Chapter 8

Systemic Unification & Feature Logic

1. Introduction

WAG is a system for the representation and processing of information at various levels of representation. At each level, the same formalism is used: system networks and realisation statements represent the *structural potential* at that level, while systemic structures represent the instantial knowledge[1]. The WAG system thus deals with two types of information at each level:

- *potential* information: the resources - system networks with associated realisation statements and feature selection conditions[2];
- *instantial* information: systemic structures (representations) -- structures of units in relation; also textual fields.

In this thesis, I have reported on representations of lexico-grammatical structures, and semantic structures (ideational, interactional and textual).[3] In work so far unpublished, I have represented context, exchange-structure, and generic structure within the WAG formalism, showing that the formalism can be applied to a wide range of phenomena.

The sub-system within WAG which deals with the representation of potential and instantial information is called the *WAG Knowledge Representation System* (WAG-KRS). It is a generalised knowledge representation system, filling the same role as systems such as Loom (MacGregor & Bates 1987), PATR (Shieber 1984), FUF (Elhadad 1991), and TFS (Emele & Zajac 1990). WAG-KRS is based strongly on the Loom system, although oriented more towards the Systemic formalism.

The WAG-KRS includes a set of basic processes, for operations on system networks and systemic structures. These can be used to process representation on any linguistic level, since a common formalism is used throughout. These processes include:

- **Resource-Management Processes**: handling the reading in of resources from data files; the storing away of this information in storage tables; and providing functions to access the resources in various ways;
- **Feature-Based Processes**: processing of features and combinations of features, e.g., finding the path of a feature, the unification of feature-paths, etc.

---

[1]Where linguistic information is not naturally represented in terms of Systemic structures, the formalism has been extended to allow alternative representations. This is the case, for instance, with *textual fields* (see chapter 5), which allow the modelling of information as membership in a set, rather than through structural representation. The WAG constraint language has been extended to allow testing or assertion of membership in the various textual fields, e.g., to test whether a particular entity is in the *relevance* textual field.

[2]The lexicon is also part of the Systemic potential, but is not connected with any one stratum.

[3]Graphological structures are at present represented procedurally, as processes for graphological construction and analysis.

- **Structure-based Processes**: processing of Systemic-structures, e.g., assertion of structural information; unification of structures; negation of structural constraints, etc.

This chapter describes the representation and processing of linguistic information within WAG-KRS.

2. Representation and Processing of Systemic Features

2.1 Computational Representations of Paths

One of the recurrent tasks in Systemic processing is to find the logical consequences of a set of features.[4] For instance, if we specify that a unit has the feature *finite*, then the logic of WAG's system network implies that it must also have the features: *grammatical-unit, clause, clause-simplex,* and *full*. This set of features is derived through a process of *backward chaining* -- working backwards from a feature (or set of features) towards the root of the system network, and collecting the features that are met along the way. The set of features which is collected is called the *path* of the feature (the 'path' between the root and the feature).

In the simple case, a path can be represented as a conjunction of features, e.g.,

```
(:and unit grammatical-unit clause clause-simplex full finite)
```

In some cases, there are several paths between a feature and the root -- system entry-conditions can be disjunctive, allowing two or more paths back to the root. For instance, in the WAG grammar, the feature *subject-inserted* is a member of a system whose entry-condition is a disjunction between the features *finite-clause* and *nonfinite-with-subject* (e.g., *I asked the man to go*). The feature thus has two paths:

```
a) (:and unit grammatical-unit clause clause-simplex full
         finite-clause subject-inserted)
b) (:and unit grammatical-unit clause clause-simplex full
         nonfinite nonfinite-with-subject subject-inserted)
```

One means of representing disjunctive paths is called *Disjoint Normal Form* (DNF) -- a logical form allowing disjunctions only at the highest level of the form. A DNF expression is thus a disjunction of conjunctive forms. The disjunctive path above would thus be represented as follows:

```
(:or (:and unit grammatical-unit clause clause-simplex full
           finite subject-inserted)
     (:and unit grammatical-unit clause clause-simplex full
           nonfinite nonfinite-with-subject subject-inserted))
```

As the number of disjunctive entry conditions in a path grows, the number of alternatives in the DNF path also increases, at an exponential rate, a combinatorial explosion. DNF representation is thus costly in time (the expansion into DNF can require a lot of time) and space (the size of the resulting form can be very large).

Kasper (1987a, 1987b) developed an alternative means of representing feature-paths which requires less space to represent, and allows quicker processing. I call this *Kasper Normal Form* (KNF). This form:

---

[4] It could be claimed that the calculation of feature paths (backward chaining) is not necessary for generation, since Penman, for instance, does not use backward chaining. However, because of this, Penman does not handle preselection properly. Also, Penman's generation does depend on backward-chaining in the semantics, to find a concept's super-concepts, but this is handled by Loom.

"is based on an a normal form that divides each description into definite and indefinite components. The definite component contains no disjunction, and the indefinite component contains a list of disjunctions that must be satisfied." (Kasper 1989, p2).

The KNF form splits the complex path into two parts:

1) The features common to all paths (definite component);

2) For each path, the features unique to that path (indefinite component).

The disjunctive path introduced above is divided as follows:

Definite: (:and unit grammatical-unit clause clause-simplex full subject-inserted)
Indefinite: (:or finite
(:and nonfinite nonfinite-with-subject))

These can be put together as a single KNF logical form:

```
(:and (:and unit grammatical-unit clause clause-simplex
            full subject-inserted)
      (:or finite
           (:and nonfinite nonfinite-with-subject)))
```

The WAG system uses KNF representation for feature paths, because it allows faster unification than DNF. For fuller descriptions of KNF, see Kasper (1987a, 1987b).

2.2 Deriving Paths (Backward Chaining)

The calculation of the path of a feature is a recursive process:

a) Find the path of the entry condition of the feature's system,

b) Add the feature to the definite features of the path.

The entry condition will be either a single feature or a complex of features. If it is a single feature, its path is calculated as above. The entry condition may, however, be complex, e.g.,

```
(:and finite (:not modal) (:or active intransitive))
```

If the entry condition is logically complex, it is calculated as follows:

- **Conjunction**: Derive the paths of each element of the conjunction, and unify these together (see next section).
- **Disjunction**: Derive the paths of each element of the disjunction, and disjoin them (see Kasper 1987a, 1987b).
- **Negation:** The feature or feature-complex contained in the negation is negated (see section 2.4).

2.3 Path Unification

Two paths can be unified to produce a single path. Path unification is used for many purposes, for instance, to check that two feature-specifications are mutually consistent. The unification of two DNF forms requires the disjunctions in each form to be expanded out, often a costly process. KNF unification first unifies the definite components, and only proceeds to unify the indefinite components if the definite information is compatible. If the definite information is not compatible, then the expansion of the disjunctions is avoided.

Another factor which makes KNF unification efficient is that the expansion of the indefinite components can be delayed. When two KNF forms are unified, the disjunctions in each form are not multiplied out, but kept as a set of simultaneous disjunctions e.g.,

```
(:and Def1 (:or A B) (:or C D))  U  (:and Def2 (:or E F G))
       => (:and (Def1 U Def2) (:or A B) (:or C D) (:or E F G))
```

The unification is accepted or rejected just on the basis of the unification of the definite components only -- the indefinite components are not checked. At various opportunistic points, the KNF form can be fully expanded, to check whether it is in fact valid or not. However, the form often becomes invalid (on the basis of the definite component) before expansion can occur, thus avoiding the need for expansion completely.

The WAG feature unification algorithm follows Kasper (1987b). See that paper for a full description. A simplified version of the WAG algorithm follows:

1) Check the compatibility of the definite component of each form. If incompatible, fail.
2) Set the definite component of the result-form to be the union of the features in the two definite components.
3) Set the Indefinite component of the result-form to the set of the disjunctions (indefinite component) from the two forms, unexpanded.

2.4 Feature Negation

Often it is necessary to assert the negation of a feature or feature-complex. For instance, we might specify that a particular role is *not* filled by a unit of a particular type, e.g., *(:type Subject (:not plural-group))*. Negation is also often used in parsing -- if we want to assert that a particular realisation statement is not used, we negate the feature to which the realisation statement is attached.

Feature paths are not allowed to contain negation, so the negation-form is expanded out, being replaced with a form with no negation. The negation of a feature in a system network is easy to calculate. If we negate a feature, then we know that either the system is not entered at all (the entry-condition is not valid), or one of the other features in the system is selected. For instance, assume the system shown in figure 8.1. If we assert *(:not a)*, then either *(:not z)* is true, or one of *b* or *c* is true, e.g.,

```
(:not a)    =>   (:or b c (:not z))
```

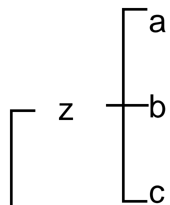

Figure 8.1: A Sample System

(:not z) can then be expanded in a similar way, until we reach a point where there is no negation left in the expression. The process will eventually get back to the root of the system network, and since the root feature of the network is by assumption true, the negation of this term simplifies to *nil.*

Negation of Feature Complexes: In the example above, the entry condition was a single feature. Often, entry conditions can be logically complex - consisting of disjunctions and conjunctions of features. Negated feature complexes can be expanded as follows:

(:not (:and A B)) => (:or (:not A) (:not B))
(:not (:or A B)) => (:and (:not A) (:not B))

2.5 Caching and Pre-calculation of Feature-paths

Feature Logic is a time-consuming process, and any step to reduce the amount of calculation is valuable. Higher efficiency can be achieved by avoiding repeated calculation of the same information. In the WAG system, some types of information are *cached* when first calculated, i.e., stored in a table. If the information is required again, it is recovered from the table rather than being re-calculated. In regards to feature logic, two types of information are cached:

- *The Path of Each System*: the path of each system is cached rather than the paths of individual features, since the path of all features in a system are identical, except for the feature itself. Since there are roughly half as many systems as features, the storage requirement is halved.
- *The Negation of Each System*: the negation of each system (more precisely, the negation of the entry condition of each system) is also cached as calculated. The negation of a feature is computed by disjoining the negation of the entry-condition with the other features in the system.

Caching still requires each calculation to be performed once during processing. For some often-used information, it is worthwhile to pre-cache (precompile) all information before processing begins. For instance, to speed up analysis, system-paths and system-negations are precompiled before parsing.

3. Representation and Processing of Systemic Structures

Systemic structures, on whichever stratum, are represented in the same manner. This section will describe the representation and processing of these structures. Systemic structures are composed of a set of *units*, which are joined together by a set of *relations*. These are the two primitives of systemic structure.

3.1 Relations

Relations join units together. The WAG system only recognises binary relations, which join one unit to another. Relations are always directed -- one unit is the *head* of the relation (the nucleus or domain), the other is the *dependent* (the satellite or range).[5] These relations are labeled in terms of the role played by the dependent unit: for instance, the relation between a process and its Actor is called an *Actor* relation.

3.2 Units

The representation of Systemic structures is based around units, not relations. The knowledge-base (the sum of all asserted structural representations) is stored as a *unit-table*, a data-structure holding the definition of each unit. A typical unit definition (for a speech-act) is shown below, and table 8.1 explains each of the fields:

[5] Non-directional relations are often used to represent the conjunction, or disjunction of units. In the WAG system, these are not represented using relations as such. Conjunction could be represented by allowing multiple occurrence of a relation, e.g., allowing two Actor roles to a process (although WAG does not yet allow this). The handling of structural disjunction in WAG will be discussed below.

Field	Description
:unit-id	A unique identifier for the unit -- used as the table key.
:features	A selection-expression for the unit, perhaps including disjunction (stored in Kasper-Normal Form).
:roles	The set of relations for which the unit is the head, along with the unit-id of the dependent of the relation (the filler).
:backpointers	The set of relations for which the unit is the dependent, along with the unit-id of the head of each relation.
:ordering	A data-structure indicating which units are known to be ordered prior, equal or after the unit (used, for instance, to order a unit relative to some number, or a point of time).
:cant-unify	A list of units with which this unit has been rerstricted from unifying with (see structural negation below).

Table 8.1: The Fields of a Unit Definition

```
#S(unit
  :id utterance-5
  :features (:and (root unit move speech-act inter-act negotiatory
                   negotiate-information propose initiate))
  :roles ((Speaking-Time unit1338)
          (Reference-Time unit1339)
          (Theme caller)
          (Proposition p5)
          (Speaker caller)
          (Hearer operator) )
  :backpointers nil
  :ordering nil
  :cant-unify nil)
```

Systemic structures are thus not represented explicitly as structures, but rather as a set of units, each unit stating its relations to other units. Relations are represented only within the *:roles* and *:backpointers* fields of the unit definitions.

3.3 Structure Unification

The most common structural operation involves the unification of units. Unification is performed as follows:

1. **Unify the feature-paths of the two units**

   If this fails, fail the unification,
   else, set the feature field of the first unit to the result.

2. **Add each role of the second unit to the first unit.**

   If the role already exists, unify the role-fillers.
   If the unification of roles fails, fail the whole unification.
   otherwise, the unification was successful.

3. **Make all Pointers to the second unit point at the first unit.**

   For each relation for which the second unit is the dependent (filler), alter the head unit to point to the first unit of the unified units.

4. **Delete the second unit.**

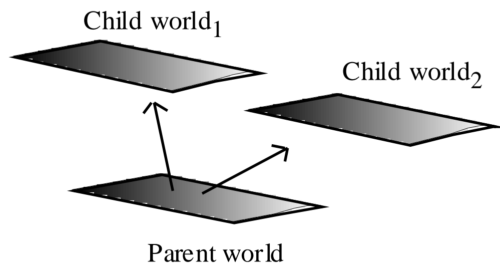

Figure 8.2: Showing Child-Worlds as a Disjunction over a Parent-World

The entry for the second unit in the unit-table is replaced with a pointer to the first unit. Thus, the unit-id of both the first and second units leads to the entry for the first, which now incorporates the sum information of both units.

3.4 Structural Disjunction

The WAG KRS allows disjunctive structural assertions to be made, e.g., the following asserts that a unit *p1* has either an Actor role (filled by a unit of type *female*), or a Senser role (filled by a unit of type *male*):

```
(tell p1 :constraint (:or (:type Actor female)
                          (:type Senser male)))
```

To handle *feature* disjunction, Kasper's KNF representation is used. However, this approach is not applicable for *structural* disjunction in WAG, since structural information is scattered throughout the unit table, rather than in a single structural form.

Structural disjunction in WAG is handled using a form of lazy copying called a *worlds* architecture.[6] This architecture assumes a *base-world*, the knowledge-base where definite structural information is asserted. When a disjunction is asserted, a *child-world* is created for each term in the disjunction, and the term asserted into that world. The child-worlds also inherit all the information asserted in the base-world (the parent-world of the child-worlds). The child-worlds are however invisible to each other, and information asserted into a child-world doesn't affect the parent-world. This situation is shown in figure 8.2. Note that child-worlds can have children of their own. The Possible-Worlds architecture offers a powerful means of handling structural disjunction in an efficient manner.

Each world is implemented as a unit-table, as described in section 3.2 above. The base-world table usually contains a large number of unit-definitions, while a child-world table only contains unit-definitions for entries which differ from its parent-world. This is called *lazy-copying* -- because only the information which differs is stored into the child-world.

This Worlds system allows us to select any existing world, and access or assert information into that world. If we wish to access a unit, what we retrieve depends on which world we are in. For instance, assume the example above, involving *p1*. If we are in the base-world, we would retrieve the information that *p1* is a unit, and no further information[7]. If we were in Child-World$_1$, we would recover that *p1* is a *material-process*, and has an *Actor* role, filled by a unit of type *female*. If we were in Child-

[6] I have learnt of the Worlds architecture through its use in the Loom knowledge representation system, although I do know of any documentation of Worlds.

[7] WAG does not automatically abstract the common information from a world's child-worlds.

$World_2$, we would recover that *p1* is a *mental-process*, and has an *Senser* role, filled by a unit of type *male*.

If we try to access a unit which is not defined in the current world, then the system will try to retrieve the unit from its parent world, as shown in figure 8.3. The system might have to look through several layers of worlds until the unit-definition is found.

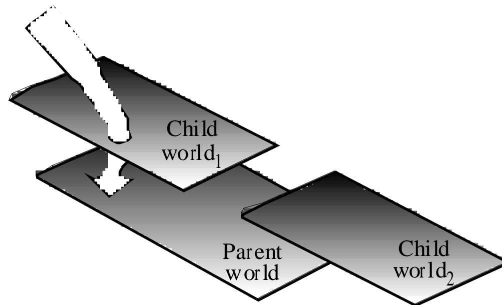

Figure 8.3: Information is Recovered from the Parent World if not found in the Child World.

3.5 Structural Negation

The WAG-KRL allows the assertion of negative statements, such as that a unit does not have a particular role, or that two units do not unify. Basically, anything which can be asserted in the positive can also be negated. The semantics of negation of different operators is described below:

Negated Require: *(:not (:require R))*: The current unit must have features which do not allow the insertion of role R. Negation of a role-chain is also possible

Negated Typing: *(:not (:type R f)) => (or (:not (:require R)) (:type R (:not f)))* Results in a disjunction: either the *:require* of the role is negated., or the Role is asserted to have a feature-description compatible with the negation of the feature.

Negated Same: *(:not (:same R1 R2))*: if the two references refer to the same unit, then the assertion will return *fail*. If the units are not at this point identical, then any attempt to unfiy the units in the future will fail. This is acheived through use of the *:cant-unify* field of the unit structure, which records the units which the unit cannot unify with.

Negated Fill: *(:not (:fill R unit-id))*: if the the unit-id is presently the filler of the role reference, then the assertion fails. Any attempt to make the unit the filler of that role later will also fail.

Negated Conjunction: *(:not (:and a b c)) => (:or (:not a) (:not b) (:not c))* The negation of a conjunction is the same as the disjunction of each of the terms negated.

Negated Disjunction: *(:not (:or a b c)) => (:and (:not a) (:not b) (:not c))* The negation of a disjunction is the same as the conjunction of each of the terms negated.

Negated Negation: *(:not (:not <constraint>)) => <constraint>*
Double negations cancel out.

4. WAG's Constraint Language

The WAG constraint language is used for several purposes, including the expression of realisation statements, and feature-selection constraints. It also allows users or programs to assert knowledge, e.g., to build up an ideation base. This section outlines the formalism of the language. In the following discussion, keep in mind that constraints are always interpreted in relation to some current unit -- structural assertions are assertions about a particular unit.

A constraint statement consists of an *operator* (e.g., *:order, :type, :same*, etc.), followed by a set of arguments. The arguments of the constraints vary, but typically, the first argument specifies some unit (for instance, a role of the current unit), and the second argument specifies some characteristic of the unit. (e.g., a feature it must have).

4.1 Referring to Units

Before exploring the various types of constraints, I will outline the various ways in which a unit can be referred to in a constraint:

Immediate Role: a role-name by itself refers to the filler of the role for the current unit, e.g., *Subject;*

Role-Chain: a chain of roles can refer to the role of a role, or the role of a role of a role, etc.; e.g., *Subject.Head;*

Back-pointer: a role-name preceded by '^' reverses the role, e.g., the ^*Subject* of a nominal-group point back to the clause. A back-pointer can occur in any position in a role-chain, e.g., *Referent.^Proposition.*;

Present unit: an '!' by itself refers to the current unit. Thus *(:type ! plural)* preselects the current unit to have a particular feature;

Reference through a Variable: a lisp variable delimited by '*' refers to the unit-id contained in the variable, e.g., given that **Speech-Act** is a variable which has previously been assigned the unit-id of the current speech-act, the speech-act can be assigned features using a constraint such as *(:type *Speech-Act* elicit)*.

4.2 Structural Operators

Tables 8.2, 8.3 and 8.4 show the operators used in WAG's constraint language. Table 8.2 shows the general operators, used on all strata. Table 8.3 shows some operators specific to textual representation, which were introduced in chapter 5. Table 8.4 shows operators which are grammar-specific -- only apply to grammatical structures. Most of the operators are standard Systemic realisation operators, as introduced in chapter 2, e.g., *insert, conflate, preselect, order*, *partition* and *lexify*. Some operators are re-labeled using WAG's nondirectional and non-process-oriented labeling. In such cases, the standard Systemic labels are shown beneath. Wherever <*role*> appears in the argument list, any of the unit-reference forms mentioned in section 4.1 can be used.

Operator	Arguments	Example	Description
:require *(insert)*	<role>	(:require Senser)	The nominated role is required to be present.
:same *(conflate)*	<role1> <role2>	(:same Modal Finite)	The nominated roles are filled by the same element, e.g., in a modal clause, both the Modal and the Finite role are filled by the same unit.
:type *(preselect, classify, inflectify)*	<role> <feature-expr>	(:type Senser conscious)	The nominated role must be filled by a unit with the specified feature-expression. The feature-expression can be a single feature, or a logical combination of features, using any combination of *and*, *or* or *not*. For instance: Subject : (:and nominal-group (:or nominative accusative) (:not wh-head))
:restrict *(outclassify)*	<role> <feature-expr>	(:restrict Subject accusative)	The nominated role must not be filled by a unit with the specified feature-expression. The feature feature-expression can be logically complex.
:fill	<role> <filler>	(:fill Actor Fred)	The nominated role must be filled by the designated filler. The filler is typically a unit-id, but can be a string, or a number. If no unit exists with the unit-id, then a unit of the designated id will be created.
<, <=, =, >=, >, <>	<role1> <role2>	(> Speaking-Time Reference-Time)	Various ordering operators. The designated units must be ordered as specified. Grammatical ordering is handled by *:order* and *:partition* -- see below in grammatical operators.

Table 8.2 General Constraint Operators

Operator	Arguments	Example	Description
:relevant	<role>	(:relevant Referent)	The nominated unit must be on the list of relevant ideational entities.
:recoverable	<role>	(:recoverable Referent)	The nominated unit must be on the list of recoverable ideational entities.
:shared-entity	<role>	(:shared-entity Referent)	The nominated unit must be on the list of shared ideational entities.

Table 8.3 Textual Constraint Operators

Operator	Arguments	Example	Description
:order	<role-list>	(:order Subj Fin)	The filler of the first role is sequenced directly before the second. Any number of elements can be sequenced in a single statement. See chapter 2 for more details.
:partition	<role-list>	(:partition Process Manner)	The roles appears in the order specified, but not necessarily immediately adjacent to each other. See chapter 2 for more details.
:lexify	<role> <lex-id>	(:lexify Deictic the-det)	The role-filler is assigned a Lexical-Item role, which is filled by the lexical-item with the specified unit-id. The word-rank unit is assigned the grammatical features from the lexical-item. Lexify overrides any preselections which may apply to the same unit.
:presume	<role>	(:presume Subject)	The specified role, while present in the structure for ordering purposes, is for other purposes not present in the structure. For instance, presumed units are not realised in generation.

Table 8.4: Grammar-Specific Constraint Operators

Operator	Syntax	Description
:and	(:and <structural-constraint1> <structural-constraint2> ....)	All the constraints in the body of the conjunction are asserted. Returns nil if any fails.
:or	(:or <structural-constraint1> <structural-constraint2> ....)	Each of the constraints in the body of the disjunction are asserted disjunctively (see section on structural disjunction below). At least one of the constraints conjoined by this operator must be true. Return nil if all fail.
:not	(:not <structural-constraint>)	The structural constraint should fail.

Table 8.5: WAG's Logical Operators

4.3 Logical Operators

Constraints can be logically complex, consisting of a logical combination of constraint-statements, e.g.,

```
(:or (:type *speech-act* propose)
     (:not (:same *Speech-Act*.Hearer Agent.Referent)
```

Table 8.5 shows the logical operators used in the WAG constraint language.

5. Summary

WAG's implementation of the Systemic formalism handles the representation of both Systemic resources (potential), and Systemic structures and fields (instantials). The formalism can be used for representation on any stratum, which has been demonstrated in this thesis, in regard to semantics (ideational, interactional and textual), and lexico-grammar.

The WAG-KRS allows various processes involving the resources and structures, including feature operations (finding the path of a feature or complex of features, and the unification of such paths), and structure operations (structural unification, disjunction and negation).

These processes and representations form a platform on which various higher level processes can be constructed. Both sentence generation and sentence analysis, reported in the remaining chapters, utilise the functionalities of the WAG-KRS.

This is the first generalised knowledge representation system based on the Systemic formalism. Most implementations using Systemic grammar use processes which are specific for one processing purpose -- generation or analysis. Those systems which take a generalised approach have translated the Systemic resources into another, non-Systemic, formalism (e.g., Bateman *et al.* 1992; Kasper 1988a, 1989).

A Worlds Architecture has been used to represent disjunctive structural information. This has proven a powerful and efficient tool for representation in an area which is often problematic.

Structural negation is also implemented for a wide range of negation possibilities. There are, however, still a few areas where negation is not complete, e.g., the negation of the unification of two units does not stop those units from being unified in the future. Further work will address these problems.