Active Workspaces: Distributed Collaborative Systems based on Guarded Attribute Grammars

Eric Badouel
Inria and LIRIMA
Campus de Beaulieu
35042 Rennes, France
eric.badouel@inria.fr

Loïc Hélouët
Inria
Campus de Beaulieu
35042 Rennes, France
loic.helouet@inria.fr

Georges-Edouard Kouamou
ENSP and LIRIMA
BP 8390, Yaoundé, Cameroon
georges.kouamou@lirima.org

Christophe Morvan
Université Paris-Est
UPEMLV, F-77454
Marne-la-Vallée, France
christophe.morvan@u-pem.fr

Robert Fondze Jr
NsaiBirni
UY1 and LIRIMA
BP 812, Yaoundé, Cameroon
nsaiobby@gmail.com

ABSTRACT
This paper presents a purely declarative approach to artifact-centric collaborative systems, a model which we introduce in two stages. First, we assume that the workspace of a user is given by a mindmap, shortened to a map, which is a tree used to visualize and organize tasks in which he or she is involved, with the information used for the resolution of these tasks. We introduce a model of guarded attribute grammar, or GAG, to help the automation of updating such a map. A GAG consists of an underlying grammar, that specifies the logical structure of the map, with semantic rules which are used both to govern the evolution of the tree structure (how an open node may be refined to a subtree) and to compute the value of some of its attributes (which derives from contextual information). The map enriched with this extra information is termed an active workspace. Second, we define collaborative systems by making the various user’s active workspaces communicate with each other. The communication uses message passing without shared memory thus enabling convenient distribution on an asynchronous architecture. We present some formal properties of the model of guarded attribute grammars, then a language for their specification and we illustrate the approach on a case study for a disease surveillance system.

1. INTRODUCTION
This paper presents a modular and purely declarative model of artifact-centric collaborative systems, which is user-centric, easily distributable on an asynchronous architecture and reconfigurable without resorting to global synchronizations.

Traditional Case Management Systems rely on workflow models. The emphasis is put on the orchestration of activities involving humans (the stakeholders) and software systems, in order to achieve some global objective. In this context, stress is often put on control and coordination of tasks required for the realization of a particular service. Such systems are usually modeled with centralized and state-based formalisms like automata, Petri nets or statecharts. They can also directly be specified with dedicated notations like BPEL [33] or BPMN [22].

One drawback of existing workflow formalisms is that data exchanged during the processing of a task play a secondary role when not simply ignored. However, data can be tightly connected with control flows and should not be overlooked. Actually, data contained in a request may influence its processing. Conversely different decisions during the treatment of a case may produce distinct output-values.

Similarly, stakeholders are frequently considered as second class citizens in workflow systems: They are modeled as plain resources, performing specific tasks for a particular case, like machines in assembly lines. As a result, workflow systems are ideal to model fixed production schemes in manufactures or organizations, but can be too rigid to model open architectures where the evolving rules and data require more flexibility.

Data-centric workflow systems were proposed by IBM [32, 24, 10]. They put stress on the exchanged documents, the so-called Business Artifacts, also known as business entities with lifecycles. An artifact is a document that conveys all the information concerning a particular case from its inception in the system until its completion. It contains all the relevant information about the entity together with a lifecycle that models its possible evolutions through the business
process. Several variants presenting the life cycle of an artifact by an automaton, a Petri net [29], or logical formulas depicting legal successors of a state [10] have been proposed. However, even these variants remain state-based centralized models in which stakeholders do not play a central role.

Guard-Stage-Milestone (GSM), a declarative model of the lifecycle of artifacts was recently introduced in [25, 11]. This model defines Guards, Stages and Milestones to control the enabling, enactment and completion of (possibly hierarchical) activities. The GSM lifecycle meta-model has been adopted as a basis of the OMG standard Case Management Model and Notation (CMMN). The GSM model allows for dynamic creation of subtasks (the stages), and handles data attributes. Furthermore, guards and milestones attached to stages provide declarative descriptions of tasks inception and termination. However, interaction with users are modeled as incoming messages from the environment, or as events from low-level (atomic) stages. In this way, users do not contribute to the choice of a workflow for a process. The semantics of GSM models is given in terms of global snapshots. Events can be handled by all stages as soon as they are produced, and guard of a stage can refer to attributes of distant stages. Thus this model is not directly executable on a distributed architecture. As highlighted in [18], distributed implementation may require restructuring the original GSM schema and relies on locking protocols to ensure that the outcome of the global execution is preserved.

This paper presents a declarative model for the specification of collaborative systems where the stakeholders interact according to an asynchronous message-based communication schema.

Case-management usually consists in assembling relevant information by calling tasks, which may in turn call subtasks. Case elicitation needs not be implemented as a sequence of successive calls to subtasks, and several subtasks can be performed in parallel. To allow as much concurrency as possible in the execution of tasks, we favor a declarative approach where task dependencies are specified without imposing a particular execution order.

Attribute grammars [28, 34] are particularly adapted to that purpose. The model proposed in this paper is a variant of attribute grammar, called Guarded Attributed Grammar (GAG). We use a notation reminiscent of unification grammars, and inspired by the work of Deransart and Maluszynski [15] relating attribute grammars with definite clause programs.

A production of a grammar is, as usual, described by a left-hand side, indicating a non-terminal to expand, and a right-hand side, describing how to expand this non-terminal. We furthermore interpret a production of the grammar as a way to decompose a task (the symbol in the left-hand side of the production) into sub-tasks associated with the symbols in its right-hand side. The semantics rules basically serve as a glue between the task and its sub-tasks by making the necessary connections between the corresponding inputs and outputs (associated respectively with inherited and synthesized attributes).

In this declarative model, the lifecycle of artifacts is left implicit. Artifacts under evaluation can be seen as incomplete structured documents, i.e., trees with open nodes corresponding to parts of the document that remain to be completed. Each open node is attached a so-called form interpreted as a task. A form consists of a task name together with some inherited attributes (data resulting from previous executions) and some synthesized attributes. The latter are variables subscribing to the values that will emerge from task execution.

Productions are guarded by patterns occurring at the inherited positions of the left-hand side symbol. Thus a production is enabled at an open node if the patterns match with the corresponding attribute values as given in the form. The evolution of the artifact thus depends both on previously computed data (stating which production is enabled) and the stakeholder’s decisions (choosing a particular production amongst those which are enabled at a given moment, and inputting associated data). Thus GAGs are both data-driven and user-centric.

Data manipulated in guarded attributed grammars are of two kinds. First, the tasks communicate using forms which are temporary information used for communication purpose only, essentially for requesting values. Second, artifacts are structured documents that record the history of cases (log of the system). An artifact grows monotonically—we never erase information. Moreover, every part of the artifact is edited by a unique stakeholder—the owner of the corresponding nodes—hence avoiding edition conflicts. These properties are instrumental to obtain a simple and robust model that can easily be implemented on a distributed asynchronous architecture.

The modeling of a distributed collaborative system using GAG proceeds in two stages. First, we represent the workspaces of the various stakeholders as the collections of the artifacts they respectively handle. An artifact is a structured document with some active parts. Indeed, an open node is associated with a task that implicitly describes the data to be further substituted to the node. For that reason these workspaces are termed active workspaces. Second, we define collaborative systems by making the various user’s active workspaces communicate with each other using asynchronous message passing.

This notion of active documents is close to the model of Active XML introduced by Abiteboul et al. [2] which consists of semi-structured documents with embedded service calls. Such an embedded service call is a query on another document, triggered when a corresponding guard is satisfied. The model of active documents can be distributed over a network of machines [1, 23]. This setting can be instantiated in many ways, according to the formalism used for specifying the guards, the query language, and the class of documents. The model of guarded attribute grammars is close to this general schema with some differences: First of all, guards in GAGs apply to the attributes of a single node while guards in AXML are properties that can be checked on a complete document. The invocation of a service in AXML creates a temporary document (called the workspace) that is removed from the document when the service call returns. In GAGs, a rule applied to solve a task adds new children to the node, and all computations performed for a task are preserved in the artifact. This provides a kind of monotony to artifacts,
an useful property for verification purpose.

The rest of the paper is organized as follows. Section 2 introduces the model of guarded attribute grammars and focuses on their use to standalone applications. The approach is extended in Section 3 to account for systems that call for external services. In this context we introduce a composition of guarded attribute grammars. Some formal properties of guarded attribute grammars are studied in Section 4. Section 5 presents some notations and constructions allowing us to cope with the complexity of real-life systems. This specification language is illustrated in Section 6 on a case study for a disease surveillance system. Finally an assessment of the model and future research directions are given in conclusion.


test

2. GUARDED ATTRIBUTE GRAMMARS

This section is devoted to a presentation of the model of guarded attribute grammars. We start with an informal presentation that shows how the rules of a guarded attribute grammar can be used to structure the workspace of a stakeholder and formalize the workspace update operations. In two subsequent subsections we respectively define the syntax and the behavior of guarded attribute grammars. The section ends with basic examples used to illustrate some of the fundamental characteristics of the model.

2.1 A Grammatical Approach to Active Workspaces

Our model of collaborative systems is centered on the notion of user’s workspace. We assume that the workspace of a user is given by a mindmap—simply call a map hereafter. It is a tree used to visualize and organize tasks in which the user is involved together with information used for the resolution of the tasks. The workspace of a given user may, in fact, consist of several maps where each map is associated with a particular service offered by the user. To simplify, one can assume that a user offers a unique service so that any workspace can be identified with its graphical representation as a map.

For instance the map shown in Fig. 1 might represent the workspace of a clinician acting in the context of a disease surveillance system.

![Figure 1: Active workspace of a clinician](image)

The service provided by the clinician is to identify the symptoms of influenza in a patient, clinically examines the patient eventually placing him under therapeutic care, declaring the suspect cases to the Disease Surveillance Center (DCS), and monitoring the patient based on subsequent requests from the epidemiologist or the biologist.

Each call to this service, namely when a new patient comes to the clinician, creates a new tree rooted at the central node of the map. This tree is an artifact that represents a structured document for recording information about the patient all along being taken over in the system. Initially the artifact is reduced to a single (open) node that bears information about the name, age and sex of the patient. An open node, graphically identified by a question mark, represents a pending task that requires the attention of the clinician. In our example the initial task of an artifact is to clinically examine the patient. This task is refined into three subtasks: clinical assessment, initial care, and case declaration.

Our first goal is to ease the work of the clinician by avoiding a manual updating of the map. In order to automate transformations of the map we must first proceed to a classification of the different nodes—indicating their sort. Intuitively two open nodes are of the same sort when they can be refined by the same subtrees—they can have the same future. It then becomes possible, depending on the sort of an open node, to associate with it specific information—the attributes of the sort—and to specify in which way the node can be developed.

We interpret a task as a problem to be solved, that can be completed by refining it into sub-tasks using business rules. In a first approximation, a (business) rule can be modelled by a production $P : s_0 \rightarrow s_1 \cdots s_n$, expressing that task $s_0$ can be reduced to subtasks $s_1$ to $s_n$. For instance the production

$$
\text{patient} \rightarrow \text{clinical_assessment} \\
\text{initial\_care} \\
\text{case\_declaration}
$$

states that a task of sort patient, the axiom of the grammar associated with the service provided by the clinician, can be refined by three subtasks whose sorts are respectively clinical_assessment, initial\_care, and case\_declaration.

If several productions with the same left-hand side $s_0$ exist then the choice of a particular production corresponds to a decision made by the user. For instance the clinician has to decide whether the case under investigation has to be declared to the Disease Surveillance Center or not. This decision can be reflected by the following two productions:

$$
s\text{uspect\_case} : \text{case\_declaration} \rightarrow \text{follow\_up} \\
\text{b}engineering\_case : \text{case\_declaration} \rightarrow
$$

If the case is reported as suspect then the clinician will have to follow up the case according to further requests of the biologist or of the epidemiologist. On the contrary, if the clinician has described the case as benign, it is closed with no follow up actions. More generally the tasks on the right-hand side of each production represent what remains to be done to resolve the task on the left-hand side in case this production is chosen.

If $P$ is the unique production having $s_0$ in its left-hand side,
then there is no real decision to make and such a rule is interpreted as a logical decomposition of the task $s_0$ into subtasks $s_1$ to $s_n$. Such a production will automatically be triggered without human intervention.

Accordingly, we model an artifact as a tree whose nodes are sorted. We write $X :: s$ to indicate that node $X$ is of sort $s$. An artifact is given by a set of equations of the form $X = P(X_1,\ldots,X_n)$, stating that $X :: s$ is a node labeled by production $P : s \rightarrow s_1 \cdots s_n$ and with successor nodes $X_1 :: s_1$ to $X_n :: s_n$. In that case node $X$ is said to be a closed node defined by equation $X = P(X_1,\ldots,X_n)$ -- we henceforth assume that we do not have two equations with the same left-hand side. A node $X :: s$ defined by no equation (i.e. that appears only in the right hand side of an equation) is an open node. It corresponds to a pending task of sort $s$.

The lifecycle of an artifact is implicitly given by the set of productions:

1. The artifact initially associated with a case is reduced to a single open node.
2. An open node $X$ of sort $s$ can be refined by choosing a production $P : s \rightarrow s_1 \cdots s_n$ that fits its sort.

The open node $X$ becomes a closed node -- defined as $X = P(X_1,\ldots,X_n)$ -- under the decision of applying production $P$ to it. In doing so task $s$ associated with $X$ is replaced by $n$ subtasks $s_1$ to $s_n$ and new open nodes $X_1 :: s_1$ to $X_n :: s_n$ are created accordingly.

3. The case has reached completion when its associated artifact is closed, i.e. it no longer contains open nodes.

Using the productions, the stakeholder can edit his workspace -- the map -- by selecting an open node -- a pending task --, choosing one of the business rules that can apply to it, and inputting some values -- information transmitted to the system.

However, plain context-free grammars are not sufficient to model the interactions and data exchanged between the various tasks associated with open nodes. For that purpose, we attach additional information to open nodes using attributes. Each sort $s \in S$ comes equipped with a set of inherited attributes and a set of synthesized attributes. Values of attributes are given by terms over a ranked alphabet. Recall that such a term is either a variable or an expression of the form $c(t_1,\ldots,t_n)$ where $c$ is a symbol of rank $n$, and $t_1,\ldots,t_n$ are terms. In particular a constant $c$, i.e. a symbol of rank $0$, will be identified with the term $c()$. We denote by $\text{var}(t)$ the set of variables used in term $t$.

**Definition 2.1 (Forms).** A form of sort $s$ is an expression $F = s(t_1,\ldots,t_n)(u_1,\ldots,u_m)$ where $t_1,\ldots,t_n$ (respectively $u_1,\ldots,u_m$) are terms over a ranked alphabet -- the alphabet of attribute’s values -- and a set of variables $\text{var}(F)$. Terms $t_1,\ldots,t_n$ give the values of the inherited attributes and $u_1,\ldots,u_m$ the values of the synthesized attributes attached to form $F$.

(Business) rules are productions where sorts are replaced by forms of the corresponding sorts. More precisely, a rule is of the form

$$s_0(p_1,\ldots,p_n)(u_1,\ldots,u_m) \rightarrow s_1(t_1^{(1)},\ldots,t_n^{(1)})(y_1^{(1)},\ldots,y_m^{(1)})$$

$$\vdots$$

$$s_k(t_1^{(k)},\ldots,t_n^{(k)})(y_1^{(k)},\ldots,y_m^{(k)})$$

where the $p_i$’s, the $u_j$’s, and the $t_j^{(r)}$’s are terms and the $y_j^{(r)}$’s are variables. The forms in the right-hand side of a rule are tasks given by forms

$$F = s(t_1,\ldots,t_n)(y_1,\ldots,y_m)$$

where the synthesized positions are (distinct) variables $y_1,\ldots,y_m$ -- i.e., they are not instantiated. The rationale is that we invoke a task by filling in the inherited positions of the form -- the entries -- and by indicating the variables that expect to receive the results returned during task execution -- the subscriptions.

Any open node is thus attached to a task. The corresponding task execution is supposed (i) to construct the tree that will refine the open node and (ii) to compute the values of the synthesized attributes -- i.e., it should return the subscribed values. A task is enacted by applying rules. More precisely, the rule can apply in an open node $X$ when its left-hand side matches with task $s_0(d_1,\ldots,d_n)(y_1,\ldots,y_m)$ attached to node $X$. For that purpose the terms $p_i$’s are used as patterns that should match the corresponding data $d_i$’s. When the rules applies, new open nodes are created and they are respectively associated with the forms -- tasks -- in the right-hand side of the rule. The values of $u_j$’s are then returned to the corresponding variables $y_j$’s that subscribed to these values. For instance applying rule (see Fig. 2)

$$R : s_0(a(x_1,x_2))(b(y_1),y_2) \rightarrow s_1(c(x_1))(y_1') s_2(x_2,y_1')(y_2')$$

to a node associated with tasks $s_0(a(t_1,t_2))(y_1, y_2)$ gives rise to the substitution $x_1 = t_1$ and $x_2 = t_2$. The two newly-created open nodes are respectively associated with the tasks $s_1(c(t_1))(y_1')$ and $s_2(t_2,y_1')(y_2')$ and the values $b(y_1')$ and $y_2'$ are substituted to the variables $y_1$ and $y_2$ respectively.

This formalism puts emphasis on a declarative (logical) decomposition of tasks to avoid overconstrained schedules. Indeed, semantic rules and guards do not prescribe any ordering on task executions. Moreover ordering of tasks depend on the exchanged data and therefore are determined at run time. In this way, the model allows as much concurrency as possible in the execution of the current pending tasks.

Furthermore the model can incrementally be designed by observing user’s daily practice and discussing with her. We can initially let the user manually develops large parts of the map and progressively improve the automation of the process by refining the classification of the nodes -- improving
2.2 Guarded Attribute Grammars Syntax

Attribute grammars, introduced by Donald Knuth in the late sixties [28], have been instrumental in the development of syntax-directed transformations and compiler design. More recently this model has been revived for the specification of structured document’s manipulations mainly in the context of web-based applications. The expression grammarware has been coined in [27] to qualify tools for the design and customization of grammars and grammar-dependent softwares. One such tool is the UUAG system developed by Swierstra and his group. They relied on purely functional implementations of attribute grammars [26, 36, 5] to build a domain specific languages (DSL) as a set of functional combiners derived from the semantic rules of an attribute grammar [16, 36, 35].

An attribute grammar is obtained from an underlying grammar by associating each sort \( s \) with a set \( \text{Att}(s) \) of attributes—which henceforth should exist for each node of the given sort—and by associating each production \( P : s \rightarrow s_1 \ldots s_n \) with semantic rules describing the functional dependencies between the attributes of a node labelled \( P \) (hence of sort \( s \)) and the attributes of its successor nodes of respective sorts \( s_1 \) to \( s_n \). We use a non-standard notation for attribute grammars, inspired from [14, 15]. Let us introduce this notation on an example before proceeding to the formal definition.

**Example 2.2 (Flattening of a Binary Tree).** Our first illustration is the classical example of the attribute grammar that computes the flattening of a binary tree, i.e., the sequence of the leaves read from left to right. The semantic rules are usually presented as shown in Fig. 2.2. The sort \( \text{bin} \) of binary trees has two attributes: The inherited attribute \( h \) contains an accumulating parameter and the synthesized attribute \( s \) eventually contains the list of leaves of the tree appended to the accumulating parameter. Which we may write as \( t \cdot s = \text{flatten}(t) ++t \cdot h \), i.e., \( t \cdot s = \text{flatten}(t) ++h \). The semantics rules stem from the identities:

\[
\text{flatten}(t)=\text{flat}(t, \text{Nil})
\]
\[
\text{flat}((\text{Fork}(t_1, t_2), h))=\text{flat}(t_1, \text{flat}(t_2, h))
\]
\[
\text{flat}((\text{Leaf}_a, h))=\text{Cons}_a(h)
\]

We present the semantics rules of Fig. 2.2 using the following syntax:

- **Root**: \( \text{root}(x) \rightarrow \text{bin}(\text{Nil})(x) \)
- **Fork**: \( \text{bin}(x)(y) \rightarrow \text{bin}(z)(y) \quad \text{bin}(x)(z) \)
- **Leaf** \( a \): \( \text{bin}(x)(\text{Cons}_a(x)) \rightarrow \)

The syntactic categories of the grammar, also called its sorts, namely \( \text{root} \) and \( \text{bin} \) are associated with their inherited attributes—given as a list of arguments: \((t_1, \ldots, t_n)\)—and their synthesized attributes—the co-arguments: \((u_1, \ldots, u_m)\). A variable \( x \) is an input variable, denoted as \( x^i \), if it appears in an inherited attribute in the left-hand side or in a synthesized attribute in the right-hand side. It corresponds to a piece of information stemming respectively from the context of the node or from the subtree rooted at the corresponding successor node. These variables should be pairwise distinct. Symmetrically a variable is an output variable, denoted as \( x^o \), if it appears in a synthesized attribute of the left-hand side or in an inherited attribute of the right-hand side. It corresponds to values computed by the semantic rules and sent respectively to the context of the node or to the subtree rooted at the corresponding successor node. Indeed, if we annotate the occurrences of variables with their polarity – input or output– one obtains:

- **Root**: \( \text{root}(x^o) \rightarrow \text{bin}(\text{Nil})(x^o) \)
- **Fork**: \( \text{bin}(x^o)(y^i) \rightarrow \text{bin}(z^o)(y^i) \quad \text{bin}(x^o)(z^o) \)
- **Leaf** \( a \): \( \text{bin}(x^o)(\text{Cons}_a(x^o)) \rightarrow \)

And if we draw an arrow from the (unique) occurrence of \( x^o \) to the (various) occurrences of \( x^i \) for each variable \( x \) to witness the data dependencies then the above rules correspond precisely to the three figures shown on the left-hand side of Table 2.2.

End of Exple 2.2

Guarded attribute grammars extend the traditional model of attribute grammars by allowing patterns rather than plain
variables –as it was the case in the above example– to represent the inherited attributes in the left-hand side of a rule. Patterns allow the semantic rules to process by case analysis based on the shape of some of the inherited attributes, and in this way to handle the interplay between the data contained in the inherited attributes– and the control—the enabling of rules.

**Definition 2.3 (Guarded Attribute Grammars).**

Given a set of sorts \( S \) with fixed inherited and synthesized attributes, a **guarded attribute grammar** (GAG) is a set of rules \( R : F_0 \rightarrow F_1 \cdots F_k \) where the \( F_i :: s_i \) are forms. A sort is **used** (respectively **defined**) if it appears in the right-hand side (resp. the left-hand side) of some rule. A guarded attribute grammar \( G \) comes with a specific set of sorts \( \text{axioms}(G) \subseteq \text{def}(G) \setminus \text{Use}(G) \) —called the **axioms** of \( G \)— that are defined and not used. They are interpreted as the provided services. Sorts which are used but not defined are interpreted as external services used by the guarded attribute grammar. The values of the inherited attributes of left-hand side \( F_0 \) are called the **patterns** of the rule. The values of synthesized attributes in the right-hand side are variables. These occurrences of variables together with the variables occurring in the patterns are called the **input occurrences** of variables. We assume that each variable has at most one input occurrence.

**Remark 2.4.** We have assumed in Def. 2.3 that axioms do not appear in the right-hand side of rules. This property will be instrumental to prove that strong-acyclicity—a property that guarantee a safe distribution of the GAG on an asynchronous architecture—can be compositionally verified. Nonetheless a specification that does not satisfy this property can easily be transformed into an equivalent specification that satisfies it: For each axiom \( s \) that occurs in the right-hand side of the rule we add a new symbol \( s' \) that becomes axioms in the place of \( s \) and we add copies of the rules associated with \( s' \)—containing \( s \) in their left-hand side—in which we replace the occurrence of \( s \) in the left-hand side by \( s' \). In this way we distinguish \( s \) used as a service by the environment of the GAG —role which is now played by \( s' \)—from its uses as an internal subtask—role played by \( s \) in the transformed GAG.

A rule of a GAG specifies the values at output positions—value of a synthesized attribute of \( s_0 \) or of an inherited attribute of \( s_1, \ldots, s_n \). We refer to these correspondences as the **semantic rules**. More precisely, the inputs are associated with (distinct) variables and the value of each output is given by a term.

A variable can have several occurrences. First it may appear (once) as an input and it may also appear in output values. The corresponding occurrence is respectively said to be in an **input or in an output position**. One can define the following transformation on rules whose effect is to annotate each occurrence of a variable so that \( x' \) (respectively \( x'' \)) stands for an occurrence of \( x \) in an input position (resp. in an output position).

\[
(F_0 \rightarrow F_1 \cdots F_k) = (F_0') \rightarrow (F_1') \cdots (F_k')
\]

\[
(x(s_1, \cdots, s_n)(u_1, \cdots, u_m)) = s((?x(t_1), \cdots, ?x(t_n))(u_1, \cdots, u_m))
\]

\[
(x(s_1, \cdots, s_n)(u_1, \cdots, u_m)) = s((!x(t_1), \cdots, !x(t_n))(u_1, \cdots, u_m))
\]

\[
(x(s_1, \cdots, s_n)) \rightarrow c((!x(t_1), \cdots, !x(t_n)))
\]

\[
(!x) \rightarrow x'
\]

The conditions stated in Definition 2.3 say that in the labelled version of a rule each variable occurs at most once in an input position, i.e., that \( c(?(F_0), !(F_1), \ldots, !(F_k)) \) is an admissible labelling of the set of forms in rule \( R \) according to the following definition.

**Definition 2.5 (Link Graph).** A labelling in \( [?, !] \) of the variables \( \text{var}(F) \) of a set of forms \( F \) is **admissible** if the labelled version of a form \( F \in F \) is given by either \( ?F \) or \( !F \) and each variable has at most one occurrence labelled with \( ? \). The occurrence \( x' \) identifies the place where the value of variable \( x \) is defined and the occurrences of \( x'' \) identify the places where this value is used. The **link graph** associated with an admissible labelling of a set of forms \( F \) is the directed graph whose vertices are the occurrences of variables with an arc from \( v_1 \) to \( v_2 \) if these vertices are occurrences of the same variable \( x \), labelled \( ? \) in \( v_1 \) and \( ! \) in \( v_2 \). This arc, depicted as follows,

\[ x \xrightarrow{?} x \]

means that the value produced in the **source vertex** \( v_1 \) should be forwarded to the **target vertex** \( v_2 \). Such an arc is called a **data link**.

**Definition 2.6 (Underlying Grammar).** The underlying grammar of a guarded attribute grammar \( G \) is the context-free grammar \( \mathcal{U}(G) = (N, T, A, P) \) where

- the non-terminal symbols \( s \in N \) are the defined sorts,
- \( T = S \setminus N \) is the set of terminal symbols —the external services—,
- \( A = \text{axioms}(G) \) is the set of axioms of the guarded attribute grammar, and
- the set of productions \( P \) is made of the underlying productions \( \mathcal{U}(R) : s_0 \rightarrow s_1 \cdots s_k \) of rules \( R : F_0 \rightarrow F_1 \cdots F_k \) with \( F_i :: s_i \).

A guarded attribute grammar is said to be **autonomous** when its underlying grammar contains no terminal symbols.

Intuitively an autonomous guarded attribute grammar represents a standalone application: It corresponds to the description of particular services, associated with the axioms, whose realizations do not rely on external services.

### 2.3 The Behavior of Autonomous Guarded Attribute Grammars

Attribute grammars are applied to input abstract syntax trees. These trees are usually produced by some parsing
algorithm during a previous stage. The semantic rules are then used to decorate the node of the input tree by attribute values. In our setting, the generation of the tree and its evaluation using the semantic rules are intertwined since the input tree represents an artifact under construction. An artifact is thus an incomplete abstract syntax tree that contains closed and open nodes. A closed node is labelled by the rule that was used to create it. An open node is associated with a form that contains all the needed information for its further refinements. The information attached to an open node consists of the sort of the node and the current value of its attributes. The synthesized attributes of an open node are undefined and are thus associated with variables.

**Definition 2.7 (Configuration).** A configuration $\Gamma$ of an autonomous guarded attribute grammar is an $S$-sorted set of nodes $X \in \text{nodes}(\Gamma)$ each of which is associated with a defining equation in one of the following forms where $\varGamma$ is a set of variables associated with $\Gamma$:

- **Closed node:** $X = R(X_1, \ldots, X_k)$ where $X_i :: s_i$ for $1 \leq i \leq k$, and $\mathcal{U}(R) : s \mapsto s_1 \ldots s_k$ is the underlying production of rule $R$. Rule $R$ is the label of node $X$ and nodes $X_1$ to $X_k$ are its successor nodes.

- **Open node:** $X = s(t_1, \ldots, t_m)(x_1, \ldots, x_m)$ where $X$ is of sort $s$ and $t_1, \ldots, t_m$ are terms with variables in $\varGamma$ that represent the values of the inherited attributes of $X$, and $x_1, \ldots, x_m$ are variables in $\varGamma$ associated with its synthesized attributes. Each variable in $\varGamma$ occurs at most once in a synthesized position. Otherwise stated $\forall \exists F \in \Gamma$ is an admissible labelling of the set of forms occurring in $\Gamma$. A node is called a root node when its sort is an axiom. Each node is the successor of a unique node, called its predecessor, except for the root nodes that are the successor of no other nodes. Hence a configuration is a set of trees – abstract-syntax trees of the underlying grammar – which we call the artifacts of the configuration. Each axiom is associated with a map made of the artifacts of the corresponding sort. A map thus collects the artifacts corresponding to a specific service of the GAG.

In order to specify the effect of applying a rule at a given node of a configuration (Definition 2.11) we first recall some notions about substitutions.

**Recall 2.8 (on Substitutions).** We identify a substitution $\sigma$ on a set of variables $\{x_1, \ldots, x_k\}$, called the domain of $\sigma$, with a system of equations

$$\sigma(x_i) = x_i \in \varGamma \text{ for } 1 \leq i \leq k$$

The set $\varGamma(x_i) = \bigcup_{1 \leq i \leq k} \varGamma(x_i)$ of variables of $\sigma$, is disjoint from the domain $\varGamma(\sigma)$ of $\sigma$. Conversely a system of equations $\{x_i = t_i \mid 1 \leq i \leq k\}$ defines a substitution $\sigma$ with $\varGamma(x_i) = t_i$ if it is in solved form, i.e., none of the variables $x_i$ appears in some of the terms $t_i$. In order to transform a system of equations $E = \{x_i = t_i \mid 1 \leq i \leq k\}$ into an equivalent system $\{x_i = t_j' \mid 1 \leq j \leq m\}$ in solved form one can iteratively replace an occurrence of a variable $x_i$ in one of the right-hand side term $t_j$ by its definition $t_j$ until no variable $x_i$ occurs in some $t_j$. This process terminates when the relation $x_i > x_j \iff x_j \in \varGamma(\sigma(x_i))$ is acyclic. One can easily verify that, under this assumption, the resulting system of equations $SF(E) = \{x_i = t_i' \mid 1 \leq i \leq n\}$ in solved form does not depend on the order in which the variables $x_i$ have been eliminated from the right-hand sides. When the above condition is met we say that the set of equations is acyclic and that it defines the substitution associated with the solved form.

**End of Recall 2.8**

The composition of two substitutions $\sigma, \sigma'$, where $\varGamma(\sigma') \cap \varGamma(\sigma) = \emptyset$, is denoted by $\sigma \sigma'$ and defined by $\sigma \sigma' = \{x = t \sigma' \mid x = t \sigma \in \Gamma\}$. Similarly, we let $\iota \sigma$ denote the configuration obtained from $\Gamma$ by replacing the defining equation $X = F$ of each open node $X$ by $X = \iota x$. We now define more precisely when a rule is enabled at a given open node of a configuration and the effect of applying the rule. First, note that variables of a rule are formal parameters whose scope is limited to the rule. They can injectively be renamed in order to avoid clashes with variables names appearing in the configuration. Therefore we always assume that the set of variables of a rule $R$ is disjoint from the set of variables of configuration $\Gamma$ when applying rule $R$ at a node of $\Gamma$. As informally stated in the previous section, a rule $R$ applies at an open node $X$ when its left-hand side $s(p_1, \ldots, p_n)(u_1, \ldots, u_m)$ matches with the definition $X = s(d_1, \ldots, d_n)(y_1, \ldots, y_m)$, namely the task attached to $X$ in $\Gamma$.

First, the patterns $p_i$ should match with the data $d_i$ according to the usual pattern matching given by the following inductive statements

- $\text{match}(c(p_1', \ldots, p_n'), c'(d_1', \ldots, d_k'))$ with $c \neq c'$ fails
- $\text{match}(c(p_1', \ldots, p_n'), c(d_1', \ldots, d_k')) = \sum_{i=1}^k \text{match}(p_i', d_i')$ of the same sort – of the whole set of patterns $p_i$, the various substitutions $\text{match}(p_i, d_i)$, when defined, are indeed concerned with disjoint sets of variables. Note also that $\text{match}(c(c'), (c)) = \emptyset$.

**Definition 2.9.** A form $F = s(p_1, \ldots, p_n)(u_1, \ldots, u_m)$ matches with a service call $F' = s(d_1, \ldots, d_n)(y_1, \ldots, y_m)$ – of the same sort – when

1. the patterns $p_i$’s matches with the data $d_i$’s, defining a substitution $\sigma_{in} = \sum_{1 \leq i \leq n} \text{match}(p_i, d_i)$,
2. the set of equations $\{y_j = u_j \sigma_{in} \mid 1 \leq j \leq m\}$ is acyclic and defines a substitution $\sigma_{out}$.

The resulting substitution $\sigma = \text{match}(F, F')$ is given by $\sigma = \sigma_{out} \cup \sigma_{in} \sigma_{out}$.

**Remark 2.10.** In most cases variables $y_j$ do not appear in expressions $d_i$. And when it is the case one has only to check that patterns $p_i$’s matches with data $d_i$’s – substitution $\sigma_{in}$ is defined – because then $\sigma_{out} = \{y_j = u_j \sigma_{in} \mid 1 \leq j \leq m\}$ since the latter is already in solved form. Moreover $\sigma = \sigma_{out} \cup \sigma_{in}$ because variables $y_j$ do not appear in expressions $\sigma_{in}(x_i)$. **End of Remark 2.10**
Definition 2.11 (Applying a Rule). Let \( R = F_0 \rightarrow F_1 \ldots F_k \) be a rule, \( \Gamma \) be a configuration, and \( X \) be an open node with definition \( X = s(d_1, \ldots, d_n)(y_1, \ldots, y_m) \) in \( \Gamma \). We assume that \( R \) and \( \Gamma \) are defined over disjoint sets of variables. We say that \( R \) is enabled in \( \Gamma \) if \( \sigma(X) \) is an open node in \( \Gamma \). Then applying rule \( R \) at node \( X \) transforms configuration \( \Gamma \) into \( \Gamma' \), denoted as \( \Gamma[R/X] \Gamma' \), with \( \Gamma' \) defined as follows:

\[
\Gamma' = \{ X = R(X_1, \ldots, X_k) \} \\
\cup \{ X_1 = F_1 \sigma, \ldots, X_k = F_k \sigma \} \\
\cup \{ X' = F \sigma \mid (X' = F) \in \Gamma \wedge X' \neq X \}
\]

where \( \sigma = \text{match}(F_0, X) \) and \( X_1, \ldots, X_k \) are new nodes added to \( \Gamma' \).

Thus the first effect of applying rule \( R \) to an open node \( X \) is that \( X \) becomes a closed node with label \( R \) and successor nodes \( X_1 \) to \( X_k \). The latter are new nodes added to \( \Gamma' \). They are associated respectively with the instances of the \( k \) forms in the right-hand side of \( R \) obtained by applying substitution \( \sigma \) to these forms. The definitions of the other nodes of \( \Gamma \) are updated using substitution \( \sigma \) or equivalently \( \sigma_{\text{out}} \). This update has no effect on the closed nodes because their defining equations in \( \Gamma \) contain no variable.

We conclude this section with two results justifying Definition 2.11. Namely, Prop. 2.12 states that if \( R \) is a rule enabled in a node \( X_0 \) of a configuration \( \Gamma \) with \( \Gamma[R/X_0] \Gamma' \) then \( \Gamma' \) is a configuration: Applying \( R \) cannot create a variable with several input occurrences. And Prop. 2.15 shows that substitution \( \sigma = \text{match}(F_0, X) \) resulting from the matching of the left-hand side \( F_0 \) to the right-hand side \( X = s(d_1, \ldots, d_n)(y_1, \ldots, y_m) \) of a rule \( R \) with the definition \( X = s(d_1, \ldots, d_n)(y_1, \ldots, y_m) \) of an open node \( X \) is the most general unifier of the set of equations \( \{ p_i = d_i \mid 1 \leq i \leq n \} \cup \{ y_j = u_j \mid 1 \leq j \leq m \} \).

Proposition 2.12. If rule \( R \) is enabled in an open node \( X_0 \) of a configuration \( \Gamma \) and \( \Gamma[R/X_0] \Gamma' \) then \( \Gamma' \) is a configuration.

Proof. Let \( R = F_0 \rightarrow F_1 \ldots F_k \) with left-hand side \( F_0 = s(p_1, \ldots, p_n)(u_1, \ldots, u_m) \) and \( X_0 = s(d_1, \ldots, d_n)(y_1, \ldots, y_m) \) be the defining equation of \( X_0 \) in \( \Gamma \). Since the values of synthesized attributes in the forms \( F_1, \ldots, F_k \) are variables (by Definition 2.3) and since these variables are unaffected by substitution \( \sigma_m \), the synthesized attribute in the resulting forms \( F_1 \sigma_m \) are variables. The substitutions \( \sigma_m \) and \( \sigma_{\text{out}} \) substitute terms to the variables \( x_1, \ldots, x_k \) appearing to the patterns and to the variables \( y_1, \ldots, y_m \) respectively. Since \( x_i \) appears in an input position in \( R \), it can appear only in an output position in the forms \( !\{F_i\} \ldots !\{F_k\} \) and thus any variable of the term \( \sigma_m(x_i) \) will appear in an output position in \( !\{F_i \sigma_m\} \). Similarly, since \( y_i \) appears in an input position in the form \( !\{s(u_1, \ldots, u_m)(y_1, \ldots, y_m)\} \), it can only appear in an output position in \( !\{F\} \) for the other forms \( F \) of \( \Gamma \). Consequently any variable of the term \( \sigma_{\text{out}}(y_i) \) will appear in an output position in \( !\{F \sigma_{\text{out}}\} \) for any equation \( X = F \) in \( \Gamma \) with \( X \neq X_0 \). It follows that the application of a rule cannot produce new occurrences of a variable in an input position and thus there cannot exist two occurrences \( x' \) of a same variable \( x \) in \( \Gamma' \). Thus applying an enabled rule defines a binary relation on configurations.

Definition 2.13. A configuration \( \Gamma' \) is directly accessible from \( \Gamma \), denoted by \( \Gamma \vdash \Gamma' \), whenever \( \Gamma[R/X] \Gamma' \) for some rule \( R \) enabled in node \( X \) of configuration \( \Gamma \). Furthermore, a configuration \( \Gamma' \) is accessible from configuration \( \Gamma \) when \( \Gamma' \vdash \Gamma' \) where \( \vdash \) is the reflexive and transitive closure of relation \( \vdash \).

Recall that a substitution \( \sigma \) unifies a set of equations \( E \) if \( t \sigma = t' \sigma \) for every equation \( t = t' \) in \( E \). A substitution \( \sigma \) is more general than a substitution \( \sigma' \), in notation \( \sigma \leq \sigma' \), if \( \sigma' = \sigma \sigma'' \) for some substitution \( \sigma'' \). If a system of equations has a some unifier, then it has \( \sigma \vdash \) to a bijective renaming of the variables in \( \sigma \) a most general unifier. In particular a set of equations of the form \( \{ x_i = t_i \mid 1 \leq i \leq n \} \) has a unifier if and only if it is acyclic. In this case, the corresponding solved form is its most general unifier.

Recall 2.14 (On Unification). We consider sets \( E = E_0 \oplus E_{\infty} \) containing equations of two kinds. An equation in \( E_0 \), denoted as \( t \equiv u \), represents a unification goal whose solution is a substitution \( \sigma \) such that \( t \sigma = u \sigma \) - substitution \( \sigma \) unifies terms \( t \) and \( u \). \( E_{\infty} \) contains equations of the form \( x = t \) where variable \( x \) occurs only there, i.e., we do not have two equations with the same variable in their left-hand side and such a variable cannot either occur in any right-hand side of an equation in \( E_{\infty} \). A solution to \( E \) is any substitution \( \sigma \) whose domain is the set of variables occurring in the right-hand sides of equations in \( E_{\infty} \) such that the compound substitution made of \( \sigma \) and the set of equations \( \{ x = t \sigma \mid t = t \in E_{\infty} \} \) unifies terms \( t \) and \( u \) for any equation \( t \equiv u \) in \( E_0 \). Two systems of equations are said to be equivalent when they have the same solutions. A unification problem is a set of such equations with \( E_{\infty} = \emptyset \), i.e., it is a set of unification goals. On the contrary \( E \) is said to be in solved form if \( E_0 = \emptyset \), thus \( E \) defines a substitution which, by definition, is the most general solution to \( E \). Solving a unification problem \( E \) consists in finding an equivalent system of equations \( E' \) in solved form. In that case \( E' \) is a most general unifier for \( E \).

Martelli and Montanari Unification algorithm [30] proceeds as follows. We pick up non deterministically one equation in \( E_0 \) and depending on its shape apply the corresponding transformation:

1. \( c(t_1, \ldots, t_n) \equiv c(u_1, \ldots, u_n) \): replace it by equations \( t_1 \equiv u_1, \ldots, t_n \equiv u_n \).
2. \( c(t_1, \ldots, t_n) \equiv c'(u_1, \ldots, u_m) \) with \( c \neq c' \): halt with failure.
3. \( x \equiv x \): delete this equation.
4. \( t \equiv x \) where \( t \) is not a variable: replace this equation by \( x \equiv t \).
5. \( x \equiv t \) where \( x \notin \text{var}(t) \): replace this equation by \( x = t \) and substitute \( x \) by \( t \) in all other equations of \( E \).
6. \( x \equiv t \) where \( x \in \text{var}(t) \) and \( x \neq t \): halt with failure.
The condition in (5) is the occur check. Thus the computation fails either if the two terms of an equation cannot be unified because their main constructors are different or because a potential solution of an equation is necessarily an infinite tree due to a recursive statement detected by the occur check. System \( E' \) obtained from \( E \) by applying one of these rules, denoted as \( E \Rightarrow E' \), is clearly equivalent to \( E \). We iterate this transformation as long as we do not encounter a failure and some equation remains in \( E' \). It can be proved that all these computations terminate and either the original unification problem \( E \) has a solution—a unifier—and every computation terminates—and henceforth produces a solved set equivalent to \( E \) describing a most general unifier of \( E' \) or \( E \) has no unifier and every computation fails. We let
\[
\sigma = \text{mgu}(\{t_i = u_i\}_{1 \leq i \leq n}) \text{ if } \{t_i \overset{?}{=} u_i\}_{1 \leq i \leq n} \Rightarrow^{*} \sigma
\]

**End of Recall 2.14**

Note that (5) and (6) are the only rules that can be applied to solve a unification problem of the form \( \{y_i \overset{?}{=} u_i \mid 1 \leq i \leq n\} \), where the \( y_i \) are distinct variables. The most general unifier exists when the occur check always holds, i.e., rule (5) always applies. The computation amounts to iteratively replacing an occurrence of a variable \( y_i \) in one of the right-hand side term \( u_i \) by its definition \( u_i \) until no variable \( y_i \) occurs in some \( u_i \), i.e., (see Recall 2.8) when this system of equations is acyclic. Hence any acyclic set of equations \( \{y_i = u_i \mid 1 \leq i \leq n\} \) defines the substitution \( \sigma = \text{mgu}(\{y_i = u_i \mid 1 \leq i \leq n\}) \).

**Proposition 2.15.** If \( F_0 = s_0(p_1, \ldots, p_n)\langle u_1, \ldots, u_m \rangle \), left-hand-side of a rule \( R \), matches with the definition \( X = s_0(d_1, \ldots, d_n)\langle y_1, \ldots, y_m \rangle \) of an open node \( X \) then substitution \( \sigma = \text{match}(F_0, X) \) is the most general unifier of the set of equations \( \{p_i = d_i \mid 1 \leq i \leq n\} \cup \{y_j = u_j \mid 1 \leq j \leq m\} \).

**Proof.** If a rule \( R \) of left-hand side \( s_0(p_1, \ldots, p_n)\langle u_1, \ldots, u_m \rangle \) is triggered in node \( X_0 = s_0(d_1, \ldots, d_n)\langle y_1, \ldots, y_m \rangle \) then by Def. 2.11 \( \{p_i \overset{?}{=} d_i\}_{1 \leq i \leq n} \cup \{y_j \overset{?}{=} u_j\}_{1 \leq j \leq m} \Rightarrow^{*} \sigma_m \cup \{y_j \overset{?}{=} u_j \sigma_m\}_{1 \leq j \leq m} \) using only the rules (1) and (5). Now, by applying iteratively rule (5) one obtains
\[
\sigma_m \cup \{y_j \overset{?}{=} u_j \sigma_m\}_{1 \leq j \leq m} \Rightarrow^{*} \sigma_m \cup \text{mgu}(\{y_j = u_j \sigma_m\}_{1 \leq j \leq m})
\]
when the set of equations \( \{y_j = u_j \sigma_m\}_{1 \leq j \leq m} \) satisfies the occur check. Then \( \sigma_m + \sigma_{out} \Rightarrow^{*} \sigma \) again by using rule (5).

**Q.E.D.**

**Remark 2.16.** The converse of Prop. 2.15 does not hold. Namely, one shall not deduce from Proposition 2.15 that the relation \( \Gamma[R/X_0] \) is defined whenever the left-hand side \( \text{lhs}(R) \) of \( R \) can be unified with the definition \( \text{def}(X_0, \Gamma) \) of \( X_0 \) in \( \Gamma \) with
\[
\Gamma' = \{X_0 = R(X_1, \ldots, X_k)\} \cup \{X_i = F_1 \sigma, \ldots, X_k = F_k \sigma\} \cup \{X = F \sigma \mid (X = F) \in \Gamma \land X \neq X_0\}
\]
where \( \sigma = \text{mgu}(\text{lhs}(R), \text{def}(X_0, \Gamma)) \), and \( X_1, \ldots, X_k \) are new nodes added to \( \Gamma' \). Indeed, when unifying \( \text{lhs}(R) \) with \( \text{def}(X_0, \Gamma) \) one may generate an equation of the form \( x = t \) where \( x \) is a variable in an inherited data \( d \) and \( t \) is an instance of a corresponding subterm in the associated pattern \( p \). This would correspond to a situation where information is sent to the context of a node through one of its inherited attribute! Stated differently, with this alternative definition some parts of the pattern \( p \) could actually be used to filter out the incoming data value \( d \) while some other parts of the same pattern would be used to transfer synthesized information to the context.

**End of Remark 2.16**

### 2.4 Some Examples

In this section we illustrate the behaviour of guarded attribute grammars with two examples. Example 2.17 describes an execution of the attribute grammar of Example 2.2. The specification in Example 2.2 is actually an ordinary attribute grammar because the inherited attributes in the left-hand sides of rules are plain variables. This example shows how data are lazily produced and send in push mode through attributes. It also illustrates the role of the data links and their dynamic evolutions. Example 2.18 illustrates the role of the guards by describing two processes acting as coroutines. The first process sends forth a list of values to the second process and it waits for an acknowledgement for each message before sending the next one.

**Example 2.17 (Example 2.2 continued).** Let us consider the attribute grammar of Example 2.2 and the initial configuration \( \Gamma_0 = \{X_0 = \text{root}(\langle x \rangle), Y_0 = \text{toor}(\langle x \rangle)\} \) shown next
\[
X_0 : \text{root} \quad Y_0 : \text{toor}
\]

The annotated version \( \!\Gamma_0 = \{\!\langle F \rangle \mid F \in \Gamma_0\} \) of configuration \( \Gamma_0 \) is
\[
\!\Gamma_0 = \{X_0 = \text{root}(\langle x \rangle), Y_0 = \text{toor}(\langle x \rangle)\}
\]
The data link from \( x' \) to \( x \) says that the list of the leaves of the tree—that will stem from node \( X_0 \)—to be synthesized at node \( X_0 \) should be forwarded to the inherited attribute of \( Y_0 \).

This tree is not defined in the initial configuration \( \Gamma_0 \). One can start developing it by applying rule \( \text{Root} : \text{root}(\langle x \rangle) \rightarrow \text{bin}(\text{Nil})\langle u \rangle \) at node \( X_0 : \text{root} \). Actually the left-hand side \( \text{root}(\langle x \rangle) \) of rule \( \text{Root} \) matches with the definition \( \text{root}(\langle x \rangle) \) of \( X_0 \) with \( \sigma_m = \emptyset \) and \( \sigma_{out} = \{x = u\} \). Thus \( \!\Gamma_0[\text{Root}/X_0] \Gamma_1 \) where the annotated configuration \( \!\Gamma_1 \) is given in Figure 4.

**Figure 4: Configuration \( \!\Gamma_1 \)**
Note that substitution $\sigma_{\text{out}} = \{ x = u \}$ replaces the data link $(x', x)$ by a new link $(u', u)$ with the same target and whose source has been moved from the synthesized attribute of $X_0$ to the synthesized attribute of $X_1$.

The tree may be refined by applying rule $\text{Fork} : \text{bin}(x)(y) \to \text{bin}(z)(y) \text{bin}(x)(z)$ at node $X_1 :: \text{bin}$ since its left-hand side $\text{bin}(x)(y)$ matches with the definition $\text{bin}(\text{Nil})(u)$ of $X_1$ with $\sigma_{\text{in}} = \{ x = \text{Nil} \}$ and $\sigma_{\text{out}} = \{ u = y \}$. Hence $\Gamma_1[\text{Fork}/X_1] \Gamma_2$ where $\Pi_2$ is given in Figure 5.

![Figure 5: Configuration $\Gamma_2$](image)

Rule $\text{Leaf}_a : \text{bin}(x)(\text{Cons}_a(x)) \to$ applies at node $X_{12}$ since its left-hand side $\text{bin}(x)(\text{Cons}_a(x))$ matches with the definition $\text{bin}(\text{Nil})(z)$ of $X_{12}$ with $\sigma_{\text{in}} = \{ x = \text{Nil} \}$ and $\sigma_{\text{out}} = \{ z = \text{Cons}_a(\text{Nil}) \}$. Hence $\Gamma_2[\text{Leaf}_a/X_{12}] \Gamma_3$ where the annotated configuration $\Pi_3$ is given in Figure 6.

![Figure 6: Configuration $\Gamma_3$](image)

As a result of substitution $\sigma_{\text{out}} = \{ z = \text{Cons}_a(\text{Nil}) \}$ the value $\text{Cons}_a(\text{Nil})$ is transmitted through the link $(z', z)$ and this link disappears.

Rule $\text{Fork} : \text{bin}(x)(u) \to \text{bin}(z)(u) \text{bin}(x)(z)$ may apply at node $X_{11}$: its left-hand side $\text{bin}(x)(u)$ matches with the definition $\text{bin}(\text{Cons}_a(\text{Nil}))(y)$ of $X_{11}$ with $\sigma_{\text{in}} = \{ x = \text{Cons}_a(\text{Nil}) \}$ and $\sigma_{\text{out}} = \{ y = u \}$. Hence $\Gamma_3[\text{Fork}/X_1] \Gamma_4$ with configuration $\Pi_4$ given in Figure 7.

![Figure 7: Configuration $\Gamma_4$](image)

Using substitution $\sigma_{\text{out}} = \{ u = \text{Cons}_a(z) \}$ the data $\text{Cons}_a(z)$ is transmitted through the link $(u', u)$ which, as a result, disappears. A new link $(z', z)$ is created so that the rest of the list, to be synthesized in node $X_{112}$ can later be forwarded to the inherited attribute of $Y_0$.

Finally one can apply rule $\text{Leaf}_b : \text{bin}(x)(\text{Cons}_a(x)) \to$ at node $X_{112}$ since its left-hand side matches with the definition $\text{bin}(\text{Cons}_a(\text{Nil}))(z)$ of $X_{112}$ with $\sigma_{\text{in}} = \{ x = \text{Cons}_a(\text{Nil}) \}$ and $\sigma_{\text{out}} = \{ z = \text{Cons}_b(\text{Cons}_a(\text{Nil})) \}$.

Therefore $\Gamma_5[\text{Leaf}_b/X_{112}] \Gamma_6$ with configuration $\Pi_6$ given in Figure 9.

Now the tree rooted at node $X_0$ is closed –and thus it no longer holds attributes– and the list of its leaves has been entirely forwarded to the inherited attribute of node $Y_0$. Note that the recipient node $Y_0$ could have been refined in parallel with the changes of configurations just described.

*End of Exple 2.17*

The above example shows that data links are used to transmit data in push mode from a source vertex $v$ –the input occurrence $x'$ of a variable $x$ to some target vertex $v'$– an output occurrence $x'$ of the same variable. These links
shown on the middle of Figure 11.

Production \(?a : q_2(a(y))(a'(x')) \leftrightarrow q_2(a(y))(x')\) applies at node \(X_2\) of \(\Gamma_2\) because its left-hand side \(q_2(a(y))(x')\) matches with the definition \(q_2(a(y))(x)\) of \(X_2\) with \(\sigma_{in} = \{y = y'\}\) and \(\sigma_{out} = \{x = x'\}\). We get configuration

\[
\Gamma_3 = \left\{ \begin{array}{l}
X = X_1 || X_2, \ \ X_1 = !a(X_{11}), \ \ X_2 = ?a(X_{21}), \\
X_{11} = q_2(x)(y'), \ \ X_{21} = q_2'(y')(x')
\end{array} \right\}
\]

shown on the right of Figure 11.

The corresponding acknowledgment may be sent and received leading to configuration

\[
\Gamma_5 = \Gamma \cup \{X_{111} = q_1(x)(y), \ X_{211} = q_2(y)(x')\}
\]

where \(\Gamma = \{X = X_1 || X_2, \ X_1 = !a(X_{11}), \ X_2 = ?a(X_{21})\}\).

The process on the left may decide to end communication by applying production \(?stop : q_1(x')(\text{stop}) \leftrightarrow \text{at} \ X_{111}\) with \(\sigma_{in} = \{x' = x\}\) and \(\sigma_{out} = \{y = \text{stop}\}\) leading to configuration

\[
\Gamma_6 = \Gamma \cup \{X_{111} = \text{stop}, \ X_{211} = q_2'(\text{stop})(x)\}
\]

The reception of this message by the process on the right corresponds to applying production \(?stop : q_2'(\text{stop})'(y) \leftrightarrow \text{at} \ X_{211}\) with \(\sigma_{in} = \emptyset\) and \(\sigma_{out} = \{x = y\}\) leading to configuration

\[
\Gamma_7 = \Gamma \cup \{X_{111} = \text{stop}, \ X_{211} = ?\text{stop}\}
\]

Note that variable \(x\) appears in an input position in \(\Gamma_6\) and has no corresponding output occurrence. This means that the value of \(x\) is not used in the configuration. When production \(?stop\) is applied in node \(X_{211}\) variable \(y\) is substituted to \(x\). Variable \(y\) has an output occurrence in production \(?stop\) and no input occurrence meaning that the corresponding output attribute is not defined by the semantic rules. As a consequence this variable simply disappears in the resulting configuration \(\Gamma_7\). If variable \(x\) was used in \(\Gamma_6\) then the output occurrences of \(x\) would have been replaced by (output occurrences) of variable \(y\) that will remain undefined – no value will be substituted to \(y\) in subsequent transformations— until these occurrences of variables may possibly disappear.  

3. COMPOSITION OF GAG

In this section we define the behavior of potentially non-autonomous guarded attributed grammars to account for systems that call for external services: A guarded attribute grammar providing some set of services may contain terminal symbols, namely symbols that do not occur in the left-hand sides of rules. These terminal symbols are interpreted as calls to external services that are associated with some other guarded attribute grammar. We introduce a composition of guarded attribute grammars and show that the behavior of the composite guarded attribute grammar can be recovered from the behavior of its components.

Recall that the behavior of an active workspace is given by a guarded attribute grammar \(G\). Its configuration is a set of
maps associated with each of the axioms, or services, of the grammar. A map contains the artifacts generated by calls to the corresponding service. We assume that each active workspace has a namespace \( ns(G) \) used for the nodes \( X \) of its configuration, the variables \( x \) occurring in the values of attributes of these nodes, and also for references to variables belonging to others active workspaces –its subscriptions. Hence we have a name generator that produces unique identifiers for each newly created variable of the configuration. Furthermore, we assume that the name of a variable determines its location, namely the active workspace it belongs to. A configuration is given by a set of equations as stated in Definition 2.7 with the following changes.

1. A node associated with a terminal symbol is associated with no equation –it corresponds a service call that initiates an artifact in another active workspace.

2. We have equations of the form \( y = x \) stating that distant variable \( y \) subscribes to the value of local variable \( x \).

3. We have equations of the form \( Y = X \) where \( Y \) is the distant node that created the artifact rooted at local node \( X \). Hence \( X \) is a root node.

Furthermore we add an input and an output buffers to each configuration \( \Gamma \). They contains messages respectively received from and sent to distant locations. A message in the input buffer \( \text{in}(\Gamma) \) is one of the following types.

1. \( Y = s(t_1, \ldots, t_n)(y_1, \ldots, y_m) \) tells that distant node \( Y \) calls service \( s \in \text{axioms}(G) \). When reading this message we create a new root node \( X \) —the root of the artifact associated with the service call. And values \( t_1, \ldots, t_n \) are assigned to the inherited attributes of node \( X \) while the distant variables \( y_1, \ldots, y_m \) subscribe to the values of its synthesized attributes. We replace variable \( Y \) by a dummy variable (wildcard: \( _{\_} \) ) when this service call is not invoked from a distant active workspace but from an external user of the system.

2. \( x = t \) tells that local variable \( x \) receives the value \( t \) from a subscription created at a distant location.

3. \( y = x \) states that distant variable \( y \) subscribes to the value of local variable \( x \).

Symmetrically, a message in the output buffer \( \text{out}(\Gamma) \) is one of the following types.

1. \( X = s(t_1, \ldots, t_n)(y_1, \ldots, y_m) \) tells that local node \( X \) calls the external service \( s \rightarrow \) a terminal symbol– with values \( t_1, \ldots, t_n \) assigned to the inherited attributes. And the local variables \( y_1, \ldots, y_m \) subscribe to the values of the synthesized attributes of the distant node where the artifact generated by this service call will be rooted at.

2. \( y = t \) tells that value \( t \) is sent to distant variable \( y \) according to a subscription made for this variable.

3. \( x = y \) states that local variable \( x \) subscribes to the value of distant variable \( y \).

The behavior of a guarded attribute grammar is given by relation \( \Gamma \xrightarrow{M} \Gamma' \) stating that event \( e \) transforms configuration \( \Gamma \) into \( \Gamma' \) and adds the set of messages \( M \) to the
output buffer. An event is the application of a rule \( R \) to a node \( X \) of configuration \( \Gamma \) or the consumption of a message from its input buffer. Let us start with the former kind of event: \( e = R/X \). Let \( X = s(t_1, \ldots, t_n)(y_1, \ldots, y_m) \in \Gamma \) and \( R = F \rightarrow F_1 \cdots F_k \) be a rule whose left-hand side matches with \( X \) producing substitution \( \sigma = \text{match}(F, X) \). All variables occurring in the definition of node \( X \) are local. We recall that in order to avoid name clashes we rename all the variables of rule \( R \) with fresh names. We use the local name generator for that purpose. Hence all variables of \( R \) are also local variables—freshly created ones. Therefore all variables used and defined by substitution \( \sigma \) are local variables. Then we let \( \Gamma' = \frac{R/X M}{\cdot \cdot \cdot} \Gamma' \)

\[
\Gamma' = \{ X = R(X_1, \ldots, X_k) \}
\cup \{ X_i = F_i \sigma \mid X_i \colon s_i \text{ and } s_i \in N \}
\cup \{ X' = F \sigma \mid (X' = F) \in \Gamma \land X' \neq X \}
\cup \{ y = y \sigma \mid (y = y_j) \in \Gamma \text{ and } y_j \sigma \text{ is a variable} \}
\cup \{ Y = X \mid (Y = X) \in \Gamma \}
\cup \{ y = y \sigma \mid (y = y_j) \in \Gamma \text{ and } y_j \sigma \text{ not a variable} \}
\]

where \( X_1, \ldots, X_k \) are new names in \( n s(G) \). Note that when a distant variable \( y \) subscribes to some synthesized attribute of node \( X \), namely \( (y = y_j) \in \Gamma \), two situations can occur depending on whether \( y_j \sigma \) is a variable or not. When \( y_j \sigma = x \) is a (local) variable the subscription \( y = y_j \) is replaced by subscription \( y = x \): Variable \( y_j \) depletes the production of the required value to \( x \). This operation is totally transparent to the location that initiated the subscription. But as soon as some value is produced – \( y_j \sigma \) is not a variable – it is immediately sent to the subscribing variable even when this value contains variables: Values are produced and send incrementally.

Let us now consider the event associated with the consumption of a message \( m \in \text{out}(\Gamma) \) in the output buffer

1. if \( m = (Y = s(t_1, \ldots, t_n)(y_1, \ldots, y_q)) \) then \( \Gamma' = \Gamma \cup \{ Y = s(t_1, \ldots, t_n)(y_1, \ldots, y_q) \}
\cup \{ y_j = \overline{y}_j \mid 1 \leq j \leq q \} \cup \{ \overline{Y} = \overline{Y} \}
\]

where \( \overline{Y} \), the variables \( x \) for \( x \in \text{var}(t_i) \) and the variables \( \overline{y}_j \) are new names in \( n s(G) \), \( \overline{t} = t[\overline{x}/x] \), and \( M = \{ \overline{x} = x \mid x \in \text{var}(t_i) \text{ and } 1 \leq i \leq n \} \).

2. if \( m = (x=t) \) then \( \Gamma' = \Gamma \cup \{ x = t[\overline{y}/y] \} \) where \( \overline{y} \) are new names in \( n s(G) \) associated with the variables \( y \) in \( t \) and \( M = \{ \overline{y} = y \mid y \in \text{var}(t) \} \).

3. if \( m = (y=x) \) then \( \Gamma' = \Gamma \cup \{ y = x \} \) and \( M = \emptyset \).

We now define the guarded attribute grammar resulting from the composition of a set of smaller guarded attribute grammars.

**Definition 3.1 (Composition of GAG).**

Let \( G_1, \ldots, G_p \) be guarded attribute grammars with disjoint sets of non terminal symbols such that each terminal symbol of a grammar \( G_i \) that belongs to another grammar \( G_j \) must be an axiom of the latter: \( T_i \cap S_j = T_i \cap axioms(G_j) \) where \( s \in T_i \cap axioms(G_j) \) means that grammar \( G_i \) uses service \( s \) of \( G_j \). Their composition, denoted as \( G = G_1 \oplus \cdots \oplus G_p \), is the guarded attribute grammar whose set of rules is the union of the rules of the \( G_i \)s and with set of axioms \( axioms(G) = \cup_{1 \leq i \leq p} axioms(G_i) \). We say that the \( G_i \) are the local grammars and \( G \) the global grammar of the composition. If some axiom of the resulting global grammar calls itself recursively we apply the transformation described in Rem. 2.4.

One may also combine this composition with a restriction operator, \( G \mid ax \), if the global grammar offers only a subset \( ax \subseteq \cup_{1 \leq i \leq p} axioms(G_i) \) of the services provided by the local grammars.

Note that the set of terminal symbols of the composition is given by

\[
T = \left( \cup_{1 \leq i \leq p} T_i \right) \setminus \left( \cup_{1 \leq i \leq p} axioms(G_i) \right)
\]

i.e., its set of external services are all external services of a local grammar but those which are provided by some other local grammar. Note also that the set of non-terminal symbols of the global grammar is the (disjoint) union of the set of non-terminal symbols of the local grammars: \( N = \cup_{1 \leq i \leq p} N_i \). This partition can be used to retrieve the local grammar by taking the rules of the global grammar whose sorts in their left-hand side belongs to the given equivalent class. Of course not every partition of the set of non-terminal symbols of a guarded attribute grammar corresponds to a decomposition into local grammars. To decompose a guarded attribute grammar into several components one can proceed as follows:

1. Select a partition \( axiom(G) \subseteq \cup_{1 \leq i \leq n} axioms_i \) of the set of axioms. These sets are intended to represent the services associated with each of the local grammars.

2. Construct the local grammar \( G_i \) associated with services \( axioms_i \), by first taking the rules whose left-hand sides are forms of sort \( s \in axioms_i \), and then iteratively adding to \( G_i \) the rules whose left-hand sides are forms of sort \( s \in N \setminus \cup_{j \not= i} axioms_j \), such that \( s \) appears in the right-hand side of a rule previously added to \( G_i \).

3. If appropriate, namely when a same rule is copied in several components, rename the non-terminal symbols of the local grammars to ensure that they have disjoint sets of non-terminal symbols.

The above transformation can duplicate rules in \( G \) in the resulting composition \( \overline{G} = G_1 \oplus \cdots \oplus G_n \) but does not radically change the original specification.

Configurations of guarded attribute grammars are enriched with subscriptions —equations of the form \( y = x \) — to enable communication between the various active workspaces. One might dispense with equations of the form \( Y = X \) in the operational semantics of guarded attribute grammars. But they facilitate the proof of correctness of this composition (Prop. 3.2) by easing the reconstruction of the global configuration from its set of local configurations. Indeed, the global configuration can be recovered as \( \Gamma = \Gamma_1 \oplus \cdots \oplus \Gamma_p \) where operator \( \oplus \) consists in taking the union of the systems of equations given as arguments and simplifying the
resulting system by elimination of the copy rules: We drop each equation of the form \( Y = X \) (respectively \( y = x \)) and replace each occurrence of \( Y \) by \( X \) (resp. of \( y \) by \( x \)).

Let \( G = G_1 \oplus \cdots \oplus G_p \) be a composition, \( G_i \) a configuration of \( G_i \) for \( 1 \leq i \leq p \) and \( \Gamma = \Gamma_1 \oplus \cdots \oplus \Gamma_p \) the corresponding global configuration. Since the location of a variable derives from its identifier we know the location of \( R/X \) \( M \) set of messages in \( M \) to be forwarded to \( G_i \). Their union \( M_i = \bigcup_{1 \leq i \leq p} M_i \) is the set of internal messages that circulate between the local grammars. The rest \( M_G = M \setminus M_i \) is the set of messages that remain in the output buffer of the global grammar—the global messages.

The join dynamics of the local grammars can be derived as follows from their individual behaviors, where \( e \) stands for \( R/X \) or a message \( m \):

1. If \( \Gamma_i \xrightarrow{e}_{M} \Gamma_i' \) then \( \Gamma \xrightarrow{e}_{M} \Gamma' \) with \( \Gamma_j = \Gamma_j' \) for \( j \neq i \).
2. If \( \Gamma \xrightarrow{m}_{M} \Gamma' \) and \( \Gamma' \xrightarrow{m}_{M'} \Gamma'' \) for \( m \in M \) then
   \[ \Gamma \xrightarrow{m}_{M \setminus \{m\} \cup M'} \Gamma''. \]

The correctness of the composition is given by the following proposition. It states that \( (i) \) every application of a rule can immediately be followed by the consumption of the local messages generated by it, and \( (ii) \) the behavior of the global grammar can be recovered from the joint behavior of its components where all internal messages are consumed immediately.

**Proposition 3.2.** Let \( G = G_1 \oplus \cdots \oplus G_p \) be a composition, \( \Gamma_1 \) a configuration of \( G_i \) for \( 1 \leq i \leq p \) and \( \Gamma = \Gamma_1 \oplus \cdots \oplus \Gamma_p \) the corresponding global configuration. Then

1. \( \Gamma \xrightarrow{R/X}_{M} \Gamma' \Rightarrow \exists \Gamma'' \text{ such that } \Gamma \xrightarrow{R/X}_{M_G} \Gamma'' \)
2. \( \Gamma \xrightarrow{e}_{M_G} \Gamma' \Rightarrow \exists \Gamma'' \text{ such that } \Gamma \xrightarrow{e}_{M_G} \Gamma'' \)

**Proof.** The first statement follows from the fact that relation \( \Gamma \xrightarrow{e}_{M} \Gamma' \) is deterministic (\( M \) and \( \Gamma' \) are uniquely defined from \( \Gamma \) and \( e \)) and the application of a rule produces, directly or indirectly, a finite number of messages.

- If \( m \) is of the form \( Y = s(t_1, \ldots, t_n)(y_1, \ldots, y_n) \), then consuming \( m \) results in adding new equations to the local configuration that receives it, and generating a set of messages of the form \( \bar{x} = x \) that can hence be consumed by the location that will receive them without generating new messages (case 3 below).
- If \( m = (x = t) \), then consumption of the message results in production of new variables, and a new (finite) set of messages of the form \( y = y \) that can then be consumed by the location that send \( m \) without producing any new message.
- If \( m = (y = x) \) then consuming the message results in adding an equation \( y = x \) to the local configuration without generating any new message.

The second statement follows from the fact that the construction of a global configuration \( \Gamma = \Gamma_1 \oplus \cdots \oplus \Gamma_p \) amounts to consuming all the local messages between the components. Q.E.D.

**Corollary 3.3.** Let \( G = G_1 \oplus \cdots \oplus G_p \) be a composition where \( G \) is an autonomous guarded attribute grammar and \( \Gamma \) be a configuration of \( G \). Then

\[ \Gamma[R/X]\Gamma' \iff \Gamma \xrightarrow{R/X}_{\emptyset} \Gamma' \]

A configuration is stable when all internal messages are consumed. The behavior of the global grammar is thus given as the restriction of the join behavior of the components to the set of stable configurations. This amounts to imposing that every event of the global configuration is immediately followed by the consumptions of the internal messages generated by the event. However if the various active workspaces are distributed at distant locations on an asynchronous architecture, one can never guarantee that no internal message remains in transit, or that some message is received but not yet consumed in some distant location. In order to ensure a correct distribution we therefore need a monotony property stating that \( (i) \) a locally enabled rule cannot become disabled by the arrival of a message and \( (ii) \) local actions and incoming messages can be swapped. We identify in the next section a class of guarded attribute grammars having this monotony property and which thus guarantees a safe implementation of distributed active workspaces.

### 4. PROPERTIES OF GAG

In this section we investigate some formal properties of guarded attribute grammars. We first turn our attention to input-enabled guarded attribute grammars to guarantee that the application of a rule in an open node is a monotonous and confluent operation. This property is instrumental for the distribution of a GAG specification on an asynchronous architecture. We then consider soundness, a classical property of case management systems stating that any case introduced in the system can reach completion. We show that this property is undecidable. Nonetheless, soundness is preserved by hierarchical composition—a restrictive form of modular composition. This opens up the ability to obtain a large class of specifications that are sound by construction, if we start from basic specifications that are known to be sound by some ad-hoc arguments.

#### 4.1 Distribution of a GAG

We say that rule \( R \) is triggered in node \( X \) if substitution \( \sigma_m \rightarrow \) given in def. 2.9 is defined: Patterns \( P_i \) match the data \( d_i \). As shown by the following example one can usually suspect a flaw in a specification when a triggered transition is not enabled due to the fact that the system of equations \( \{ y_\ell = u_j \sigma_m \mid 1 \leq j \leq m \} \) is cyclic.

**Example 4.1.** Let us consider the guarded attribute grammar given by the following rules:

\[
\begin{align*}
P : & \quad s_0()() \rightarrow s_1(a(x))() \quad s_2(x)() \\
Q : & \quad s_1(y)(a(y)) \rightarrow \\
R : & \quad s_2(a(z))() \rightarrow 
\end{align*}
\]
Applying $P$ in node $X_0$ of configuration $\Gamma_0 = \{ X_0 = s_0(\cdot) \}$ leads to configuration

$$\Gamma_1 = \{ X_0 = P(X_1, X_2); X_1 = s_1(a(x))(x); X_2 = s_2(x)(\cdot) \}$$

Rule $Q$ is triggered in node $X_1$ with $\sigma_m = \{ y = a(x) \}$ but the occur check fails because variable $x$ occurs in $a(y)\sigma_m = a(a(x))$. Alternatively, we could drop the occur check and instead adapt the fixed point semantics for attribute evaluation defined in [7, 31] in order to cope with infinite data structures. More precisely, we let $\sigma_{out}$ be defined as the least solution of system of equations $\{ y_j = u_j\sigma_m \mid 1 \leq j \leq m \}$ — assuming these equations are guarded, i.e., that there is no cycle of copy rules in the link graph of any accessible configuration. In that case the infinite tree $\sigma$ is substituted to variable $x$ and the unique maximal computation associated with the grammar is given by the infinite tree $P(Q, R^\infty)$. In Definition 2.11 we have chosen to restrict to finite data structures which seems a reasonable assumption in view of the nature of systems we want to model. The occur check is used to avoid recursive definitions of attribute values. The given example, whose most natural interpretation is given in Example 4.2, is used to illustrate the nature of systems we want to model. The occur check is enabled in configuration $\Gamma$ because configuration $\Gamma$ is given by the infinite tree $P(Q, R^\infty)$.

Substitution $\sigma_{in}$, given by pattern matching, is monotonous w.r.t. incoming information and thus it causes no problem for a distributed implementation of a model. However substitution $\sigma_{out}$ is not monotonic since it may become undefined when information coming from a distant location makes the match of output attributes a cyclic set of equations, as illustrated by example 4.2.

**Definition 4.4.** An autonomous guarded attribute grammar is **input-enabled** if every rule that is triggered in an accessible configuration is also enabled.

If every form $s(d_1, \ldots, d_n)(y_1, \ldots, y_m)$ occurring in some reachable configuration is such that variables $y_j$ do not appear in expressions $d_i$ then by Remark 2.10 the guarded attribute grammar is input-enabled — moreover $\sigma = \sigma_{out} \cup \sigma_{in}$ for every enabled rule. This property is clearly satisfied for guarded L-attributed grammars which consequently constitute a class of input-enabled guarded attribute grammars.

**Proposition 4.6 (Monotony).** Let $\Gamma$ be an accessible configuration of an input-enabled GAG, $X = s(d_1, \ldots, d_n)(y_1, \ldots, y_m) \in \Gamma$ and $\sigma$ be the substitution induced by some sequence starting from $\Gamma$. Then $\Gamma[P/X] \Gamma'$ implies $\Gamma[\sigma[P/X]] \Gamma' \sigma$. 

We first tackle the problem of safe distribution of a GAG specification on an asynchronous architecture by limiting ourselves to standalone systems. Hence to autonomous guarded attribute grammars. At the end of the section we show that this property can be verified in a modular fashion if the grammar is given as the composition of local (and thus non-autonomous) grammars.

**Definition 4.3 (Accessible Configurations).** Let $G$ be an accessible guarded attribute grammar. A case $c = s(t_1, \ldots, t_n)(x_1, \ldots, x_m)$ is a ground instantiation of service $s$, an axiom of the grammar, i.e., the values $t_i$ of the inherited attributes are ground terms. It means that it is a service call which already contains all the information coming from the environment of the guarded attribute grammar. An **initial configuration** is any configuration $\Gamma_0(c) = \{ X_0 = c \}$ associated with a case $c$. An **accessible configuration** is any configuration accessible from an initial configuration.
The former is trivial and the latter follows by induction on the length of the computation of the most general unifier \( \rightarrow^* \) using rule (5) only. Note that the assumption that the guarded attribute grammar is input-enabled is crucial because in the general case it could happen that the set \( \{ y_j = u_j \sigma_m \mid 1 \leq j \leq m \} \) satisfies the occur check whereas the set \( \{ y_j = u_j \sigma_m \sigma \mid 1 \leq j \leq m \} \) does not satisfy the occur check.

Proposition 4.6 is instrumental for the distributed implementation of guarded attribute grammars. Namely it states that new information coming from a distant asynchronous location refining the value of some input occurrences of variables of an enabled rule do not prevent the rule to apply. Thus a rule that is locally enabled can freely be applied regardless of information that might further refine the current partial configuration. It means that conflicts arise only from the existence of two distinct rules enabled in the same open node. Hence the only form of non-determinism corresponds to the decision of a stakeholder to apply one particular rule among those enabled in a configuration. This is expressed by the following confluence property.

**Corollary 4.7.** Let \( \Gamma \) be an accessible configuration of an input enabled GAG. If \( \Gamma(P/X)\Gamma_1 \) and \( \Gamma(Q/Y)\Gamma_2 \) with \( X \neq Y \) then \( \Gamma_2(P/X)\Gamma_3 \) and \( \Gamma_1(Q/Y)\Gamma_3 \) for some configuration \( \Gamma_3 \).

Note that, by Corollary 4.7, the artifact contains a full history of the case in the sense that one can reconstruct from the artifact the complete sequence of applications of rules leading to the resolution of the case —up to the commutation of independent elements in the sequence.

**Remark 4.8.** We might have considered a more symmetrical presentation in Definition 2.3 by allowing patterns for synthesized attributes in the right-hand sides of rules with the effect of creating forms in a configuration with patterns in their co-arguments. These patterns would express constraints on synthesized values. This extension could be acceptable if one sticks to purely centralized models. However, as soon as one wants to distribute the model on an asynchronous architecture, one cannot avoid such a constraint to be further refined due to a transformation occurring in a distant location. Then the monotony property (Proposition 4.6) is lost: A locally enabled rule can later be disabled when a constraint on a synthesized value gets a refined value. This is why we required synthesized attributes in the right-hand side of a rule to be given by plain variables in order to prohibit constraints on synthesized values.

**Definition 4.9.** Let \( s \) be a sort of a guarded attribute grammar with \( n \) inherited attributes and \( m \) synthesized attributes. We let \((j,i) \in SI(s)\) where \( 1 \leq i \leq n \) and \( 1 \leq j \leq m \) if there exists \( X = s(d_1, \ldots, d_n)(y_1, \ldots, y_m) \in \Gamma \) where \( \Gamma \) is an accessible configuration and \( y_j \in d_i \). If \( R \) is a rule with left-hand side \( s(p_1, \ldots, p_n)(u_1, \ldots, u_m) \) we let \((i,j) \in IS(R)\) if there exists a variable \( x \in \text{var}(R) \) such that \( x \in \text{var}(d_i) \cap \text{var}(u_j) \). The guarded attribute grammar \( G \) is said to be acyclic if for every sort \( s \) and rule \( R \) whose left-hand side is a form of sort \( s \) the graph \( G(s,R) = SI(s) \cup IS(R) \) is acyclic.

**Proposition 4.10.** An acyclic guarded attribute grammar is input-enabled.

**Proof.** Suppose \( R \) is triggered in node \( X \) with substitution \( \sigma_m \) such that \( y_j \in u_i \sigma_m \) then \((i,j) \in G(s,R)\). Then the fact that occur check fails for the set \( \{ y_j \mid 1 \leq j \leq m \} \) entails that one can find a cycle in \( G(s,R) \).

Relation \( SI(s) \) still takes into account the whole set of accessible configurations. The following definition provides an overapproximation of this relation given by a fixpoint computation.

**Definition 4.11.** The graph of local dependencies of a rule \( R : F_0 \rightarrow F_1 \cdots F_i \) is the directed graph \( GLD(R) \) that records the data dependencies between the occurrences of attributes given by the semantics rules. We designate the occurrences of attributes of \( R \) as follows: We let \( k(i) \) (respectively \( k(j) \)) denote the occurrence of the \( i \)th inherited attribute (resp. the \( j \)th synthesized attribute) in \( F_i \). If \( s \) is a sort with \( n \) inherited attributes and \( m \) synthesized attributes we define the relations \( TS(s) \) and \( SI(s) \) over \([1,n] \times [1,m] \) and \([1,m] \times [1,n] \) respectively as the least relations such that:

1. For every rule \( R : F_0 \rightarrow F_1 \cdots F_i \) where form \( F_i \) is of sort \( s_i \) and for every \( k \in [1,\ell] \)

\[
\{(i,j) \mid (k(j), k(i)) \in GLD(R)^k \} \subseteq SI(s_k)
\]

where graph \( GLD(R)^k \) is given as the transitive closure of

\[
GLD(R) \cup \{(0(j), 0(i)) \mid (j,i) \in SI(s_0)\}
\]

\[
\cup \{(k'(i), k'(j)) \mid k' \in [1,\ell], k' \neq k, (i,j) \in TS(s_{k'})\}
\]

2. For every rule \( R : F_0 \rightarrow F_1 \cdots F_i \) where form \( F_i \) is of sort \( s_i \)

\[
\{(i,j) \mid (0(i), 0(j)) \in GLD(R)^0 \} \subseteq TS(s_0)
\]

where graph \( GLD(R)^0 \) is given as the transitive closure of

\[
GLD(R) \cup \{(k(i), k(j)) \mid k \in [1,\ell], (i,j) \in TS(s_k)\}
\]

End of Remark 4.8
The guarded attribute grammar $G$ is said to be strongly-acyclic if for every sort $s$ and rule $R$ whose left-hand side is a form of sort $s$ the graph $\overline{G}(s, R) = \overline{SI}(s) \cup IS(R)$ is acyclic.

**Proposition 4.12.** A strongly-acyclic guarded attribute grammar is acyclic and hence input-enabled.

**Proof.** The proof is analog to the proof that a strongly non-circular attribute grammar is non-circular and it goes as follows. We let $(i, j) \in IS(s)$ when $\text{var}(\sigma) \cap \text{var}(\gamma, \sigma) \neq \emptyset$ for some form $F = s(d_1, \ldots, d_n)\{y_1, \ldots, y_m\}$ of sort $s$ and where $\sigma$ is the substitution induced by a firing sequence starting from configuration $\{X = F\}$. Then we show by induction on the length of the firing sequence leading to the accessible configuration that $IS(s) \subseteq \overline{TS}(s)$ and $SI(s) \subseteq \overline{SI}(s)$.

Note that the following two inclusions are strict strongly-acyclic $GAG \subsetneq$ acyclic $GAG \subseteq$ input enabled $GAG$

Indeed the reader may easily check that the guarded attribute grammar with rules

\[
\begin{align*}
  A(x)(z) &\rightarrow B(a(x, y))(y, z) \\
  B(a(x, y))(x, y) &\rightarrow
\end{align*}
\]

is cyclic and input-enabled whereas guarded attribute grammar with rules

\[
\begin{align*}
  A(x)(z) &\rightarrow B(y, x)(z, y) \\
  A(x)(z) &\rightarrow B(x, y)(y, z) \\
  B(x, y)(x, y) &\rightarrow
\end{align*}
\]

is acyclic but not strongly-acyclic. Attribute grammars arising from real situations are almost always strongly non-circular so that this assumption is not really restrictive. Similarly we are confident that most of the guarded attribute grammars that we shall use in practise will be input-enabled and that most of the input-enabled guarded attribute grammars are in fact strongly-acyclic. Thus most of the specifications are distributable and in most cases, this can be proved by checking strong non-circularity.

Let us conclude this section by addressing the modularity of strong-acyclicity. This property (see Def. 4.11) however was defined for autonomous guarded attribute grammars viewed as standalone applications. Here, the initial configuration is associated with a case $c = s(t_1, \ldots, t_n)\{y_1, \ldots, y_m\}$ introduced by the external environment. And the information transmitted to the inherited attributes are given by ground terms. Even though one can imagine that these values are introduced gradually they do not depend on the values that will be returned to the subscribing variables $y_1, \ldots, y_m$. It is indeed reasonable to assume that when an user enters a new case in the system she instanciates the inherited information and then waits for the returned values. Things go differently if the autonomous guarded attribute grammar is not a standalone application but a component in a larger specification. In that case, a call to the service provided by this component is of the form $s(t_1, \ldots, t_n)\{y_1, \ldots, y_m\}$ where the values transmitted to the inherited attributes may depend (directly or indirectly) on the subscribing variables. Definition 4.11 should be amended to incorporate these dependencies and strong-acyclicity can be lost. Therefore a component which is strongly-acyclic when used as a standalone application might lose this property when it appears as an individual component of a larger specification. Thus if the component is already implemented as a collection of subcomponents distributed on an asynchronous architecture, the correctness of this distribution can be lost if the component takes part in a larger system.

To avoid this pitfall we follow a standard contract based approach, where each component can be developed independently as long as it conforms to contraints given by assume/guarantee conditions. Assuming some properties of the environment, this approach allows to preserve properties of assembled components. In our case, we show that strong-acyclicity is preserved by composition.

**Definition 4.13.** Let $s$ be a sort with $n$ inherited attributes and $m$ synthesized attributes. We let $\text{IS}(s) = [1, n] \times [1, m]$ and $\text{SI}(s) = [1, m] \times [1, n]$ denote the set of (potential) dependencies between inherited and synthesized attributes of $s$. Let $G$ be a guarded attribute grammar with axioms $s_1, \ldots, s_k$ and terminal symbols $s_1', \ldots, s_k'$. An assume/guarantee condition for $G$ is a pair $(a, g) \in AG(G)$ with $a \in IS(s_1) \times \cdots \times IS(s_k) \times IS(s_1') \times \cdots \times IS(s_k')$ and $g \in IS(s_1) \times \cdots \times IS(s_k) \times IS(s_1') \times \cdots \times IS(s_k')$. Equivalently it is given by the data $SI(s) \subseteq SI(s)$ and $IS(s) \subseteq IS(s)$ for $s \in \text{axioms}(G)$. The guarded attribute grammar $G$ is strongly-acyclic w.r.t. assume/guarantee condition $(a, g)$ if the modified fixed-point computation of Def. 4.11, where constraints $SI(s) \subseteq SI(s)$ for $s \in \text{axioms}(G)$ and $IS(s) \subseteq \overline{TS}(s)$ for $s \in T$ are added, allows to conclude strong-acyclicity with $\overline{IS}(s) \subseteq IS(s)$ for $s \in \text{axioms}(G)$ and $\overline{SI}(s) \subseteq SI(s)$ for $s \in T$.

The data $SI(s) \subseteq SI(s)$ and $IS(s) \subseteq IS(s)$ give an over-approximation of the attribute dependencies, the so-called ‘potential’ dependencies, for the axioms and the terminal symbols. They define a contract of the guarded attribute grammar. This contract splits into assumptions about its environment –SI dependencies for the axioms and IS dependencies for the terminal symbols– and guarantees offered in return to the environment –IS dependencies for the axioms and SI dependencies for the terminal symbols. Thus strongly-acyclicity of a guarded attribute grammar $G$ w.r.t. assume/guarantee condition $(a, g)$ means that when the environment satisfies the assume condition, grammar $G$ is strongly-acyclic and satisfies the guarantee condition.

The following result states the modularity of strong-acyclicity.

**Proposition 4.14.** Let $G = G_0 \oplus \cdots \oplus G_p$ be a composition of guarded attribute grammars. Let $SI(s) \subseteq SI(s)$ and $IS(s) \subseteq IS(s)$ be assumptions on the (potential) dependencies between attributes where sort $s$ ranges over the set of axioms and terminal symbols of the components $G_i$ –thus containing also the axioms and terminal symbols of global grammar $G$. These constraints restrict to assume/guarantee conditions $(a_i, g_i) \in AG(G_i)$ for every local grammar and for the global grammar as well: $(a, g) \in AG(G)$. Then $G$ is.
strongly-acyclic w.r.t. \((a, g)\) when each local grammar \(G_i\) is strongly-acyclic w.r.t. \((a_i, g_i)\).

**Proof.** The fact that the fixed-point computation for the global grammar can be computed componentwise follows from the fact that for each local grammar no rule apply locally to a terminal symbol \(s\) and consequently rule 3 in Def. 4.11 never applies for \(s\) and the value \(SI(s)\) is left unmodified during the fixpoint computation, it keeps its initial value \(IS(s)\). Similarly, rule 2 in Def. 4.11 never applies for an axiom \(s\) and \(TS(s)\) keeps its initial value \(IS(s)\).

Q.E.D.

Conversely if the global grammar is strongly-acyclic w.r.t. some assume/guarantee condition \((a, g)\) then the values of \(SI(s)\) and \(IS(s)\) produced at the the end of the fixpoint computation allows to complement the assume/guarantee conditions with respect to which the local grammars are strongly-acyclic. The issue is how to guess some correct assume/guarantee conditions in the first place. One can imagine that some knowledge about the problem at hand can help to derive the potential attribute dependencies, and that we can use them to type the components for their future reuse in larger specifications. In many cases however there is no such dependencies and the assume/guarantee conditions are given by empty relations.

**Definition 4.15.** A composition \(G = G_0 \oplus \cdots \oplus G_p\) of guarded attribute grammars is distributable if each local grammar (and hence also the global grammar) is strongly-acyclic w.r.t. the empty assume/guarantee condition.

This condition might seems rather restrictive but it is not. Indeed if \((i, j) \notin IS(s)\) (similarly for \((i, j) \notin SI(s)\)) does not mean that the \(j^{th}\) synthesized attribute does not depend on the value received by the \(i^{th}\) inherited attribute—the inherited value influences the behaviour of the component and hence has an impact on the values that will be returned in synthesized attributes. It rather says that one should not return in the value of a synthesized attribute some data directly extracted from the value of inherited attributes, which is the most common situation. The emptiness of assume/guarantee gives us a criterion for a distributable decomposition of a guarded attribute grammar: It indicates the places where a specification can safely be split into smaller pieces. We shall denote a distributable composition as:

\[
G = (G_1, \ldots, G_p)
\]

where

\[
G_1 :: \%\text{definition of } G_1
\]

\[
G_p :: \%\text{definition of } G_p
\]

### 4.2 Soundness

A specification is sound if every case can reach completion no matter how its execution started. Recall from Def. 4.3 that a case \(c = s_0(t_1, \ldots, t_n)(x_1, \ldots, x_m)\) is a ground instantiation of service \(s_0\), an axiom of the grammar. And, an accessible configuration is any configuration accessible from a configuration \(\Gamma_0(c) = \{X_0 = c\}\) associated with a case \(c\) (an initial configuration).

**Definition 4.16.** A configuration is closed if it contains only closed nodes. An autonomous guarded attribute grammar is sound if a closed configuration is accessible from any accessible configuration.

We consider the finite sequences \((\Gamma_i)_{0 \leq i \leq n}\) and the infinite sequences \((\Gamma_i)_{0 \leq i \in \omega}\) of accessible configurations such that \(\Gamma_i[\Gamma_{i+1}]\). A finite and maximal sequence is said to be terminal. Hence a terminal sequence leads to a configuration that enables no rule. Soundness can then be rephrased by the two following conditions.

1. Every terminal sequence leads to a closed configuration.
2. Every configuration on an infinite sequence also belongs to some terminal sequence.

Soundness can unfortunately be proved indecidable by a simple encoding of Minsky machines.

**Proposition 4.17.** Soundness of guarded attribute grammar is undecidable.

**Proof.** We consider the following presentation of the Minsky machines. We have two registers \(r_1\) and \(r_2\) holding integer values. Integers are encoded with the constant zero and the unary operator succ. The machine is given by a finite list of instructions instr, for \(i = 1, \ldots, N\) of one of the three following forms

1. \(\text{INC}(r, i)\): increment register \(r\) and go to instruction \(i\).
2. \(\text{JZDEC}(r, i, j)\): if the value of register \(r\) is 0 then go to instruction \(i\) else decrement the value of the register and go to instruction \(j\).
3. \(\text{HALT}\): terminate.

We associate a Minsky machine with a guarded attribute grammar whose sorts corresponds bijectively its instructions, \(S = \{s_1, \ldots, s_N\}\), with the following encoding of the program instructions by rules:

1. If instr\(_k\) = INC\((r_1, i)\) then add rule
   \[
   \text{Inc}(k, 1, i) : s_k(x, y) \rightarrow s_i(\text{succ}(x), y)
   \]
2. If instr\(_k\) = INC\((r_2, i)\) then add rule
   \[
   \text{Inc}(k, 2, i) : s_k(x, y) \rightarrow s_i(x, \text{succ}(y))
   \]
3. If instr\(_k\) = JZDEC\((r_1, i, j)\) then add the rules
   \[
   \text{JZ}(k, 1, i) : s_k(\text{zero}, y) \rightarrow s_i(\text{zero}, y)
   \]
   \[
   \text{Dec}(k, 1, j) : s_k(\text{succ}(x), y) \rightarrow s_j(x, y)
   \]
4. If instr\(_k\) = JZDEC\((r_2, i, j)\) then add the rules
   \[
   \text{JZ}(k, 2, i) : s_k(x, \text{zero}) \rightarrow s_i(x, \text{zero})
   \]
   \[
   \text{Dec}(k, 2, j) : s_k(x, \text{succ}(y)) \rightarrow s_j(x, y)
   \]
5. If instr_k = HALT then add rule
   \[ \text{Halt}(k) : s_k(x, y) \rightarrow \]

Since there is a unique maximal firing sequence from the initial configuration \( \Gamma_0 = \{ X_0 = s_1(\text{zero}, \text{zero}) \} \) the corresponding guarded attribute grammar is sound if and only if the computation of the corresponding Minsky machine terminates.

Q.E.D.

This result is not surprising as most of non-trivial properties of an expressive enough formalism are indeed undecidable. The above encoding uses very simple features of the model: All guards are of depth at most one, the system is deterministic, and there are only inherited attributes (and in fact only two of them)! This leaves no hope to find syntactic restrictions to characterize an effective subclass of sound specifications.

Even though this problem is undecidable, soundness can still be proven for a given specification using ad-hoc arguments. Furthermore we show now that a restricted form of composition of GAG, called \textit{hierarchical composition} preserves soundness. This result allows the construction of a large class of specifications which are sound by construction.

\textbf{Definition 4.18} \textit{(Hierarchical Composition).} A composition \( G = G_0 \oplus \cdots \oplus G_n \) is \textit{hierarchical} if each GAG \( G_i \) has a unique axiom \( s_i \), the local grammars \( C_i = G_i \) for \( 1 \leq i \leq n \) are autonomous, and the terminal symbols of \( K = G_0 \) are \( \{ s_1, \ldots, s_n \} \).

We interpret \( K \) as a \textit{connector} that defines the possible orchestrations of the services associated with components \( C_1, \ldots, C_n \) and denote \( G = K(C_1, \ldots, C_n) \) such a composition. In this situation, depicted in Fig. 12, the connector provides a global service \( s_0 \) through an orchestration of the components. For that purpose the connector can make service calls to the services \( s_1, \ldots, s_n \) delivered by the components. The components are autonomous and therefore can not call their respective services. They can however communicate with each other information via attributes but only indirectly using the connector. Thus orchestration between the various components is fully encoded into the connector specification. The resulting composition, \( C = K(C_1, \cdots, C_n) \), is also an autonomous guarded attribute grammar and thus one can iterate this construction to obtain more complex hierarchical decompositions.

Soundness was defined in Def. 4.16 for autonomous guarded attribute grammars. We adapt this definition for a connector as follows.

\textbf{Definition 4.19.} A guarded attribute grammar is a \textit{connector} if it satisfies the following two conditions:

1. \textbf{Weak-Soundness.} For every accessible configuration \( \Gamma \), there exists a configuration accessible from \( \Gamma \) all of whose open nodes have sorts that are terminal symbols.

2. \textbf{Independence w.r.t. Components.} For any accessible configuration \( \Gamma \), open node \( X \), and substitution \( \sigma \) for variables subscribing to external services (i.e., occurring in a synthesized position in an open node of \( \Gamma \) whose sort is a terminal symbol), and for any rule \( R \) one has \( \Gamma[R/X] \not\Rightarrow \Gamma[\sigma[R/X]] \), which means that a pattern of a rule never tests values produced by an external service.

Intuitively Independence w.r.t. Components means that components can exchange information through the connector but this information has no impact on the choices of rules to apply within the connector.

\textbf{Proposition 4.20.} Let \( C = K(C_1, \ldots, C_n) \) be a hierarchical composition where \( K \) is a connector and the components \( C_1, \ldots, C_n \) are sound. If the global grammar is input-enabled —for instance if the composition is distributable— then it is also a sound (autonomous) guarded attribute grammar.

\textbf{Proof.} Let \( \Gamma \) be an accessible configuration of \( C \). By Independence w.r.t. Components there exists sequences of rule applications \( \Gamma_0[\ast]\Gamma_1[\ast]\Gamma_2[\ast]\Gamma_3[\ast] \) where rules in the first sequence belong to the connector, and in the second sequence belong to the components. By Weak-Soundness, as \( \Gamma_1 \) is a configuration of the connector, one can find a sequence of rule applications in the connector \( \Gamma_1[\ast]\Gamma_2[\ast] \) such that the sort of an open node in \( \Gamma_2 \) is a terminal symbol of the connector (i.e., lies in \( I(K) \)). By Confluence (which follows from Input-Enabledness) there exists a configuration \( \Gamma_3 \) which is accessible both from \( \Gamma \) and from \( \Gamma_2 \). The sorts of open nodes of \( \Gamma_3 \) are found in the components. By Soundness of the components and Monotony (which also follows from Input-Enabledness) \( \Gamma_3[\ast] \Gamma_4 \) where \( \Gamma_4 \) is a closed configuration.

Q.E.D.

\section{Towards a Language for the Specification of GAGs}

In this section we introduce some syntax elements to outline a specification language for guarded attribute grammars that is expressive enough to describe realistic applications. Our purpose is not to fully design such a specification language. This would require more thorough investigations and, in particular, the implementation of some typing mechanism for the manipulated values. We only intend to introduce some syntactic sugar and constructs which allow to describe large and complex specifications in a more concise and friendlier way. This syntax is used in Section 6 where a case study is presented.

First, we introduce in Section 5.1 a functional notation for business rules, inspired from monadic programming in Haskell.
So far, a guarded attribute grammar was presented as a task rewriting system, a convenient formalism for formal manipulations. However, rewriting systems are not necessarily perceived as a handy programming notation despite their similarity with logic programming. In Section 5.2 we give the opportunity to write generic rules, namely rules that contain parameters whose instantiations can generate a potentially large set of similar rules. This is particularly useful to formalize the notion of role: When several stakeholders play a similar role they can use the same generic local grammar instantiated with their respective identities to distinguish them from one another. Section 5.3 introduces a feature that allows the designer to extend the formalism by adding combinators, a technique that can be used to customize the notations in order to derive domain specific languages adapted to the particular user needs.

5.1 A Functional Notation

In order to ease the writing of rules we introduce a syntax inspired from monadic computations in Haskell. More precisely, we restate rule

$$\text{sort}(p_1, \ldots, p_n) \langle u_1, \ldots, u_m \rangle \Rightarrow$$

$$\text{sort}_1(t_1^{(1)}, \ldots, t_{n_1}^{(1)}) \langle y_1^{(1)}, \ldots, y_{m_1}^{(1)} \rangle$$

$$\ldots$$

$$\text{sort}_k(t_1^{(k)}, \ldots, t_{n_k}^{(k)}) \langle y_1^{(k)}, \ldots, y_{m_k}^{(k)} \rangle$$

as

$$\text{sort}(p_1, \ldots, p_n) =$$

$$\text{do} \ (y_1^{(1)}, \ldots, y_{m_1}^{(1)}) \leftarrow \text{sort}_1(t_1^{(1)}, \ldots, t_{n_1}^{(1)})$$

$$\ldots$$

$$\ (y_1^{(k)}, \ldots, y_{m_k}^{(k)}) \leftarrow \text{sort}_k(t_1^{(k)}, \ldots, t_{n_k}^{(k)})$$

$$\text{return} \ (u_1, \ldots, u_m)$$

This functional presentation stresses out the operational purpose of business rules: Each task has an input –inherited attributes– seen as parameters and an output –synthesized attributes– seen as returned values. This notation however can confuse Haskell programmers for two reasons.

First, recall from Definition 2.3 that an input occurrence of a variable is either a variable occurring in a pattern $p_i$ or a variable occurring as a subscription in the right-hand side of the rule or, in this alternative presentation, in the left-hand side of a generator

$$\ (y_1^{(i)}, \ldots, y_{m_i}^{(i)}) \leftarrow \text{sort}_i(t_1^{(i)}, \ldots, t_{n_i}^{(i)})$$

Note that, using this $\text{do}$ notation, a guarded attribute grammar is left-attributed (Def. 4.5) precisely when every variable is defined before used: each output occurrence of a variable is preceded by its corresponding input occurrence. For guarded attribute grammar which are not left-attributed we thus find some variables which are used before being defined. This is incompatible with monadic programming in Haskell where the scope of a variable occurring in the left-hand side of a generator is the part of the $\text{do}$ expression that follows the generator, including the return statement.

Second, a Haskell monadic expression is evaluated in $\text{pull}$ mode: If the output returned by the $\text{do}$ expression does not use the values of the variables defined by a given generator, then this generator is not evaluated at all. By contrast, a rule of a guarded attribute grammar is evaluated in $\text{push}$ mode: When rule is applied, we create one open node for every generator. Then users can continue to develop these nodes with the effect of gradually refining the returned values.

**Example 5.1.** Consider the GAG of Example 2.2:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Root</strong></td>
<td>$\text{root()} \Rightarrow \text{do} \ (x) \leftarrow \text{bin}(\text{Nil}) \ $ \text{return} \ (x)</td>
</tr>
<tr>
<td><strong>Fork</strong></td>
<td>$\text{bin}(x) \Rightarrow \text{do} \ (z) \leftarrow \text{bin}(x) \ (y) \leftarrow \text{bin}(z) \ $ \text{return} \ (y)</td>
</tr>
<tr>
<td><strong>Leaf</strong></td>
<td>$\text{bin}(x) \Rightarrow \text{return} \ (\text{Cons}_a(x))$</td>
</tr>
</tbody>
</table>

This syntactical translation into the functional notation is the following:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Root</strong></td>
<td>$\text{root()} = \text{do} \ (x) \leftarrow \text{bin}(\text{Nil}) \ $ \text{return} \ (x)</td>
</tr>
<tr>
<td><strong>Fork</strong></td>
<td>$\text{bin}(x) = \text{do} \ (z) \leftarrow \text{bin}(x) \ (y) \leftarrow \text{bin}(z) \ $ \text{return} \ (y)</td>
</tr>
<tr>
<td><strong>Leaf</strong></td>
<td>$\text{bin}(x) = \text{return} \ (\text{Cons}_a(x))$</td>
</tr>
</tbody>
</table>

which we write more simply as

<table>
<thead>
<tr>
<th>Rule</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Root</strong></td>
<td>$\text{root()} = \text{bin}(\text{Nil})$</td>
</tr>
<tr>
<td><strong>Fork</strong></td>
<td>$\text{bin}(x) = \text{do} \ (z) \leftarrow \text{bin}(x) \ (y) \leftarrow \text{bin}(z) \ $ \text{return} \ (y)</td>
</tr>
<tr>
<td><strong>Leaf</strong></td>
<td>$\text{bin}(x) = \text{return} \ (\text{Cons}_a(x))$</td>
</tr>
</tbody>
</table>

using the simplification rules given below.

End of Exple 5.1

(SR₁) When the returned value is the result of the last generator, one replaces these two instructions by the last call, e.g.:

$$\text{do} \ (y) \leftarrow \text{bin} (\text{Nil}) \ $ \text{return} \ (y) \Leftrightarrow \text{do} \ \text{bin} (\text{Nil})$$

(SR₂) When the $\text{do}$ sequence is reduced to a unique item –e.g. : a call or a $\text{return}$ statement– one omits the $\text{do}$ instruction, e.g.:

$$\text{do} \ \text{bin} (\text{Nil}) \Leftrightarrow \text{bin} (\text{Nil})$$

$$\text{do} \ \text{return} \ (\text{Cons}_a(x)) \Leftrightarrow \text{return} \ (\text{Cons}_a(x))$$

5.2 Parametric Rules

It may happen that several components of a composition $G = G_1 \oplus \cdots \oplus G_k$ are associated with stakeholders that play the same role in the system and consequently use the same set of rules. To illustrate this situation let us consider the editorial process of a scholarly journal. The Editor of the journal has to make an editorial decision about a submitted paper by resorting to scientific evaluations produced by independent reviewers. The first local grammar consists of rules governing the activities of the Editor. The other local grammars describe the activities of the various reviewers. Suppose that the role of a reviewer was previously described by a guarded attribute grammar “evaluate” with an homonymous axiom corresponding to the service “making a scientific evaluation of a paper” –It is actually convenient to name a GAG by its axiom when this axiom is unique. All the components associated with reviewers are thus given by this specification. However the components
must have disjoint sets of non-terminal symbols so that the
distribution schema described in Section 3 works properly.
For that purpose, each actual reviewer is attached to a dis-
joint copy of grammar ‘evaluate’. Technically, a parameter
reviewer is added to each of the non-terminal symbols of
grammar ‘evaluate’. Each component is then obtained by in-
stanciating the parameter by an identifier of the correspond-
ing reviewer. For instance evaluate[reviewer] is a generic
sort and an actual sort is evaluate[Paul] where Paul is a re-
viewer. This allows to distinguish activities evaluate[Paul]
and evaluate[Ann] that correspond to sending a paper for
evaluation to Paul and to Ann respectively. We write –using
a hierarchical form of composition:

editorial\_decision(\text{evaluate}[\text{reviewer}]) \quad \text{where}
editor = \text{Paul} \mid \text{Ann} \mid \ldots
editorial\_decision :: \%Grammar editorial\_decision
evaluate :: \%Description of the grammar evaluate

This contract promotes an ordinary GAG to a generic one. At
the same time, it alleviates the notations by locating the use
of parameters: The sort evaluate[\text{reviewer}] appears in
grammar ‘editorial\_decision’ but grammar ‘evaluate’ simply
uses sort evaluate. Such a construction can be iterated. For
instance if the journal has several editors, one may write

editorial\_process(editorial\_decision[editor]) \quad \text{where}
editor = \text{Mary} \mid \text{Frank} \mid \ldots
editorial\_process :: \%Grammar editorial\_process
editorial\_decision ::
editorial\_decision(\text{evaluate}[\text{reviewer}]) \quad \text{where}
editor = \text{Paul} \mid \text{Ann} \mid \ldots
editorial\_decision :: \%...
evaluate :: \%...

The convention to name a GAG by its axiom entails some
overloading of notations because in a hierarchical composi-
tion, the axiom of a global grammar coincides with the
axiom of the connector. Nevertheless this overloading cre-
ates no confusion and it simplifies the notations. The above
description makes it clear that the editorial process relies on
editorial decisions made by editors and that such a decision
depends on evaluations made by reviewers.

Parameters are instanciated by the connector. For instance
a rule in grammar ‘editorial\_process’ states that the Editor
takes a decision based on evaluation reports produced by
two referees. This rule can be written as:

Evaluate\_Submission[\text{reviewer}_1, \text{reviewer}_2]
\quad \text{where not(}\text{not(}\text{reviewer}_1 = \text{reviewer}_2)\quad \text{:
submission(}\text{article}) =
do \quad \text{report}_1 \leftarrow \text{evaluate}[\text{reviewer}_1]
\quad \text{report}_2 \leftarrow \text{evaluate}[\text{reviewer}_2]
\quad \text{decide(}\text{report}_1, \text{report}_2)

Evaluate\_Submission[\text{reviewer}_1, \text{reviewer}_2] is a generic rule,
the corresponding actual rules are obtained by choosing val-
ues for the parameters that conform to the condition given
in the where clause. Thus it defines as many rules as pairs
of distinct reviewers. We’d rather write the above rule on
the form

Evaluate\_Submission :
\quad \text{submission(}\text{article}) =
\quad \text{input (}\text{reviewer}_1, \text{reviewer}_2)
\quad \text{where not(}\text{not(}\text{reviewer}_1 = \text{reviewer}_2)
\quad \text{do report}_1 \leftarrow \text{evaluate}[\text{reviewer}_1]
\quad \quad \text{report}_2 \leftarrow \text{evaluate}[\text{reviewer}_2]
\quad \quad \text{decide(}\text{report}_1, \text{report}_2)

The input clause serves to highlight those parameters of
the rule that are instanciated by the user when she applies this
rule at an open node associated with task submission(\text{article}).

The corresponding instance of the generic rule, for instance

Evaluate\_Submission[\text{Paul}, \text{Ann}]

which is the actual rule selected by the user, will subse-
quently label the node. In this manner the information
about the selection of the reviewers is stored in the arti-
fact. Parameters in the input clause enable user to input
any kind of data and not solely instances of roles. For in-
stance one may find the following rule in the specification of
the reviewer:

Accept :
\quad \text{evaluate(}\text{article}) =
\quad \text{input (}\text{msg} :: \text{String})
\quad \text{do report} \leftarrow \text{review(}\text{article})
\quad \quad \text{return (}\text{Yes}(\text{msg}, \text{report})\text{)}

With this rule the reviewer informs the Editor that he ac-
cepts to review the paper. The returned value is formed
with the Yes constructor, witnessing acceptance, with two
arguments: A complementary (optional) message and a link
to the report that the reviewer commits himself to subse-
quently produce. In this way the specification generates
an infinite set of actual rules since there exists an infinite
number of potential messages –including the empty one. In
practice however the parameters will correspond either to a
specific role in the system –whose instantiations are finite
in number – or some kind of data – a message, a report, a
decision, etc.– whose values should be kept in the artifact
but has no impact on the subsequent behavior of the sys-
tem. Therefore it will be possible to abstract the parameter
values to end up with a finite guarded attribute grammar
with the same behavior.

If we expand the global grammar ‘editorial\_decision’ the
rules of grammar ‘evaluate’ are promoted to generic rules. In
particular the above rule gives rise to the following generic
rule

Accept :
\quad \text{evaluate}[\text{reviewer}](\text{article}) =
\quad \text{input (}\text{msg} :: \text{String})
\quad \text{do report} \leftarrow \text{review}[\text{reviewer}](\text{article})
\quad \quad \text{return (}\text{Yes}(\text{msg}, \text{report})\text{)}

Note that by contrast to parameters that appear in an input
clause, the parameters in the left-hand side of the rule do not
correspond to user choices, simply because they are already
instanciated in the current configuration. For instance, in
an open node associated with task evaluate[\text{Paul}](\text{article}),
which belongs to the active workspace of Paul, only the in-
stance of the rule associated with Paul can apply: Mary has
(SR₃) We omit the do statement if it merely returns the value(s) introduced by the preceding input clause:

\[
\begin{align*}
\text{input}(x) & \quad \Leftrightarrow \quad \text{input}(x) \\
\text{return}(x) & \quad \Leftrightarrow \quad \text{input}(x)
\end{align*}
\]

### 5.3 Combinators

Since we use a variant of attribute grammars and a notation inspired from monadic computations in Haskell the question naturally arises whether one can implement our specification language as a set of monadic combinators, in the line of the Parsec library [19] for functional parsers. In the present state of things it is not at all certain that such a goal is easily achievable or even feasible. Nonetheless, some useful combinators can be introduced to tailor the specification language towards more specific application domains.

For instance, one can introduce an iteration schema given by combinator many defined as follows. If s is a sort with inherited and synthesized attributes of respective types a and b, many s stands for a new sort whose inherited and synthesized attributes are lists of elements of type a and b respectively, associated with rules

\[
\begin{align*}
\text{many } s \ (\text{Cons}(\text{head}, \text{tail})) & \quad = \quad \text{do } \text{head}' \leftarrow s(\text{head}) \\
& \quad \quad \text{tail}' \leftarrow \text{many } s \ (\text{tail}) \\
& \quad \quad \text{return} \ (\text{Cons}(\text{head}', \text{tail}')) \\
\text{many } s \ (\text{Nil}) & \quad = \quad \text{return} \ (\text{Nil})
\end{align*}
\]

This is only syntactic sugar: a GAG uses a set of sorts with no a priori structure, but one can equip the set of sorts with a set of combinators together with a type system to constraint their usage — for instance many s :: a* → b* when s :: a → b- and with rules which conform to this type system.

The following example, adapted from Example 2.18, uses the above combinator to describe two recurring tasks that communicate through lazy lists:

\[
\begin{align*}
(s, s') & \quad :: \quad () \sim () \\
\text{when } s & \quad :: \quad a \sim b \\
\quad & \quad s' :: b \sim a \\
(s, s') & \quad = \quad \text{input}(x) \quad \text{where} \quad x :: a \\
& \quad \quad \text{do } y s \leftarrow \text{many } s \ (\text{Cons}(x, x s)) \\
& \quad \quad \quad x s \leftarrow \text{many } s' \ (y s) \\
& \quad \quad \text{return} ()
\end{align*}
\]

The corresponding compound process uses no information from and returns no information to its surrounding environment. We can improve on this example by replacing this combinator by a (potentially infinite) set of combinators using functions as extra parameters.

This combinator has two parameters given by variables in and out and thus it actually generate a potentially infinite set of rules according to the actual functions used to instantiate the parameters. Still, if we assume that all such parametric combinators are totally instantiated in a global GAG specification then we guarantee that we end up with a finite specification.

For instance we may define the derived combinator:

\[
\begin{align*}
[s, s'] & \quad = \quad (s, s') \text{id zip } a \sim (a \times b)^* \\
\text{when } s & \quad :: \quad a \sim b \\
\quad & \quad s' :: b \sim a \\
\end{align*}
\]

where id :: a → a is the identity function and zip :: a* × b* → (a × b)* is the function given by:

\[
\begin{align*}
\text{zip } \text{Nil} \ y s & \quad = \quad \text{Nil} \\
\text{zip } x s \ \text{Nil} & \quad = \quad \text{Nil} \\
\text{zip } \text{Cons}(x, x s) \ \text{Cons}(y, y s) & \quad = \quad \text{Cons}((x, y), \text{zip } x s \ y s)
\end{align*}
\]

This combinator can equivalently be specified as

\[
\begin{align*}
[s, s'] & \quad :: \quad a \sim (a \times b)^* \\
\text{when } s & \quad :: \quad a \sim b \\
\quad & \quad s' :: b \sim a \\
[s, s'] & \quad = \quad \text{do } y s \leftarrow \text{many } s \ (\text{Cons}(x, x s)) \\
& \quad \quad x s \leftarrow \text{many } s' \ (y s) \\
& \quad \quad \text{return} \ (\text{zip } x s \ y s)
\end{align*}
\]

However, this specification is not a GAG because the semantic rules are no longer given by plain terms but they also include some basic functions (like zip). One can impose that functions that instantiate the parameters of combinators are basic functions for list and tuples manipulations. These functions are, in any case, necessary to describe the various plumbing operations between the parametric combinators and their context of use. Since these functions are lazily evaluated it does not really affect the operational semantic of GAG.

### 6. A DISEASE SURVEILLANCE SYSTEM

In this section, we illustrate the model of Active workspaces based on Guarded attribute grammars with the notations introduced in Section 5 on a collaborative case management system. The real world scenario is observed from a disease surveillance system. Disease Surveillance as defined by the Centers for Disease Control and Prevention (CDC) [8] is the ongoing, systematic collection, analysis, interpretation, and dissemination of data about a health-related event for use in public health action to reduce morbidity and mortality and to improve wellbeing. It is a complex process that uses data from a plethora of sources and distributes its activities over several geographically dispersed actors with heterogeneous
The disease surveillance system consists of both the physicians and the Disease Surveillance Center (DSC). The latter proceeds to the analyses of the suspect cases transmitted by physicians.

diseaseSurveillance :: (visit[physician], caseAnalysis)
  where
  physician = Alice | Bob | ...
  visit :: % Role of a physician
  caseAnalysis ::
    caseAnalysis(laboratoryAnalysis[biologist],
      dataAnalysis[epidemiologist])
     where
      epidemiologist = Ann | Paul | ...
      biologist = Frank | Mary | ...
  
  caseAnalysis :: % Disease Surveillance Center
  laboratoryAnalysis :: % Role of a biologist
  dataAnalysis :: % Role of an epidemiologist

The various components are modeled in detail in the following paragraphs. The following naming conventions are used throughout this section. Services (grammar axioms) are written in **bold**, sorts of internal (local) rules are unformatted, variable names are *italized*, and constructors are unformatted with first letter capitalized. Apart from Constructors, no other identifier has its first letter capitalized.

**Role of a Physician (visit).**

The physician receives patients, clinically examines them and if necessary, declares them as suspect cases of influenza.

\[
\text{visit}(\text{patient}, \text{alarm}) =
\]
\[
\text{do (symps)} \leftarrow \text{clinicalAssessment}(\text{patient})
\]
\[
() \leftarrow \text{initialCare(symps)}
\]
\[
\text{caseDeclaration}(\text{patient}, \text{symps}, \text{alarm})
\]

First the physician fills out a form to report the symptoms observed in the patient. This information is a parameter of the rule (**input** clause) and thus is recorded in the artifact associated with the patient.

\[
\text{clinicalAssessment}(\text{patient}) = \text{input} (\text{symps})
\]

Note that we use here the simplication rule \(\text{SR}_3\) meaning that the input value—the symptoms—is returned as a result of the clinical assessment. Then the physician can prescribe appropriate treatments.

\[
\text{initialCare}(\text{symps}) = \text{input} (\text{care})
\]
\[
\text{return} ()
\]

Again the prescribed treatments are recorded in the artifact of the patient but this information is not returned as a result since it will not be used later.

If the symptoms correspond with the Influenza declaration criteria, the physician extracts some samples to be sent to a biologist for laboratory analysis and declares the patient as a suspect case. And he commits himself to later run some further verifications on the case acmCheck— if required by the DSC, and the correspondinh results checkRes are send back to the DSC to complete the case analysis.

\[
\text{caseDeclaration}(\text{patient}, \text{symps}, \text{alarm}) =
\]
\[
\text{input} (\text{samples})
\]
\[
\text{do (alarm)} \leftarrow \text{caseAnalysis} (\text{SuspectCase (patient, symps, samples)},
\]
\[
\text{checkRes})
\]
\[
\text{return} ()
\]

Note that this rule is not left-attributed: There are mutual dependencies between the subtasks \text{caseAnalysis} and checkRes which are executed as coroutines: the caseAnalysis may produce an alarm that triggers the acmCheck, and the acmCheck returns check results which are used to continue the case analysis. If the patient does not have the influenza symptoms, the following rule is used instead.

\[
\text{caseDeclaration}(\text{patient, symps, _}) = \text{return} ()
\]

Here we have replaced variable \text{alarm} by a dummy variable (the wildcard) to stress that this variable is not used in the rule—and in fact it will not be assigned a value either since
physician who declared the current case. This service is devoted to the investigative activities to a set of queries sent to the
and with contextual variables. In this case study, we limit the aggregation tasks on the entire declaration database with the
posing the alarm data with information from other sources
better characterize the outbreak alarms. These investigative
break alarms). This is followed by investigation tasks to
aim of detecting aberrations from normal behaviour (out-
This service runs a number of automated data analyses and
lance related computations on the reported data in order
to detect and investigate disease outbreaks. These analyses
can produce an alarm.

\[ \text{acmCheck(Alarm(info, todo))} = \text{input (checkRes)} \]

Else, the following rule applies

\[ \text{acmCheck(NoAlarm)} = \text{return (}_-) \]

Again the wildcard indicates a variable that is not instantiated—it is not defined by the rule—and whose value is not expected elsewhere.

Disease Surveillance Center.

If the patient is reported as a suspect case of influenza, a biologist and an epidemiologist are assigned to respectively
carry out biological analyses for samples sent by the physi-
cian and to do statistical analyses and other disease surveil-
lance related computations on the reported data in order
to detect and investigate disease outbreaks. These analyses
can produce an alarm.

\[ \text{caseAnalysis(SuspectCase(patient, symps, samples), checkRes)} = \]
\[ \text{input (bio, epi)} \]
\[ \text{do (labRes) \leftarrow laboratoryAnalysis[bio](samples)} \]
\[ \text{dataAnalysis[epi](patient, symps, labRes, checkRes)} \]

Note the use of simplification rule SR2 due to the fact that the result returned by caseAnalysis is the result of data-
Analysis.

Role of a Biologist (laboratoryAnalysis).

The biologist receives the samples, verifies their conditioning and runs the requested analyses. The results are returned for used in confirming or revoking the outbreak alarm. For simplicity, we suppose and only model the case where the sample is well conditioned in which case the corresponding grammar is reduced to rule

\[ \text{laboratoryAnalysis(samples)} = \text{input (labResult)} \]

Role of an Epidemiologist (dataAnalysis).

This service runs a number of automated data analyses and aggregation tasks on the entire declaration database with the aim of detecting aberrations from normal behaviour (out-
break alarms). This is followed by investigation tasks to better characterize the outbreak alarms. These investigative
tasks (aspecific counter measure checks) may involve super-
posing the alarm data with information from other sources and with contextual variables. In this case study, we limit the investigative activities to a set of queries sent to the
physician who declared the current case. This service is de-

\[ \text{dataAnalysis(patient, symps, labResult, checkResult)} = \]
\[ \text{do (ack) \leftarrow storeCaseData(patient, symps)} \]
\[ \text{automatedAnalysis(ack, labResult, checkResult)} \]

The epidemiologist stores the current case data in the surve-
llance database.

\[ \text{storeCaseData(patient, symps)} = \text{return (Ack)} \]

This triggers automated analyses on the entire database of declared suspect cases. The following rule is used if the analyses raise an alarm.

\[ \text{automatedAnalysis(Ack, labResult, checkResult)} = \]
\[ \text{input (info, todo)} \]
\[ \text{do () \leftarrow notifyAuth(info)} \]
\[ () \leftarrow outbreakDecl(labResult, checkResult) \]
\[ \text{return (Alarm(info, todo))} \]

Note that the alarm is emitted in the first place, prior to no-
tifying authorities and the outbreak declaration. And these
two tasks can be performed concurrently. This clearly il-
lustrates that the elements of a do body—including the
return statement—are not necessarily executed in their order
of appearance.

If on the other hand no alarm is raised the following rule is
used instead.

\[ \text{automatedAnalysis(Ack, _ _, _ _)} = \text{return (NoAlarm)} \]

Observe that in both versions of rule automatedAnalysis
pattern Ack is checked in order to ensure that the database
has been updated according to the current case. This is an
illustration of side effects as discussed in observation (O1),
at the end of this section.

Raised alarms are immediately notified to public health au-
thorities.

\[ \text{notifyAuth(info)} = \text{return ()} \]

As earlier stated, the alarm contains a list of queries that
will need to be answered by the physician who declared the
suspect case. Rich with the investigation results and the lab-
oratory results provided respectively by the physician and the
biologist, the epidemiologist runs a number of alert anal-
yses to confirm the outbreak alarm and declare an outbreak
alert. He then analyses the risks involved and the public
health impact of the outbreak. This information is used to
propose appropriate counter measures which he communi-
cates to public health authorities for action.

\[ \text{outbreakDecl(labResult, checkResult)} = \]
\[ \text{input (alertInfos)} \]
\[ \text{do (risks) \leftarrow riskAnalysis(alertInfos)} \]
\[ (counterM) \leftarrow \text{defineCounterMeasures(risks)} \]
\[ \text{feedback(alertInfos, counterM)} \]

\[ \text{riskAnalysis(alertInfos)} = \text{input (risks)} \]
\[ \text{defineCounterMeasures(risks)} = \text{input (counterM)} \]
feedback(alertInfos, counterM) =
  input (mail_list)
sendFeedback(mail_list, alertInfos, counterM)

sendFeedback(mail_list, alertInfos, counterM) = return ()

If, conversely, the outbreak alert is not confirmed the following
alternative rule is chosen.

outbreakDecl(labResult, checkResult) = return ()

Let us conclude this case study with some observations.

(O1) Some tasks may have side-effects. For instance send-
feedback forwards a message about the alert and the
proposed counter measures to the email addresses given
in mail_list. Also, storeCaseData stores the declared
data in some local database on which the automated
analysis is run. The automated analysis of the database
produces information that guides the epidemiologist
in her choice of raising an alarm or not (i.e., in choosing
which rule to apply for automatedAnalysis). Such
side-effects are not described in the model but they
can easily be attached to rules when necessary.

(O2) Rules of the form

\[
\text{task}(\text{args}) = \text{input} (\text{results})
\]

corresponding to tasks that merely return values input-
putted by the user can be used for incremental speci-
fications. Indeed if the rules specifying the resolution
of task(args) are not yet designed, one can used the
above temporary rule to obtain an approximate speci-
fication that can be executed and tested even though
it will require the user to manually input the expected
results. Then this temporary rule can progressively be
refined to obtain a new rule where the results are no
longer inputted by the user but synthesized from in-
termediate results produced by new subtasks that are
introduced for that purpose.

(O3) The given specification is sound, which can be easily
verified from the fact that the underlying grammar
is non-recursive: A task never calls itself directly or
indirectly. This situation, which is relatively common,
provides a large family of sound specifications from
which many other sound specifications can be built
using hierarchical composition.

7. CONCLUSION

In this paper, we have proposed a declarative model of artifact-
centric collaborative systems. The key idea was to represent
the workspace of a stakeholder by a set of (mind)maps asso-
ciated with the services that she delivers. Each map consists
of the set of artifacts created by the invocations of the cor-
responding service. An artifact records all the information
related to the treatment of the related service call. It con-
tains open nodes corresponding to pending tasks that require
user’s attention. In this manner each user has a global view
of the activities in which she is involved, including all rel-
vent information needed for the treatment of the pending
tasks.

Using a variant of attribute grammars, called guarded attri-
butte grammars, one can automate the flow of information
in collaborative activities with business rules that put empha-
sis on user’s decisions. We gave an in-depth descrip-
tion of this model through its syntax and its behaviour.
We paid attention to two crucial properties of this model.
First, the input-enabled GAG satisfy a monotony property
that allows to distribute the model on an asynchronous ar-
chitecture. Input-enabledness is undecidable but we have
identified the sufficient condition of strongly-acyclicity that
can be checked very efficiently by a fixpoint computation of
an over-approximation of attribute dependencies. Second,
soundness is a property that asserts that any case intro-
duced in the system can reach completion. This property
is also undecidable but we have defined a hierarchical com-
position of GAGs that preserves soundness and thus allows
to build large specifications that are sound by construction
if one starts from small components which are known to be
sound.

We have introduced some notations and constructs paving
the way towards an expressive and user-friendly specification
language for active workspaces. Also, we have demonstrated
the expressive power and exemplified the key concepts of
active workspaces on a case study for a disease surveillance
system.

In the rest of this section we list key features of the model
and draw some future research directions.

7.1 Assessment of the Model

In a nutshell active workspaces and guarded attribute gram-
mars provide a modular, declarative, user-centric, data-driven,
distributed and reconfigurable model of case management. It
favors flexible design and execution of business process since
it possesses (to varying degrees) all four forms of Process
Flexibility proposed in [40].

Concurrency.

The lifecycle of a business artifact is implicitly represented
by the grammar productions. A production decomposes a
task into new subtasks and specifies constraints between
their attributes in the form of the so-called semantic rules.
The subtasks may then evolve independently as long as the
semantic rules are satisfied. The order of execution, which
may depend on value that are computed during process ex-
cution, need not (and cannot in general) be determined
statically. For that reason, this model dynamically allows
maximal concurrency. In comparison, models in which the
lifecycle of artifacts are represented by finite automata con-
strain concurrency among tasks in an artificial way.

Modularity.

The GAG approach also facilitates a modular description of
business processes. For instance, when laboratory test re-
quests are sent to the biologist, the physician needs not know
about the subprocess through which the specimen will pass
before results are finally produced. For instance, the service
laboratoryAnalysis can –in a more refined specification– be
modeled by a large set of rules including specimen purifi-
cation, and several biological and computational processes. However, following a top-down approach, one simply introduces an attribute in which the results should eventually be synthesized and delegate the actual production of the expected outcome to an additional set of rules. The identification of the different roles involved in the business process can also contribute to enhance modularity. Finally, some techniques borrowed from attribute grammars, like description composition [20, 21], decomposition by aspects [42, 41] or higher-order attribute grammars [43, 38], may also contribute to better modular designs.

Reconfiguration.

The workflow can be reconfigured at run time: New business rules (associated with productions of the grammar) can be added to the system without disturbing the current cases. By contrast, run time reconfiguration of workflows modeled by Petri nets (or similar models) is known to be a complex issue [13, 17]. One can also add “macro rules” corresponding to specific compositions of rules. For instance if the Editor-in-chief wants to handle the evaluation of a paper, he can decide to act as an associate editor and as a referee for this particular submission. However, this means forwarding the corresponding case to himself as an associate editor and then asking himself as a referee if she is willing to write a report. A more direct way to model this decision is to encapsulate these steps in a compound macro production that bypasses the intermediate communications. More generally compound rules can be introduced for handling unusual behaviors that deviates from the nominal workflow.

Logged information.

When a case is terminated, the corresponding artifact collects all relevant information of its history. Nodes are labeled by instances of the productions that have lead to the completion of the case. Henceforth, they record the decisions (the choices among the allowed productions) together with information associated with these decisions. In the case of the editorial process, a terminated case contains the names of the referees, the evaluation reports, the editorial decision, etc. A terminated case is a tree whose branches reflect causal dependencies among subactivities used to solve a case, while abstracting from concurrent subactivities. The artifacts can be collected in a log which may be used for the purpose of process mining [39] either for process discovery (by inferring a GAG from a set of artifacts using common patterns in their tree structure) or for conformance checking (by inspection of the logs produced during simulations of a model or the executions of an actual implementation).

Distribution.

Guarded attributed grammars can easily be implemented on a distributed architecture without complex communication mechanisms –like shared memory or FIFO channels. Stakeholders in a business process own open nodes, and communicate asynchronously with other stakeholders via messages. Moreover there are no edition conflicts since each part of an artifact is edited by the unique owner of the corresponding node. Moreover, the temporary information stored by the attributes attached to open nodes no longer exist when the case has reached completion. Closing nodes eliminates temporary information without resorting to any complex mechanism of distributed garbage collection.

7.2 Future Works

An immediate milestone is the design of a prototype of the Active Workspaces runtime environment. This prototype shall contain support tools (editor, parser, checker, simulators ...) to analyze, implement, and run GAG descriptions of AW systems. The following research directions are also considered:

Coupling GAG systems with external features.

In Section 6, we have imagined a scenario where the reported cases are stored in some local database and the analysis and investigation tasks directly access and use these data. Such implicit side-effects of rules abound –see Observation (O1) Section 6. They generally do not conflict with the GAG specification but rather complement it, in basically two ways. First, they allow to associate real-world activities with a rule, like extracting samples from a patient –caseDeclaration-, sending messages –sendFeedback-, performing verifications –acmCheck, riskAnalysis-, etc. Second, they may be used to extract information from the current artifacts to build dashboards or to feed some local database that are later used to guide the user on her choice of the rule to apply for a pending task. They may, in a more coercive fashion, suggest a specific rule to apply or even inhibit some of the rules. Some information from dashboards or contained in a local database can also be used to populate some input parameters of a rule in place of the stakeholder. The actions of the stakeholder, namely choosing which rule to apply and the values to input in the system are left unspecified in the GAG specification –they constitute its only form on non-determinism. Side-effects can thus complement the GAG specification by providing an additional support to the stakeholder in this regard. Nonetheless, if we resort to a distant database or web services then it will be necessary to put some restrictions on the allowed queries to preserve monotony and thus to guarantee a safe distribution of the specification. Similarly we must be careful, if side-effects can inhibit some rules, that this does not jeopardize soundness. By the way, it is important to dispose of a language to describe side-effects of rules and in particular a language for making queries on active workspaces. The ideal solution would be to implement GAG as a domain specific language embedded into a general purpose language. In that case we could directly write the side-effects in the host language.

Development methodology.

We need to develop a support for the derivation of a GAG specification from a problem description. Object-oriented programming uses, for that purpose, normalized notations and diagrams for specifying the involved classes, uses cases, activities and collaborations. A modeling language for GAG should concentrate on the central concepts of the model: The artifacts, task decomposition, user’s decisions, user’s
communication. For the latter one may use concepts of speech act theory [4, 45] for classifying business rules in terms of assertions, orders, requests, commitments, etc. As far as artifacts are concerned, one can observe on the two examples of the paper (Editorial process and Disease surveil-
ance) that we have very few completed artifacts, once the case’s specific information contained in the artifacts have been abstracted. One can try to extract the business rules –task decomposition and semantic rules– starting from these archetypal artifacts and answering the following questions: What are the dependencies between data field values? Who produces these values? What information does one need to produce that value? Can one identify the conditions that justify variabilities between similar artifacts? etc. As a for-
malism for distributed collaborative systems the GAG model should also come with a complete method for elaborating the procedure going from a problem description, through the implementation, to the deployment on a distributed asyn-
cchronous architecture. Just as with Software Processes [37], this method will provide notations which describe how to identify relevant information (roles, data, processes ...) and propose appropriate representation tools to add expres-
siveness to the textual descriptions of collaborative case man-
gagement systems.

Applicability and Pertinence.

The development of case studies is very important to check the pertinence, level of applicability, and practical limita-
tions of the GAG model. These case studies are also impor-
tant to refined the specification language that we have sketched in Section 5 and to extract from practise useful con-
cepts for a modeling language. We continue our study on Disease surveillance that we intend to effectively implement on a real situation in collaboration with epidemiologists from Centre Pasteur in Cameroon. Besides Disease surveillance, it is also useful to develop more representative case studies of distributed collaborative systems. The following two ex-
amples are representative case studies in which our model can provide advantages over existing techniques. Report-
ing systems where several stakeholders collaborate to build a report. The grammar can reflect the structure of the re-
port, the identification of stakeholders and their respective contributions. The semantic rules implement the automatic assembly of the report from the bits provided by the dis-
tributed stakeholders. Most often, many information to be inserted in a report are already available. Semantics rules avoid redundancies and reduce workload: You write only once each piece of information, it is then collected in a syn-
thesized attribute for further use. Guarded attribute gram-
mars also avoids email overload —a problem that appears frequently when you have to coordinate a group of people to complete a task— since most of the communication is di-
rectly made between the active workspaces. A distributed distance learning which is a highly decentralized system deployed mostly in degraded environments (where Internet connection is not always available) with most stakeholders working off-line and synchronizing their activities upon es-


8. REFERENCES


http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf.


