A Service-Level Agreement Approach Towards Termination Analysis of Service-Oriented Systems

Mandy Weißbach and Wolf Zimmermann
Institute of Computer Science, University of Halle
06120 Halle (Saale), Germany
Email: {weissbach, zimmermann}@informatik.uni-halle.de

***Abstract*—Classical approaches for program analysis as, e.g., termination analysis usually do not take into account modern software approaches such as service-oriented systems or cloud computing. Instead, they have a monolithic view on the software system as a single completely available program. As first step to enable such analyses also in a service-oriented or cloud computing context, respectively, this paper considers termination. Since termination is a service quality attribute, we consider a service-level agreement approach that allows dynamic bindings to software services. In contrast to many other service-level agreements, termination is a binary attribute that cannot be measured quantitatively (as, e.g., reliability or response time). The proposed approach shows how clients of services can verify the information provided by the services.**

Keywords-Termination; Software Services; Service Level Agreement; Verification.

I. Introduction and Motivation

The vision of cloud computing is (among others) that there are numerous software services in the cloud that can be used by clients to fullfil their functionality. These services are functionally equivalent in the sense of the context of the client. However, they might differ in their non-functional properties. Thus clients may negotiate service level agreements on non-functional properties such as, e.g., reliability, availability, response times, etc. The literature on service-oriented computing and cloud computing offers already numerous techniques that clients may monitor these quality attributes, see, e.g., [1] for an overview. However, there are other service quality attributes such as, e.g., termination of the services and/or the client, robustness (i.e., neither the service nor the client aborts due to uncaught exceptions), or absence of deadlocks. In contrast to the above mentioned service quality attributes, these attributes have a binary character: either the services or clients satisfy the quality attribute or not. In this work, we consider in particular the termination of software services and the clients using software services. Petri-Net based approaches towards deadlock analysis are usually based on termination of the services [2]–[4].

Remark: At first glance, it seems that there is no need for a termination analysis in service-oriented systems (except possibly for deadlock analysis) because one might think that after a certain time the client might switch to another, functionally equivalent, service in the cloud. However, there are situations where this approach cannot be applied. First, the approach doesn't work if none of the functionally equivalent services has (for the client) satisfactory quality attributes. In this case the chosen service becomes the single candidate and its termination is an important property for the client. A second reason is the choice of the time period: If the period is fixed to just a few seconds or minutes, this might be a reasonable approach. It might work well in business applications. However, in scientific computing or bioinformatics there are computation intensive applications and if these are offered as services they might run for hours or days. Fixing the time period to a few seconds or minutes implies that any functionally equivalent service fails to be finished within this time. On the other hand, it doesn't make sense to switch after a few hours or days to another service that possibly requires even more execution time than the originally chosen service. Thus, in these situations it is better to know that the service terminates and will deliver an answer. Third, a termination proof for clients may require information on the effect on results of services being called, e.g., their size. This size change information must also be included in the analysis and has a rather different character than simple termination. □

The techniques that enable the clients to check whether service-level agreements are obeyed cannot be applied in the context of binary quality attributes. Consider for example termination: if a client has not yet a response from a service, the client cannot conclude that the service doesn't terminate. The service might respond within the next second. On the other hand, the client cannot reason on its own termination behaviour without provision of adequate information from the services. This information must be correct. Thus, the challenge is how the client can verify that the requested information is correct. The situation becomes even more difficult if a client uses a service A and the service A uses a service B, etc. In this case termination of the client may indirectly depend on service B and service A needs to request information on termination of B as well as additional information to prove its termination.

In this paper, we assume that there are no recursive call-backs, i.e., recursion over service boundaries are excluded. Furthermore, in order to use well-known termination analysis approaches, we exclude service-internal parallelism since this is still an open issue in classical termination analysis. Thus, we tackle in this paper termination analysis of service-oriented systems in dynamically changing environments where recursive call-backs and service-internal parallelism is being excluded.

 ISBN: 978-1-61208-153-3

The paper is organized as follows: Section II introduces into classical termination analysis. The following Section III shows how this approach can be extended in a service-oriented or cloud computing context, respectively, using a service-level agreement approach. In Section IV, we show how clients can verify the results by combining the approach of Section III with approaches used for verification of Web pages. Section V discusses related work and Section VI concludes this work.

II. Termination Analysis

Although termination of programs is undecidable (known as halting problem), there is a lot of work on *conservative* analysis of program termination. A conservative termination analysis guaruantees termination in the case of a positive answer. However, a negative answer may be false. It should be interpreted as *termination cannot be proven*. Thus, termination analysis does not implement the halting problem but it only provides a one-sided solution (similar to model checking) . The following discussion shows that there are a number of works (it just mentions the most important ones) on termination analysis. Each of them is conservative and assume that the whole program to be analyzed is available to them.

The first works on termination analysis or the related field of automatic complexity analysis go back to [5] for pureLisp programs. This was generalized to first-order functional languages [6] and to object-oriented imperative programs [7], [8]. Works on automatic complexity analysis as well as on termination analysis are based on the notion of a termination function. This is a function from program states to natural numbers that strictly decreases when executing the body of loop or when a procedure is called recursively. Since there is no infinite descending chain in the natural numbers, a termination function ensures loop or recursion termination, respectively. More recent work on termination analysis focuses on automatic derivation of termination functions, which is often called the size-change principle, cf. [9]–[12]. Instead introducing into these methods, we informally demonstrate termination analysis by the example in Figure 1. In a service-oriented architecture the four classes will be considered as services (implemented by web services, cf. Figure 2. `Calendar` contains to public procedures `first()` and `next(Month month)` which together can be used to iterate over all 12 months of a year. The class `List` is a classical list implementation with a sentinel `empty`. `MSales` has access to a customer database. The procedure `sales(Month month, Year year)` uses this customer database to calculate the sales of month `month` in year `year`. Procedure `sales(Year year)` calculates the sales of year `year` by summing up the sales of all months of `year`.

Suppose the termination of procedure `YSales.sales` has to be analyzed. Note that all the steps (except possibly the provision of terminations functions which have to annotated) can be performed automatically according to the above mentioned works.

```
class YSales {
  private Msales msales;
  public int sales(Year year) {
    Month month=Calendar.first();
    int sum=0;
    while (month!=Month.complete) {
      int amount=msales.sales(month,year);
      sum += amount;
      month=Calendar.next(month);
    }
    return sum;
  }
}
class Calendar {
  public Month first() { return Month.jan; }
  public Month next(Month m) {
    if (m==Month.jan) return Month.feb;
    ...
    if (m==Month.dec) return Month.complete
  }
}
class MSales {
  private static CustomerDatabase db;
  public int sales(Month month,Year year) {
    List cl=db.getCustomers(month,year);
    int sum=0;
    while (cl!=List.empty) {
      int amount=cl.hd();
      sum += amount;
      cl=cl.tl();
    }
    return sum;
  }
}
class List {
  private int head;
  private List tail;
  static List empty=new List();
  public int hd() { return head; }
  public List tl() {
    return (tail==NULL?empty:tail);
  }
}
```

Figure 1. Sales-Example

Step 1 Analyze each non-recursively called procedure for termination:

Since this procedure calls procedures `MSales.sales`, `Calendar.first`, and `Calendar.next`, these procedures have to be analyzed for termination.

Step 2 Analyze each loop and each recursively called procedure for termination by deriving/introducing an adequate termination function:

The loop termination of the loop in `YSales.sales` apparantly depends on the variable `month`. The termination function φ defined by

$$\varphi(\texttt{month}) \triangleq 13 - sz(\texttt{month}) \qquad (1)$$

where $sz(\texttt{month})$ is the number of the month (i.e., $sz(\texttt{Jan}) = 1, sz(\texttt{Feb} = 2)$, etc. and $sz(\texttt{complete}) = 13$) proves termination. This is because

$$\varphi(\texttt{next}(\texttt{month}))) = \varphi(\texttt{month}) - 1 \qquad (2)$$

(2) can be derived by determining a size change function for `next` with the notion of size defined by (1), i.e., a function $\varphi\_\texttt{next} : \mathbb{N} \rightarrow \mathbb{N}$ such that $\varphi\_\texttt{next}(\varphi(\texttt{month})) = \varphi(\texttt{next}(\texttt{month})))$.

Step 3 Determine the necessary size change functions for procedures:

By a simple case analysis it can be determined that $\varphi\_\texttt{next}$ is defined by $\varphi\_\texttt{next}(n) = n - 1$ thereby proving that the termination function φ decreases by 1 during loop termination.

 ISBN: 978-1-61208-153-3

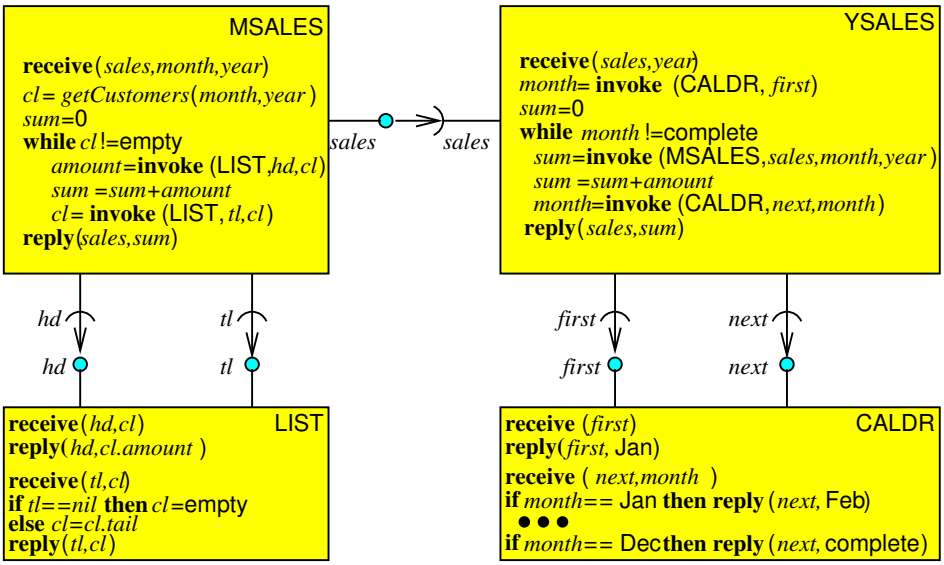

Figure 2. A Service-Oriented Architecture for the Sales Example

The termination of `Calendar.first` and `Calendar.next` can be derived directly as they neither contain a loop nor a procedure call. The termination analysis for `MSales.sales` must follow the same approach as mentioned above:

Step 1: The procedure `MSales.sales` calls procedure `CustumerDatabase.getCustomers`, `List.hd`, and `List.tl`. The latter two terminate since they neither call a procedure nor contain a loop. The former terminates since it executes a database query (not shown in Figure 1).

Step 2: The termination of the loop in `MSales.sales` depends on the length of the list `cl`, i.e., the termination function is recursively defined by

$$\psi(\texttt{cl}) \triangleq \begin{cases} 0 & \text{if } \texttt{cl} = \texttt{NULL} \\ 1 + \psi(\texttt{cl.tail}) & \text{otherwise} \end{cases} \quad (3)$$

This termination function requires the determination of the size change function $\psi\_\texttt{tl} : \mathbb{N} \to \mathbb{N}$ such that

$$\psi\_\texttt{tl}(\psi(\texttt{cl})) = \psi(\texttt{tl}(\texttt{cl})) \quad (4)$$

Step 3: The analysis yields that $\psi\_\texttt{tl}(n) = n - 1$ which completes the proof of termination of the loop in `MSales.sales`

In a nutshell, the termination argument for `YSales.sales` is as follows:

- `YSales.sales` terminates because each procedure called in the body terminates and the loop terminates
- The loop terminates because (1) is a termination function
- φ is a termination function because of (2) which proves that φ strictly decreases after executing the loop body

The steps presented in this section can be formalized as proof rules (see [14] for a short summary). These rules are usually the formal basis for the correctness of termination analysis. If the proof succeeds the program terminates. However, a program may terminate althoug a termination analysis cannot find a proof.

III. An SLA approach for Termination Analysis

The goal of this section is to apply the approach of Section II in a service-oriented context. It is demonstrated by the service-oriented architecture shown in Figure 2 which corresponds to the example in Figure 1 and is implemented by three web services MSALES (with interface *sales*), LIST (with interfaces *hd* and *tl*), CALDR (with interfaces *first* and *next*), and a client YSALES.

Note that the implementations of the web services are not known to their clients. Thus, a termination analysis cannot directly follow the approach as described in Section II. In particular, Step 1 cannot analyze the termination of services being called but it must rely on the information of the termination provided by the called service. For example the invocation of the services CALDR.*first*, CALDR.*next*, and MSALES.*sales* require termination, and the providing web services must know this information. Note that the client YSALES is not aware of the fact that the termination of MSALES.*sales* depends on the termination of LIST.*hd* and LIST.*tl*. Furthermore, in order to proof termination of the loop in YSALES, the service CALDR must provide a strictly decreasing size change function for *next*. The decision whether it must be decreasing or increasing, or how fast it must be decreasing or increasing for proving termination depends on the loop body.

Thus, in a service-oriented setting, a client needs to have the following infomation when it analyzes its termination:

- The information on the termination of each service called

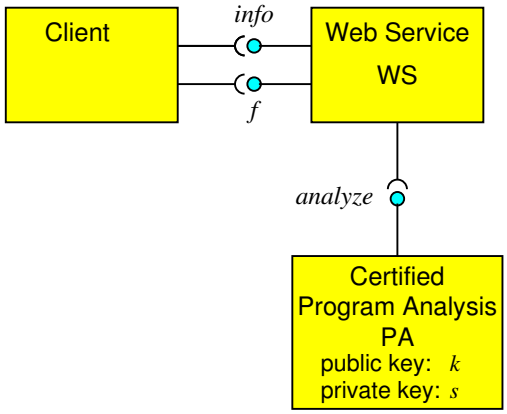

Figure 3. Verifiable Program Analysis Information

- Adequate context-dependent size change functions for those services whose calls influence termination of loops or recursive calls.

While the first information can be provided by the web service providing a called service, the latter must be individually requested by the client while analyzing the termination behaviour of the client. In both cases, the client relies on the correctness of the information provided by the web service.

IV. Certification of Informations Provided by Software Services

A problem with the approach in Section III is the validity of the information on termination of services as well as the validity of the size change function. In contrast to quantitative properties such as, e.g., reliability, availability, or response time, the client has no possibility to check a service level such as termination or the validity of size change functions. We first present an approach that considers basic web services, i.e., they don't use other Web Services. Then, we extend the approach to web services using other Web Services where the use-relation is acyclic.

A. Basic Web Services

Figure 3 shows an approach that may solve this problem. First, a certified program analysis service PA is needed for the analysis of the web service. Second, a public-key infrastructure is needed for enabling the client to verify the results of the analysis. The program analysis service PA must be known to the client as a certified analysis tool. With this infrastructure, a termination/program analysis as discussed in Section III can be implemented such that the client can verify the results from the web service WS:

Step 1: The client requests from web service WS via the service *info* information on the termination or size change of f (as discussed in Section III).

Step 2: Web service WS encrypts its source text with the public key of the certified program analysis service PA and sends it via the interface *analyze* together with the requested analysis to PA.

Step 3: The certified program analysis service PA decrypts the source text of WS with its private key s, performs the requested program analysis, signs the result with its private key s, and returns the signed result to WS.

Step 4: Web service WS returns the signed result to the client together with the public key k of the certified program analysis service PA.

Step 5: The client can decrypt the information with the key k and since the key k is unambiguous, it can verify that the information is obtained by the certified program analysis service PA.

With this approach, the client can verify that the certified program analysis performed its analysis. The encryption in Step 2 is needed because implementers of web services don't want to publish their implementation. With the encryption, the source text is only available to the certified program analysis service PA.

For this approach, the trusted base is certainly the certified program analysis service PA. However, it is not guaranteed that PA really analyzes the source text of WS. A malicious web service WS might send another source text whose analysis results errorneously indicate the client termination or provides an adequate size change function. Currently, we are not aware of a technology that ensures that the WS sends the correct source text to PA.

However, it is possible to make it more difficult for WS to be malicious by keeping the analysis request secret to WS. This can be achieved changing the protocol of the SLA: The client first notifies WS that it wants to perform a program analysis. Then WS returns a public key k of a certified program analysis service PA. The client can use k to verify that PA is indeed certified. Finally, the analysis request is encrypted with k. The above implementation needs only to be changed at Step 3 where the analysis request must be decrypted. If we trust PA and the public key infrastructure, then it is impossible for WS to decrypt the analysis request.

B. Composed Web Services

The approach in Section IV-A doesn't consider the situation as shown in Figure 4. Web service WS_1 uses as a client web service WS_2 and the client is not aware of this usage. Thus, the termination analysis (or other program analyzes) of WS_1 requires the analysis of WS_2 (including possibly the analysis of size change functions).

For the termination analysis or the analysis of size change functions of WS_1's service, web service WS_1 acts as a client of web service WS_2. Hence WS_1 negotiates termination and size change functions with WS_2 as described in Section IV-A. However, this information is needed by the certified program analysis. For example, if the program analysis requires for the termination of f information on the termination of g or a size change function for g where g is an external service call of WS_1, then this information is passed to WS_1 via the interface *painfo* (encrypted with the public key k_1 of WS_1 for security reasons). Service WS_1 decrypts the analysis request and passes it as described in Section IV-A to WS_2.

 ISBN: 978-1-61208-153-3

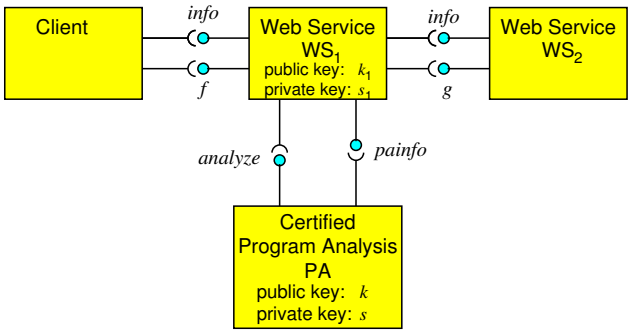

Figure 4. Program Analysis on Composed Web Services

Then, WS_2 returns the information on termination of g or the requested size change function for g, respectively. In contrast to the approach in Section IV-A, the result is not decrypted and verified by WS_1. Instead it is passed to the certified program analysis (as the return value of the service $info$ of WS_2) and the certified program analysis verifies whether the analysis results for g can be trusted. Note, that this approach does not require that WS_2 uses the same program analysis service as WS_1.

Apparently, with this approach WS_2 may use a web service WS_3, etc. However, the approach is limited to acyclic architectures. Otherwise, the termination analysis itsself would run into an infinite loop which practically would have the same effect as a Denial-Of-Service attack to the services.

V. Related Work

There is a need for program analysis of service-oriented systems. Canfora, *et al.* [13] states it as a key challenge for software reverse engineering. Currently, there are not many works on program analysis of service-oriented systems – in particular we are not aware of any work on termination analysis of service-oriented systems except [14]. This work is based on interface descriptions of web services containing termination information and size change function. Furthermore, it doesn't verify the information provided by the interface descriptions.

One of the few works considering program analysis is [15], [16]. They consider response time in terms of some notion of input size. Information on response time is provided by the web service interfaces. Their approach generalizes the approach of [17] for the analysis of software complexity of BPEL processes towards response time. For invocations of other services [15], [16] use the information provided by the corresponding service descriptions. However, they don't verify this information and it seems that size change functions play no role in their approach.

For functional verification of web service contracts, [18] discusses a similar approach using a public key infrastructure. Apparently, contracts should be part of web service interface descriptions and are not part of service-level agreements. In contrast to our approach, they require that the analyzers are located on the same machine as the service implementations, respectively. This one-platform approach allows to take into account the operating system and the compiler.

VI. Conclusion and Future Work

This paper has presented a termination analysis of service-oriented systems in a dynamic changing environment. This goal was achieved by using an SLA approach. It was shown that the service has to provide two kinds of informations: its termination and size change functions requested by the client which enables the client to prove its termination. In contrast to quantitative service qualities, these informations cannot be verified by the client. Therefore, a certification process similar to the verification of web pages has been added in order to ensure that the information has been derived from certified tools.

One property of the approach is the violation of the black-box paradigm of services because they must offer their source to a program analysis service. However, we consider such program analysis services as a trusted institutions (analogous to institutions certifying web pages). In any case, the clients never see implementation details of the used services.

Our approach may be used for the analysis of other binary quality attributes which can be verified by program analyses or model checking approaches. Currently, it excludes cycles in the architecture, i.e., there are no recursive call-backs. Such cycles would lead to an infinite loop while negotiating the service-level agreement. We also assume that the services have no internal parallelism. The next steps will be to drop these assumptions and to consider other binary quality attributes.

Another challenge is to prevent malicious analysis results from the web service to be analyzed. As pointed out in Section IV, a web service may send the wrong source text to the program analysis service. We have presented an approach that keeps the requested analysis secret to the web service but this only makes it more difficult to the web service to cheat. A secure approach must enable the client to verify that the source text given to the program analysis service is identical to the source text of the web service. A possible solution might be that the web service signs its source text with its digital signature when sending it to the program analysis. In this case, at least liability is possible if the wrong source text was sent.

Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful comments.

References

[1] K. Candan, W. Li, T. Phan, and M. Zhou, "Frontiers in Information and Software as Services," in *IEEE International Conference on Data Engineering*. IEEE, 2009, pp. 1761–1768.

[2] W. M. P. van der Aalst, "The application of Petri nets to workflow management," *The Journal of Circuits, Systems and Computers*, vol. 8, no. 1, pp. 21–66, 1998.

 ISBN: 978-1-61208-153-3

[3] W. M. P. van der Aalst, A. P. Barros, A. H. M. ter Hofstede, and B. Kiepuszewski, "Advanced workflow patterns," in *CooplS '02: Proceedings of the 7th International Conference on Cooperative Information Systems*. London, UK: Springer-Verlag, 2000, pp. 18–29.

[4] W. Reisig, "Modeling- and analysis techniques for web services and business processes," in *In FMOODS*, 2005, pp. 243–258.

[5] B. Wegbreit, "Mechanical program analysis," *Communications of the ACM*, vol. 18, no. 9, pp. 528 – 539, 1975.

[6] W. Zimmermann, *Automatische Komplexitätsanalyse funktionaler Programme*, ser. Informatik-Fachberichte. Springer, 1990.

[7] H. Schmidt and W. Zimmermann, "Reasoning about complexity of object-oriented programs," in *Programming Concepts, Methods and Calculi*, ser. IFIP Transactions, E.-R. Olderog, Ed., vol. A–56, 1994, pp. 553–572.

[8] H. Schmidt and W. Zimmermann, "A complexity calculus for object-oriented programs," *Journal of Object-Oriented Systems*, vol. 1, no. 2, pp. 117–147, 1994.

[9] A. M. Ben-Amram and C. S. Lee, "Program termination analysis in polynomial time," *ACM Transactions on Programming Languages and Systems*, vol. 29, pp. 5:1–5:37, January 2007.

[10] A. M. Ben-Amram, "Size-change termination, monotonicity constraints and ranking functions," in *Proceedings of the 21st International Conference on Computer Aided Verification*, ser. CAV '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 109–123.

[11] M. Codish, C. Fuhs, J. Giesl, and P. Schneider-Kamp, "Lazy abstraction for size-change termination," in *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*, ser. LPAR'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 217–232.

[12] F. Spoto, F. Mesnard, and E. Payet, "A termination analyzer for java bytecode based on path-length," *ACM Transactions on Programming Languages and Systems*, vol. 32, pp. 8:1–8:70, March 2010.

[13] G. Canfora, M. Di Penta, and L. Cerulo, "Achievements and challenges in software reverse engineering," *Commun. ACM*, vol. 54, pp. 142–151, April 2011.

[14] M. Weißbach and W. Zimmermann, "Termination analysis of business process workflows," in *Proceedings of the 5th International Workshop on Enhanced Web Service Technologies*, ser. WEWST '10. New York, NY, USA: ACM, 2010, pp. 18–25.

[15] D. Ivanovic, M. Carro, and M. Hermenegildo, "An initial proposal for data-aware resource analysis of orchestrations with applications to predictive monitoring," in *Proceedings of the 2nd Workshop on Monitoring, Adaptation and Beyond (MONA+), Lecture Notes in Computer Science. Springer*, 2010.

[16] D. Ivanovic, M. Carro, and M. Hermenegildo, "Towards Data-Aware QoS-driven Adaptation for Service Orchestrations," in *2010 IEEE International Conference on Web Services*. IEEE, 2010, pp. 107–114.

[17] J. Cardoso, "Complexity analysis of BPEL web processes," *Software Process Improvement and Practice*, vol. 12, no. 1, pp. 35–49, 2007.

[18] J. Lyle, "Trustable remote verification of web services," *Trusted Computing*, pp. 153–168, 2009.

 ISBN: 978-1-61208-153-3