results emerged, the tool constantly grew and eventually we ran into severe maintenance problems.

With Wendy, we completely rethought our tool development process. We came to the conclusion that it would be beneficial to switch from a single multipurpose tool to the family of single-purpose tools which we sketched in Sect. 3. The advantages of the single-purpose paradigm became such evident that we managed to copy most functionality of Fiona in the new tool family in less than a year.

The main lesson of this section is:

Lesson 8. *Separate functionality into a multitude of single-purpose tools rather than integrating it into a single tool.*

The decisive advantage of the single-purpose approach is that data structures are significantly simpler than in a multipurpose tool. In our case, many algorithms roughly shared the same core data structures (in particular, annotated automata). However, each of the algorithms required some subtle changes in the general scheme, or required certain constraints. In effect, we dived into an increasingly deep class hierarchy. As this hierarchy became less and less transparent, programmers did no longer oversee which invariants they could rely on for certain data structures and so a number of redundant computations occurred. It become increasingly difficult to familiarize students with the existing code and to add functionality. Cross-dependencies caused severe problems for maintenance and testing.

Wendy was an attempt to flee this self-magnifying loop. It reimplements the core tasks of Fiona: computation of a correctly interacting partner and computation of an annotated automaton which represents all correctly interacting partners. We achieved the following results:

- Wendy's core functionality could be implemented within 1,500 lines of code instead of more than 30,000 lines of code in Fiona.
- Wendy was coded within two weeks compared to several years of developing Fiona.
- Wendy solves problems 10 to 100 times faster than Fiona although it is used on the same basic algorithmic idea. At the same time, the memory consumption could be decreased by a similar factor.
- Wendy has shallow data structures and its code is easy to understand. For instance, the whole functionality is implemented in six classes and there was no need for any class hierarchies (viz. inheritance). In contrast, Fiona consists of over 50 classes with an up to 5-level class inheritance.

The data structures could be kept shallow as they no longer needed to fit the needs of many tool components at once.

Lesson 9. *Single-purpose tools yield simpler data structures which in turn simplifies maintenance and testability.*

The clearer structure of the tool helped us to design a set of tests with more than 95 % code coverage. A similar endeavor would have been hopeless for Fiona as the various interfering features would have caused an astronomical number of test cases. For the same lack of interference with other features, it is much easier to repeat bugs on simple samples rather than big input files thus speeding up debugging.

With Wendy's 1,500 lines of code, it was quite easy to delegate further implementation tasks to students. Excited by the efficiency gains of Wendy, we span off more functionality from Fiona. The pattern described above repeated to all the tools which now form the SERVICE-TECHNOLOGY.ORG family. In addition, we observed that the separation into many autonomous tools actually led to tremendous speed-ups in the response to errors. As we assigned whole tools to students (what we could do because of the light weight of each tool), responsibilities for errors were fully transparent. Moreover, we recognized that the students committed much more to "their tool" than they used to commit to "their changes in the code" of the big tool. It was easier to get them to work. In the old tool, they hesitated to touch code as they feared "to break everything".

These observations may be typical for an academic context. Here, much of the programming work is carried out by students. On the one hand, their commitment to tool development is only part of their duties and only for a very limited time interval. Programming is often just a side-product of writing a thesis or a paper. This way, skills and experience do not grow over time and senior software architects are rare. On the other hand, the algorithmic solutions to be implemented tend to be rather involved as they stem from theoretically challenging problems. Hence, for successful tool development it is critical to provide an architectural environment that is as simple as possible.

Lesson 10. *Single purpose tools have just the right granularity for tool development in an academic context.*

Apparently, other groups have come to similar conclusions. At least, the plugin concepts in ProM and Oryx also seem to follow the idea that the unit of code to be delegated to a single student should be extremely well encapsulated. However, their plugin concept seems to require a very well designed and mature core unit which does not exist in our family. Furthermore, these tools offer very sophisticated graphical user interfaces which not only justifies the need of a central framework, but also a higher level of abstraction in the interface between the plugins and this framework. In our tool family, however, performance is the key factor and we decided to keep user interaction and abstraction layers to a minimum.

Lesson 11. *Single purpose tools permit tool development at varying speed and sophistication.*

In our case, the transition from a monolithic tool to a family of tools led to smaller release cycles. We were able to implement concepts such as code reviews, four-eyes programming and other *extreme programming* [6] techniques. In particular, the core control logic of each individual tool tends to be rather simple and linear.

Of course, one may object that we just shifted complexity from a single tool to the interplay between the tools. One issue could be the definition of formats for exchanging

definitions. Surprisingly, this was a minor issue so far in our family, for the following simple reason:

Lesson 12. *Data exchanged between tools of the family are typically tied to precisely defined core concepts of the underlying theory.*

As the theoretical concepts are only to a small degree subject to continuous improvements, the exchange formats remain rather stable. Another issue with single purpose tools could be to find out what the single purpose of such a tool would be. Again the answer might be surprising.

Lesson 13. *It is not overly important to precisely understand the purpose of a single-purpose tool in early development stages.*

If, after some time of development, it turns out that the functionality of a tool should have been split into separate tools, just split then! This is what we have exercised with Fiona. Originally thought as a tool to decide controllability, Fiona grew over time into a tool for synthesizing partners and operating guidelines, for verifying substitutability, for synthesizing adapters, and other tasks. Only then we realized that this functionality would be better provided by more than one tool and only then could we plan these new tools. Only with the experience from Fiona we have been able to define the functionality that should go into the Petri net API. The costs for refactoring are affordable and provide a canonical opportunity for revising design decisions.

In addition, a certain modularity in the tool family is suggested by a corresponding modularity in the underlying theory. In the single-purpose paradigm, a cited result in the theory is often copied by a call of the related tool. Here, Marlene is an excellent example. Marlene is devoted to the computation of an adapter between otherwise incompatible services P and R. Theory suggests [22] that the adapter A should be composed of two parts, E and C. E implements the individual adaptation activities that can be performed *in principle* by the adapter, based on a specification which is part of the input to Marlene. C controls, based on the actual exchanged messages, when to execute the available activities in E. According to the theory, C is just a correctly interacting partner of the composition $P \oplus E \oplus R$. Hence, Marlene transforms the input into a Petri net E, calls Wendy on $P \oplus E \oplus R$ to synthesize C and compiles the resulting C to the final adapter $A = E \oplus C$. Every optimization or extension that is done to Wendy is thus immediately inherited by Marlene.

A third concern of the single-purpose paradigm could be the fear of divergence between the tools which could compromise interoperability. However, despite available tools for software engineering, a whole tool will always be more encapsulated than a single method.

Lesson 14. *It is easier to let different versions of a tool coexist than to maintain different versions of a function or method within a single tool.*

We think that this is an advantage of a tool family even compared to a plugin based architecture where it is nontrivial to run various versions of a plugin concurrently.

7 Conclusion

We introduced SERVICE-TECHNOLOGY.ORG as a tool family which provides prototype implementations for several theoretical results on correctness of business processes and services. The core technology providers in this family have been successfully exposed to real-world problems. These real-world problems were accessible, because we have several compilers which mediate between theoretical core formalisms and industrial languages. In this paper, we shared several insight that we observed upon tool development. As our development included a significant refactoring from a monolithic tool toward an interoperable tool family, we are able to directly compare different solutions. Our findings particularly concern the specifics of academic tool development.

References

1. Aalst, W.M.P.v.d.: The application of Petri nets to workflow management. Journal of Circuits, Systems and Computers 8(1), 21–66 (1998)
2. Aalst, W.M.P.v.d., Dongen, B.F.v., Günther, C.W., Mans, R.S., de Medeiros, A.K.A., Rozinat, A., Rubin, V., Song, M., Verbeek, H.M.W.E., Weijters, A.J.M.M.: ProM 4.0: Comprehensive support for *real* process analysis. In: ICATPN 2007. pp. 484–494. LNCS 4546, Springer (2007)
3. Aalst, W.M.P.v.d., Hofstede, A.H.M.t.: YAWL: yet another workflow language. Inf. Syst. 30(4), 245–275 (2005)
4. Alves, A., et al.: Web Services Business Process Execution Language Version 2.0. OASIS Standard, OASIS (2007)
5. Badouel, E., Darondeau, P.: Theory of regions. In: Advanced Course on Petri Nets. pp. 529–586. LNCS 1491, Springer (1996)
6. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, 2nd edition edn. (2005)
7. Berthelot, G., Lri-Iie: Checking properties of nets using transformation. In: Advances in Petri Nets 1985. pp. 19–40. LNCS 222, Springer (1986)
8. Billington, J., Christensen, S., Hee, K.M.v., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The Petri net markup language: Concepts, technology, and tools. In: ICATPN 2003. pp. 483–505. LNCS 2679, Springer (2003)
9. Carmona, J., Cortadella, J., Kishinevsky, M.: Genet: a tool for the synthesis and mining of Petri nets. In: ACSD 2009. pp. 181–185. IEEE (2009)
10. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
11. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers. Trans. Inf. and Syst. E80-D(3), 315–325 (1997)
12. Dadam, P., Reichert, M.: The ADEPT project: a decade of research and development for robust and flexible process support. Computer Science - R&D 23(2), 81–97 (2009)
13. Decker, G., Kopp, O., Leymann, F., Weske, M.: BPEL4Chor: Extending BPEL for modeling choreographies. In: ICWS 2007. pp. 296–303. IEEE (2007)
14. Decker, G., Overdick, H., Weske, M.: Oryx - an open modeling platform for the BPM community. In: BPM 2008. pp. 382–385. LNCS 5240, Springer (2008)
15. Desel, J., Esparza, J.: Free Choice Petri Nets. Cambridge University Press (1995)
16. Dumas, M., Spork, M., Wang, K.: Adapt or perish: Algebra and visual notation for service interface adaptation. In: BPM 2006. pp. 65–80. LNCS 4102, Springer (2006)
17. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT. pp. 502–518. LNCS 2919, Springer (2003)
18. Fahland, D.: Translating UML2 Activity Diagrams Petri nets for analyzing IBM WebSphere Business Modeler process models. Informatik-Berichte 226, Humboldt-Universität zu Berlin, Berlin, Germany (2008)

19. Fahland, D., Favre, C., Jobstmann, B., Koehler, J., Lohmann, N., Völzer, H., Wolf, K.: Instantaneous soundness checking of industrial business process models. In: BPM 2009. pp. 278–293. LNCS 5701, Springer (2009)
20. Free Software Foundation: lpsolve: Mixed Integer Linear Programming (MILP) Solver. `http://lpsolve.sourceforge.net`
21. Fu, X., Bultan, T., Su, J.: Conversation protocols: a formalism for specification and verification of reactive electronic services. Theor. Comput. Sci. 328(1-2), 19–37 (2004)
22. Gierds, C., Mooij, A.J., Wolf, K.: Specifying and generating behavioral service adapter based on transformation rules. Preprint CS-02-08, Universität Rostock, Rostock, Germany (2008)
23. Gottschalk, K.: Web Services Architecture Overview. IBM whitepaper, IBM developerWorks (2000), `http://ibm.com/developerWorks/web/library/w-ovr`
24. Hamez, A., Hillah, L., Kordon, F., Linard, A., Paviot-Adet, E., Renault, X., Thierry-Mieg, Y.: New features in CPN-AMI 3: focusing on the analysis of complex distributed systems. In: ACSD. pp. 273–275. IEEE (2006)
25. Hee, K.M.v., Oanea, O., Post, R., Somers, L.J., Werf, J.M.E.M.v.d.: Yasper: a tool for workflow modeling and analysis. In: ACSD 2006. pp. 279–282. IEEE (2006)
26. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to Petri nets. In: BPM 2005. pp. 220–235. LNCS 3649, Springer (2005)
27. Kaschner, K.: Safira: Implementing set algebra for service behavior. In: ZEUS 2010. pp. 49–56. CEUR Workshop Proceedings Vol. 563, CEUR-WS.org (2010)
28. Kaschner, K., Wolf, K.: Set algebra for service behavior: Applications and constructions. In: BPM 2009. pp. 193–210. LNCS 5701, Springer (2009)
29. Kiepuszewski, B., Hofstede, A.H.M.t., Aalst, W.M.P.v.d.: Fundamentals of control flow in workflows. Acta Inf. 39(3), 143–209 (2003)
30. Liske, N., Lohmann, N., Stahl, C., Wolf, K.: Another approach to service instance migration. In: ICSOC 2009. pp. 607–621. LNCS 5900, Springer (2009)
31. Lohmann, N.: Correcting deadlocking service choreographies using a simulation-based graph edit distance. In: BPM 2008. pp. 132–147. LNCS 5240, Springer (2008)
32. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In: WS-FM 2007. pp. 77–91. LNCS 4937, Springer (2008)
33. Lohmann, N., Kleine, J.: Fully-automatic translation of open workflow net models into simple abstract BPEL processes. In: Modellierung 2008. Lecture Notes in Informatics (LNI), vol. P-127, pp. 57–72. GI (2008)
34. Lohmann, N., Kopp, O., Leymann, F., Reisig, W.: Analyzing BPEL4Chor: Verification and participant synthesis. In: WS-FM 2007. pp. 46–60. LNCS 4937, Springer (2008)
35. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting BPEL processes. In: BPM 2006. pp. 17–32. LNCS 4102, Springer (2006)
36. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: ICATPN 2007. pp. 321–341. LNCS 4546, Springer (2007)
37. Lohmann, N., Verbeek, H., Dijkman, R.M.: Petri net transformations for business processes – a survey. LNCS ToPNoC II(5460), 46–63 (2009), special Issue on Concurrency in Process-Aware Information Systems
38. Lohmann, N., Verbeek, H., Ouyang, C., Stahl, C.: Comparing and evaluating Petri net semantics for BPEL. Int. J. Business Process Integration and Management 4(1), 60–73 (2009)
39. Lohmann, N., Weinberg, D.: Wendy: A tool to synthesize partners for services. In: PETRI NETS 2010. pp. 297–307. LNCS 6128, Springer (2010)
40. Lohmann, N., Wolf, K.: Realizability is controllability. In: WS-FM 2009. pp. 110–127. LNCS 6194, Springer (2010)
41. Massuthe, P., Weinberg, D.: Fiona: A tool to analyze interacting open nets. In: AWPN 2008. pp. 99–104. CEUR Workshop Proceedings Vol. 380, CEUR-WS.org (2008)

42. Mendling, J., Moser, M., Neumann, G., Verbeek, H.M.W., van Dongen, B.F., van der Aalst, W.M.P.: Faulty EPCs in the SAP reference model. In: BPM 2006. pp. 451–457. LNCS 4102, Springer (2006)
43. Mennicke, S., Oanea, O., Wolf, K.: Decomposition into open nets. In: AWPN 2009. CEUR Workshop Proceedings, vol. 501, pp. 29–34. CEUR-WS.org (2009)
44. Milner, R.: Communication and concurrency. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
45. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)
46. Oanea, O., Wimmel, H., Wolf, K.: New algorithms for deciding the siphon-trap property. In: PETRI NETS 2010. LNCS, Springer (2010)
47. Ouyang, C., Verbeek, E., Aalst, W.M.P.v.d., Breutel, S., Dumas, M., Hofstede, A.H.M.t.: Formal semantics and analysis of control flow in WS-BPEL. Sci. Comput. Program. 67(2-3), 162–198 (2007)
48. Prasad, M.R., Biere, A., Gupta, A.: A survey of recent advances in SAT-based formal verification. STTT 7(2), 156–173 (2005)
49. Reichert, M., Rinderle-Ma, S., Dadam, P.: Flexibility in process-aware information systems. LNCS ToPNoC II(5460), 115–135 (2009), special Issue on Concurrency in Process-Aware Information Systems
50. Schröter, C., Schwoon, S., Esparza, J.: The Model-Checking Kit. In: ICATPN 2003. pp. 463–472. LNCS 2679, Springer (2003)
51. Somenzi, F.: CUDD: CU Decision Diagram Package, `http://vlsi.colorado.edu/~fabio/CUDD/`
52. Stahl, C., Massuthe, P., Bretschneider, J.: Deciding substitutability of services with operating guidelines. LNCS ToPNoC II(5460), 172–191 (2009), special Issue on Concurrency in Process-Aware Information Systems
53. Stahl, C., Reisig, W., Krstic, M.: Hazard detection in a GALS wrapper: A case study. In: ACSD'05. pp. 234–243. IEEE (2005)
54. Sürmeli, J.: Profiling services with static analysis. In: AWPN 2009. CEUR Workshop Proceedings, vol. 501, pp. 35–40. CEUR-WS.org (2009)
55. Talcott, C.L., Dill, D.L.: Multiple representations of biological processes. T. Comp. Sys. Biology LNCS 4220(VI), 221–245 (2006)
56. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and more focused control-flow analysis for business process models through SESE decomposition. In: ICSOC 2007. pp. 43–55. LNCS 4749, Springer (2007)
57. Verbeek, H.M.W., Aalst, W.M.P.v.d., Hofstede, A.H.M.t.: Verifying workflows with cancellation regions and OR-joins: An approach based on relaxed soundness and invariants. Comput. J. 50(3), 294–314 (2007)
58. Verbeek, H.M.W., Basten, T., Aalst, W.M.P.v.d.: Diagnosing workflow processes using Woflan. Comput. J. 44(4), 246–279 (2001)
59. Weber, M., Kindler, E.: The Petri Net Kernel. In: Petri Net Technology for Communication-Based Systems. pp. 109–124. LNCS 2472, Springer (2003)
60. Weinberg, D.: Efficient controllability analysis of open nets. In: WS-FM 2008. pp. 224–239. LNCS 5387, Springer (2008)
61. Wolf, K.: Generating Petri net state spaces. In: ICATPN 2007. pp. 29–42. LNCS 4546, Springer (2007)
62. Wolf, K.: Does my service have partners? LNCS T. Petri Nets and Other Models of Concurrency 5460(2), 152–171 (2009)
63. Wolf, K., Stahl, C., Ott, J., Danitz, R.: Verifying livelock freedom in an SOA scenario. In: ACSD 2009. pp. 168–177. IEEE (2009)