Automated analysis of conflicts in WS–Agreement

Carlos Müller, Manuel Resinas, and Antonio Ruiz-Cortés

***Abstract*—WS–Agreement is one of the most widely used SLA specifications. An advantage of WS–Agreement over other agreement metamodels is that it allows one to define conditional and optional term sets inside an agreement document, which are commonly found features in real-world agreements. Unfortunately, they increase the complexity of the automated detection and explanation of conflicts between SLA terms, leading to new kind of conflicts that are not supported by current techniques. Furthermore, creating a general-purpose conflict analyser in WS–Agreement is a hard task since it should understand the semantics of an unbounded number of languages that can be used in the eight extension points that WS–Agreement includes for the sake of flexibility. In this article we address these issues by providing a conflict classification for SLAs that includes new conflicts derived from the use of conditional and optional term sets; and a novel, language-agnostic technique based on constraint satisfaction problems to automatically detect and explain these conflicts. In pursuing these results, we defined some WS–Agreement concepts as well as a fully-fledged WS–Agreement-compliant language. The developed technique and its reference implementation have been thoroughly validated.**

***Index Terms*—Service Level Agreement, SLA, WS–Agreement, Conflict Management, Consistency.**

I. Introduction and Motivation

Service level agreements (SLA) play an important role during the entire life-cycle of a service-based application (SBA) because they establish the functional and quality aspects of SBAs between providers and consumers or service integrators [2]. Although there is no commonly accepted way to describe SLAs, it ranges over a variety of approaches that include: the use of natural language in SLAs intended to be used only by humans[1]; the use of formal languages with the intention of analyzing some properties of the SLA [3]–[5], and the use of XML documents in standardization efforts aimed at making the interoperability between consumers and providers easier [6]–[8]. One of the most widely used SLA standards is WS–Agreement [6], a proposed recommendation of the Open Grid Forum[2] that provides a protocol for establishing agreements between two parties using an extensible XML language for specifying agreements; and agreement templates to facilitate discovery of compatible agreement parties.

An advantage of WS–Agreement over other agreement metamodels is that it includes several aspects that are necessary to model real-world agreements, namely: (1) conditional terms, i.e. terms whose service level objective (SLO), which defines the agreed quality of service, is subject to a qualifying condition (QC); and (2) terms compositors (TC) that enable the inclusion of optional term sets inside an agreement document. For instance, the AmazonS3 SLA guarantees a DataDurability >= 99.999999999 %, but only if the consumer does not request the Reduced Redundancy Storage (RRS) (which is cheaper than default redundancy check), i.e., "RRS is false" is the QC of the agreement term that guarantees the DataDurability. In addition, the SLA provides several client support features such as an online reporting support, a turn around time of 15 minutes to solve problems, a phone support, an extended support and so on. All of them can be modelled as optional terms by means of TCs to allow clients to choose a customized support plan. Unfortunately, these aspects increase the complexity of the agreements and, hence, the complexity of the tools to manage them. As a result they are not usually supported despite their importance to model real-world SLAs. In fact, a comparison of eight WS–Agreement implementations [9] reveals that only one of them (Phosphorus-WSAG4U) supports TCs and only another different one (AssessGrid) supports QCs.

A part of the SLA management whose complexity increases significantly is the automated detection and explanation of conflicts that may appear between SLA terms, such as inconsistencies, a type of conflict that involves semantics contradictions between terms. This is a key part of the agreement creation process and it has received significant attention by the research community. In particular, a number of approaches focus on detecting inconsistencies that affect the whole agreement [3], [4]. Other proposals go further and also report some explanation for such inconsistencies [1]. However, these proposals are not well suited to deal with SLAs that include QCs a TCs because of two reasons. First, not all proposals are able to deal with the additional complexity that these aspects involve. For instance, [3], [5] do not consider neither QCs nor TCs. Second, these aspects lead to types of conflicts different than inconsistencies that are not supported by current techniques. These new types of conflicts arise because QCs and TCs introduce optional terms and, hence, conflicts may appear in optional parts of the SLA without invalidating the whole document. Thus, a proposal that only detect inconsistencies that affect the whole agreement may incorrectly consider an SLA as conflict-free if its conflicts only affect optional

Manuscript received July XX, 2012; revised Month YY, 2012.
Authors's affiliation: ETS. Ingeniería Informática, University of Seville, 41012 Sevilla (Spain), {cmuller, resinas, aruiz}@us.es.
This work is an extended version of the earlier paper [1]. This work has been partially supported by the Spanish Government under the CICYT projects SETI (TIN2009–07366); and by the Andalusian Government under the projects THEOS (TIC–5906) and ISABEL (P07–TIC–2533).

[1]For instance, the SLAs of companies such as Amazon (aws.amazon.com/s3-sla/) or Google(www.google.com/apps/intl/en/terms/sla.html)

[2]www.ogf.org

or conditional terms.

Furthermore, the extension mechanism of WS–Agreement poses an additional problem in the automated detection and explanation of conflicts in WS–Agreement documents. Contrary to what could be thought, WS–Agreement does not provide a fully-fledged language for specifying agreement documents. Instead, it defines a general–purpose schema that can and must be complemented with a set of languages to specify several parts of the agreement document such as service description terms or service level objectives. Many different languages such as JSDL (Job Submission Description Language) and the WSLA [8] expression language, have been used for this task since WS–Agreement do not impose additional constraints on their expressiveness. In this paper, we call sublanguages (SubLs) to those languages that can be used within a WS-Agreement document.

Consequently, the first thing one has to do to create a WS-Agreement document is to decide which are the SubLs that are going to be used in it. We call a WS-Agreement Configuration (WSAC) to the set of SubLs used within a given WS-Agreement document. For instance, the WSAC used by the WSAG4J framework[3] involves using JSDL as the language for specifying service description terms and JEXL as the language for specifying service level objectives. SWAPS [10] also proposes a WSAC by using WSDL-S/OWL as SubLs for the service functionality, and WSLA [8] expression language as SLOs SubL. A complete example of WSAC is included in Section IV-A.

These extension points are a great advantage of WS–Agreement since they make it possible to use WS–Agreement in an unbounded number of different ways [11]. However, as usual, this advantage comes with a price for WS–Agreement users: the difficulty of creating a general-purpose tool since such a tool should deal with the great variety of SubLs that can be used in different WS–Agreement documents. This problem gets even harder for conflict analysis since the tool should understand not only the SubLs syntaxes, but also their semantics.

In this paper, we face the aforementioned problems to enable the automated detection and explanation of conflicts in WS–Agreement documents. In particular, this paper contributes to the research on SLAs analysis in general and WS–Agreement in particular in the following ways:

First, we propose a novel classification of conflicts for SLAs as well as a rigorous definition for each of them. The classification identifies three kinds of conflicts that can be found in two possible scopes including the only conflict already identified in the literature (i.e., inconsistencies that affect the whole agreement) and those we have found while applying our proposal to several scenarios (cf. Sec. VI). Furthermore, the classification is WSAC-agnostic.

Second, we propose a technique based on constraint satisfaction problems (CSP) [12] that allows the use of any off-the-shelf CSP solver to automatically detect and explain these conflicts in WS–Agreement documents. The only requirement is that the semantics of the SubLs used in the WS-Agreement document can be interpreted as a CSP. This requirement is quite reasonable since, according to [4], it can be assumed that for automated analysis purposes any SLA can be interpreted as a CSP regardless of the specification language. In this paper, to illustrate this technique, we detail how this mapping can be done with WS–Agreement documents specified with a WSAC called iAgree [13]. The main features of the SubLs of iAgree are: (1) they are endowed with a precise semantics provided by the semantic mapping to CSPs (cf. Sec. IV-B), (2) they are general-purpose, meaning that they can be used regardless of the domain of the agreement, and (3) they are expressive enough to define many real-world SLAs (cf. Sec. VI).

The developed technique comes with a publicly-available reference implementation, SLA Explainer from now on, that support WS–Agreement documents specified with iAgree WSAC. Therefore iAgree is the first general-purpose WSAC proposed with support for conflicts detection and explanation. Furthermore, an advantage of using iAgree is that the mechanism to detect and explain conflicts can be directly used with any WS-Agreement document whose WSAC is iAgree or can be mapped into iAgree. In our validation, we have successfully mapped the WSACs used in WSAG4J framework and SWAPS (cf. Appendices A and E), which suggests that iAgree is generic enough to accommodate a significant variety of WSACs, although a detailed evaluation in this direction is out of the scope of this paper. We are convinced that as well as the inherent value of our techniques, the development of adaptable and easy to use open source tooling support has been a deciding factor for the adoption of SLA Explainer by users from different domains and with different WSACs and needs. We are aware that SLA Explainer is merely a first step, but it simplifies the development of tools that have to manage agreement documents in general and WS–Agreement documents in particular because it ensures the assumption that said documents are free of conflicts. SLA Explainer has been validated not only by other researchers in the community, but also in a real SOA-Governance system of our Regional Government, and in an end-user monitoring platform [14].

The paper is organized as follows. Section II provides an overview of WS–Agreement and introduces our definition of variants in a WS–Agreement document. Our conflict classification is described in Section III. Section IV discusses the process to automate the detection and explanation of conflicts by using CSP. Section V describes the major features and implementation issues of SLA Explainer and a prototypical implementation, while Section VI reports on its use in a number of different scenarios. Related work is discussed in Section VII and, finally, conclusions and future work are drawn in Section VIII.

II. Understanding WS–Agreement Documents

In this section, we briefly discuss the structure and semantics of the XML–based schema proposed in WS–

[3] WS–Agreement framework for Java developed by GRAAP members http://packcs-e0.scai.fraunhofer.de/wsag4j/index.html.

Agreement recommendation [6] for agreement documents as well as the SubLs that are necessary to get a WSAC. Fig. 2 depicts an example of WS–Agreement document using a human-friendly syntax instead of XML for the sake of clarity. The WSAC used in the example is our general-purpose WSAC so-called iAgree [13].

A. WS–Agreement Documents: An Overview

The WS–Agreement recommendation specifies two types of agreement documents to reach an agreement between the interested parties: *agreement offers* and *templates*.

The most common usage scenario of WS–Agreement starts with a pre-defined agreement template. A template is a partially completed offer that specifies customizable aspects of the documents, and rules that must be followed in creating an agreement, which are called agreement creation constraints. Fig. 2 depicts a template created by a company that provides a language translation service. Once the template is published, any customer demanding the translation service creates an agreement offer compliant with the published template and initiates the process by sending it to the provider. Finally, the provider either accepts or rejects the agreement offer. If accepted, such an offer becomes the agreement[4] that regulates the service provision. As can be seen, the customer initiates the interaction and therefore it plays the role initiator in the agreement creation process, whereas the provider plays the role responder.

Although the previous is the usual usage scenario, two additional considerations are included in WS–Agreement. First, the publication of a template is optional and thus, any party may send an agreement offer to other party without any template published. However, the acceptance of an agreement offer is more likely if it is defined based on a previously published template. Second, although the service provider usually acts as responder, WS-Agreement also allows consumers to play the role of responders by publishing templates with the service they intend to consume and some desired guarantees.

The general structure of WS–Agreement documents is depicted in Fig. 1 from an abstract point of view. The small white boxes represent the SubLs a user must define to describe the corresponding element of WS–Agreement documents. Thus, a WS–Agreement document is composed of an *agreement identifier*, an *agreement context*, and *agreement terms*. Templates can also include *agreement creation constraints* to restrict the space of *template–compliant* offers that can be created from them.

The agreement context contains information about whether the service provider is the agreement *initiator* or *responder*, which is the information that is included in Fig. 2. In addition, it optionally provides information about the agreement parties endpoints, the agreement's lifetime, references to the template from which the offer is created if this is the case; and any other relevant information such as the metrics datatype and domain included Fig. 2.

[4]From now on, we will use SLA and agreement as synonyms.

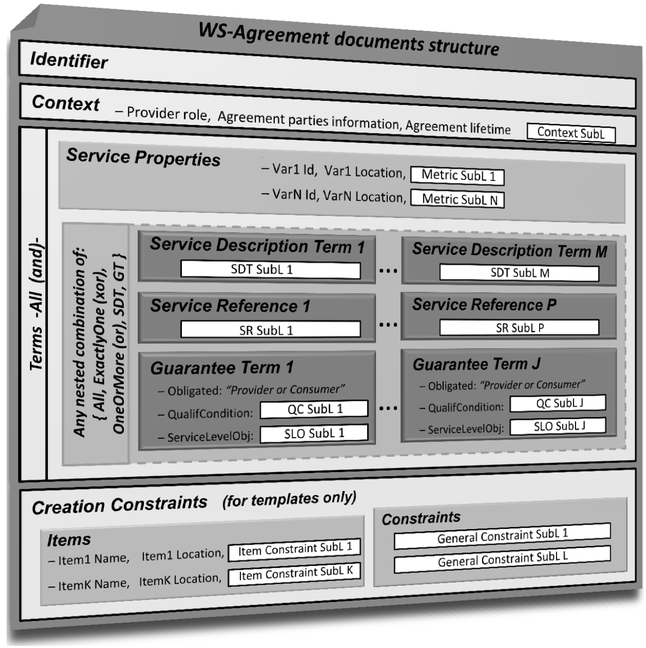

Fig. 1. WS–Agreement documents structure overview.

```
Template — Translate it!
  Context:
   Responder: Translator, as ServiceProvider
   Metrics [generalMetrics, Translation.metrics]
      Id, Uri: string
      Percent: float [0..100]
      Time: integer [1..100] //in minutes
      Translations:set {ES_to_EN–UK, ES_to_EN–US, ES_to_FR,
                        EN–UK_to_ES, EN–US_to_ES, FR_to_ES, Auto}

All
  SDT1: Service Description for TranslationService
      InputFile, OutputFile: Uri
      TranslationLangs: Translations = "ES_to_EN–UK";
      InputFileSize: integer
      ImageTranslation: bool
      Cost: integer

  SP1: Service Properties for TranslationService
      TranslationTime — measured by Time
                      — related to SDT1
      //Description: % of source text typos
      InputErrors — measured by Percent
                  — related to SDT1/InputFile
  SR1: Service Reference for TranslationService
    www.translator.com/translator.wsdl //endpoint

  GTTranslationTime1: Guaranteed by ServiceProvider
     QC: NOT ImageTranslation
     SLO: TranslationTime <= 1min

  GTTranslationTime2: Guaranteed by ServiceProvider
     QC: ImageTranslation
     SLO: TranslationTime <= 2min

  GTInputErrors: Guaranteed by ServiceConsumer
     SLO: InputErrors <= 1

Creation Constraints:
  C1: TranslationLangs belongs {ES_to_EN–UK, ES_to_FR,
                   EN–UK_to_ES, FR_to_ES, Auto};
  C2 [InputFileSizeConst]: InputFileSize <= 50 pages;
  C3 [CostConst]: Cost [0..unbounded];
  C4 [BasicCost]: Cost = InputFileSize * 1;
     onlyIf (NOT ImageTranslation);
  C5 [ImageCost]: Cost = InputFileSize * 2;
     onlyIf (ImageTranslation);
  C6 [MinSize]: NOT ImageTranslation;
     onlyIf (InputFileSize < 30);
```

Fig. 2. WS–Agreement template of the translation service scenario using a human-friendly syntax

The terms section of a WS–Agreement document describes both the characteristics of the services to be provided and the guarantees on such services. In WS–Agreement there are two kinds of terms, namely service terms and guarantee terms.

Service terms describe those service features that are relevant for an agreement and can be divided into *service description terms*, *service properties*, and *service references*. Service description terms (SDT) define the features of the service that will be delivered under an agreement. In addition, when they appear in templates, as in document of Fig. 2 for `TranslationLangs`, they are considered as default values that can be changed by the other party according to their domains and the constraints included in the template (see creation constraints at the end of the section). How these service description terms are organized and expressed must be defined in one or more SubLs. For instance, in our example, we use the iAgree SubL syntax to describe the translation service by means of a set of attribute–value pairs, namely: the supported translations (`TranslationLangs`), the input and output files, the size limit of the text to translate, whether the consumer wants to translate images or not, and the service cost. Service properties (SP) define named, service–related sets of monitorable *variables* that can be used for the specification of guarantee terms and must be therefore considered for agreement monitoring. All variables must be related to a service description term or a part of it and they may include a metric definition to specify the semantics and type of a variable with one or more SubLs. In our example, the context includes such metrics definition for the two defined properties: (1) `TranslationTime` represents the time the service translation takes; and (2) `InputErrors` represents the percentage of spelling and grammar errors in the `InputFile`. Finally, Service references provide endpoint references for the services under agreement and they also need one or more specific SubLs to be defined.

Guarantee terms (GT) describe the *SLOs* that an *obligated party*, usually the service provider, must fulfil as part of the agreement (see *guaranteed by...* in figures). The SLO of a guarantee term is an assertion defined over monitorable variables defined in the service properties section of the agreement document, and over external factors such as date, time, etc. Each guarantee term can be guarded by an optional *qualifying condition* (QC) that expresses a precondition under which the guarantee holds. Thus, using QCs is the way WS–Agreement allows for defining conditional terms. Both QCs and SLOs can be expressed using any suitable assertion language and so, users must specify SubLs for them. The structure of a guarantee term also includes a scope to which the guarantee is applied and a list of *business values*. However, these two elements are out of the scope of this paper. In our example, the provider assures diverse translation times depending on the selection of the translation of images. It also includes a guarantee term to assure that the source text to translate sent by the consumer has less than 1% errors of typos.

Finally, creation constraints specify the mandatory presence of specific elements in the offers created from the template, and their acceptable values. To this end, the template may define specific items using XML Schema [15] to delimit the possible value assignments for the service features defined in the SDTs or, if the creation constraint involves several elements, they can be specified using any suitable constraints language that must be described by means of one or more SubLs. Since items can be considered as a particular case of constraints, we just use the latter in Fig. 2. For instance, simple constraints are used to set the allowed translations, or the size of the text to be translated, while more advanced constraints depending on the selected optional features are also included to specify how the cost of the service is calculated.

B. Variants in a WS–Agreement Document.

The terms in a WS–Agreement document can be grouped using *and*, *or* and *xor*–like compositors. In an *and*–like (*All*) compositor, every comprised term or compositor is mandatory. i.e., all of them must be fulfilled. All terms at the top level of the terms section must be inside an *and*–like compositor, as depicted in Fig. 2. In an *or*–like (*OneOrMore*) compositor, every comprised term or compositor is optional. i.e., a set of them, at least one, must be fulfilled. Finally, in a *xor*–like (*ExactlyOne*) compositor, every comprised term or compositor is alternative. i.e., only one of them must be fulfilled.

Furthermore, term compositors can be nested, thus enabling the specification of alternative branches with potentially complex nesting within the agreement terms. Choices expressed using compositors can be exercised by the party that makes the next step in the agreement creation process, i.e., by the agreement initiator if it is creating an offer from a template, by the agreement responder if it is creating an agreement from an offer, or by the service provider if it is delivering the service according to a previously created agreement. For instance, document of Fig. 3 offers the alternative of either choosing GT1 or choosing GT2 and one or more terms between GT3 and GT4. SDT1 is not an optional term since it is in the top-level and-like compositor.

Note that parties can exercise the choice but it is not mandatory to do so. In other words, the term compositors can remain in an offer and even in a final agreement, exactly as they were defined in the former template.

Given this structure of term compositors in a WS–Agreement document, it is possible to analyse them to identify all of the sets of terms that can be chosen in the next step of the agreement creation process. For instance, from the document of Fig. 3, it is possible to choose the sets of terms depicted in Fig. 4. We call *variant* to each of those sets of terms that can be chosen in one WS–Agreement document. Consequently, term compositors can be seen as a means to define the variability of a WS–Agreement document in terms of the variants that can be chosen by the party that makes the next step in the agreement creation process. To obtain these variants it is necessary to follow a recursive procedure that iterates through the nested

```
Template – Translate it!
  Context:
    Responder: Translator, as ServiceProvider
...
All
    SDT1: Service Description for TranslationService
      ... //as in Figure 2
    Exactly One between:
      GT1: Guaranteed by ServiceProvider
        QC: TranslationLangs = FR_to_ES
        SLO: TranslationTime <= 2min
      All
        GT2: Guaranteed by ServiceProvider
          QC: TranslationLangs belongs {EN–UK_to_ES}
          SLO: TranslationTime <= 1min
        One Or More between:
          GT3: Guaranteed by ServiceProvider
            SLO: ImageTranslation
          GT4: Guaranteed by ServiceConsumer
            SLO: InputErrors <= 1%
```

Fig. 3. An example of nested term compositors

```
Variant 1: SDT1, GT1
Variant 2: SDT1, GT2, GT3
Variant 3: SDT1, GT2, GT4
Variant 4: SDT1, GT2, GT3, GT4
```

Fig. 4. Enumeration of all of the variants defined by the term compositors in document of Fig. 3

term compositors, and processes them according to their semantics. More specifically, let $variants(t)$ be a function that, given a term, returns the set of variants that it defines, then $variants(t)$ can be defined as follows:

If term t is not a composite term, then $variants(t) = \{t\}$.

If term t is a composite term, $C(T)$, where T is the set of composed terms, $T = \{t_1, \ldots, t_n\}$, then

$$variants(C(T)) = \left\{ \begin{array}{ll} \bigcup_{p \in \mathcal{P}(T) - \varnothing} variants(All(p)) & (1) \\ \{\{\bigcup_{i=1}^{n} j_i\} \mid \bigwedge_{i=1}^{n} j_i \in variants(t_i)\} & (2) \\ \bigcup_{i=1}^{n} variants(t_i) & (3) \end{array} \right\}$$

where $\mathcal{P}(S)$ is the power set of S and (1), (2) and (3) depends on the type of term compositor $C(t_1, \ldots, t_n)$:

(1) If term compositor C is *OneOrMore*, then every composed term is optional, but there must be at least one term selected. Therefore, the variants are all combinations of the composed terms. For instance, in the previous example, the variants of the one or more compositor are $\{GT3, GT4, \{GT3, GT4\}\}$. Note that, in the definition, the use of $variants(All(p))$ is necessary due to the nesting of term compositors, i.e., the terms nested in a one or more compositor could be term compositors themselves.

(2) If term compositor C is *All*, then every composed term is mandatory. Therefore, the variants should always contain one variant of each of the composed terms. For instance, in the previous example, the variants of the nested All compositor always contain one variant of $GT2$, which is $\{GT2\}$, and one variant of $OneOrMore(GT3, GT4)$, which are $\{GT3, GT4, \{GT3, GT4\}\}$. Consequently, the variants of this compositor are: $\{\{GT2, GT3\}, \{GT2, GT4\}, \{GT2, GT3, GT4\}\}$.

(3) If term compositor C is *ExactlyOne*, then every composed term is an alternative. Therefore, the variants of the ExactlyOne compositor are all of the variants of each composed terms. For instance, in the previous example, the variants of the ExactlyOne compositor are the union of the variants of $GT1$, which is $\{GT1\}$, and the variants of $All(OneOrMore(GT2, GT3))$, calculated above. Hence, the variants of the ExactlyOne compositor are: $\{GT1, \{GT2, GT3\}, \{GT2, GT4\}, \{GT2, GT3, GT4\}\}$.

III. Classification of Conflicts

In this paper, we propose a novel classification that includes all the conflicts that we have found in our use of WS–Agreement. Conflicts in WS–Agreement can be classified attending to two different criteria: the type of conflict and its scope.

A. Types of conflicts

All of the literature related to conflicts in SLAs focus on inconsistencies exclusively. However, the use of qualifying conditions in guarantee terms leads to other types of conflicts, namely dead terms and conditionally inconsistent terms. Moreover, according to the experience of users validating our proposal (see Sec. VI), these conflicts are more common than inconsistencies since they are harder to detect because they usually involve relationships amongst several properties. Next, we detail these types of conflicts.

Inconsistencies: A contradiction between terms, parts of terms, or creation constraints, and all of these amongst themselves (e.g., "`InputErrors < 1 & InputErrors = 1`" in the same expression) constitute an inconsistency of the WS–Agreement document. This means that it is impossible to find a satisfactory assignment to the variables that appear in those terms or creation constraints. The consequence is that the whole document (or one or more variants) is invalidated because it will never be fulfilled regardless of the way the service is provided. For instance, document of Fig. 5 includes an inconsistency between the SLO of `GT4` and the creation constraint `C2` because they state contradictory expressions. This contradiction may occur in the real world if the provider tries to obtain a minimum benefit by imposing a minimum file size in the `GT4` term, due to the translation service cost depends on such size (see the cost creation constraints of Fig. 2). However, the template owner skips the `C2` creation constraint by mistake.

Dead term: This conflict is caused when the condition of a conditional term never holds in a document or one or more variants. In other words, a dead term is a guarantee term whose qualifying condition has a contradiction with itself or one or more terms and/or creation constraints of the document making the term dead because its SLO can never be applied since its precondition never holds. For instance, document of Fig. 6, includes a dead term (`GT5`) because its qualifying condition can never be satisfied (by the SLO of the `GT3` guarantee term). This contradiction

```
Template – Translate it! version 1.1
...
   GT4: Guaranteed by ServiceConsumer
      SLO: InputFileSize > 30

Creation Constraints:
...
  C2: [InputFileSizeConst] InputFileSize <= 30 //in pages
...
```

Fig. 5. Document with an inconsistency between a GT and a creation constraint

```
Template – Translate it! version 1.2
...
   GT3: Guaranteed by ServiceConsumer
      SLO: InputErrors <= 1
...
   GT5: Guaranteed by ServiceProvider
      QC: InputErrors > 1
      SLO: TranslationTime <= 1min
...
```

Fig. 6. Document with a dead GT caused by another GT

```
Template – Translate it! version 1.3
...
   GT6: Guaranteed by ServiceConsumer
      SLO: InputFileSize >= 30
...
   GT12: Guaranteed by ServiceConsumer
      QC: TranslationLangs = FR_to_ES
      SLO: InputFileSize < 20
...
```

Fig. 7. Document with a conditionally inconsistent GT caused by another GT

```
Template – Translate it! version 1.4
...
   GT3: Guaranteed by ServiceConsumer
      SLO: InputErrors <= 1
...
   Exactly One between:
      GT1: Guaranteed by ServiceProvider
         QC: TranslationLangs = ES_to_DE
         SLO: TranslationTime <= 2min

      GT2: Guaranteed by ServiceProvider
         QC: TranslationLangs belongs {EN–UK_to_ES,
                                       FR_to_ES, Auto}
         SLO: TranslationTime <= 1min
...
Creation Constraints:
  C1: TranslationLangs belongs {ES_to_EN–UK, ES_to_FR,
                         EN–UK_to_ES, FR_to_ES, Auto};
  ...
```

Fig. 8. WS–Agreement document with a partial dead term originated by a contradiction between the QC of `GT1` and the creation constraint

may occur in the real world if the template owner is the provider and he/she only considers their guaranteed terms (`GT5`) while editing, skipping terms guaranteed by the other party (`GT3`), by mistake.

Conditionally inconsistent term: This conflict is caused when a conditional term makes the document inconsistent when its condition holds, which contradicts usual expectations. In other words, a conditionally inconsistent term is a guarantee term with the following characteristics: (1) it is not inconsistent, and (2) when its qualifying condition holds, then its SLO does not hold because of a contradiction within itself, with the qualifying condition or with other terms or creation constraints of the WS–Agreement document. Consequently, the conditionally inconsistent term is one that when the qualifying condition holds, the SLO and, hence, the guarantee term, does not hold. For instance, the `GT12` term of Fig. 7 is conditionally inconsistent because when its qualifying condition enables the SLO, such SLO is contradictory with the `GT6` SLO. This contradiction may occur in the real world if the template owner tries to obtain a minimum benefit by imposing a minimum file size (as in the `GT6` term), since the translation service cost depends on such size (see creation constraints of Fig. 2). However, in a further revision of the template, the provider decides to reduce the file size of `FR_to_ES` translations due to technical problems. The provider skips that `GT6` term applies to every language and therefore, the conditionally inconsistent GT12 does not allow reaching agreements for any `FR_to_ES` translations.

B. Scope of conflicts

As stated in Sec. II-B, term compositors enable the definition of variants in a WS–Agreement document. Therefore, depending on the term or terms that are involved in a conflict, it may affect all of the variants of a WS–Agreement document or just some of them.

Total conflicts: When a conflict affects all of the variants of a document. For instance, assuming that there are no term compositors in the examples of the previous section, the scope of all those conflicts is total.

Partial conflicts: When a conflict only affects a subset of the variants of a document. For instance, document of Fig. 8 has a set of terms composed by the ExactlyOne term compositor. It defines two variants, namely: `{GT3, GT1, C1}` and `{GT3, GT2, C1}`. The first variant includes a dead term because the QC of `GT1` never holds since it contradicts the creation constraint. However, in the second variant there is no conflict because it does not contain guarantee term `GT1`. Therefore the scope of the conflict is partial since it only affects one variant of the document.

Another example of partial conflicts can be found in document of Fig. 9. It has a set of optional guarantee terms (`GT1`, `GT2` and `GT3`) by the OneOrMore term compositor. However, `GT2` and `GT3` are contradictory. Therefore, every variant that contains both GTs is inconsistent. However, since they are optional, there are other variants in which they do not appear together. Consequently, in this document there is an inconsistency conflict with a partial scope.

IV. Automating the Detection and Explanation

In the previous section, we have provided a description of the different kinds of conflicts that can be found in a WS–Agreement document. However, its semantics has been defined in an intuitive way. In this section, we provide a precise definition of them by means of a semantic mapping. A semantic mapping is a way to provide semantics to a model, in this case WS–Agreement documents, by mapping the concepts into a semantic domain, i.e., a target domain whose semantics has been formally defined [16].

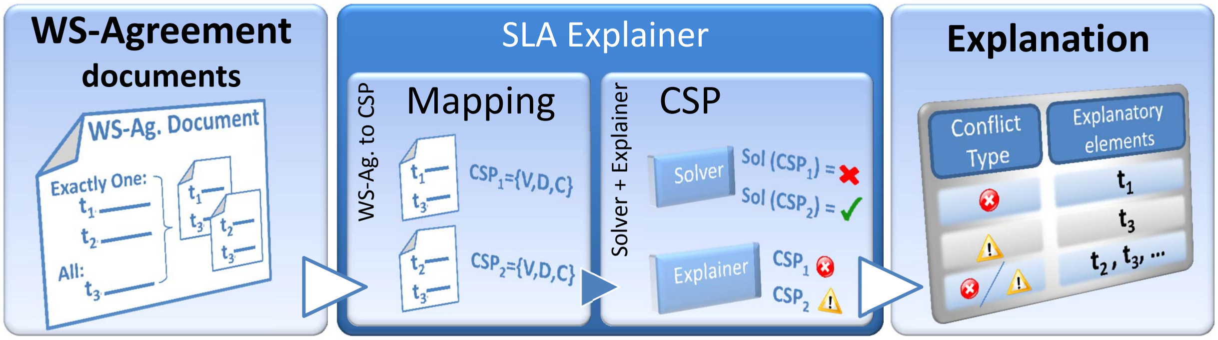

Fig. 10. Automated process to detect and explain conflicts in WS–Agreement documents.

```
AgreementOffer TranslationServiceOffer version 1.1
                for Template Translate it! version 1.0
...
    SP1: Service Properties for TranslationService
        ...
        //Description: % of target text typos
        OutputErrors - measured by Percent
                     - related to SDT1/OutputFile
...
    GTTranslationTime2: Guaranteed by ServiceProvider
        QC: ImageTranslation
        SLO: TranslationTime <= 2min
...
    One Or More between:
        GT1: Guaranteed by ServiceProvider
            SLO: OutputErrors <= 1%

        GT2: Guaranteed by ServiceConsumer
            SLO: InputErrors <= 1%

        GT3: Guaranteed by ServiceConsumer
            SLO: InputErrors > 1%
```

Fig. 9. WS–Agreement document with a partial inconsistency

The advantage of defining such semantic mapping is that it allows one to use the techniques specific to the target semantic domain for analysing the source models [17]. This can be done following a process such as that depicted in Fig. 10. In that process, a WS–Agreement model is mapped into a semantic domain, in which an automated technique is used to identify and explain the conflicts in a WS–Agreement model, and then, these results are traced back and returned as elements of the WS–Agreement metamodel.

There are several possible semantic domains into which WS–Agreement documents can be mapped such as description logics [10] or event calculus [18]. In our case, we have chosen constraint satisfaction problems (CSP) [12] as the semantic domain for our mapping. A CSP is a three–tuple of the form (V, D, C) where $V \neq \varnothing$ is a finite set of variables, $D \neq \varnothing$ is a finite set of domains (one for each variable) and C is a constraint defined on V. Consider, for instance, the CSP: $(\{a, b\}, \{[0, 2], [0, 2]\}, \{a + b < 4\})$. A solution of such CSP is whatever valid assignment of all elements in V that satisfies C. $(2, 0)$ is a possible solution of previous example since it verifies that $2 + 0 < 4$.

CSPs have been chosen because of two major reasons. Firstly, the most significant part of an agreement is a set of constraints that are set on some properties or descriptions of the service and, therefore, CSPs can be used to describe this problem in a very natural way as we have shown in previous work [4]. Secondly, there is a plethora of CSP solvers available that supports a wide range of constraints and can be used to automatically analyse the WS–Agreement document in an efficient manner.

A. *iAgree: An Intermediate WS–Agreement Configuration*

To enable the automated analysis of conflicts, the semantic mapping of a WS–Agreement document into a CSP must include both the mapping of the general-purpose schema and the mapping of each SubL used in the document since an important part of the semantics of a WS–Agreement document is included in the SubLs chosen for its extension points. However, it is not feasible to provide a mapping from all possible SubLs to a CSP. Therefore, a practical mechanism must be provided to enable SubLs developers to implement these mappings.

The solution we propose is to use an Intermediate WS–AGREEment configuration (iAgree). iAgree is a WSAC that has been designed with two main features. First, it is a general-purpose WSAC, which means that its SubLs can be used in any domain unlike other SubLs such as JSDL that are domain-specific. Second, the SubLs chosen in iAgree are generic and expressive enough to make it easier to map other WSAC into it. Appendices A and E include iAgree documents mapped from the WSACs used in WSAG4J framework and SWAPS [10], respectively. We conjecture that these mappings are easier to develop than direct mappings to CSPs because the concepts used and the result obtained (another WS–Agreement) are closer semantically than CSPs. Furthermore, WS–Agreement documents could be written directly in iAgree if it fits the users' needs. Following this approach, we just need to provide one unique full semantic mapping from iAgree WS–Agreement documents into CSPs. Other WSACs shall benefit of the semantics and automated conflict analysis of iAgree by defining a mapping from them to iAgree.

iAgree uses the following SubLs. The service description term SubL defines services as attribute–value pairs as it is depicted in Fig. 2 and also provide the domain of the attribute, i.e., a function $domain_D(attribute)$. Note that if an XML element can be flattened in terms of attribute–value pairs and its domain has been defined in a XMLSchema, then it can be easily mapped into this SubL. For instance,

V = { TranslationLangs, InputFile, OutputFile, InputFileSize
ImageTranslation, Cost, TranslationTime, InputErrors }
D = { set {ES_to_EN-UK, ... , Auto}, string, string, integer,
boolean, float, integer [1..100], float [0..100] }
C = { NOT ImageTranslation ⇒ TranslationTime<=1,
ImageTranslation ⇒ TranslationTime<=2, InputErrors<=1,
TranslationLangs=[ES_to_EN-UK,...,Auto], InputFileSize<=50,
Cost>0, NOT ImageTranslation ⇒ Cost=InputFileSize*1,
ImageTranslation ⇒ Cost=InputFileSize*2,
InputFileSize<30 ⇒ NOT ImageTranslation }

Fig. 11. CSP generated from mapping Template of Fig. 2

the JSDL example included in appendix B is flattened to attribute–value pairs in the `APPLICATION_STD_1` SDT in appendix A.

The metric SubL, which is used to define the metric of a service property, provides the domain of the service property, i.e., it describes its data type and its allowed values. From an abstract point of view, the SubL provides a function $domain_M(metric)$ to obtain such domain. For instance, Document of Fig. 2 specifies that service property `InputErrors` is measured by `Percent`. Therefore, as can be seen in the metric section of Fig. 2, there must be some metric model, that states `Percent` must be interpreted as the domain: "real value between 0 and 100", i.e., $domain_M(\texttt{Percent}) = \{x \in \mathbb{R} : x \geq 0 \land x \leq 100\}$.

The remaining SubLs are expression languages to specify SLOs, qualifying conditions, and creation constraints, respectively. In this case, the SubLs define assertions using logical, relational, and algebraic operators defined on the domain of the service descriptions, service properties and literals. In practice, the expression language used shall be restricted by the types of expressions that a particular CSP solver can use. Sec. V-B provides more details about the expression language that we have used in our tool.

B. iAgree to CSP Mapping

A WS–Agreement document Δ specified with iAgree WSAC describes a service by means of a set of attribute–value pairs and defines some service guarantees for a set of service properties whose domain has been previously defined. The semantics of such WS–Agreement documents can be modelled in a CSP by means of the semantic mapping $map(\Delta)$. The basic idea is to map the service descriptions and service properties into CSP variables, whereas GTs and creation constraints are mapped into CSP constraints. By doing so, the set of CSP solutions represents the range of values that may take the service descriptions and properties in the next step of the agreement creation process, i.e., when creating an offer from a template, or creating an agreement from an offer. This mapping is summarised in Table I and detailed as follows:

TABLE I
TERMS MAPPING FROM IAGREE TO CSP

iAgree Element	CSP Mapping
Template / AgreementOffer . . . **Context** : . . .	Not mapped into the CSP
SDT: **Service Description for** "name" attribute – **measured by** metric = value	*-in Templates-* $V \leftarrow V \cup attribute$ $D \leftarrow D \cup domain_D(attribute)$
	-in AgreementOffers- $V \leftarrow V \cup attribute$ $D \leftarrow D \cup domain_D(attribute)$ $C \leftarrow C \cup (attribute = value)$
SP: **Service Properties** property – **measured by** metric – **related to** . . .	$V \leftarrow V \cup property$ $D \leftarrow D \cup domain_M(metric)$ $C \leftarrow C$
SR: **Service Reference**	Not mapped into the CSP
GT: **Guaranteed by ServiceProvider / Consumer** **QC**: QCExpr **SLO**: SLOExpr	$V \leftarrow V$ $D \leftarrow D$ $C \leftarrow C \cup (QCExpr \Rightarrow SLOExpr)$
GT: **Guaranteed by ServiceProvider / Consumer** **SLO**: SLOExpr	$V \leftarrow V$ $D \leftarrow D$ $C \leftarrow C \cup (SLOExpr)$
Creation Constraints: name: GCExpr	$V \leftarrow V$ $D \leftarrow D$ $C \leftarrow C \cup (GCExpr)$

Service description terms are mapped into variables, domains and, if they are part of an offer, into constraints. Specifically, for each SDT attribute–value pair, the attribute is added into the set of variables of the CSP; its domain, which is obtained from function $domain_D$ defined in the SDT SubL, is added into the set of domains of the CSP, and if the document is an offer, a constraint is added into the set of constraints in order to assign the specified value to the variable. If the document is a template, SDTs represent default values and, hence, they are not mapped into constraints.

Service properties are mapped into CSP variables and their respective domain. The domain is obtained from the function $domain_M$ defined in the metric SubL. These variables are used in the guarantee terms. Note that we just depict the mapping of one variable in the Table for simplicity. If a service property defines more variables, then the same mapping should be applied for each variable. For instance the `InputErrors` property is mapped as a variable with the `[0..100]` domain as Fig. 11 shows.

Guarantee terms are always mapped as CSP constraints. If there is a qualifying condition, the guarantee term is mapped as an implication between qualifying condition and the SLO to represent the fact that the SLO can be applied only if the qualifying condition holds. For instance the `NOT ImageTranslation ⇒ TranslationTime<=1` constraint mapped in Fig. 11 from `GTranslationTime1` of Fig. 2. Otherwise, only the SLO expression is added into the set of constraints of the CSP. For instance the `InputErrors<=1` constraint mapped in Fig. 11 from `GTInputErrors` of Fig. 2.

Constraints are directly added into the CSP constraints. For instance, `InputFileSize<30 ⇒ NOT ImageTranslation` in Fig. 11. We do not include a mapping for items mentioned in Sec. II-A because we consider them a particular case of current constraints.

The mapping function $map(\Delta)$ assumes that there are no composite terms in the WS–Agreement document and, hence, all terms are composed by an `All` compositor. There are two approaches to take the composite term structure of WS–Agreement documents into account: The first one involves translating the semantics of term compositors into one CSP. In this case, the result is one CSP that uses logical operators OR and NOT to represent the semantics of the composite term structure of a WS–Agreement document. The second approach involves using the concept of variant to translate one WS–Agreement document into several CSPs, one for each variant of the document.

From a semantic point of view, both solutions are equivalent. However, from an operational point of view, the first approach makes it much harder to identify partial conflicts and to obtain explanations for the conflicts. Therefore, in this paper we take the second approach and define an extended mapping function for composite terms, $map_c(\Delta_c)$, as a function that, given a WS–Agreement document with composite terms Δ_c, returns a set of CSPs, one for each variant of Δ_c: $map_c(\Delta_c) = \{map(x) : x \in variants(\Delta_c)\}$, where $variants(\Delta_c)$ is the function that returns the set of variants defined by a WS–Agreement document Δ_c that was defined in Sec. II.

C. CSP-based analysis of conflicts

The advantage of the previous semantic mapping is that we can use the techniques specific to the semantic domain of CSPs to analyse the conflicts in a WS–Agreement document as depicted in Fig. 10. In our case, to analyse these conflicts we need two analysis techniques that have been widely used in CSPs: find solutions and explaining the lack of solutions. The former $solve(V, D, C)$ involves finding a valid assignment of values to all of the variables of the CSP V so that it satisfies its constraints C. To this end, many heuristics and techniques have been developed to obtain these solutions in an efficient manner. The latter technique $explain(V, D, C)$ involves providing an explanation when such solution is not possible. This explanation is a minimal set of constraints $c \subseteq C$ that makes it impossible to find a valid assignment of all elements in V that satisfies c, i.e., that makes $solve(V, D, c) = \varnothing$. For instance, for the CSP: $(\{a, b, d\}, \{[0..2], [0..2], [0..2]\}, \{a+b < 1, a = 1, d > 1\})$, the resulting c would be $\{a + b < 1, a = 1\}$ because the minimum allowed value for b is 0. On the basis of these two operations, we can provide a precise semantics to the conflicts described in Sec. III as follows:

Inconsistent terms. A WS–Agreement document has inconsistent terms, i.e., it has contradictions between its terms or creation constraints, if the mapped CSP has no solutions: $inconsistent(\Delta) \Leftrightarrow solve(map(\Delta)) = \varnothing$. For instance, the inconsistent term of template of Fig. 5 would be detected, by means of its CSP mapping as follows:

solve({InputFileSize}, {int}, {InputFileSize>=30,
InputFileSize<30, InputFileSize<=50}) = ∅

Furthermore, the inconsistent terms are explained by the result of tracing back the set of constraints c returned by $inconsistent_{exp}(\Delta) = explain(map(\Delta)) = c$

Then, previous example would be explained as follows:

explain({InputFileSize}, {int}, {InputFileSize>=30,
InputFileSize<30, InputFileSize<=50}) =
= {InputFileSize>=30, InputFileSize<30}

Dead terms. A guarantee term is a dead term if it can never be applied if all of the mandatory terms of the agreement are fulfilled, i.e., if its QC can never be true provided that the other terms of the agreement are fulfilled. Therefore, to detect that a term is dead, we just have to check whether its QC contradicts the remaining terms of the agreement. This can be expressed in terms of a CSP as follows: let GT_i be a guarantee term whose qualifying condition is QC_{GT_i}, GT_i is a dead term if adding its QC as a new constraint to the document it makes it inconsistent: $dead(GT_i, \Delta) \Leftrightarrow solve(map(\Delta)) \neq \varnothing \wedge solve(V, D, (C \setminus map(GT_i) \cup QC_{GT_i})) = \varnothing$. For instance, the dead term of template of Fig. 6 would be detected as follows:

solve({TranslationTime,InputErrors}, {int[1..100],float[0..100]},
{InputErrors>1⇒TranslationTime<=1, InputErrors<=1}) ≠ ∅
∧ **solve**({TranslationTime,InputErrors}, {int[1..100],
float[0..100]}, {InputErrors>1, InputErrors<=1}) = ∅

A dead term is explained by the result of tracing back the set of constraints c returned by: $dead_{exp}(GT_i, \Delta) = explain(V, D, (C \setminus map(GT_i) \cup QC_{GT_i})) = c$

Then, previous example would be explained as follows:

explain({TranslationTime,InputErrors}, {int[1..100],
float[0..100]}, {InputErrors>1,InputErrors<=1}) =
= {InputErrors>1,InputErrors<=1}

Conditionally inconsistent terms. A guarantee term is a conditionally inconsistent term if when its QC is true (i.e., it is enabled), its SLO is always false (i.e., it cannot be fulfilled). Consequently, to detect that a term is conditionally inconsistent, we have to check whether its QC and SLO contradict each other taking into account the other agreement terms. In terms of a CSP, this can be expressed as follows: let GT_i be a guarantee term whose qualifying condition is QC_{GT_i} and service level objective is SLO_{GT_i}, GT_i is a conditionally inconsistent term if: $condInconsistent(GT_i, \Delta) \Leftrightarrow solve(map(\Delta)) \neq \varnothing \wedge solve(V, D, (C \setminus map(GT_i) \cup QC_{GT_i} \cup SLO_{GT_i})) = \varnothing$. For instance, the conditionally inconsistent term of offer of Fig. 7 would be detected as follows:

solve({TranslationLangs,InputFileSize}, {set{ES_to_EN-UK,...
,Auto},int}, {TranslationLangs=FR_to_ES ⇒
⇒ InputFileSize<20, InputFileSize>=30}) ≠ ∅
∧ **solve**({TranslationLangs,InputFileSize},{set{ES_to_ EN-UK,...
,Auto},int}, {TranslationLangs=FR_to_ES,
InputFileSize<20, InputFileSize>=30}) = ∅

A conditionally inconsistent term is explained by tracing back the set of constraints c returned by: $condInconsistent_{exp}(GT_i, \Delta) = explain(V, D, (C \setminus$

$map(GT_i) \cup QC_{GT_i} \cup SLO_{GT_i})) = c$

Then, previous example would be explained as follows:

explain({TranslationLangs,InputFileSize},{set{ES_to_EN-UK,... ,Auto},int}, {TranslationLangs=FR_to_ES, InputFileSize<20, InputFileSize>=30}) =
= {InputFileSize<20, InputFileSize>=30}

These definitions can be easily extended for a document with term compositors by obtaining the variants of the document and, then, applying the previous definitions to each of the variants of the document. If the same conflict is found in all of the variants, then the scope of the conflict is total. If the conflict is found in, at least one of the variants, but not all of them, then the scope is partial.

For instance, the partial dead term of template of Fig. 8 would be detected by means of the CSP mapping of its two variants $map(\Delta_1)$ and $map(\Delta_2)$:

$map(\Delta_1)$ = ({TranslationTime, TranslationLangs},
{int[1..100], set{ES_to_EN-UK,...,Auto}},
{TranslationLangs=ES_to_DE $\Rightarrow$ TranslationTime<=2,
TranslationLangs in {ES_to_EN-UK,ES_to_FR,
EN-UK_to_ES,FR_to_ES,Auto}})
$map(\Delta_2)$ = ({TranslationTime, TranslationLangs},
{int[1..100], set{ES_to_EN-UK,...,Auto}},
{TranslationLangs in {EN-UK_to_ES, EN-US_to_ES,
FR_to_ES, Auto} $\Rightarrow$ TranslationTime<=1,
TranslationLangs in {ES_to_EN-UK,ES_to_FR,
EN-UK_to_ES,FR_to_ES,Auto}})

While $map(\Delta_2)$ has not conflicts, a partial dead term would be detected and explained in $map(\Delta_1)$, as follows:

solve($map(\Delta_1)$) $\neq \varnothing$
$\wedge$ **solve**({TranslationTime, TranslationLangs},
{ int[1..100], set{ES_to_EN-UK,...,Auto}},
{TranslationLangs=ES_to_DE,
TranslationLangs in {ES_to_EN-UK,ES_to_FR,
EN-UK_to_ES,FR_to_ES,Auto}})= $\varnothing$
explain({TranslationTime, TranslationLangs},
{ int[1..100], set{ES_to_EN-UK,...,Auto}},
{TranslationLangs=ES_to_DE
TranslationLangs in {ES_to_EN-UK,ES_to_FR,
EN-UK_to_ES,FR_to_ES,Auto}})=
{TranslationLangs=ES_to_DE
TranslationLangs in {ES_to_EN-UK,ES_to_FR,
EN-UK_to_ES,FR_to_ES,Auto}}

V. SLA Explainer

The technique developed in the previous section has a reference implementation named SLA Explainer, which automatically detects and explains all types of conflicts. Fig. 12 depicts the overall architecture of such implementation. SLA Explainer extends the desktop application developed as a proof-of-concept of the technique to detect and explain inconsistent terms in WS–Agreement developed

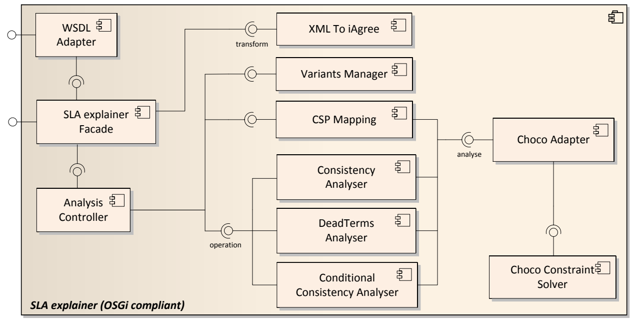

Fig. 12. SLA Explainer components diagram.

previously [1]. In this extension, external users have played a pivotal role as they have provided high–value input we have taken into account in order to prioritize some non–functional requirements. We hereby comment some of the aspects where the end-users input has been most important in the design of SLA Explainer.

A. Features

The main features provided by the components of SLA Explainer included in Fig. 12 are: (1) *ready-to-use* by the development of a user-friendly web application (see Sec. V-D) to test and use the conflict analysis provided by SLA Explainer with fewer effort; (2) *functional suitability* by supporting the analysis of expressive WS–Agreement documents with conditional and optional terms by the `Variants Manager`, arithmetic-logic expressions inside SLOs, etc [13]; (3) *understandability* by supporting a plain-text notation of iAgree [13] that makes reading and writing WS–Agreementdocuments easier for humans (such notation is serialised internally to the XML standard syntax of WS–Agreement by the `XML to iAgree` component); (4) *interoperability* through a triple distribution model (Java library, OSGi[5] service, and web service) through the `SLA explainer Facade` component; and (5) *CSP solver independence* by protecting our design from the possible variations derived from using a different CSP Solver than Choco constraint solver[6], which is the one SLA Explainer includes by default. For this reason, we have designed the interface `Analyser` with which interacts both the `CSP Mapping` component, which performs the semantic mapping function to map WS-Agreement documents into the concrete CSP used, and the components that analyse each conflict. In our implementation of SLA Explainer, interface `Analyser` is implemented by component `Choco Adapter`, which interacts with Choco constraint solver. Therefore, the only requirement to add support for a new CSP engine is to provide an implementation to interface `Analyser`.

[5]www.osgi.org
[6]Laburthe et al. Choco constraint solver. http://www.emn.fr/z-info/choco-solver/index.html

B. Implementation Issues

SLA Explainer allows the detection and explanation of all kind of conflicts mentioned in this paper for WS–Agreement documents specified with iAgree. However, some implementation issues such as the used CSP solver or the way service properties metrics are expressed, affect to the supported type of variables and constraints. For instance, the open source Choco CSP solver only implements complete support for integer variables. We have made the following implementation decisions on the iAgree SubLs.

As mentioned in Sec. IV-A, SDTs must be modelled as attribute–value pairs, independently of their domain–specific, internal (possibly hierarchical) organization. We consider that the advantages of using a generic SubL for SDTs are more relevant than the potential expressiveness of more specific SubLs like JSDL or WSDL for example. Applying this generic approach of attribute–value pairs in SDTs, we can automatically process any SLA document independently of the specific nature of the services to be agreed upon.

According to WS–Agreement, the (mathematical) domain of service property variables can be specified by *domain–specific metrics*. We have developed a simple yet effective metrics SubL for the definition of global metrics that can be used in iAgree or any other WS–Agreement document. Examples of the metrics definition are shown in the context of Fig. 2.

Finally, regarding logic assertions SubLs for guarantee Terms and creation constraint, we have designed a plain–text predicate–oriented SubL that can be easily fed into any CSP solver. The abstract syntax of iAgree of the predicate–oriented SubL is shown below, where *ID* is the identifier of an integer variable and *lit* is a literal value.[7]

$P ::= P\ op_L\ P \mid T$, predicate, where $op_L \in \{\wedge \mid \vee \mid \neg \mid \Rightarrow \mid \Leftrightarrow\}$
$T ::= E\ op_C\ E$, term, where $op_C \in \{ = \mid \neq \mid > \mid \geq \mid < \mid \leq \}$
$E ::= E\ op_A\ E \mid var \mid lit$, expression, where op_A is an algebraic operator defined on the domain of variables and literals

C. SLA Explainer Front-end

We have developed a front-end for iAgree that is available on-line and can be used from any platform including mobile devices. It uses the WSDL interface[8] to consume SLA Explainer as a web service. The front-end offers the following features (the number in brackets refers to the corresponding part of Fig. 13): (1) an user-friendly iAgree notation with syntax highlighting and a line numbered editor; (2) skeletons for the creation of agreement documents ("New template/offer" items of File menu), specially developed for users not familiar with WS–Agreement; (3) several document preloaded in order to reduce the learning curve (note that every example of current paper is included); (4) analysis operations to launch the conflict checking and explaining, and to get the number of variants; and (5) the result of the analysis is shown in a message (the message shown corresponds to the inconsistency explanation operation). Moreover, as Fig. 14 depicts using the document of Fig. 9 as example, we have developed a view to launch several analysis operations consecutively by using a simple JavaScript notation, and the result is shown in a log window.

[7]Technical report [13] includes several examples of predicates.
[8]http://www.isa.us.es:8081/ADAService?wsdl

Fig. 13. Edition capabilities of the iAgree front-end to try SLA Explainer (http://labs.isa.us.es/apps/iagreeeditor/)

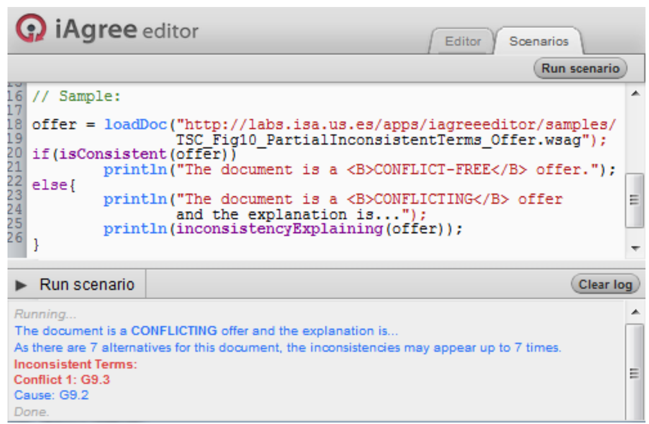

Fig. 14. Scenarios view of the iAgree front-end to try SLA Explainer (http://labs.isa.us.es/apps/iagreeeditor/)

D. SLA Explainer Verification

A test suite has been developed comprising 780 implementation–independent test cases [19] to verify the functionality of SLA Explainer. The test cases have been designed in terms of the inputs (i.e. an agreement offer) and expected outputs of the detection or explanation of conflicts under test. We used three popular black–box techniques from the software testing community to design our test cases, namely: equivalence partitioning, pairwise testing, and error guessing. The test cases helped to find: (1) a couple of errors detected while handling documents, for instance documents with variants were not correctly managed because a `NullPointerException` were thrown and it was solved with an adequate exception treatment; and

(2) a dozen of errors detected while analysing documents, for instance attribute-value pairs of SDTs in templates were considered as value assignments, instead of default values.

VI. Validation

An important validation effort has been carried out in the pursuit of real–world usefulness for SLA Explainer. As major conclusion we can claim that the results are useful not only for other colleagues in the community but also to be used in real end-user platforms and as a tool to help students. In the following, the main hints of this validation effort are highlighted.

Our first validation scenario was focused on the agreement creation process of WS–Agreement protocol. We revised WSAG4J, a framework developed by members of the GRAAP working group of the Open Grid Forum to provide an implementation of the WS–Agreement protocol. We noticed that WSAG4J lacked mechanisms to detect and explain conflicts in WS–Agreement documents which might led to undesired situations: SLAs with inconsistencies, dead terms and conditionally inconsistent term, and failures during the agreement creation process due to inconsistent templates or offers. It was necessary to build an adapter to SLA Explainer to deal with the documents written in the WSAC of WSAG4J. Two alternatives to build this adapter were considered: mapping the WSAC of WSAG4J to iAgree, or mapping the WSAC of WSAG4J to CSP. We chose the former for the sake of simplicity. Examples of this mapping can be found in Appendix A.

As a second validation effort we had the opportunity to use SLA Explainer in the SOA–Governance System used by our Regional Governmental Organization (see Appendix C). SLA Explainer was integrated into an SLA Management Infrastructure to automatically analyse the agreement documents and to enhance editing tools with debugging (detection and explanation) capabilities. During the evaluation we were aware of an unidentified kind of conflict that we coined as conditionally inconsistent term. We asked ourselves about the existence of other unidentified kinds of conflicts and we decided to conduct an experiment with our M.Sc students. The students had to choose a real-world service such as PayPal, Amazon EC2, Business Process Management Systems, NetOpen EAI, *et cetera* and create its corresponding template and a compliant offer. A total number of 46 WS–Agreement documents, 24 templates and 22 offers, were created. Amongst them, there were agreement documents with up to 14 service properties, 9 guarantee terms, 5 conditional guarantee terms, 12 variants, 4 items creation constraints or 14 general creation constraints. The conclusion we obtained from this exercise was that real-world SLAs make an intensive use of qualify conditions and variants in real-world SLAs. Furthermore, we also found another unidentified kind of conflict: dead term. All documents created by the students are available in a public repository[9].

[9]Available at http://www.isa.us.es/ada in the SLARepository link

VII. Related Work

This section is organized according to the two major contributions of this paper, namely: a conflict classification for WS–Agreement as well as a rigorous definition for each conflict, and a novel constraint satisfaction problem (CSP)–based technique to detect and explain every single type of conflict of that classification.

Although there are conflict taxonomies in other application domains, we have not found any one neither in SLA domain in general, nor WS–Agreement in particular. In fact, the only type of conflict we have found regarding SLAs is the inconsistency conflict. Our first steps in defining our classification were strongly influenced by [20]. In this work, an error classification of feature models and a CSP–based technique to automatically detect and explain errors in such models was proposed. A feature model allows to represent all the products of a Software Product Line in a compact way. Three major type of errors were identified in [20]: dead features, full-mandatory features and void model. A dead feature is a feature that despite of being defined in a feature model, it will never appear in a product in the software product line. We realized that this concept could be translated to our context so that we could say that a dead term is a term that despite of being defined in an agreement document, it will never be applied during the agreement lifetime. The adjective "dead" has been also used with a very similar meaning to denote activities that cannot be reached in web services choreographies because their executions would lead to violate at least one choreography constraint [21], so we consider that it is an adjective that perfectly grasps the idea we wished to transmit.

We have not found any other correspondence so direct in the case of inconsistencies and conditional inconsistencies, although we have found some very complete and complex taxonomies with very similar, but not the same, type of errors. For instance, in the field of Information System Management, according to the taxonomy proposed in [22], our inconsistencies and dead term conflicts can be considered as *mutex* and our conditionally inconsistent term conflict can be considered as *inconsistence configuration*. In order to avoid misunderstandings, we have decided to use "inconsistency" because it is commonly used in SLAs [1], and "conditionally inconsistent term" because it transmits the meaning of the error type in a more precise way.

As far as we know, our proposal is novel in giving rigorous semantics to WS-Agreements documents and their elements using a semantic mapping to CSP. The proposal of Oldham et al. [10] is the closest to ours because they provide semantics to an specific WSAC by means of ontologies to automate the agreement compliance checking between the parties at negotiation phase of the WS–Agreement life-cycle. However, they do not support nested term compositors. There are some previous formalisations of WS–Agreement [23], [24] but all of them are focused on the protocol-side of WS–Agreement where a formal definition of the offers and templates is not necessary.

The novel CSP–based technique to detect and explain the conflicts of our classification extends our preliminary proposal to detect and explain only inconsistent terms on WS–Agreement offers [1]. Although WS–Agreement templates, dead or conditionally inconsistent terms, and even conflict scope were not considered in such previous work, it is the unique proposal we find in the literature that proposes conflicts explanation in WS–Agreement.

There are also a couple of proposals that are able to detect inconsistencies but neither dead nor conditionally inconsistent terms conflicts. In the first one [4], we already proposed a constraint-based approach to detect only inconsistencies between terms; and in the second one [3], Braga et al. proposes a state-search and model-checking technique that is able to analyse SLA models detecting inconsistent SLOs. In both cases, authors dealt with non–WS–Agreement documents that do not include either conditional terms nor term compositors, although conditional terms could be expressed as logical entailment. Furthermore, they do not provide any explanation of conflicts.

Another problem that could be interpreted in terms of conflicts between several documents is the agreement violation detection during agreement monitoring. In this case, the conflicts are checked between the SLA previously agreed by parties and a document that includes the result of monitoring the agreed service. Some proposals that allow the detection of agreement violations are: [25]–[29]. As for proposals that not only detect, but also propose to carry out an action when a violation has been detected: [30] proposes to renegotiate the SLA; [31] proposes monetary penalties, impact on potential future agreements between the parties and the enforced re-running of the agreed service; [32] proposes actions which take into account not only penalty clauses, but rewards clauses also; and finally [18] provides explanation of violations based on the events of service–based systems. All these proposals can benefit from our SLA Explainer. Actually, proposal [14] extends the monitoring technique proposed in [28] with SLA Explainer to explain the violations, i.e., which metrics and SLOs have been violated.

Finally, there are approaches of the SLA domain in general, and WS–Agreement in particular, that provide detection and explanation for a different kind of conflicts; those not in a unique agreement document but between several documents. This is the case of [4] where a technique to determine the compliance between documents described in a non–WS–Agreement notation is introduced. In [33] this technique is revised to be applicable in WS–Agreement and a new technique to explain the non-compliance conflicts between templates and agreement offers is introduced.

VIII. Conclusions and Future Work

Three main conclusions can be drawn from this paper. First, the use of qualifying conditions and term compositors are at the root of two new kinds of conflicts: dead terms and conditionally inconsistent terms. Furthermore, the use of alternative term compositors leads to situations where the conflicts affect only partially the agreement document. In turn, in order to deal with partial conflicts the notion of variants in a WS–Agreement document have been rigorously defined. These findings are valid for any SLA specification model incorporating conditional and alternative terms.

Second, we show how constraint programming can be used to provide a rigorous definition for all kinds of conflicts in WS–Agreement documents identified to date. More specifically, the definitions rely solely on two analysis techniques widely used in constraint programming: finding solutions and explaining the lack of solutions. Both techniques are generally incorporated by many solvers which means our solution can be easily shared and reproduced.

Third, we conjecture that both the use of ad-hoc WSACs and the lack of a commonly accepted general-purpose WSAC, are delaying the adoption of WS–Agreement by researchers and practitioners. In this regard, we are confident that iAgree will provide a foundation on which general-purpose tools for WS–Agreement can be built. Furthermore, apart from our technique's inherent value, having developed open source tooling support which can be quickly integrated (as it is shown in [13]) and easy-to-use is a determining factor to achieve a useful result and settles the basis to spread the use of WS–Agreement.

As far as we know, this is the first work that explores the detection and explaining of conflicts in WS–Agreements documents with conditional and optional terms, and thus a lot of work remains to be done. We mention only a twofold direction: to explore the new conflicts that poses in dealing with temporal-aware terms [34], [35] and penalties [36].

Acknowledgments

We would like to thank for his helpful comments and revisions: F. Casati, A. Metzger, O. Wäldrich, W. Ziegler, P. Wieder, A. Durán, O. Martín-Diaz, P. Fernández, S. Segura, J. M. Alcaraz, M. Montali, M. Aiello, F. Raimondi, P. Dolog, and the technical staff for his work and dedication, specially to J. García and A. Jurado.

References

[1] C. Müller, A. Ruiz-Cortés, and M. Resinas, "An Initial Approach to Explaining SLA Inconsistencies," in *Proc. of the 6^{th} ICSOC'08*, ser. LNCS, vol. 5364, pp. 394–406.

[2] P. Plebani, A. Metzger and O. Sammodi, "Validated principles, techniques and methodologies for specifying end-to-end quality (S-Cube deliverable CD-JRA-1.3.6)," 2012.

[3] C. Braga, F. Chalub, and A. Sztajnberg, "A Formal Semantics for a Quality of Service Contract Language," *Electronic Notes in Theoretical Computer Science*, vol. 203, no. 7, pp. 103–120, 2009.

[4] A. Ruiz-Cortés, O. Martín-Díaz, A. Durán, and M. Toro, "Improving the Automatic Procurement of Web Services using Constraint Programming," *Int. Journal on Cooperative Information Systems*, vol. 14, no. 4, 2005.

[5] M. Comuzzi and B. Pernici, "A framework for qos-based web service contracting," *ACM Trans. Web*, vol. 3, pp. 10:1–10:52, 2009.

[6] Andrieux et al., "Web Services Agreement Specification (WS-Agreement) (v. gfd-r.192)," 2011, OGF - Grid Resource Allocation Agreement Protocol WG.

[7] C. K. et al., "SLA@SOI: SLA Management and Foundations," SLA@SOI consortium, Tech. Rep. Deliverable D.A5.a, 2009.

[8] A. Keller and H. Ludwig, "The WSLA Framework : Specifying and Monitoring Service Level Agreements for Web Services," *Network*, vol. 11, no. 1, pp. 57–81, 2003.

[9] Dominic Battré et al., "WS-Agreement Specification Version 1.0 Experience Document (v. gfd-e.167)," 2010, OGF - Grid Resource Allocation Agreement Protocol WG.

[10] N. Oldham, K. Verma, A. Sheth, and F. Hakimpour, "Semantic WS-Agreement Partner Selection," in *15^{th} International WWW Conf., 697–706*. ACM Press, 2006.

[11] D. Battré, M. Hovestadt, and O. Wäldrich, "Lessons learned from implementing ws-agreement," in *Grids and Service-Oriented Architectures for Service Level Agreements*.

[12] E. Tsang, *Foundations of Constraint Satisfaction*. A. Press, 1995.

[13] C. Müller, A. Durán, M. Resinas, A. Ruiz-Cortés, and O. Martín-Díaz, "Experiences from Building a WS–Agreement Document Analyzer Tool," ISA Research Group, Tech. Rep., 2010.

[14] C. Müller, M. Oriol, M. Rodríguez, X. Franch, J. Marco, M. Resinas, and A. Ruiz-Cortés, "SALMonADA: A platform for Monitoring and Explaining Violations of WS–Agreement–compliant Documents," in *4^{th} Workshop on Principles of Engineering Service–Oriented Systems (PESOS'12)*, 2012.

[15] W3C, "Xml schema part 0: Primer second edition," http://www.w3.org/TR/xmlschema-0/.

[16] D. Harel and B. Rumpe, "Meaningful modeling: What's the semantics of "semantics"?" *IEEE Computer*, vol. 37, no. 10, 2004.

[17] J. Rivera, E. Guerra, J. de Lara, and A. Vallecillo, "Analyzing rule-based behavioral semantics of visual modeling languages with Maude," in *Software Language Engineering 2008*, ser. Lecture Notes in Computer Science, vol. 5452/2009. Springer, 2009, pp. 54–73.

[18] K. Mahbub and G. Spanoudakis, "Monitoring ws-agreements: An event calculus-based approach," in *Test and Analysis of Web Services, Chapter 10*, 2007, pp. 265–306.

[19] C. Müller, S. Segura, and A. Ruiz-Cortés, "A test suite for agreement document analyser," ISA Research Group, Tech. Rep., 2012.

[20] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro, "Automated error analysis for the agilization of feature modeling," *Journal of Systems and Software*, vol. 81, no. 6, pp. 883–896, 2008.

[21] M. Montali, M. Pesic, W. M. P. v. d. Aalst, F. Chesani, P. Mello, and S. Storari, "Declarative specification and verification of service choreographiess," *ACM Trans. Web*, vol. 4, pp. 3:1–3:62, 2010.

[22] J. M. A. Calero, J. M. M. Pérez, J. B. Bernabé, F. J. G. Clemente, G. M. Pérez, and A. F. G. Skarmeta, "Detection of semantic conflicts in ontology and rule-based information systems," *Data K. Eng.*, vol. 69, no. 11, pp. 1117 – 1137, 2010.

[23] M. Aiello, G. Frankova, and D. Malfatti, "What's in an Agreement? An Analysis and an Extension of WS-Agreement." in *ICSOC'05*.

[24] G. D. Modica, O. Tomarchio, and L. Vita, "Dynamic SLAs Management in Service Oriented Environments," *Journal of Systems and Software*, vol. 82, pp. 759–771, 2009.

[25] F. Raimondi, J. Skene, and W. Emmerich, "Efficient Online Monitoring of Web-Service SLAs," in *16^{th} ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Atlanta, GE: ACM Press, Nov. 2009, pp. 170–180.

[26] J. Skene, D. Lamanna, and W. Emmerich, "Precise Service Level Agreements," in *26^{th} Intl. Conf. on Software Engineering*. IEEE Computer Society Press, May 2004, pp. 179–188.

[27] S. Lamparter, S. Luckner, and S. Mutschler, "Formal Specification of Web Service Contracts for Automated Contracting and Monitoring," in *40^{th} Hawaii International Conference on System Sciences*. IEEE Computer Society Press, 2007.

[28] M. Oriol, X. Franch, J. Marco, and D. Ameller, "Monitoring adaptable soa-systems using salmon," in *Workshop on Service Monitoring, Adaptation and Beyond (Mona+)*, 2008, pp. 19–28.

[29] C. Chen, L. Li, and J. Wei, "AOP Based Trustable SLA Compliance Monitoring for Web Services," in *Seventh International Conference on Quality Software, QSIC'07*, Oct. 2007, pp. 225–230.

[30] D. Gmach, S. Krompass, A. Scholz, M. Wimmer, and A. Kemper, "Adaptive quality of service management for enterprise services," *ACM Trans. Web*, vol. 2, pp. 8:1–8:46, March 2008.

[31] O. F. Rana, M. Warnier, T. B. Quillinan, F. Brazier, and D. Cojocarasu, "Managing violations in service level agreements," in *Grid Middleware and Services Chapter Title - Managing Violations in Service Level Agreements*, 2008, pp. 349–358.

[32] M. Schäfer, P. Dolog, and W. Nejdl, "An environment for flexible advanced compensations of web service transactions," *ACM Trans. Web*, vol. 2, pp. 14:1–14:36, May 2008.

[33] C. Müller, M. Resinas, and A. Ruiz-Cortés, "Explaining the Non-Compliance between Templates and Agreement Offers in WS-Agreement*," in *Proc. of the 7^{th} ICSOC'09*, ser. LNCS, vol. 5900.

[34] O. Martín-Díaz, A. Ruiz-Cortés, A. Durán, and C. Müller, "An approach to temporal-aware procurement of web services," in *Proc. of The 3^{rd} International Conference on Service-Oriented Computing (ICSOC)*, 2005, pp. 170–184.

[35] C. Müller, O. Martín-Díaz, A. Ruiz-Cortés, M. Resinas, and P. Fernández, "Improving Temporal-Awareness of WS-Agreement," in *Proc. of the 5^{th} International Conference on Service-Oriented Computing (ICSOC)*. Springer Verlag, 2007, pp. 193–206.

[36] O. F. Rana, M. Warnier, T. B. Quillinan, and F. M. T. Brazier, "Monitoring and reputation mechanisms for service level agreements," in *Grid Economics and Business Models (GECON)*, 2008, pp. 125–139.

Carlos Müller is a Research Assistant and member of the Applied Software Engineering Group (ISA, www.isa.us.es) at University of Sevilla, Spain. His current research line includes service oriented computing, specifically the automated analysis of service level agreements and the application of such analysis at SLA design and monitoring.

Manuel Resinas is a Lecturer at the University of Sevilla, Spain. He obtained his PhD in Computer Science from this University. His current research lines include analysis and management of service level agreements, business process compliance, and process performance management. Previously, he worked on automated negotiation of service level agreements.

Antonio Ruiz-Cortés is Associate Professor and Head of the Applied Software Engineering Group (ISA, www.isa.us.es) at University of Sevilla, Spain. He obtained his PhD (with Honours) in Computer Science from this University. His current research lines include service oriented computing, software product lines, and business process management. Previously, he worked on requirements engineering and multiparty interaction.