Applied Computing Review

Sep. 2015, Vol. 15, No. 3

Frontmatter

Selected Research Articles

SIGAPP FY'15 Quarterly Report

July 2015 – September 2015
Jiman Hong

Mission

To further the interests of the computing professionals engaged in the development of new computing applications and to transfer the capabilities of computing technology to new problem domains.

Officers

Chair	Jiman Hong Soongsil University, South Korea
Vice Chair	Tei-Wei Kuo National Taiwan University, Taiwan
Secretary	Maria Lencastre University of Pernambuco, Brazil
Treasurer	John Kim Utica College, USA
Webmaster	Hisham Haddad Kennesaw State University, USA
Program Coordinator	Irene Frawley ACM HQ, USA

Notice to Contributing Authors

By submitting your article for distribution in this Special Interest Group publication, you hereby grant to ACM the following non-exclusive, perpetual, worldwide rights:

- to publish in print on condition of acceptance by the editor
- to digitize and post your article in the electronic version of this publication
- to include the article in the ACM Digital Library and in any Digital Library related services
- to allow users to make a personal copy of the article for noncommercial, educational or research purposes

However, as a contributing author, you retain copyright to your article and ACM will refer requests for republication directly to you.

SAC 2016 Preview

The 31st Annual Edition of the ACM Symposium on Applied Computing (SAC) will be held in Pisa, Italy, in the City Congress Center, located in the heart of the city. The conference is hosted by University of Pisa and Scuola Superior Sant'Anna. The conference starts on Monday April 4th with the *Tutorials Program* and continues with the *Technical Program* till Friday April 8th, 2016. The *Student Research Competition (SRC) Program* is planned for Tuesday (display session) and Wednesday (presentations session), respectively, and the *Posters Program* will take place on Thursday. The paper submission system is open at the following location: http://www.acm.org/conferences/sac/sac2016/submission.html. Details of submissions, acceptance rate, and country of participation will be reported in the next issue of ACR.

The conference will include the fourth annual *Student Research Competition (SRC)* program. *SRC* is designed to provide graduate students the opportunity to meet and exchange ideas with researchers and practitioners in their areas of interest. Active graduate students seeking feedback from the scientific community on their research ideas are invited to submit research abstracts of their original un-published and in-progress research work in areas of experimental computing and application development related to SAC 2016 Tracks. Accepted research abstracts will be published in the conference proceedings. Authors of accepted abstracts are eligible to apply for the SIGAPP *Student Travel Award Program (STAP)*. A designated committee will judge the display and presentations of abstracts and select the top three winners for medallions and cash prizes ($200, $300, and $500) provided by ACM. The winners will be recognized during the banquet event. We encourage you to share this information with your graduate students.

The local organizing committee is composed of Professors Giorgio Buttazzo (Scuola Superiore Sant'Anna), Alessio Bechini (University of Pisa), and Ettore Ricciardi (ISTI-CNR, Pisa). To facilitate attendees travel, the local organizing committee will provide plentiful information and help. Once finalized, details will be posted on SAC 2016 website. As the planning is underway, we are excited to have SAC 2016 in Pisa, Italy. We invite you to join us this April to meet other attendees, enjoy the conference programs, and have a pleasant stay in Pisa. We hope to see you there.

Best Regards to all,

Hisham Haddad
Kennesaw State University
Member of SAC Steering Committee
Member of SAC 2016 Organizing Committee

Next Issue

The planned release for the next issue of ACR is December 2015.

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
nsairobby@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.

CCS Concepts

•Information systems → Enterprise information systems; Collaborative and social computing systems and tools; Asynchronous editors; •Theory of computation → Interactive computation; Distributed computing models; Grammars and context-free languages; •Software and its engineering → Specification languages;

Keywords

Business Artifacts, Case Management, Attribute Grammars

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 instancied 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.

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.

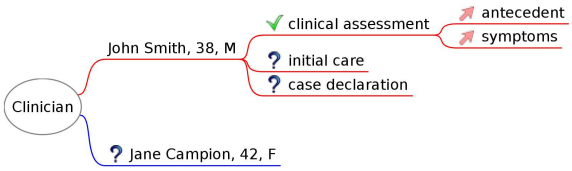


Figure 1: Active workspace of a clinician

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* $P : s_0 \to s_1 \cdots s_n$ expressing that task s_0 can be reduced to subtasks s_1 to s_n. For instance the production

$$\begin{array}{rl} \textbf{patient} \to & \textbf{clinical\_assessment} \\ & \textbf{initial\_care} \\ & \textbf{case\_declaration} \end{array}$$

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:

$$\begin{array}{lll} \text{suspect\_case} & : & \textbf{case\_declaration} \to \textbf{follow\_up} \\ \text{benign\_case} & : & \textbf{case\_declaration} \to \end{array}$$

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,

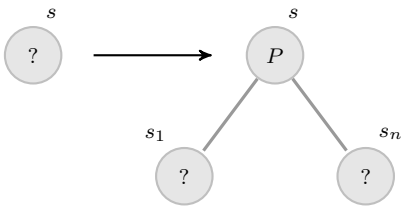
then there is no real decision to make and such a rule is interpreted as a logical decomposition of the task s_0 into substasks s_1 to s_n. Such a production will automatically be triggered without human intervention.

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 $P : s \to s_1 \cdots s_n$ 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 $P : s \to s_1 \ldots s_n$ 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 $var(t)$ the set of variables used in term t.

Definition 2.1 (Forms). *A **form** of sort s is an expression $F = s(t_1, \ldots, t_n)\langle u_1, \ldots, u_m\rangle$ 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 $var(F)$. 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

$$s_0(p_1, \ldots, p_n)\langle u_1, \ldots, u_m\rangle \to s_1(t_1^{(1)}, \ldots, t_{n_1}^{(1)})\langle y_1^{(1)}, \ldots, y_{m_1}^{(1)}\rangle \\ \vdots \\ s_k(t_1^{(k)}, \ldots, t_{n_k}^{(k)})\langle y_1^{(k)}, \ldots, y_{m_k}^{(k)}\rangle$$

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

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

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 $s_0(d_1, \ldots, d_n)\langle y_1, \ldots, y_m\rangle$ 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)

$$R : s_0(a(x_1, x_2))\langle b(y_1'), y_2'\rangle \to s_1(c(x_1))\langle y_1'\rangle \;\; s_2(x_2, y_1')\langle y_2'\rangle$$

to a node associated with tasks $s_0(a(t_1, t_2))\langle y_1, y_2\rangle$ 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 $s_1(c(t_1))\langle y_1'\rangle$ and $s_2(t_2, y_1')\langle y_2'\rangle$ 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

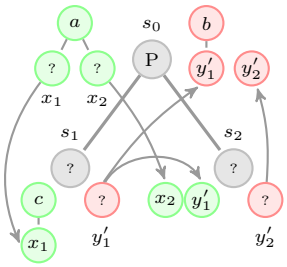


Figure 2: A business rule

our knowledge on the ontology of the system– and introducing new business rules when recurrent patterns of activities are detected.

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 *grammareware* 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 developped 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 combinators 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 $Att(s)$ of *attributes* —which henceforth should exist for each node of the given sort— and by associating each production $P : s \to s_1 \dots s_n$ 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 *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 $t \cdot s = \mathit{flatten}(t) +\!\!+ t \cdot h$, i.e., $t \cdot s = \mathit{flat}(t, t \cdot h)$ where $\mathit{flat}(t, h) = \mathit{flatten}(t) +\!\!+ h$. The semantics rules stem from the identities:

$$\mathit{flatten}(t) = \mathit{flat}(t, Nil)$$
$$\mathit{flat}(\mathrm{Fork}(t_1, t_2), h) = \mathit{flat}(t_1, \mathit{flat}(t_2, h))$$
$$\mathit{flat}(\mathrm{Leaf}_a, h) = \mathrm{Cons}_a(h)$$

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

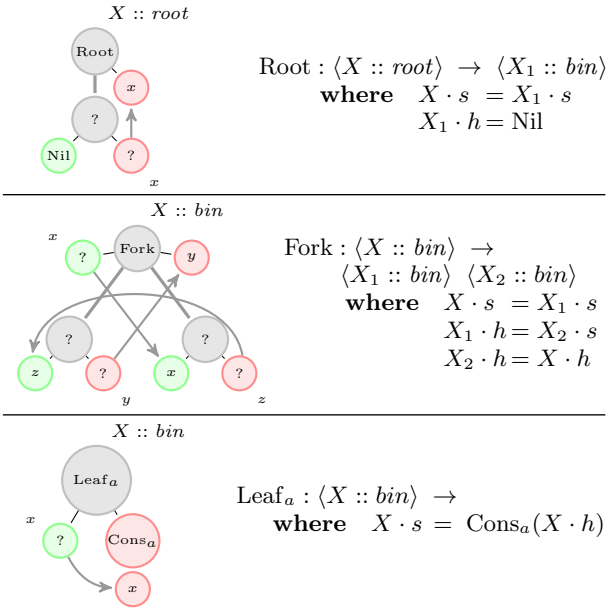


Figure 3: Flattening of a binary tree

syntax:

$$\begin{array}{lll} \mathrm{Root} & : & \mathit{root}()\langle x \rangle \to \mathit{bin}(Nil)\langle x \rangle \\ \mathrm{Fork} & : & \mathit{bin}(x)\langle y \rangle \to \mathit{bin}(z)\langle y \rangle \ \mathit{bin}(x)\langle z \rangle \\ \mathrm{Leaf}_a & : & \mathit{bin}(x)\langle \mathrm{Cons}_a(x) \rangle \to \end{array}$$

The *syntactic categories* of the grammar, also called its *sorts*, namely *root* and *bin* are associated with their inherited attributes –given as a list of arguments: $(t_1, \dots, t_n)$– and their synthesized attributes –the co-arguments:$\langle u_1, \dots, u_m \rangle$. A variable x is an *input variable*, denoted as $x^?$, 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 $x^!$, 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 the subtree rooted at the corresponding successor node. Indeed, if we annotate the occurrences of variables with their polarity – input or output– one obtains:

$$\begin{array}{lll} \mathrm{Root} & : & \mathit{root}()\langle x^! \rangle \to \mathit{bin}(Nil)\langle x^? \rangle \\ \mathrm{Fork} & : & \mathit{bin}(x^?)\langle y^! \rangle \to \mathit{bin}(z^!)\langle y^? \rangle \ \mathit{bin}(x^!)\langle z^? \rangle \\ \mathrm{Leaf}_a & : & \mathit{bin}(x^?)\langle \mathrm{Cons}_a(x^!) \rangle \to \end{array}$$

And if we draw an arrow from the (unique) occurrence of $x^?$ to the (various) occurrences of $x^!$ 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 $R : F_0 \to F_1 \cdots F_k$ 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* $\mathbf{axioms}(G) \subseteq \mathrm{def}(G) \setminus \mathrm{Use}(G)$ *–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.

End of Remark 2.4

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).

$$
\begin{aligned}
!(F_0 \to F_1 \cdots F_k) &= ?(F_0) \to !(F_1) \cdots !(F_k) \\
?(s(t_1, \ldots t_n)\langle u_1, \ldots u_m\rangle) &= s(?(t_1), \ldots ?(t_n))\langle !(u_1), \ldots !(u_m)\rangle \\
!(s(t_1, \ldots t_n)\langle u_1, \ldots u_m\rangle) &= s(!(t_1), \ldots !(t_n))\langle ?(u_1), \ldots ?(u_m)\rangle \\
?(c(t_1, \ldots t_n)) &= c(?(t_1), \ldots ?(t_n)) \\
!(c(t_1, \ldots t_n)) &= c(!(t_1), \ldots !(t_n)) \\
?(x) &= x^? \\
!(x) &= x^!
\end{aligned}
$$

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 $\{?(F_0), !(F_1), \ldots, !(F_k)\}$ 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 $var(\mathcal{F})$ of a set of forms $\mathcal{F}$ is **admissible** if the labelled version of a form $F \in \mathcal{F}$ 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 $\mathcal{F}$ 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 a same variable x, labelled $?$ in v_1 and $!$ in v_2. This arc, depicted as follows,*

*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 $\mathcal{U}(G) = (N, T, A, \mathcal{P})$ 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–,*
- *$A =$ $\mathbf{axioms}(G)$ is the set of axioms of the guarded attribute grammar, and*
- *the set of productions $\mathcal{P}$ is made of the underlying productions $\mathcal{U}(R) : s_0 \to s_1 \cdots s_k$ of rules $R : F_0 \to F_1 \cdots F_k$ 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 doe 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** Γ of an autonomous guarded attribute grammar is an S-sorted set of nodes $X \in nodes(\Gamma)$ each of which is associated with a defining equation in one of the following form where $var(\Gamma)$ is a set of variables associated with Γ:*

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

Open node: *$X = s(t_1, \ldots, t_n)\langle x_1, \ldots, x_m\rangle$ where X is of sort s and $t_1, \ldots, t_k$ are terms with variables in $var(\Gamma)$ that represent the values of the inherited attributes of X, and $x_1, \ldots, x_m$ are variables in $var(\Gamma)$ associated with its synthesized attributes.*

Each variable in $var(\Gamma)$ occurs at most once in a synthesized position. Otherwise stated $!\Gamma = \{!F \mid F \in \Gamma\}$ is an admissible labelling of the set of forms occurring in Γ. 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 σ on a set of variables $\{x_1, \ldots, x_k\}$, called the **domain** of σ, with a system of equations

$$\{x_i = \sigma(x_i) \mid 1 \leq i \leq k\}$$

The set $\mathrm{var}(\sigma) = \bigcup_{1 \leq i \leq k} \mathrm{var}(\sigma(x_i))$ of variables of σ, is disjoint from the domain $\mathrm{dom}(\sigma)$ of σ. Conversely a system of equations $\{x_i = t_i \mid 1 \leq i \leq k\}$ defines a substitution σ with $\sigma(x_i) = t_i$ if it is in **solved form**, i.e., none of the variables x_i appears in some of the terms t_j. 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_i until no variable x_i occurs in some t_j. This process terminates when the relation $x_i \succ x_j \Leftrightarrow x_j \in \mathrm{var}(\sigma(x_i))$ is acyclic. One can easily verify that, under this assumption, the resulting system of equation $SF(E) = \{x_i = t'_i \mid 1 \leq i \leq n\}$ 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 σ, σ', where $\mathrm{var}(\sigma') \cap \mathrm{dom}(\sigma) = \emptyset$, is denoted by $\sigma\sigma'$ and defined by $\sigma\sigma' = \{x = t\sigma' | x = t \in \sigma\}$. Similarly, we let $\Gamma\sigma$ denote the configuration obtained from Γ by replacing the defining equation $X = F$ of each open node X by $X = F\sigma$.

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 Γ when applying rule R at a node of Γ. As informally stated in the previous section, a rule R applies at an open node X when its left-hand side $s(p_1, \ldots, p_n)\langle u_1, \ldots u_m\rangle$ matches with the definition $X = s(d_1, \ldots, d_n)\langle y_1, \ldots, y_m\rangle$, namely the task attached to X in Γ.

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

$\mathbf{match}(c(p'_1, \ldots, p'_k), c'(d'_1, \ldots, d'_{k'}))$ with $c \neq c'$ fails
$\mathbf{match}(c(p'_1, \ldots, p'_k), c(d'_1, \ldots, d'_k)) = \sum_{i=1}^{k} \mathbf{match}(p'_i, d'_i)$
$\mathbf{match}(x, d) = \{x = d\}$

where the sum $\sigma = \sum_{i=1}^{k} \sigma_i$ of substitutions σ_i is defined and equal to $\bigcup_{i \in 1..k} \sigma_i$ when all substitutions σ_i are defined and associated with disjoint sets of variables. Note that since no variable occurs twice in the whole set of patterns p_i, the various substitutions $\mathbf{match}(p_i, d_i)$, when defined, are indeed concerned with disjoint sets of variables. Note also that $\mathbf{match}(c(), c()) = \emptyset$.

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

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

The resulting substitution $\sigma = \mathbf{match}(F, F')$ 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 σ_{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 $R = F_0 \to F_1 \dots F_k$ be a rule, Γ be a configuration, and X be an open node with definition $X = s(d_1, \dots, d_n)\langle y_1, \dots, y_m\rangle$ in Γ. We assume that R and Γ are defined over disjoint sets of variables. We say that R is* **enabled** *in X and write $\Gamma[R/X\rangle$, if the left-hand side of R matches with the definition of X. Then applying rule R at node X transforms configuration Γ into Γ', denoted as $\Gamma[R/X\rangle\Gamma'$, with Γ' defined as follows:*

$$\begin{array}{rcl} \Gamma' & = & \{X = R(X_1, \dots, X_k)\} \\ & \cup & \{X_1 = F_1\sigma, \dots, X_k = F_k\sigma\} \\ & \cup & \{X' = F\sigma \mid (X' = F) \in \Gamma \ \wedge \ X' \neq X\} \end{array}$$

where $\sigma = \mathbf{match}(F_0, X)$ and $X_1, \dots, X_k$ are new nodes added to Γ'.

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 Γ'. They are associated respectively with the instances of the k forms in the right-hand side of R obtained by applying substitution σ to these forms. The definitions of the other nodes of Γ are updated using substitution σ –or equivalently σ_{out}. This update has no effect on the closed nodes because their defining equations in Γ 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 Γ with $\Gamma[R/X_0\rangle\Gamma'$ then Γ' is a configuration: Applying R cannot create a variable with several input occurrences. And Prop. 2.15 shows that substitution $\sigma = \mathbf{match}(F_0, X)$ resulting from the matching of the left-hand side $F_0 = s(p_1, \dots, p_n)\langle u_1, \dots, u_m\rangle$ of a rule R with the definition $X = s(d_1, \dots, d_n)\langle y_1, \dots, y_m\rangle$ 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 Γ and $\Gamma[R/X_0\rangle\Gamma'$ then Γ' is a configuration.*

PROOF. Let $R = F_0 \to F_1 \dots F_k$ with left-hand side $F_0 = s(p_1, \dots, p_n)\langle u_1, \dots u_m\rangle$ and $X_0 = s(d_1, \dots, d_n)\langle y_1, \dots, y_m\rangle$ be the defining equation of X_0 in Γ. Since the values of synthesized attributes in the forms $F_1, \dots, F_k$ are variables (by Definition 2.3) and since these variables are unaffected by substitution σ_{in} the synthesized attribute in the resulting forms $F_j\sigma_{in}$ are variables. The substitutions σ_{in} and σ_{out} substitute terms to the variables $x_1, \dots, x_k$ appearing to the patterns and to the variables $y_1, \dots, y_m$ respectively. Since x_i appears in an input position in R, it can appear only in an output position in the forms $!(F_1), \dots !(F_k)$ and thus any variable of the term $\sigma_{in}(x_i)$ will appear in an output position in $!(F_i\sigma_{in})$. Similarly, since y_i appears in an input position in the form $!(s(u_1, \dots, u_n)\langle y_1, \dots, y_m\rangle)$, it can only appear in an output position in $!(F)$ for the others forms F of Γ. Consequently any variable of the term $\sigma_{out}(y_i)$ will appear in an output position in $!(F\sigma_{out})$ for any equation $X = F$ in Γ 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 Γ'. *Q.E.D.*

Thus applying an enabled rule defines a binary relation on configurations.

DEFINITION 2.13. *A configuration Γ' is* ***directly accessible*** *from Γ, denoted by $\Gamma[\rangle\Gamma'$, whenever $\Gamma[R/X\rangle\Gamma'$ for some rule R enabled in node X of configuration Γ. Furthermore, a configuration Γ' is* ***accessible*** *from configuration Γ when $\Gamma[*\rangle\Gamma'$ where $[*\rangle$ is the reflexive and transitive closure of relation $[\rangle$.*

Recall that a substitution σ unifies a set of equations E if $t\sigma = t'\sigma$ for every equation $t = t'$ in E. A substitution σ is more general than a substitution σ', in notation $\sigma \leq \sigma'$, if $\sigma' = \sigma\sigma''$ for some substitution σ''. If a system of equations has a some unifier, then it has –up to an bijective renaming of the variables in σ– 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 $E = E_? \uplus E_=$ containing equations of two kinds. An equation in $E_?$, denoted as $t \stackrel{?}{=} u$, represents a *unification goal* whose solution is a substitution σ such that $t\sigma = u\sigma$ –substitution σ unifies terms t and u. $E_=$ 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 $E_=$. A *solution* to E is any substitution σ whose domain is the set of variables occurring in the right-hand sides of equations in $E_=$ such that the compound substitution made of σ and the set of equations $\{x = t\sigma \mid x = t \in E_=\}$ unifies terms t and u for any equation $t \stackrel{?}{=} u$ in $E_?$. 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 $E_= = \emptyset$, i.e., it is a set of unification goals. On the contrary E is said to be in *solved form* if $E_? = \emptyset$, 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 $E_?$ and depending on its shape apply the corresponding transformation:

1. $c(t_1, \dots, t_n) \stackrel{?}{=} c(u_1, \dots, u_n)$: replace it by equations $t_1 \stackrel{?}{=} u_1, \dots, t_1 \stackrel{?}{=} u_1$.
2. $c(t_1, \dots, t_n) \stackrel{?}{=} c'(u_1, \dots, u_m)$ with $c \neq c'$: halt with failure.
3. $x \stackrel{?}{=} x$: delete this equation.
4. $t \stackrel{?}{=} x$ where t is not a variable: replace this equation by $x \stackrel{?}{=} t$.
5. $x \stackrel{?}{=} t$ where $x \notin \mathrm{var}(t)$: replace this equation by $x = t$ and substitute x by t in all other equations of E.
6. $x \stackrel{?}{=} t$ where $x \in \mathrm{var}(t)$ 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

$$\sigma = \mathbf{mgu}(\{t_i = u_i\}_{1\leq i\leq n}) \ \textbf{iff} \ \left\{t_i \stackrel{?}{=} u_i\right\}_{1\leq i\leq n} \Rightarrow^* \sigma$$

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 $\{y_i \stackrel{?}{=} u_i \mid 1 \leq i \leq n\}$, 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_j by its definition u_i until no variable y_i occurs in some u_j, i.e., (see Recall 2.8) when this system of equation is acyclic. Hence any acyclic set of equations $\{y_i = u_i \mid 1 \leq i \leq n\}$ defines the substitution $\sigma = \mathbf{mgu}(\{y_i = u_i \mid 1 \leq i \leq n\})$.

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 $\sigma = \mathbf{match}(F_0, X)$ 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 $\left\{p_i \stackrel{?}{=} d_i\right\}_{1\leq i\leq n} \cup \left\{y_j \stackrel{?}{=} u_j\right\}_{1\leq j\leq m} \Rightarrow^* \sigma_{in} \cup \left\{y_j \stackrel{?}{=} u_j\sigma_{in}\right\}_{1\leq j\leq m}$ using only the rules (1) and (5). Now, by applying iteratively rule (5) one obtains

$$\sigma_{in} \cup \left\{y_j \stackrel{?}{=} u_j\sigma_{in}\right\}_{1\leq j\leq m} \Rightarrow^* \sigma_{in} \cup \mathbf{mgu}\,\{y_j = u_j\sigma_{in}\}_{1\leq j\leq m}$$

when the set of equations $\{y_j = u_j\sigma_{in}\}_{1\leq j\leq m}$ satisfies the occur check. Then $\sigma_{in} + \sigma_{out} \Rightarrow^* \sigma$ 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 $\Gamma[R/X_0\rangle\Gamma'$ is defined whenever the left-hand side $\mathrm{lhs}(R)$ of R can be unified with the definition $\mathrm{def}(X_0, \Gamma)$ of X_0 in Γ with

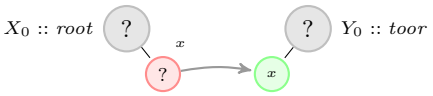
$$\begin{array}{rcl} \Gamma' & = & \{X_0 = R(X_1, \ldots, X_k)\} \\ & \cup & \{X_1 = F_1\sigma, \ldots, X_k = F_k\sigma\} \\ & \cup & \{X = F\sigma \mid (X = F) \in \Gamma \ \wedge \ X \neq X_0\} \end{array}$$

where $\sigma = \mathbf{mgu}(\mathrm{lhs}(R), \mathrm{def}(X_0, \Gamma))$, and $X_1, \ldots, X_k$ are new nodes added to Γ'. Indeed, when unifying $\mathrm{lhs}(R)$ with $\mathrm{def}(X_0, \Gamma)$ one may generate an equation of the form $x = t$ where x is a variable in an inherited data d_i and t is an instance of a corresponding subterm in the associated pattern p_i. 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_i could actually be used to filter out the incoming data value d_i while some other parts of the same pattern would be used to transfert 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 $\Gamma_0 = \{X_0 = root()\langle x\rangle, Y_0 = toor(x)\langle\rangle\}$ shown next



The annotated version $!\Gamma_0 = \{!F \mid F \in \Gamma_0\}$ of configuration Γ_0 is

$$!\Gamma_0 = \left\{X_0 = root()\langle x^?\rangle, Y_0 = toor(x^!)\langle\rangle\right\}$$

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 intial configuration Γ_0. One can start developping it by applying rule Root : $root()\langle u\rangle \rightarrow bin(Nil)\langle u\rangle$ at node X_0 :: *root*. Actually the left-hand side $root()\langle u\rangle$ of rule Root matches with the definition $root()\langle x\rangle$ of X_0 with $\sigma_{in} = \emptyset$ and $\sigma_{out} = \{x = u\}$. Thus $\Gamma_0[\mathrm{Root}/X_0\rangle\Gamma_1$ where the annotated configuration $!\Gamma_1$ is given in Figure 4.

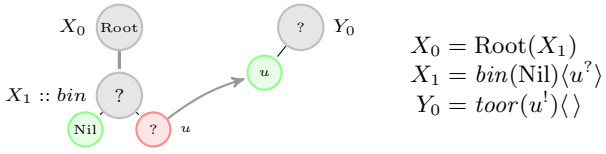


Figure 4: Configuration Γ_1

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

$$\text{Fork} : bin(x)\langle y\rangle \to bin(z)\langle y\rangle\ bin(x)\langle z\rangle$$

at node $X_1 :: bin$ since its left-hand side $bin(x)\langle y\rangle$ matches with the definition $bin(\text{Nil})\langle u\rangle$ of X_1 with $\sigma_{in} = \{x = Nil\}$ and $\sigma_{out} = \{u = y\}$. Hence $\Gamma_1[\text{Fork}/X_1\rangle\Gamma_2$ where $!\Gamma_2$ is given in Figure 5.

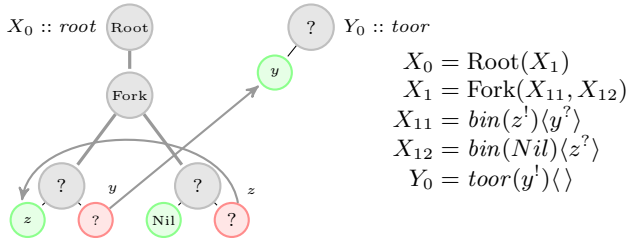


Figure 5: Configuration Γ_2

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

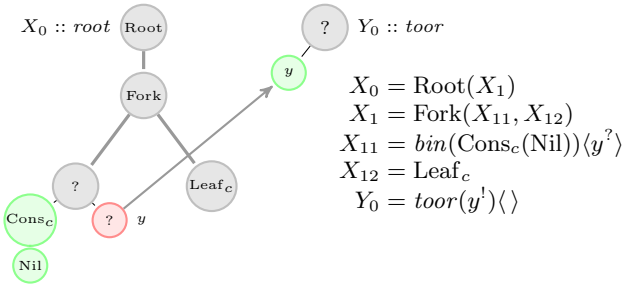


Figure 6: Configuration Γ_3

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

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

Rule $\text{Leaf}_a : bin(x)\langle\text{Cons}_a(x)\rangle \to$ applies at node X_{111} since its left-hand side $bin(x)\langle\text{Cons}_a(x)\rangle$ matches with the definition $bin(z)\langle u\rangle$ of X_{111} with $\sigma_{in} = \{x = z\}$ and $\sigma_{out} = \{u = \text{Cons}_a(z)\}$. Hence $\Gamma_4[\text{Leaf}_a/X_{111}\rangle\Gamma_5$ with configuration $!\Gamma_5$ given in Figure 8.

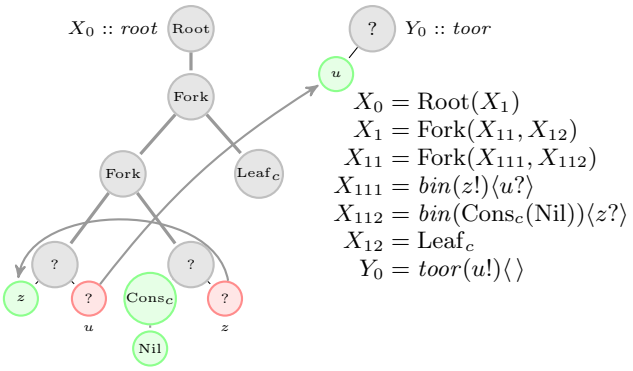


Figure 7: Configuration Γ_4

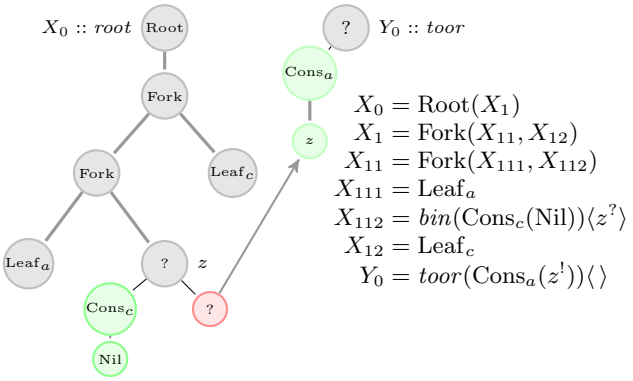


Figure 8: Configuration Γ_5

Using substitution $\sigma_{out} = \{u = \text{Cons}_a(z)\}$ 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.

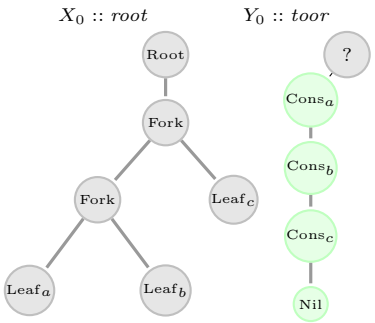
Finally one can apply rule $\text{Leaf}_b : bin(x)\langle\text{Cons}_a(x)\rangle \to$ at node X_{112} since its left-hand side matches with the definition $bin(\text{Cons}_c(\text{Nil}))\langle z\rangle$ of X_{112} with $\sigma_{in} = \{x = \text{Cons}_c(\text{Nil})\}$ and $\sigma_{out} = \{z = \text{Cons}_b(\text{Cons}_c(\text{Nil}))\}$.

Therefore $\Gamma_5[\text{Leaf}_b/X_{112}\rangle\Gamma_6$ with configuration $!\Gamma_6$ 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



$$
\begin{aligned}
X_0 &= \mathrm{Root}(X_1)\\
X_1 &= \mathrm{Fork}(X_{11}, X_{12})\\
X_{11} &= \mathrm{Fork}(X_{111}, X_{112})\\
X_{111} &= \mathrm{Leaf}_a\\
X_{112} &= \mathrm{Leaf}_b\\
X_{12} &= \mathrm{Leaf}_c\\
Y_0 &= \mathit{toor}(\mathrm{Cons}_a(\mathrm{Cons}_b(\mathrm{Cons}_c(\mathrm{Nil}))))\langle\,\rangle
\end{aligned}
$$

Figure 9: Configuration Γ_6

$(x^!, x^?)$ are *transient* in the sense that they disappear as soon as variable x gets defined by the substitution σ_{out} induced by the application of a rule in some open node of the current configuration. If $\sigma_{out}(x)$ is a term t, not reduced to a variable, with variables $x_1, \ldots, x_k$ then vertex v' is refined by the term $t[x_i^!/x_i]$ and new vertices v'_i —associated with these new occurrences of x_i in an output position— are created. The original data link $(x^?, x^!)$ is replaced by all the corresponding instances of $(x_i^?, x_i^!)$. Consequently, a target is replaced by new targets which are the recipients for the subsequent pieces of information –maybe none because no new links are created when t contains no variable. If the term t is a variable y then the link $(x^?, x^!)$ is replaced by the link $(y^?, y^!)$ with the same target and whose source, the (unique) occurrence $x^?$ of variable x, is replaced by the (unique) occurrence $y^?$ of variable y. Therefore the direction of the flow of information is in both cases preserved: Channels can be viewed as "generalized streams" –that can fork or vanish– through which information is pushed incrementally.

Example 2.18. Figure 10 shows a guarded attribute grammar that represents two coroutines communicating through lazy streams. Each process alternatively sends and receives data. More precisely the second process sends an acknowlegment –message $?b$– upon reception of a message sent by the left process. Initially or after reception of an acknowlegment of its previous message the left process can either send a new message or terminate the communication.

Production $!a : q_1(x')\langle a(y')\rangle \leftarrow q_2(x')\langle y'\rangle$ applies at node X_1 of configuration

$$\Gamma_1 = \{X = X_1 \| X_2,\ \ X_1 = q_1(x)\langle y\rangle,\ \ X_2 = q'_2(y)\langle x\rangle\}$$

shown in Figure 11 because its left-hand side $q_1(x')\langle a(y')\rangle$ matches with the definition $q_1(x)\langle y\rangle$ of X_1 with $\sigma_{in} = \{x' = x\}$ and $\sigma_{out} = \{y = a(y')\}$. We get configuration

$$\Gamma_2 \ = \ \left\{ \begin{array}{l} X = X_1 \| X_2,\ \ X_1 = !a(X_{11}),\\ X_2 = q'_2(a(y'))\langle x\rangle,\ \ X_{11} = q_2(x)\langle y'\rangle \end{array} \right\}$$

shown on the middle of Figure 11.

Production $?a : q'_2(a(y))\langle x'\rangle \leftarrow q'_1(y)\langle x'\rangle$ applies at node X_2 of Γ_2 because its left-hand side $q'_2(a(y))\langle x'\rangle$ matches with the definition $q'_2(a(y'))\langle x\rangle$ of X_2 with $\sigma_{in} = \{y = y'\}$ and $\sigma_{out} = \{x = x'\}$. We get configuration

$$\Gamma_3 \ = \ \left\{ \begin{array}{l} X = X_1 \| X_2,\ \ X_1 = !a(X_{11}),\ \ X_2 = ?a(X_{21}),\\ X_{11} = q_2(x')\langle y'\rangle,\ \ X_{21} = q'_1(y')\langle x'\rangle \ \} \end{array} \right\}$$

shown on the right of Figure 11.

The corresponding acknowlegment may be sent and received leading to configuration

$$\Gamma_5 = \Gamma \cup \left\{X_{111} = q_1(x)\langle y\rangle,\ \ X_{211} = q'_2(y)\langle x\rangle\right\}.$$

$$\mathit{where}\ \ \Gamma \ = \ \left\{ \begin{array}{l} X = X_1 \| X_2,\ X_1 = !a(X_{11}),\ X_2 = ?a(X_{21}),\\ X_{21} = !b(X_{211}),\ \ X_{11} = ?b(X_{111}) \end{array} \right\}.$$

The process on the left may decide to end communication by applying production $!\mathrm{stop} : q_1(x')\langle \mathrm{stop}\rangle \leftarrow$ at X_{111} with $\sigma_{in} = \{x' = x\}$ and $\sigma_{out} = \{y = \mathrm{stop}\}$ leading to configuration

$$\Gamma_6 = \Gamma \cup \left\{X_{111} = !\mathrm{stop},\ \ X_{211} = q'_2(\mathrm{stop})\langle x\rangle\right\}.$$

The reception of this message by the process on the right corresponds to applying production $?\mathrm{stop} : q'_2(\mathrm{stop})\langle y\rangle \leftarrow$ at X_{211} with $\sigma_{in} = \emptyset$ and $\sigma_{out} = \{x = y\}$ leading to configuration

$$\Gamma_7 = \Gamma \cup \{X_{111} = !\mathrm{stop},\ \ X_{211} = ?\mathrm{stop}\}.$$

Note that variable x appears in an input position in Γ_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 occurence meaning that the corresponding output attribute is not defined by the semantic rules. As a consequence this variable simply disappears in the resulting configuration Γ_7. If variable x was used in Γ_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.

End of Exple 2.18

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

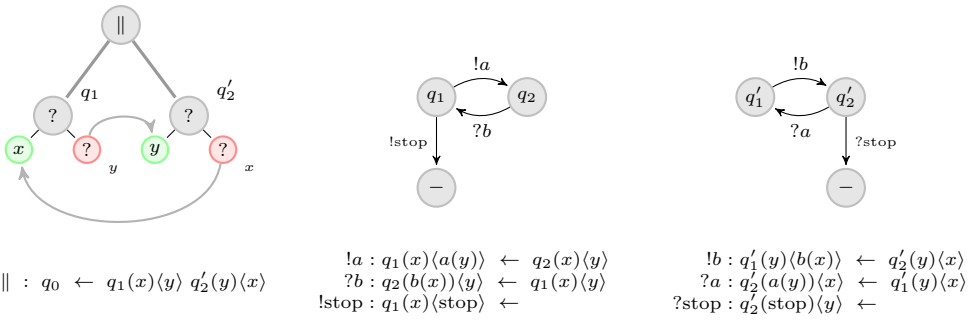


Figure 10: Coroutines with lazy streams

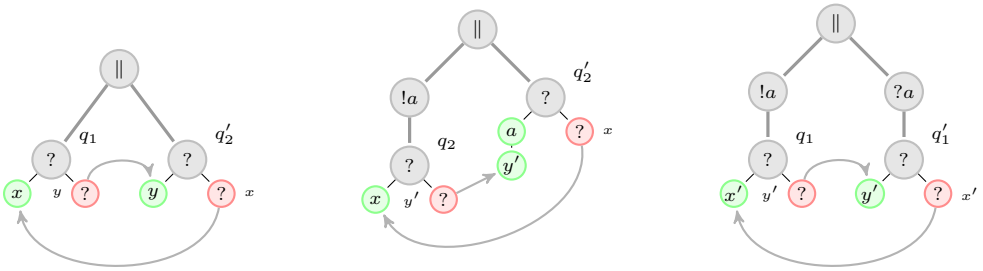


Figure 11: $\Gamma_1[!a/X_1\rangle\Gamma_2[?a/X_2\rangle\Gamma_3$

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.

Futhermore we add an input and an output buffers to each configuration Γ. They contains messages respectively received from and send to distant locations. A message in the input buffer $in(\Gamma)$ is one of the following types.

1. $Y = s(t_1, \ldots, t_n)\langle y_1, \ldots, y_m\rangle$ tells that distant node Y calls service $s \in$ **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 distants 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 $out(\Gamma)$ is one of the following types.

1. $X = s(t_1, \ldots, t_n)\langle y_1, \ldots, y_m\rangle$ tells that local node X calls the external service s –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 $\Gamma \xrightarrow[M]{e} \Gamma'$ stating that event e transforms configuration Γ into Γ' 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 Γ or the consumption of a message from its input buffer. Let us start with the former kind of event: $e = R/X$. let $X = s(t_1, \ldots, t_n)\langle y_1, \ldots, y_m\rangle \in \Gamma$ and $R = F \to F_1 \cdots F_k$ be a rule whose left-hand side matches with X producing substitution $\sigma = \mathbf{match}(F, X)$. 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 σ are local variables. Then we let $\Gamma \xrightarrow[M]{R/X} \Gamma'$ where

$$\begin{array}{rcl}
\Gamma' & = & \{X = R(X_1, \ldots, X_k)\} \\
& \cup & \{X_i = F_i\sigma \mid X_i :: s_i \text{ and } s_i \in N\} \\
& \cup & \{X' = F\sigma \mid (X' = F) \in \Gamma \wedge X' \neq X\} \\
& \cup & \{y = y_j\sigma \mid (y = y_j) \in \Gamma \text{ and } y_j\sigma \text{ is a variable}\} \\
& \cup & \{Y = X \mid (Y = X) \in \Gamma\} \\
M & = & \{X_i = F_i\sigma \mid X_i :: s_i \text{ and } s_i \in T\} \\
& \cup & \{y = y_j\sigma \mid (y = y_j) \in \Gamma \text{ and } y_j\sigma \text{ not a variable}\}
\end{array}$$

where $X_1, \ldots, X_k$ are new names in $ns(G)$. Note that when a distant variable y subscribes to some synthesized attribute of node X, namely $(y = y_i) \in \Gamma$, 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 subcription $y = y_i$ is replaced by subcription $y = x$: Variable y_i delegates 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 $m \in out(\Gamma)$ in the output buffer.

1. If $m = (Y = s(t_1, \ldots, t_n)\langle y_1, \ldots, y_q\rangle)$ then

$$\begin{array}{l}
\Gamma' = \Gamma \cup \left\{\overline{Y} = s(\overline{t}_1, \ldots, \overline{t}_n)\langle \overline{y}_1, \ldots, \overline{y}_q\rangle\right\} \\
\qquad \cup \{y_j = \overline{y_j} \mid 1 \leq j \leq q\} \cup \left\{Y = \overline{Y}\right\}
\end{array}$$

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

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

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

We know 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: $T_i \cap S_j = T_i \cap \mathbf{axioms}(G_j)$ where $s \in T_i \cap \mathbf{axioms}(G_j)$ 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_is and with set of axioms $\mathbf{axioms}(G) = \cup_{1 \leq i \leq p}\mathbf{axioms}(G_j)$. 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, $G \downarrow_{\mathbf{ax}}$, if the global grammar offers only a subset $\mathbf{ax} \subseteq \cup_{1 \leq i \leq p}\mathbf{axioms}(G_j)$ of the services provided by the local grammars.

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

$$T = (\cup_{1 \leq i \leq p} T_i) \setminus (\cup_{1 \leq i \leq p}\mathbf{axioms}(G_j))$$

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 $\mathbf{axiom}(G) \subseteq \cup_{1 \leq i \leq n}\mathbf{axioms}_i$ 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 $\mathbf{axioms}_i$ by first taking the rules whose left-hand sides are forms of sort $s \in \mathbf{axioms}_i$, and then iteratively adding to G_i the rules whose left-hand sides are forms of sort $s \in N \setminus \cup_{j \neq i}\mathbf{axioms}_i$ 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 , Γ_i be 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 destination of every message in the output buffer of a component. If M is a set of messages we let $M_i \subseteq M$ denote the set of messages in M to be forwarded to G_i. Their union $M_\iota = \cup_{1 \leq i \leq p} M_i$ is the set of *internal* messages that circulate between the local grammars. The rest $M_G = M \setminus M_\iota$ 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 $\Gamma_i \xrightarrow[M]{e} \Gamma_i'$ then $\Gamma \xRightarrow[M]{e} \Gamma'$ with $\Gamma_j = \Gamma_j'$ for $j \neq i$.
2. If $\Gamma \xRightarrow[M]{e} \Gamma'$ and $\Gamma' \xRightarrow[M']{m} \Gamma''$ for $m \in M$ then
$$\Gamma \xRightarrow[M \setminus \{m\} \cup M']{e} \Gamma''.$$

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, Γ_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. Then*

1. $\Gamma \xRightarrow[M]{R/X} \Gamma' \implies \exists!\, \Gamma''$ *such that* $\Gamma \xRightarrow[M_G]{R/X} \Gamma''$
2. $\Gamma \xrightarrow[M_G]{e} \Gamma' \iff \Gamma \xRightarrow[M_G]{e} \Gamma'$

PROOF. The first statement follows from the fact that relation $\Gamma \xRightarrow[M]{e} \Gamma'$ is deterministic (M and Γ' are uniquely defined from Γ and e) and the application of a rule produces, directly or indirectly, a finite number of messages.

- If m is of the form $Y = s(t_1, \ldots, t_n)\langle y_1, \ldots, y_n \rangle$, 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 $\bar{y} = y$ 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 Γ be a configuration of G. Then*
$$\Gamma[R/X\rangle\Gamma' \iff \Gamma \xRightarrow[\emptyset]{R/X} \Gamma'$$

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 disbled by the arrival of a message and *(ii)* local actions and incomming 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 guarante 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 σ_{in} –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 $\{y_j = u_j\sigma_{in} \mid 1 \leq j \leq m\}$ is cyclic.

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

$$\begin{array}{lll} P: & s_0(\,)\langle\,\rangle \to s_1(a(x))\langle x\rangle & s_2(x)\langle\,\rangle \\ Q: & s_1(y)\langle a(y)\rangle & \to \\ R: & s_2(a(z))\langle\,\rangle & \to \end{array}$$

Applying P in node X_0 of configuration $\Gamma_0 = \{X_0 = s_0(\,)\langle\rangle\}$ leads to configuration

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

Rule Q is triggered in node X_1 with $\sigma_{in} = \{y = a(x)\}$ but the occur check fails because variable x occurs in $a(y)\sigma_{in} = a(a(x))$. 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 σ_{out} be defined as the least solution of system of equations $\{y_i = u_j\sigma_{in} \mid 1 \leq j \leq m\}$ — 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 a^ω is substituted to variable x and the unique maximal computation associated with the grammar is given by the infinite tree $P(Q, R^\omega)$. 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 by fixed point computation, should in that respect be considered as ill-formed. And indeed this guarded attribute grammar is not *sound* —a notion presented in Section 4.2— because configuration Γ_1 is not closed (it still contains open nodes) but yet it is a terminal configuration that enables no rule. Hence it represents a case that can not be terminated.

End of Exple 4.1

The fact that triggered rules are not enabled can also impact the distributability of a grammar as shown by the following example.

Example 4.2. Let us consider the GAG with the following rules:

$$\begin{array}{ll} P: s(\,)\langle\,\rangle & \to s_1(x)\langle y\rangle\ \ s_2(y)\langle x\rangle \\ Q: s_1(z)\langle a(z)\rangle & \to \\ R: s_2(u)\langle a(u)\rangle & \to \end{array}$$

Rule P is enabled in configuration $\Gamma_0 = \{X_0 = s(\,)\langle\,\rangle\}$ with $\Gamma_0[P/X_0\rangle\Gamma_1$ where

$$\Gamma_1 = \{X_0 = P(X_1, X_2);\ X_1 = s_1(x)\langle y\rangle,\ X_2 = s_2(y)\langle x\rangle\}\,.$$

In configuration Γ_1 rules Q and R are enabled in nodes X_1 and X_2 respectively with $\Gamma_1[Q/X_1\rangle\Gamma_2$ where

$$\Gamma_2 = \{X_0 = P(X_1, X_2);\ X_1 = Q,\ X_2 = s_2(a(x))\langle x\rangle\}$$

and $\Gamma_1[R/X_2\rangle\Gamma_3$ where

$$\Gamma_3 = \{X_0 = P(X_1, X_2);\ X_1 = s_2(a(y))\langle y\rangle,\ X_2 = R\}$$

Now rule R is triggered but not enabled in node X_2 of configuration Γ_2 because of the cyclicity of $\{x = a(a(x))\}$. Similarly, rule Q is triggered but not enabled in node X_3 of configuration Γ_3. There is a conflict between the application of rules R and Q in configuration Γ_1. When the grammar is distributed in such a way that X_1 and X_2 have distinct locations, the specification is not implementable.

End of Exple 4.2

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 autonmous guarded attribute grammar. A* ***case*** *$c = s(t_1, \ldots, t_n)\langle x_1, \ldots, x_m\rangle$ is a ground instanciation 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.*

Substitution σ_{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 σ_{out} is not monotonous 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 $s(d_1, \ldots, d_n)\langle y_1, \ldots, y_m\rangle$ 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 garded attribute grammars.

Definition 4.5 (L-attributed Grammars). *A guarded attribute grammar is* left-attributed*, in short a LGAG, if any variable that is used in an inherited position in some form F of the right-hand side of a rule is either a variable defined in a pattern in the left-hand side of the rule or a variable occurring at a synthesized position in a form which appears at the left of F, i.e., inherited information flows from top-to-bottom and left-to-right between sibling nodes.*

We call the *substitution induced by a sequence* $\Gamma[*\rangle\Gamma'$ the corresponding composition of the various substitutions associated respectively with each of the individual steps in the sequence. If X is an open node in both Γ and Γ', i.e., no rules are applied at node X in the sequence, then we get $X = s(d_1\sigma, \ldots, d_n\sigma)\langle y_1, \ldots, y_m\rangle \in \Gamma'$ where

$$X = s(d_1, \ldots, d_n)\langle y_1, \ldots, y_m\rangle \in \Gamma$$

and σ is the substitution induced by the sequence.

Proposition 4.6 (Monotony). *Let Γ be an accessible configuration of an input-enabled GAG, $X = s(d_1, \ldots, d_n)\langle y_1, \ldots, y_m\rangle \in \Gamma$ and σ be the substitution induced by some sequence starting from Γ. Then*

$$\Gamma[P/X\rangle\Gamma' \quad \text{implies} \quad \Gamma\sigma[P/X\rangle\Gamma'\sigma.$$

PROOF. Direct consequence of Definition 2.3 due to the fact that

1. $\mathbf{match}(p, d\sigma) = \mathbf{match}(p, d)\sigma$, and

2. $\mathbf{mgu}(\{y_j = u_j\sigma \mid 1 \leq j \leq m\}) = \mathbf{mgu}(\{y_j = u_j \mid 1 \leq j \leq m\})\sigma$.

The former is trivial and the latter follows by induction on the length of the computation of the most general unifier –relation $\Rightarrow^*$ 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 $\{y_j = u_j\sigma_{in} \mid 1 \leq j \leq m\}$ satisfies the occur check whereas the set $\{y_j = u_j(\sigma_{in}\sigma) \mid 1 \leq j \leq m\}$ does not satisfy the occur check. *Q.E.D.*

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 Γ be an accessible configuration of an input enabled GAG. If $\Gamma[P/X\rangle\Gamma_1$ and $\Gamma[Q/Y\rangle\Gamma_2$ with $X \neq Y$ then $\Gamma_2[P/X\rangle\Gamma_3$ and $\Gamma_1[Q/Y\rangle\Gamma_3$ for some configuration Γ_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.

End of Remark 4.8

It is difficult to verify input-enabledness as the whole set of accessible configurations is involved in this condition. Nevertheless one can find a sufficient condition for input enabledness, similar to the strong non-circularity of attribute grammars [9], that can be checked by a simple fixpoint computation.

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 $X = s(d_1, \ldots, d_n)\langle y_1, \ldots, y_m\rangle \in \Gamma$ where Γ is an accessible configuration and $y_j \in d_i$. If R is a rule with left-hand side $s(p_1, \ldots, p_n)\langle u_1, \ldots, u_m\rangle$ we let $(i, j) \in IS(R)$ if there exists a variable $x \in \mathrm{var}(R)$ such that $x \in \mathrm{var}(d_i) \cap \mathrm{var}(u_j)$. 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 σ_{in} such that $y_j \in u_i\sigma_{in}$ 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 $G(s, R)$. *Q.E.D.*

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 $R : F_0 \to F_1 \cdots F_\ell$ 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 $k\langle j\rangle$) denote the occurrence of the i^{th} inherited attribute (resp. the j^{th} synthesized attribute) in F_k. If s is a sort with n inherited attributes and m synthesized attributes we define the relations $\overline{IS(s)}$ and $\overline{SI(s)}$ over $[1, n] \times [1, m]$ and $[1, m] \times [1, n]$ respectively as the least relations such that:*

1. *For every rule $R : F_0 \to F_1 \cdots F_\ell$ where form F_i is of sort s_i and for every $k \in [1, \ell]$*

$$\left\{(j, i) \;\middle|\; (k\langle j\rangle, k(i)) \in GLD(R)^k\right\} \subseteq \overline{SI(s_k)}$$

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

$$GLD(R) \cup \left\{(0\langle j\rangle, 0(i)) \;\middle|\; (j, i) \in \overline{SI(s_0)}\right\}$$
$$\cup \left\{(k'(i), k'\langle j\rangle) \;\middle|\; k' \in [1, \ell],\ k' \neq k,\ (i, j) \in \overline{IS(s_{k'})}\right\}$$

2. *For every rule $R : F_0 \to F_1 \cdots F_\ell$ where form F_i is of sort s_i*

$$\left\{(i, j) \;\middle|\; (0(i), 0\langle j\rangle) \in GLD(R)^0\right\} \subseteq \overline{IS(s_0)}$$

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

$$GLD(R) \cup \left\{(k(i), k\langle j\rangle) \;\middle|\; k \in [1, \ell],\ (i, j) \in \overline{IS(s_k)}\right\}$$

*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 $\mathrm{var}(d_i\sigma) \cap \mathrm{var}(y_j\sigma) \neq \emptyset$ for some form $F = s(d_1,\ldots,d_n)\langle y_1,\ldots,y_m\rangle$ of sort s and where σ 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 $IS(s) \subseteq \overline{IS(s)}$ and $SI(s) \subseteq \overline{SI(s)}$. *Q.E.D.*

Note that the following two inclusions are strict

strongly-acyclic GAG $\subsetneq$ acyclic GAG $\subsetneq$ input enabled GAG

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

$$\begin{cases} A(x)\langle z\rangle \to B(a(x,y))\langle y,z\rangle \\ B(a(x,y))\langle x,y\rangle \to \end{cases}$$

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

$$\begin{cases} A(x)\langle z\rangle \to B(y,x)\langle z,y\rangle \\ A(x)\langle z\rangle \to B(x,y)\langle y,z\rangle \\ B(x,y)\langle x,y\rangle \to \end{cases}$$

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 $c = s(t_1,\ldots,t_n)\langle y_1,\ldots,y_m\rangle$ 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 $s(t_1,\ldots,t_n)\langle y_1,\ldots,y_m\rangle$ 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 developped 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 $\mathbf{IS}(s) = [1,n]\times[1,m]$ and $\mathbf{SI}(s) = [1,m]\times[1,n]$ 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 $(a,g) \in \mathbf{AG}(G)$ with $a \in \mathbf{SI}(s_1)\times\cdots\times\mathbf{SI}(s_k)\times\mathbf{IS}(s'_1)\times\cdots\times\mathbf{IS}(s'_{k'})$ and $g \in \mathbf{IS}(s_1)\times\cdots\times\mathbf{IS}(s_k)\times\mathbf{SI}(s'_1)\times\cdots\times\mathbf{SI}(s'_{k'})$. Equivalently it is given by the data $SI(s) \in \mathbf{SI}(s)$ and $IS(s) \in \mathbf{IS}(s)$ for $s \in \mathbf{axioms}(G) \cup T$. 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 $SI(s) \subseteq \overline{SI(s)}$ for $s \in \mathbf{axioms}(G)$ and $IS(s) \subseteq \overline{IS(s)}$ for $s \in T$ are added, allows to conclude strong-acyclicity with $\overline{IS(s)} \subseteq IS(s)$ for $s \in \mathbf{axioms}(G)$ and $\overline{SI(s)} \subseteq SI(s)$ for $s \in T$.*

The data $SI(s) \in \mathbf{SI}(s)$ and $IS(s) \in \mathbf{IS}(s)$ 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 $SI(s) \in \mathbf{SI}(s)$ and $IS(s) \in \mathbf{IS}(s)$ 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 $(a_i,g_i) \in \mathbf{AG}(G_i)$ for every local grammar and for the global grammar as well: $(a,g) \in \mathbf{AG}(G)$. 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 $\overline{SI(s)}$ is left unmodified during the fixpoint computation, it keeps its initial value $SI(s)$. Similarly, rule 2 in Def. 4.11 never applies for an axiom s and $\overline{IS(s)}$ 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 $\overline{SI(s)}$ and $\overline{IS(s)}$ 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 $(i, j) \notin IS(s)$ (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:

$$
\begin{array}{ll}
G = \langle G_1, \ldots, G_p \rangle & \\
\textbf{where} & G_1 :: \%\text{definition of } G_1 \\
& \vdots \\
& G_p :: \%\text{definition of } G_p
\end{array}
$$

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 $c = s_0(t_1, \ldots, t_n)\langle x_1, \ldots, x_m \rangle$ is a ground instanciation 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 $(\Gamma_i)_{0<i\leq n}$ and the infinite sequences $(\Gamma_i)_{0<i<\omega}$ of accessible configurations such that $\Gamma_i[\rangle\Gamma_{i+1}$. 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 $instr_i$ for $i = 1, \ldots, N$ of one of the three following forms

1. **INC(r, i):** increment register r and go to instruction i.
2. **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. **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 $instr_k = \text{INC}(r_1, i)$ then add rule
$$\text{Inc}(k, 1, i) : s_k(x, y) \to s_i(\mathbf{succ}(x), y)$$
2. If $instr_k = \text{INC}(r_2, i)$ then add rule
$$\text{Inc}(k, 2, i) : s_k(x, y) \to s_i(x, \mathbf{succ}(y))$$
3. If $instr_k = \text{JZDEC}(r_1, i, j)$ then add the rules
$$\begin{array}{ll} \text{Jz}(k, 1, i) & : s_k(\mathbf{zero}, y) \to s_i(\mathbf{zero}, y) \\ \text{Dec}(k, 1, j) & : s_k(\mathbf{succ}(x), y) \to s_j(x, y) \end{array}$$
4. If $instr_k = \text{JZDEC}(r_2, i, j)$ then add the rules
$$\begin{array}{ll} \text{Jz}(k, 2, i) & : s_k(x, \mathbf{zero}) \to s_i(x, \mathbf{zero}) \\ \text{Dec}(k, 2, j) & : s_k(x, \mathbf{succ}(y)) \to s_j(x, y) \end{array}$$

5. If $instr_k = \text{HALT}$ then add rule

$$\text{Halt}(k) : s_k(x, y) \rightarrow$$

Since there is a unique maximal firing sequence from the initial configuration $\Gamma_0 = \{X_0 = s_1(\mathbf{zero}, \mathbf{zero})\}$ 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 *hierarchical composition* preserves soundness. This result allows the construction of a large class of specifications which are sound by construction.

DEFINITION 4.18 (HIERARCHICAL COMPOSITION). *A composition $G = G_0 \oplus \cdots \oplus G_n$ is **hiearchical** 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 *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 hiarchical decompositions.

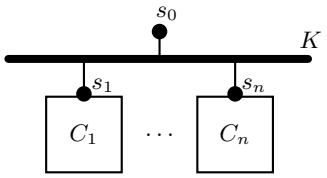


Figure 12: A hierarchical composition of GAGs

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

DEFINITION 4.19. *A guarded attribute grammar is a **connector** if it satisfies the following two conditions:*

1. ***Weak-Soundness.** For every accessible configuration Γ there exists a configuration accessible from Γ all of whose open nodes have sorts that are terminal symbols.*
2. ***Independence w.r.t. Components.** For any accessible configuration Γ, open node X, and substitution σ for variables subscribing to external services (i.e., occurring in a synthesized position in an open node of Γ whose sort is a terminal symbol), and for any rule R one has $\Gamma[R/X\rangle \Leftrightarrow \Gamma\sigma[R/X\rangle$, 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.

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.*

PROOF. Let Γ be an accessible configuration of C. By Independence w.r.t. Components there exists sequences of rule applications $\Gamma_0[*\rangle\Gamma_1[*\rangle\Gamma$ where rules in the first sequence belong to the connector, and in the second sequence belong to the components. By Weak-Soundness, as Γ_1 is a configuration of the connector, one can find a sequence of rule applications in the connector $\Gamma_1[*\rangle\Gamma_2$ such that the sort of an open node in Γ_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 Γ_3 which is accessible both from Γ and from Γ_2. The sorts of open nodes of Γ_3 are found in the components. By Soundness of the components and Monotony (which also follows from Input-Enabledness) $\Gamma_3[*\rangle\Gamma_4$ where Γ_4 is a closed configuration. *Q.E.D.*

5. TOWARDS A LANGUAGE FOR THE SPECIFICATION OF GAG

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 syntatic 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 instanciations 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

$$\begin{aligned} &sort(p_1,\ldots,p_n)\langle u_1,\ldots,u_m\rangle \rightarrow \\ &\qquad sort_1(t_1^{(1)},\ldots,t_{n_1}^{(1)})\langle y_1^{(1)},\ldots,y_{m_1}^{(1)}\rangle \\ &\qquad \cdots \\ &\qquad sort_k(t_1^{(k)},\ldots,t_{n_k}^{(k)})\langle y_1^{(k)},\ldots,y_{m_k}^{(k)}\rangle \end{aligned}$$

as

$$\begin{aligned} &sort(p_1,\ldots,p_n) \;= \\ &\quad \textbf{do} \quad (y_1^{(1)},\ldots,y_{m_1}^{(1)}) \leftarrow sort_1(t_1^{(1)},\ldots,t_{n_1}^{(1)}) \\ &\qquad \cdots \\ &\qquad (y_1^{(k)},\ldots,y_{m_k}^{(k)}) \leftarrow sort_k(t_1^{(k)},\ldots,t_{n_k}^{(k)}) \\ &\qquad \textbf{return}\ (u_1,\ldots,u_m) \end{aligned}$$

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*

$$(y_1^{(j)},\ldots,y_{m_j}^{(j)}) \leftarrow sort_j(t_1^{(j)},\ldots,t_{n_j}^{(j)})$$

Note that, using this **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 **do** expression that follows the generator, including the return statement.

Second, a Haskell monadic expression is evaluated in *pull* mode: If the output returned by the **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 *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:

$$\begin{aligned} &\text{Root} \;:\; root()\langle x\rangle \rightarrow bin(\text{Nil})\langle x\rangle \\ &\text{Fork} \;:\; bin(x)\langle y\rangle \rightarrow bin(z)\langle y\rangle\ bin(x)\langle z\rangle \\ &\text{Leaf}_a \;:\; bin(x)\langle \text{Cons}_a(x)\rangle \rightarrow \end{aligned}$$

Its syntactical translation into the functional notation is the following:

$$\begin{array}{lllll} \text{Root}: & \text{root}() & = & \textbf{do} & (x) \leftarrow bin(\text{Nil}) \\ & & & & \textbf{return}\ (x) \\ \text{Fork}: & \text{bin}(x) & = & \textbf{do} & (z) \leftarrow bin(x) \\ & & & & (y) \leftarrow bin(z) \\ & & & & \textbf{return}\ (y) \\ \text{Leaf}_a & \text{bin}(x) & = & \textbf{do} & \textbf{return}\ (\text{Cons}_a(x)) \end{array}$$

which we write more simply as

$$\begin{array}{lllll} \text{Root}: & \text{root}() & = & & bin(\text{Nil}) \\ \text{Fork}: & \text{bin}(x) & = & \textbf{do} & (z) \leftarrow bin(x) \\ & & & & (y) \leftarrow bin(z) \\ & & & & \textbf{return}\ (y) \\ \text{Leaf}_a & \text{bin}(x) & = & \textbf{return} & (\text{Cons}_a(x)) \end{array}$$

using the simplification rules given below.

End of Exple 5.1

($\mathbf{SR}_1$) When the returned value is the result of the last generator, one replaces these two instructions by the last call, e.g.:

$$\begin{array}{lll} \textbf{do} & (y) \leftarrow bin\,(\text{Nil}) & \\ & \textbf{return}\ (y) & \Leftrightarrow \quad \textbf{do}\ bin\,(\text{Nil}) \end{array}$$

($\mathbf{SR}_2$) When the **do** sequence is reduced to a unique item –either a call or a **return** statement– one omits the **do** instruction, e.g.:

$$\begin{array}{lll} \textbf{do}\ bin(\text{Nil}) & \Leftrightarrow & bin(\text{Nil}) \\ \textbf{do}\ \textbf{return}\ (\text{Cons}_a(x)) & \Leftrightarrow & \textbf{return}\ (\text{Cons}_a(x)) \end{array}$$

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 *disjoint* 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 instanciating the parameter by an identifier of the corresponding reviewer. For instance evaluate[*reviewer*] is a generic sort and an actual sort is evaluate[Paul] where Paul is a reviewer. 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(evaluate[*reviewer*]) **where**
reviewer = Paul | Ann | ...
editorial_decision :: %Grammar editorial_decision
evaluate :: %Description of the grammar evaluate

This contruct 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[*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*]) **where**
editor = Mary | Frank | ...
editorial_process :: %Grammar editorial_process
editorial_decision ::
  editorial_decision(evaluate[*reviewer*]) **where**
  reviewer = Paul | Ann | ...
  editorial_decision :: %...
  evaluate :: %...

The convention to name a GAG by its axiom entails some overloading of notations because in a hierarchical composition, the axiom of a global grammar coincides with the axiom of the connector. Nevertheless this overloading creates 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[$reviewer1, reviewer2$]
where not($reviewer1 = reviewer2$) :
submission($article$) =
  do $report1 \leftarrow$ evaluate[$reviewer1$]
    $report2 \leftarrow$ evaluate[$reviewer2$]
    decide($report1, report2$)

Evaluate_Submission[$reviewer1, reviewer2$] is a *generic rule*, the corresponding actual rules are obtained by choosing values 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 :
submission($article$) =
  input ($reviewer1, reviewer2$)
  where not($reviewer1 = reviewer2$)
  do $report1 \leftarrow$ evaluate[$reviewer1$]
    $report2 \leftarrow$ evaluate[$reviewer2$]
    decide($report1, report2$)

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($article$). The corresponding instance of the generic rule, for instance

Evaluate_Submission[Paul, Ann]

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

Accept :
evaluate($article$) =
  input ($msg :: String$)
  do $report \leftarrow$ review($article$)
    return (Yes($msg, report$))

With this rule the reviewer informs the Editor that he accepts 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 subsequently 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 system. 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 :
evaluate[$reviewer$]($article$) =
  input ($msg :: String$)
  do $report \leftarrow$ review[$reviewer$]($article$)
    return (Yes($msg, report$))

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[Paul]($article$), which belongs to the active workspace of Paul, only the instance of the rule associated with Paul can apply: Mary has

no possibility to accept to review a paper that was not send to her.

All parameters of rules appear either in an **input** clause or in its left-hand side. Therefore they are always left implicite in the name of the rule.

We add the following rule to the simplification rules introduced in Sect. 5.1:

($\mathbf{SR}_3$) We omit the **do** statement if it merely returns the value(s) introduced by the preceding **input** clause:

$$\begin{array}{l}\textbf{input}\ (x)\\ \textbf{return}\ (x)\end{array} \quad \Leftrightarrow \quad \textbf{input}\ (x)$$

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

$$\begin{array}{lll}\textbf{many}\ s\ (\mathrm{Cons}(\mathrm{head},\mathrm{tail})) = \textbf{do} & \mathrm{head}' \leftarrow s(\mathrm{head}) \\ & \mathrm{tail}' \leftarrow \textbf{many}\ s\ (\mathrm{tail}) \\ & \textbf{return}\ (\mathrm{Cons}(\mathrm{head}',\mathrm{tail}')) \\ \textbf{many}\ s\ (\mathrm{Nil}) = \textbf{return}\ (\mathrm{Nil}) & \end{array}$$

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** $s :: a^* \rightsquigarrow b^*$ when $s :: a \rightsquigarrow b$– 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:

$$\begin{array}{lll}\langle s, s'\rangle :: () \rightsquigarrow () & & \\ \textbf{when} \quad s :: a \rightsquigarrow b & & \\ \qquad\quad s' :: b \rightsquigarrow a & & \\ \langle s, s'\rangle = & \textbf{input}\ (x)\ \textbf{where}\ x :: a & \\ & \textbf{do} \quad ys \leftarrow \textbf{many}\ s\ (\mathrm{Cons}(x, xs)) & \\ & \qquad xs \leftarrow \textbf{many}\ s'\ (ys) & \\ & \qquad \textbf{return}\ () & \end{array}$$

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:

$$\begin{array}{lll}\langle s, s'\rangle\ in\ out :: c \rightsquigarrow d & & \\ \textbf{when} \quad s :: a \rightsquigarrow b & & \\ \qquad\quad s' :: b \rightsquigarrow a & & \\ \qquad\quad in :: c \to a & & \\ \qquad\quad out :: a^* \times b^* \to d & & \\ \langle s, s'\rangle\ in\ out = & \textbf{do} \quad ys \leftarrow \textbf{many}\ s\ (\mathrm{Cons}(in\ x, xs)) & \\ & \qquad xs \leftarrow \textbf{many}\ s'\ (ys) & \\ & \qquad \textbf{return}\ (out\ (xs, ys)) & \end{array}$$

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:

$$\begin{array}{ll}[s, s'] = \langle s, s'\rangle id\ zip :: a \rightsquigarrow (a \times b)^* & \\ \textbf{when} \quad s :: a \rightsquigarrow b & \\ \qquad\quad s' :: b \rightsquigarrow a & \end{array}$$

where $id :: a \to a$ is the identity function and $zip :: a^* \times b^* \to (a \times b)^*$ is the function given by:

$$\begin{array}{l} zip\ \mathrm{Nil}\ ys = \mathrm{Nil} \\ zip\ xs\ \mathrm{Nil} = \mathrm{Nil} \\ zip\ \mathrm{Cons}(x, xs)\ \mathrm{Cons}(y, ys) = \mathrm{Cons}((x, y), zip\ xs\ ys) \end{array}$$

This combinator can equivalently be specified as

$$\begin{array}{lll}[s, s'] :: a \rightsquigarrow (a \times b)^* & & \\ \textbf{when} \quad s :: a \rightsquigarrow b & & \\ \qquad\quad s' :: b \rightsquigarrow a & & \\ [s, s'] = & \textbf{do} \quad ys \leftarrow \textbf{many}\ s\ (\mathrm{Cons}(x, xs)) & \\ & \qquad xs \leftarrow \textbf{many}\ s'\ (ys) & \\ & \qquad \textbf{return}\ (zip\ xs\ ys) & \end{array}$$

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

profiles [6, 12, 44, 3]. Each actor plays a specific role in the system and offers a number of services needed by other stakeholders. Futhermore, the overall flow of tasks and activities is highly dependent on the available data and other contextual variables. For the purpose of the illustration, the scenario has been slightly adapted and does not therefore necessarily depict what happens in a real surveillance system.

The modeled scenario describes a situation in which three sets of actors with distinct roles (Epidemiologist, Physician, Biologist) actively participate in the surveillance and investigation of outbreaks of Influenza. An artifact in this scenario contains all the information pertaining to the treatment of a suspect case. The process starts with patient visits at a physician's office. The physician receives patients, registers the signs and symptoms, and verifies whether they correspond with those contained in the Influenza declaration criteria. If the verification is successful, he immediately declares the patient as a suspect case to the Disease Surveillance Center (DSC) (**caseDeclaration**). If the declared data contains saliva samples, the latter are sent to the biologist for laboratory analysis (**laboratoryAnalysis**). In parallel, the data is automatically analyzed by an epidemiologist (**dataAnalysis**) and eventually, outbreak alarms are produced. A number of verification tasks are run on the data and the analysis results to ascertain the alarm. The epidemiologist eventually creates a list of actions (*todo* list) that will have to be carried out by the physician to complete the alarm verification (**acmCheck**). Alongside these activities, he immediately informs Public Health Officials of the situation. Results from the laboratory analyses carried out by the biologist are used together with the results from the above checks to either confirm or revoke the outbreak alarm and produce an outbreak alert (**outbreakDecl**). Based on the outbreak characteristic data, the epidemiologist analyses the risks related to the outbreak alert and proposes appropriate counter measures.

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 case analysis is given by a hierarchical composition combining the roles of biologists and epidemiologists through an homonymous connector —the connector and the compound specification have the same axiom associated with service **caseAnalysis**.

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.

```
visit(patient, alarm) =
 do (symps) ← clinicalAssessment(patient)
    () ← initialCare(symps)
    caseDeclaration(patient, symps, 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.

```
clinicalAssessment(patient) = input (symps)
```

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

```
initialCare(symps) = input (care)
                     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.

```
caseDeclaration(patient, symps, alarm) =
 input (samples)
 do (alarm) ← caseAnalysis (SuspectCase (patient,
                                         symps,
                                         samples),
                            checkRes)
    (checkRes) ← acmCheck(alarm)
    return ()
```

Note that this rule is not left-attributed: There are mutual dependencies between the subtasks **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.

```
caseDeclaration(patient, symps, _) = return ()
```

Here we have replaced variable *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

the case is not reported to the DSC and thus the analysis, from which the alarm could have been produced, is not executed for this case.

Finally if an alarm is raised –*alarm*=Alarm(*info*,*todo*)– containing analysis information together with a list of verification tasks, then the physician performs the corresponding checks on the case and transmits the results.

$$\text{acmCheck}(\text{Alarm}(\mathit{info}, \mathit{todo})) \;=\; \textbf{input}\ (\mathit{checkRes})$$

Else, the following rule applies

$$\text{acmCheck}(\text{NoAlarm}) \;=\; \textbf{return}\ (\_)$$

Again the wildcard indicates a variable that is not instanciated –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 physician and to do statistical analyses and other disease surveillance related computations on the reported data in order to detect and investigate disease outbreaks. These analyses can produce an alarm.

caseAnalysis (SuspectCase(*patient*, *symps*, *samples*), *checkRes*) =
  input (*bio*, *epi*)
  do (*labRes*) ← **laboratoryAnalysis**[*bio*](*samples*)
    dataAnalysis[*epi*](*patient*, *symps*, *labRes*, *checkRes*)

Note the use of simplification rule $\mathbf{SR}_2$ due to the fact that the result returned by caseAnalysis is the result of **dataAnalysis**.

*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

$$\textbf{laboratoryAnalysis}(\mathit{samples}) \;=\; \textbf{input}\ (\mathit{labResult})$$

*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 (outbreak alarms). This is followed by investigation tasks to better characterize the outbreak alarms. These investigative tasks (aspecific counter measure checks) may involve superposing 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 defined by the following rule,

dataAnalysis(*patient*, *symps*, *labResult*, *checkResult*) =
  do (*ack*) ← storeCaseData(*patient*, *symps*)
    automatedAnalysis(*ack*, *labResult*, *checkResult*)

The epidemiologist stores the current case data in the surveillance database.

$$\text{storeCaseData}(\mathit{patient}, \mathit{symps}) \;=\; \textbf{return}\ (\text{Ack})$$

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

automatedAnalysis(Ack, *labResult*, *checkResult*) =
  input (*info*, *todo*)
  do () ← notifyAuth(*info*)
    () ← outbreakDecl(*labResult*, *checkResult*)
    return (Alarm(*info*, *todo*))

Note that the alarm is emitted in the first place, prior to notifying authorities and the outbreak declaration. And these two tasks can be performed concurrently. This clearly illustrates 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.

$$\text{automatedAnalysis}(\text{Ack}, \_, \_) \;=\; \textbf{return}\ (\text{NoAlarm})$$

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 authorities.

$$\text{notifyAuth}(\mathit{info}) \;=\; \textbf{return}\ ()$$

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 laboratory results provided respectively by the physician and the biologist, the epidemiologist runs a number of alert analyses 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 communicates to public health authorities for action.

outbreakDecl(*labResult*, *checkResult*) =
  input (*alertInfos*)
  do (*risks*) ← riskAnalysis(*alertInfos*)
    (*counterM*) ← defineCounterMeasures(*risks*)
    feedback(*alertInfos*, *counterM*)

$$\text{riskAnalysis}(\mathit{alertInfos}) \;=\; \textbf{input}\ (\mathit{risks})$$

$$\text{defineCounterMeasures}(\mathit{risks}) \;=\; \textbf{input}\ (\mathit{counterM})$$

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 sendFeedback 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 dabase 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

task($args$) = **input** ($results$)

corresponding to tasks that merely return values inputted by the user can be used for incremental specifications. 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 specification that can be executed and tested even though it will require the user to manually input the expected results. Then this temporary rule can progessively be refined to obtain a new rule where the results are no longer inputted by the user but synthesized from intermediate 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 hiearchical 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 associated with the services that she delivers. Each map consists of the set of artifacts created by the invocations of the corresponding service. An artifact records all the information related to the treatment of the related service call. It contains 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 relevant information needed for the treatment of the pending tasks.

Using a variant of attribute grammars, called guarded attribute grammars, one can automate the flow of information in collaborative activities with business rules that put emphasis on user's decisions. We gave an in-depth description 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 architecture. 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 introduced in the system can reach completion. This property is also undecidable but we have defined a hierarchical composition 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 grammars 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 execution, 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 constrain concurrency among tasks in an artificial way.

Modularity.

The GAG approach also facilitates a modular description of business processes. For instance, when laboratory test requests 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 contributes to enhance modularity. Finally, some techniques borrowed from attribute grammars, like descriptional 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. Henthforth, 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. Morevover, 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 coercitive 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 surveillance) 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 formalism 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 asynchronous 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 expressiveness to the textual descriptions of collaborative case management systems.

Applicability and Pertinence.

The development of case studies is very important to check the pertinence, level of applicability, and practical limitations of the GAG model. These case studies are also important to refined the specification language that we have sketched in Section 5 and to extract from practise useful concepts 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 examples are representative case studies in which our model can provide advantages over existing techniques. **Reporting systems** where several stakeholders collaborate to build a report. The grammar can reflect the structure of the report, the identification of stakeholders and their respective contributions. The semantic rules implement the automatic assembly of the report from the bits provided by the distributed 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 synthesized attribute for further use. Guarded attribute grammars 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 directly 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 establishment of an Internet connection. Also, the declarative decomposition of learning activities, which do not impose particular execution order, gives more flexibility in the design and description or learning activities and more freedom to the learner in the learning path.

8. REFERENCES

[1] S. Abiteboul, J. Baumgarten, A. Bonifati, G. Cobena, C. Cremarenco, F. Dragan, I. Manolescu, T. Milo, and N. Preda. Managing distributed workspaces with active XML. In *VLDB*, pages 1061–1064, 2003.

[2] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: A data-centric perspective on web services. In *BDA'02*, 2002.

[3] P. Astagneau and T. Ancelle. *Surveillance épidémiologique*. Lavoisier, 2011.

[4] J.L. Austin. *How to Do Things with Words*. Oxford Univ. Press, 1962.

[5] K. Backhouse. A functional semantics of attribute grammars. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 2280 of *LNCS*, pages 142–157. Springer, 2002.

[6] H. Chen, D. Zeng, and P. Yan. *Infectious Disease Informatics: Syndromic Surveillance for Public Health and Bio-Defense*. Springer, 1st edition, 2009.

[7] L.M. Chirica and D.F. Martin. An order-algebraic definition of Knuthian semantics. *Mathematical Systems Theory*, 13:1–27, 1979.

[8] World Health Organization / Centers For Disease Control. Technical Guidelines for Intergrated Disease Surveillance and Response in the African Region. Technical report, WHO/CDC, Georgia, USA, 2001.

[9] B. Courcelle and P. Franchi-Zannettacci. Attribute grammars and recursive program schemes *(i)* and *(ii)*. *Theor. Comput. Sci.*, 17:163–191 and 235–257, 1982.

[10] E. Damaggio, A. Deutsch, and V. Vianu. Artifact systems with data dependencies and arithmetic. *ACM Trans. Database Syst.*, 37(3):22, 2012.

[11] E. Damaggio, R. Hull, and R. Vaculín. On the equivalence of incremental and fixpoint semantics for business artifacts with guard-stage-milestone lifecycles. *Inf. Syst.*, 38(4):561–584, 2013.

[12] V. Dato, R. Shephard, and M.M. Wagner. Outbreaks and investigations. In M.M. Wagner, A.W. Moore, and R.M. Aryel, editors, *Handbook of Biosurveillance*, pages 13 – 26. Academic Press, Burlington, 2006.

[13] G. De Michelis and C.A. A. Ellis. Computer supported cooperative work and Petri nets. In *Advanced Course on Petri Nets, Dagstuhl 1996*, volume 1492 of *LNCS*, pages 125–153. Springer, 1998.

[14] P. Deransart and J. Maluszynski. Relating logic programs and attribute grammars. *J. Log. Program.*, 2(2): 119–155, 1985.

[15] P. Deransart and J. Maluszynski. *A grammatical view of logic programming*. MIT Press, 1993.

[16] S. Doaitse Swierstra, P.R. Azero Alcocer, and J. Saraiva. Designing and implementing combinator languages. In *Advanced Functional Programming*, pages 150–206, 1998.

[17] C.A. Ellis and K. Keddara. Ml-dews: Modeling language to support dynamic evolution within workflow

systems. *Computer Supported Cooperative Work*, 9(3/4):293–333, 2000.

[18] R. Eshuis, R. Hull, Y. Sun, and R. Vaculín. Splitting GSM schemas: A framework for outsourcing of declarative artifact systems. In *BPM*, vol. 8094 of *LNCS*, pages 259–274. Springer, 2013.

[19] J. Fokker. Functional parsers. *Advanced Functional Programming, First International Spring School*, vol. 925 of *LNCS*, pages 1–23, Springer 1995.

[20] H. Ganzinger. Increasing modularity and language-independency in automatically generated compilers. *Sci. Comput. Program.*, 3(3):223–278, 1983.

[21] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *SIGPLAN Symposium on Compiler Construction*, pages 157–170. ACM, 1984.

[22] Object Management Group. Business process model and notation, v 2.0. Technical report, OMG, 2011. http://www.bpmn.org/.

[23] L. Hélouët and A. Benveniste. Document based modeling of web services choreographies using active XML. In *ICWS*, pages 291–298. IEEE Computer Society, 2010.

[24] R. Hull. Artifact-centric business process models: Brief survey of research results and challenges. In *OTM 2008*, volume 5332 of *LNCS*, pages 1152–1163. Springer, 2008.

[25] R. Hull, E. Damaggio, R. De Masellis, F. Fournier, M. Gupta, F.T. Heath, S. Hobson, M.H. Linehan, S. Maradugu, A. Nigam, P.N. Sukaviriya, and R. Vaculín. Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In *Fifth ACM International Conference on Distributed Event-Based Systems, DEBS 2011*, pages 51–62. ACM, 2011.

[26] T. Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture, FPCA*, vol. 274 of *LNCS*, pages 154–173. Springer, 1987.

[27] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. on Soft. Eng. Meth.*, 14(3): 331–380, 2005.

[28] D.E. Knuth. Semantics of context free languages. *Mathematical System Theory*, 2(2):127–145, 1968.

[29] N. Lohmann and K. Wolf. Artifact-centric choreographies. In *Service-Oriented Computing - 8th Int. Conf., ICSOC 2010.*, pages 32–46, 2010.

[30] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2): 258–282, 1982.

[31] B.H. Mayoh. Attribute grammars and mathematical semantics. *SIAM J. Comput.*, 10(3):503–518, 1981.

[32] A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Syst. J.*, 42:428–445, July 2003.

[33] OASIS. Web services business process execution language. Tech. report, OASIS, 2007. docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf.

[34] J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.

[35] J. Saraiva and S.D. Swierstra. Generating spreadsheet-like tools from strong attribute grammars. In *Generative Programming and Component Engineering, GPCE 2003*, vol. 2830 of *LNCS*, pages 307–323. Springer, 2003.

[36] J. Saraiva, S.D. Swierstra, and M.F. Kuiper. Functional incremental attribute evaluation. In *Compiler Construction' 2000*, vol. 1781 of *LNCS*, pages 279–294. Springer, 2000.

[37] I. Sommerville. *Software Engineering*. Addison-Wesley, 9th edition, 2011.

[38] S.D. Swierstra and H. Vogt. Higher order attribute grammars. In *Attribute Grammars, Applications and Systems*, vol. 545 of *LNCS*, pages 256–296. Springer, 1991.

[39] W.M.P. van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[40] W.M.P. van der Aalst. Business Process Management: A Comprehensive Survey. *ISRN Software Engineering*, 2013:1–37, 2013.

[41] E. Van Wyk. Implementing aspect-oriented programming constructs as modular language extensions. *Sci. Comput. Program.*, 68(1):38–61, 2007.

[42] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Compiler Construc.*, ETAPS, Grenoble, France, vol. 2304 of *LNCS*, pages 128–142, Springer 2002.

[43] H. Vogt, S.D. Swierstra, and M.F. Kuiper. Higher-order attribute grammars. In *ACM SIGPLAN'89 Conference on Programming Language Design and Implementation* (PLDI), Portland, Oregon, pages 131–145, 1989.

[44] M.M. Wagner, L.S. Gresham, and V. Dato. Case detection, outbreak detection, and outbreak characterization. In M.M. Wagner, A.W. Moore, and R.M. Aryel, editors, *Handbook of Biosurveillance*, pages 27–50. Academic Press, Burlington, 2006.

[45] T. Winograd and F. Flores. *Understanding Computer and Cognition: A new Foundation for Design*. Ablex Publishing Corporation, Norwood, New Jersey , 1986.

ABOUT THE AUTHORS:

Eric Badouel is researcher at Inria since 1990. He obtained a phD in Computer science in February 1990 and an Habilitation à diriger les recherches in April 1999 from the University of Rennes I. His research activities cover formal models of concurrent systems, Petri net synthesis, verification of opacity for dynamic systems and workflows, theory of interfaces, and artifact-centric collaborative systems. He is Scientific officer for the Africa and Middle East region at the European and International Partnerships Departement of Inria, co-director of Lirima, the Inria International Laboratory in africa, and co-Editor in Chief of ARIMA Journal.

Loic Hélouët is Chargé de Recherche at INRIA Rennes since 2001. He obtained his Ph.D in computer science from université de Rennes 1 in 2000, and an habilitation à diriger des recherches in 2013. His main research interests are distributed systems modeling, algorithmics and verification problems, with a strong emphasis on concurrency models.

Georges-Edouard Kouamou is a Lecturer at National Advanced School of Engineering since 2001. He obtained a PhD in Computer Science in October 2002 from the University of Yaoundé I. Its main research discipline concerns Software Engineering with an emphasis on software architectures, models and their transformation (Model Driven Soft. Eng.), distributed and collaborative systems modelling.

Christophe Morvan received his PhD in 2001 and habilitation à diriger des recherches in 2014. He joined Université de Marne-la-Vallée in 2002 as an Associate Professor. Since 2006 he his performing most of his research at INRIA Rennes. He has studied several classes of infinite graphs like rational, automatic or regular graphs. Recently he has been focusing on partial observation problems for such models.

Nsaibirni Robert Fondze Jr (MSc. in Computer Sc.) is currently PhD Student in the Department of Computer Sciences, University of Yaoundé 1. Member of the IDASCO and ALOCO teams of LIRIMA (www.lirima.org). His PhD Research Topic is the study of an architectural platform for early warning disease surveillance. His research Interests are Health Information Systems, BPM, Software Engineering.

Modeling and Validating Self-adaptive Service-oriented Applications

Paolo Arcaini
Charles University in Prague
Faculty of Mathematics and Physics, Czech Republic
arcaini@d3s.mff.cuni.cz

Elvinia Riccobene
Dept. of Computer Science
Università degli Studi di Milano, Italy
elvinia.riccobene@unimi.it

Patrizia Scandurra
Dept. of Managment, Information and Production Engineering
Università degli Studi di Bergamo, Italy
patrizia.scandurra@unibg.it

ABSTRACT

Self-adaptive and autonomous behaviors are becoming more and more important in the context of service-oriented applications, and formal modeling self-adaptive service-oriented components is highly required to assure quality properties.

This paper enhances the formal framework SCA-ASM for modeling and validating distributed self-adaptive service-oriented applications. We explain how modeling an SCA-ASM component able to monitor and react to environmental changes (context-awareness) and to internal changes (self-awareness), and present the operators for expressing and coordinating self-adaptive behaviors in a distributed setting. We also support techniques for validating adaptation scenarios, and getting feedback of the correctness of the adaptation logic as implemented by the managing SCA-ASM components over the managed ones. As a proof-of-concepts, we use self-adaptive SCA-ASMs for modeling and validating a decentralized traffic monitoring system.

CCS Concepts

•General and reference → Validation; •Software and its engineering → Formal methods; Requirements analysis;

Keywords

Self-adaptation; service-oriented applications; formal modeling; validation; SCA-ASM

1. INTRODUCTION

Service-oriented applications are playing so far an important role in several application domains (e.g., information technology, health care, robotics, defense and aerospace, to name a few). Cloud service providers, in particular, are expanding their offerings to include the entire traditional IT stack, ranging from foundational hardware and platforms to application components, software services, and whole software applications.

Self-adaptation (SA) [12, 14, 22] is recognized as an effective approach to deal with the increasing complexity, uncertainty and dynamicity of modern service-oriented applications where service components can appear and disappear, may become temporarily or permanently unavailable, may continuously change their behavior. A self-adaptive component is able to autonomously adapt its behavior to achieve required goals and functionality, on the basis of its internal dynamics and changing conditions in the environment.

To assure the required quality properties (e.g., flexibility, robustness, etc.), techniques for modeling and reasoning about adaptation capabilities of service-oriented components and their assemblies (or compositions) are greatly in demand, both at design time and at runtime. However, the current answer to this request is very limited [34], and this work would be a step forward along this research line.

Formal frameworks for modeling service-oriented applications already exist. Among them, SCA-ASM [28, 26] is based on the combined use of the OASIS standard SCA (*Service Component Architecture*) [29] to model the architecture and the assembly of the application, and the ASM (*Abstract State Machine*) formal method [8] to specify services' bevavior. The SCA-ASM language provides modeling primitives to represent service component assemblies, to express internal service computation, interaction and orchestration, as well as fault and compensation handling.

By adopting the *decentralized MAPE-K control loop model* of SA [22], we here show how SCA-ASM can be used to formalize and execute components exposing self-adaptive behavior in a decentralized manner and with partial shared knowledge. In particular, we explain how modeling in an SCA-ASM component its capacity to cope with continuously changing conditions of the environment (context-awareness), and to react to internal changes (self-awareness); we introduce operators for expressing and coordinating self-adaptive behavior in a distributed setting.

SCA-ASM enhanced with modeling features for SA provides a formal framework for the rigorous development of self-adaptive service-oriented applications. The proposed framework offers several advantages with respect to the current state of art (see Sect. 8 for related works). SCA-ASM allows modeling both *structure* and *behavior* of service com-

ponents in a unique framework integrating architectural and behavioral views. Based on the practical and scientifically well-founded ASM formal method, SCA-ASM models are executable and without mathematical overkill. The framework supports the design principle of *separation of concerns* and different adaptive behaviors can be introduced separately and in an incremental way, discovering possible side effects. Finally, by exploiting the prototyping/simulation environment for SCA-ASM [26], components can be *executed* already at high level of formalization, without caring about implementation details. Early validation by model simulation is a great means for evaluating architectural choices and alternative designs with limited implementation effort. Moreover, the mathematical foundation of the method facilitates *reasoning* about component behavior in order to guarantee correctness of its adaptation logic on the basis of the adaptation concerns.

A first overview of the self-adaptive SCA-ASM framework was presented in [27]. This article extends the earlier work in [27] by improving and extending the description of the modeling framework (modeling constructs and patterns) for the specification of self-adaptive service components with decentralized adaptation control. Through a case study, we also describe here the formal techniques supported by our framework for validating adaptation scenarios and getting feedback (already at system design time) of the correctness of the adaptation logic. Formal validation of quality properties is possible in terms of model simulation [17] and construction of execution scenarios [11]. We here mainly focus on *flexibility* (i.e., the ability of the system to dynamically adapt to changing conditions in the environment) and *robustness* (i.e., the ability of the system to cope autonomously with errors during execution).

This paper is organized as follows. Sect. 2 briefly introduces a reference model for SA. Sect. 3 describes the running case study of a Traffic Monitoring Application (TMA). Sect. 4 provides a background on the SCA-ASM modeling language for service oriented applications. Sect. 5 shows how to model SA in SCA-ASM in a decentralized manner. Sect. 6 reports, as case study, the results of modeling the architecture and the behavior of the TMA example. Sect. 7 presents the results of the TMA SCA-ASM model validation by simulating adaptation scenarios. Sect. 8 presents contributions related to our work. Sect. 9 discusses some lessons learned, while Sect. 10 concludes the paper and outlines future research directions of our work.

2. REFERENCE MODEL FOR SA

According to the reference model FORMS [35] and the study in [36], SA is based on the design principle of *separation of concerns*. As shown in Fig. 1 (adapted from [36]), a self-adaptive system is situated in an environment (both physical and software entities) and basically consists of a two-layer architecture: a *managed subsystem* layer that comprises the application logic, and a *managing subsystem* on top of the managed subsystem comprising the adaptation logic. This last realizes a *feedback loop* that monitors the environment and the managed subsystem, and adapts the latter when necessary, such as to deal with particular types of faults (self-heal), self-optimize when operating conditions

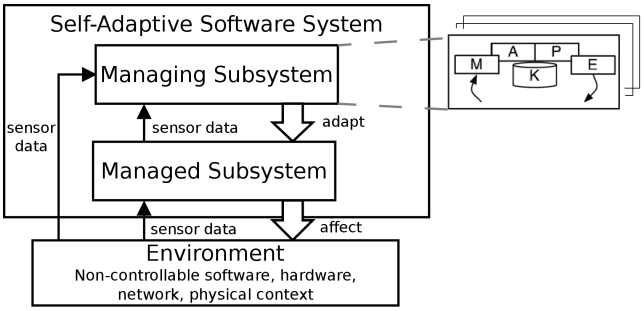


Figure 1: A self-adaptive software system (adapted from [36])

change, self-reconfigure when a goal changes, etc. Typically, the managing subsystem is conceived as a set of interacting feedback loops, one per each self-adaptation aspect (or concern). Other layers can be added to the system where higher-level managing subsystems manage underlying subsystems, which can be managing systems themselves.

A common approach to realize a feedback loop is by means of a MAPE-K (Monitor-Analyze-Plan-Execute over a Knowledge base) [22] loop. A component Knowledge (K) maintains data of the managed system and environment, adaptation goals, and other relevant states that are shared by the MAPE components. A component Monitor (M) gathers particular data from the underlying managed system and the environment through *probes* (or *sensors*) of the managed system, and saves data in the Knowledge. A component Analyze (A) performs data analysis to check whether an adaptation is required. If so, it triggers a component Plan (P) that composes a workflow of adaptation actions necessary to achieve the system's goals. These actions are then carried out by a component Execution (E) through *effectors* (or *actuators*) of the managed system.

Computations M, A, P, and E may be made by multiple components that coordinate with one another to adapt the system when needed, i.e., they may be decentralized throughout the multiple MAPE-K loops [36]. These MAPE components can communicate explicitly or indirectly by sharing information in the knowledge repository.

3. RUNNING CASE STUDY

As case study to illustrate the proposed modeling and validation framework, we adopt the Traffic Monitoring Application (TMA) presented in [20, 33]. It is an example of decentralized adaptation control in the application domain of traffic monitoring.

A number of intelligent cameras are distributed along a road (see Fig. 2). A camera is able to measure the traffic conditions and decide whether there is a traffic jam within its viewing range. A camera is endowed with a data processing unit and a communication unit to interact with other cameras. Traffic jams can span the viewing range of multiple cameras and can dynamically grow and dissolve. The dynamic nature of the traffic requires dynamic cameras collaborations. Since there is no central control, cameras have to collaborate in organizations to observe larger phenom-

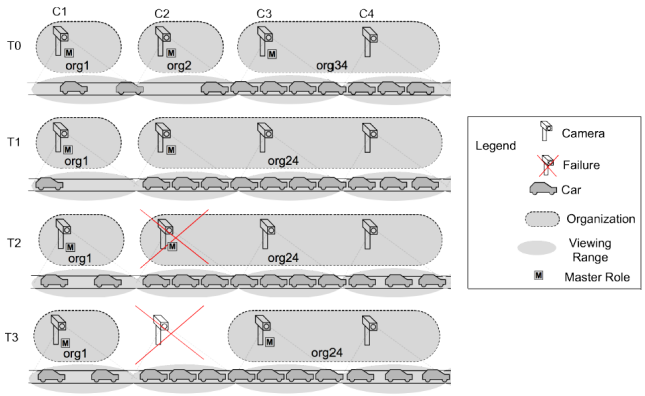


Figure 2: Adaptation scenarios (adapted from [20])

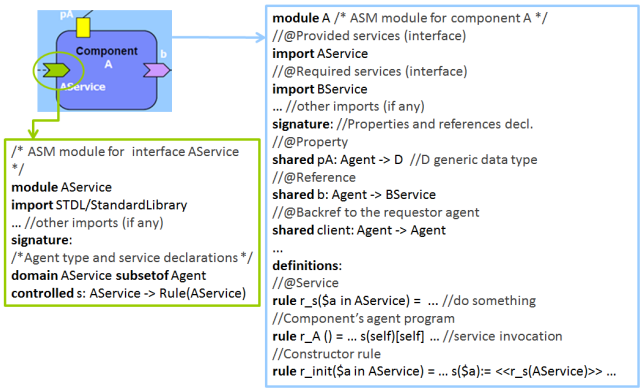


Figure 3: SCA-ASM component shape

ena such as areas affected by traffic jam on the basis of the head and tail of the cameras' viewing range, and aggregate the monitored data. Cameras enter or leave an organization whenever the traffic jam enters or leaves their viewing range. Possible clients of this monitoring application may be traffic light controllers, assistance systems that inform drivers about expected travel time delays, systems for collecting data for long term structural decision making, etc.

From a behavioral perspective, there are two main adaptation concerns. The first one is *flexibility* of the dynamic adaptation of an organization. See, for example, the scenario in Fig. 2 from configuration T0 to T1, where camera 2 joins the organization of cameras 3 and 4 after it monitors a traffic jam. The second one is related to *robustness* due to camera failures, i.e., when a camera becomes unresponsive without sending any incorrect data. This scenario is shown in Fig. 2 from T2 to T3, where camera 2 fails.

4. BACKGROUND ON SCA-ASM

The SCA-ASM language has been defined [28, 26] for modeling service-oriented applications. It complements the SCA component model with the ASM (Abstract State Machine) model of computation to provide an ASM-based formal and executable description of services *internal behavior*, *orchestration* and *interactions*.

An *SCA-ASM service-oriented component* appears as depicted in Fig. 3. It consists of an SCA graphical front-end to model the structure of the component, and an ASM model to specify the component's behavior.

The *SCA model* (`Component A`) graphically represents: the *services* (`AService`) provided by the component, the *references* (`b`) (functions required by the component) wired to services provided by other components, the *properties* (`pA`) allowing for the configuration of a component implementation and *bindings* that specify access mechanisms used by services and references according to some technology/protocol. Services and references are typed by *interfaces*.

The *ASM model*, given in terms of ASM modules – one for each service interface (`module AService`), and one to describe the component (`module A`) –, is used to express how a component behaves to provide its own services and to interact with the other components.

An ASM is an extension of a FSM [8] where unstructured control states are replaced by states comprising arbitrary complex data, and transitions are expressed by rules describing how data change from one state to the next. Formally speaking, an ASM *state* is a multi-sorted first-order structure, i.e. domains of objects with functions and predicates defined on them. State function values are saved into *locations* which may be updated from one state to another by firing a set of *transition rules* (or machine program). The SCA-ASM language exploits the *distributed multi-agents ASM* computational model where a number of running agents may interact in a synchronous or asynchronous way.

An SCA-ASM service-oriented component is endowed with (at least) one ASM agent computing according with the component's business partner (or *role*). Each agent a executes its own, possibly the same but differently instantiated, $program(a)$[1] that specifies the agent's behavior and the services provided by the component in terms of ASM transition rules. Components' agents are able to interact with other agents by providing/requiring services to/from other service-oriented components' agents.

The interface module of an SCA-ASM component `A` (see the skeleton reported on the left of Fig. 3) declares a dynamic controlled function `s` that returns a service as a rule. Such a *service function* represents a provided *service rule* since, once initialized, it leads to binding each invocation of `s` (read: application of function `s`) to a service rule `r_s` of the component. It can be dynamically updated (at runtime) as a built-in service adaptation mechanism (see Sect. 5.2).

In the ASM module for components (see the skeleton reported on the right of Fig. 3), the `@annotations` are used to denote SCA concepts (i.e., references, properties, services, etc.). Both ASM modules are expressed by using the textual notation ASMETA/AsmetaL[2].

[1]In ASM, a function *program* on *Agent* indicates the named transition rule associated with an agent as its behavior. It is used to dynamically associate and change behavior to agents.

[2]Two grammatical conventions must be recalled: a variable identifier starts with $; a rule identifier begins with "r_".

Table 1: SCA-ASM rule constructors

BASIC COMPUTATION	
Skip rule	**skip** do nothing
Update rule	$f(t_1, \ldots, t_n) := t$ update the value of f at $t_1, \ldots, t_n$ to t
Call rule	$R[x_1, \ldots, x_n]$ call rule R with parameters $x_1, \ldots, x_n$
Let rule	**let** $x = t$ **in** R assign the value of t to x and then execute R
COORDINATION	
Conditional rule	**it** ϕ **then** R_1 **else** R_2 **endif** if ϕ is true, then execute rule R_1, otherwise R_2
Iterate rule	**while** ϕ **do** R execute rule R until ϕ is true
Seq rule	**seq** $R_1 \ldots R_n$ **endseq** rules $R_1 \ldots R_n$ are executed in sequence without exposing intermediate updates
Parallel rule	**par** $R_1 \ldots R_n$ **endpar** rules $R_1 \ldots R_n$ are executed in parallel
Forall rule	**forall** x **with** ϕ **do** $R(x)$ forall x satisfying ϕ execute R
Choose rule	**choose** x **with** ϕ **do** $R(x)$ choose an x satisfying ϕ and then execute R
Split rule	**forall** $n \in N$ **do** $R(n)$ split N times the execution of R
Spawn rule	**spawn** child **with** R create a child agent with program R
COMMUNICATION	
Send rule	**send**[*lnk,R,snd*] send data *snd* to *lnk* in reference to rule R (no blocking, no acknowledgment)
Receive rule	**receive**[*lnk,R,rcv*] receive data *rcv* from *lnk* in reference to R (blocks until data are received, no ack)
SendReceive rule	**sendreceive**[*lnk,R,snd,rcv*] send data *snd* to *lnk* in reference to R waits for data *rcv* to be sent back (no ack)
Reply rule	**replay**[*lnk,R,snd*] returns data *snd* to *lnk*, as response of R request received from *lnk* (no ack)

ASM rule constructors and predefined ASM rules (i.e., named ASM rules made available as model library) are used as basic SCA-ASM behavioral primitives. They are reported in Table 1 and include: rule constructors to specify basic *computation* actions (location update, service invocation, or do nothing) and *coordination* such as conditional actions (*if-then-else*), parallel actions (*par*), sequential actions (*seq*), iterations (*iterate*, *while*, *recwhile*), actions to create child agents (*spawn*), non-determinism (existential quantification *choose*) and unrestricted synchronous parallelism (universal quantification *forall*), etc.; *communication* rules – *send*, *receive*, *send-receive* and *reply* – providing synchronous/asynchronous service invocation styles (corresponding, respectively, to the *request-response* and *one-way* interaction patterns of the SCA standard). Communication relies on a dynamic domain *Message* that represents message instances managed by an *abstract message-passing* mechanism: components communicate over wires and a message encapsulates information about the partner *link* and the referenced *service name* and *data* transferred.

SCA-ASM rule constructors can be combined to model specific interaction and orchestration patterns in well structured and modularized entities.

SCA-ASM modeling constructs for fault/compensation handling are also supported (see [28, 26]), but are not reported here since related to *fault tolerance* concepts that we do not take into account in the adaptation model presented here.

An SCA-ASM design environment [30, 9, 26] was developed by integrating the Eclipse-based SCA Composite Designer, the SCA runtime platform Tuscany [32], and the simulator ASMETA/AsmetaS [17, 4, 5]. This environment allows a designer to graphically model, compose, deploy, and execute heterogeneous service-oriented applications in a technologically agnostic way.

5. SELF-ADAPTIVE SCA-ASM

Here we introduce the concept of *self-adaptive SCA-ASM assembly* to model distributed self-adaptive service-oriented applications from the perspective of distributed and interacting MAPE-K feedback control loops.

According to the reference model for SA in Sect. 2, we distinguish *managed SCA-ASM components* encapsulating the functional logic of the application and *managing SCA-ASM components* encapsulating the logic of adaptation.

A managed SCA-ASM component is an SCA-ASM service component as presented in Sect. 4.

A managing SCA-ASM component is, instead, a *self-adaptive SCA-ASM component* endowed, besides the functionalities of an SCA-ASM service component, with mechanisms/operators to monitor the environment and itself, and to perform adaptation actions.

The assembly of the self-adaptive components should expose a certain number of MAPE-K loops

$$\{MAPE(adj_1), \ldots, MAPE(adj_n)\}$$

one per each adaptation concern adj_i. The four computations of each $MAPE(adj_i)$ are formalized in terms of ASM transition rules distributed among those managing SCA-ASM components involved in the execution of the loop.

Therefore, the program of an agent a associated to a managing SCA-ASM component has the form

$$program(a) = \otimes(R^a_{MAPE(adj_1)}, \ldots, R^a_{MAPE(adj_k)})$$

where the operator $\otimes$ is an SCA-ASM coordination rule constructor (most of the times **par** o **seq** for parallel or sequential actions coordination), and rules $R^a_{MAPE(adj_j)}, j = 1 \ldots k$, specify the behavioral contributions of the agent a to the k loops the component is involved in.

These rules are annotated (as comments //) with appropriate labels @M_c (for context-aware monitoring), @M_s (for self-aware monitoring), @A (for analyzing), @P (for planning), and @E (for execution), depending on the role of the agent a in the loop. These labels (better explained in the following subsections) show how MAPE computations are distributed among the agents. For reasoning and simulation scopes, these annotations may be extracted from comments and used by a runtime platform.

An adaptation concern *adj* is also characterized by a *knowledge* $K(adj)$. Therefore, further SCA-ASM components represent the knowledge of the MAPE loops. These components are used by the managing components as *reflective model* to save and share data (through suitable access services) of the

managed subsystem and the environment, and other information relevant for the MAPE computations. In principle we can have a *Knowledge* component for each *MAPE*(*adj*). However, it is possible to collapse more knowledge components into one (especially when more MAPE-K loops are performed by the same managing component).

The notion of *environment* is directly supported in SCA-ASM through *monitored functions*[3] of the ASM theory [8]. *Probes* are represented by actions of managed components' agents and consist into reading and reporting values of monitored functions. Similarly, *actuators* are actions of managed agents that update the value of *controlled functions*[4] on the base of the plan decided by the managing agents.

In the rest of this section, we describe how the self-adaptive SCA-ASM component model realizes the following key requirements for SA (see the reference model FORMS [35]):

- How a component system *monitors* the environment and itself;
- How a component system *adapts* itself;
- How to *coordinate* monitoring and adaptation in a distributed setting.

5.1 Monitor and analyze computations

According to [35], an *update computation* perceives the state of the environment, while a *monitor computation* perceives the state of the managed subsystem to update the system model (the observed data from the system) in the knowledge. Update computations and monitor computations may trigger analyze computations when particular conditions hold. An *analyze computation* of a MAPE-K loop assesses the collected data from the environment and/or the system itself to determine the system's ability to satisfy its objectives. Update computations in combination with analyze computations provide for context-awareness [35], which is a key property of self-adaptive systems. Monitor and analyze computations provide for self-awareness [35], which is another key property of self-adaptive systems.

In SCA-ASM, we prefer the terms *context-aware monitoring* and *self-aware monitoring* to denote update computations and monitor computations, respectively. In managing components, these computations are explicitly captured by ASM rules (or portions of rules) annotated with @M_c and @M_s, respectively. Context-aware monitoring consists of looking at values of ASM monitored locations (the environment), while self-aware monitoring consists of looking at values of ASM controlled locations (internal locations) of the managed system.

Context/self-aware monitoring computations may have two different rule schemes (1 and 2) depending on the *decentralized* or *centralized* control of the loop's computations, respectively:

$$\textbf{if } Cond \textbf{ then } Updates_K \qquad //@M\_c[s] \tag{1}$$

$$\textbf{if } Cond \textbf{ then } Analyze \qquad //@M\_c[s] \tag{2}$$

In both schemes, *Cond*, the condition under which the rule is applied, is an arbitrary first-order formula over monitored (in case of context-awareness) and/or controlled (in case of self-awareness) locations of the managed SCA-ASM. In the decentralized control scheme (1), $Updates_K$ is a finite set of transition rules simultaneously executed; they may consist of function updates $f(t_1, \ldots, t_n) := t$ changing (or defining, if there was none) the value of the knowledge location represented by the function f at the given parameters, and/or of call rules for more complex computations. Knowledge updates have to be implemented in concrete terms as invocations of appropriate access services of the knowledge component of the MAPE-K loop. Such knowledge updates (i.e., once the corresponding services executions complete) may then trigger an analyze activity executed by other managing components' agents. In case of centralized control (2), *Analyze* is an ASM transition rule for an analyze computation (see schemes 3, 4) triggered by the monitoring computation and executed by the same component's agent in a waterfall style.

An *analyze computation* is specified by an ASM conditional rule annotated with @A:

$$\textbf{if } Cond_K \textbf{ then } Updates_K \qquad //@A \tag{3}$$

It involves the evaluation of a first order formula $Cond_K$, to determine if a violation of the system's goals occurs and an adaptation plan has to be triggered. This formula can be arbitrary complex and expresses the logic relationship of certain knowledge location values that must be true in order that violating situation holds. In a decentralized control, $Updates_K$ are updates of knowledge functions that may trigger planning activity executed by other components' agents. Alternatively, in centralized mode (schema 4), a planning computation can be directly executed by the same component's agent in a waterfall style:

$$\textbf{if } Cond_K \textbf{ then } Plan \qquad //@A \tag{4}$$

where *Plan* is a transition rule for planning (see next section).

Note that complex planning computations may be missing in a MAPE-K loop and therefore the transition from an analyze computation to an execute computation may be direct. In this case, the ASM rules schemes 3 and 4 will trigger an execute activity indirectly (by knowledge updates) or directly (by executing an execute computation).

5.2 Plan and execute computations

In a MAPE-K loop, analyze computations may trigger *plan computations* when a particular analysis discovers a violation of the system's objectives. A plan computation creates/selects a procedure to enact a necessary adaptation in the managed system. The plan computation can be a single action or a complex workflow. Then, as decided by the plan computation, an *execute computation* carries out the adaptation actions on the managed system using effectors.

In SCA-ASM, plan computations are executed by managing agents and consist of ASM rules annotated with @P. These rules can be conditional rules or may adopt a more complex

[3]Locations updated by the environment and read by an ASM.
[4]Locations read and written by an ASM [8].

ASM rule scheme.These rules determine the desired adaptation actions and trigger (by sending information or setting values of the shared knowledge) the managing agent(s) responsible to execute such adaptations or directly invoke execute computations.

An *execute computation* is an ASM rule annotated with @E and usually made of atomic adaptation actions. In SCA-ASM, a set of atomic adaptation actions allow runtime adaptation to be expressed both at architectural and at behavioral level.

Structural adaptation actions.

- add/remove components to/from a composite;
- add/remove component services, references, properties to/from a component;
- add/remove wires to bind/unbind reference-service interfaces;
- add/remove wires to delegate/undelegate service-service and reference-reference interfaces.

These architectural re-configurations are effectively executed by an SCA runtime platform with dynamic introspection and adaptation capabilities (e.g., the FraSCAti platform [31]).

Behavioral adaptation actions.

- change values of managed component's properties (by firing update rules);
- change the knowledge (by invoking appropriate services of the knowledge);
- stop/start components (by updating the function *status* of a component's *life cycle* to the value INIT/EXITED [26]);
- instantiate a new agent within a component to introduce a new concurrent behavior (by a spawn rule);
- change a component's agent behavior dynamically (by updating the function $program(a)$ of agent a to a new rule r);
- change a component's service behavior dynamically (by updating the service function $s(a)$ of an agent a to a new service rule $r_s{}'$).

@E can also label non-atomic SCA-ASM rules modeling exception or compensation handlers [26] to be executed, respectively, in case of fault or to rollback partially executed actions.

5.3 Distributed coordination

Distributed adaptive service-oriented applications are specified as *self-adaptive SCA-ASM assembly* that is the composition of managed SCA-ASM components, managing self-adaptive SCA-ASM components, plus SCA-ASM components for the shared MAPE-K loops' knowledge.

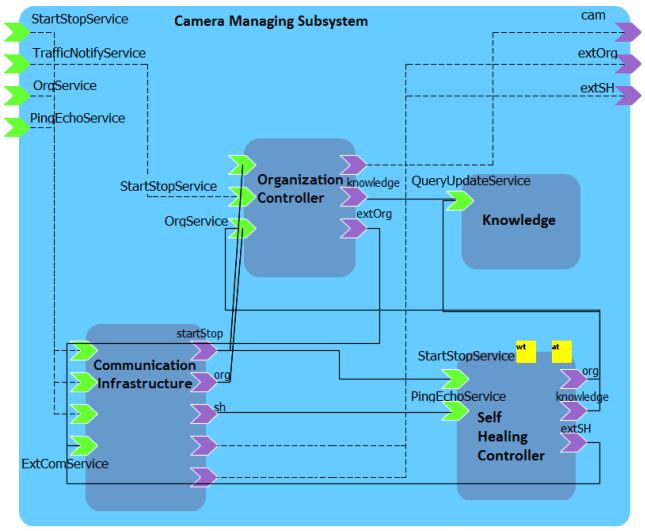


Figure 4: SCA model: Camera *Managing Subsystem*

The computational model is that of a *multi-agent ASM* where multiple components' agents interact in parallel in a synchronous/asynchronous way.

Moreover, MAPE computations may be enhanced with support for distribution through coordination [36]. Cooperation and competition are forms of interactions among concurrent MAPE computations. So, interactive MAPE-K loops may require developing coordination models explicitly. To this purpose, SCA-ASM components' agents may adopt recurrent coordination patterns for distributed control (e.g., master-slaves pattern, hierarchical control pattern, alternate pattern, etc.) as formalized in [7].

6. SCA-ASM MODEL OF THE TMA

In this section we exemplify the self-adaptive SCA-ASM assembly by our running case study.

6.1 System architecture

Fig. 4 shows the SCA architecture[5] of the managing subsystem of each camera system as adapted from [20]. According to the vision provided in Sect. 2, the managed subsystem is the local camera, while the managing subsystem corresponds essentially to the components *Self-healing Controller* and *Organization Controller*. The local camera (represented as an external reference *cam*) provides the functionality to detects traffic jams and inform clients. Traffic detection is carried out by the camera through a sensor that detects traffic jam and in case of "congestion" or "no longer congestion" notifies the organization controller (see the service interface `TrafficNotifyService` in Fig. 4).

Cameras collaboration is managed by organization controllers using a *master/slave* control model. For each organiza-

[5]The case study presents Event-Driven interactions not yet supported in the SCA runtime Tuscany. We model events in SCA-ASM as asynchronous service requests. An event type is defined in the component implementation that can receive it as a rule annotated with @service and with a skip body.

tion, one of the organization controllers is elected as master, whereas the other controllers of the organization are slaves. The master is responsible for managing the dynamics of that organization by synchronizing with all of the slaves. To deal, instead, with camera failures and therefore support robustness, the *Self-Healing Controller* detects failures of other cameras based on a broadcast *ping-echo* mechanism.

To coordinate with other cameras, the managing subsystem can rely on the services provided by the component *Communication Infrastructure*. Precisely, to detect failures, the self-healing controller needs to coordinate with the self-healing controllers of other cameras using, respectively, the service interface `ExtComService`, to forward outgoing remote messages to the communication infrastructure, and `PingEchoService` to pass remote messages that arrive at the communication infrastructure to the local self-healing controller. Similarly, the organization controller needs to coordinate with organization controllers of other dependent cameras using the interfaces `ExtComService` and `OrgService`.

A component *Knowledge* is used to share and maintain information of the managed subsystem, the environment, and other relevant state for the MAPE computations. Actually, components interact directly via communication primitives and also indirectly via the knowledge. Note that this case study, as it is presented, does not require explicit adaptations of the architecture, but only changes to the knowledge repository. When a camera fails and remains silent, all dependent cameras will be aware of it and temporarily pause the relevant communications towards it without physically removing the connecting wires.

6.2 Self-adaptive components bevavior

We specify the self-adaptive behavior of the overall distributed camera system using three main MAPE-K loops[6]:

$\{MAPE(flexibility), MAPE(intFailure), MAPE(extFailure)\}$

The first MAPE-K loop deals with the flexibility concern to restructure organizations in case of congestion and it is handled by the organization controllers of each camera. The second MAPE-K loop deals with the robustness concern to restructure organizations in case of failures of other cameras (silent cameras). It is handled by both the organization controllers and the self-healing controllers of each camera. Finally, the third MAPE-K loop deals with the robustness concern to restructure organizations in case of internal failure of the camera. The two managing agents of a camera are both involved in this loop.

Below, we describe the behavior of the organization controller by reporting some fragments of its SCA-ASM implementation. The complete SCA-ASM specification is available online [30].

Organization controller. A master/slave control model is adopted to structure organizations in case of congestion. To keep the master election policy simple, we assume every camera has a unique ID that is a monotonically increasing function on the traffic direction and the camera with the lowest ID becomes master. Algorithms to elect a new master, like the Bully and the Ring algorithms [16], are out of the scope of this paper.

[6]The vision of MAPE-K loops we take here slightly differ from the one taken in [20].

Listing 1: Program of each organization controller

```
macro rule r_masterBevavior =
    seq
        r_selfFailureAdapt[] //Adaptation due to internal failure
        r_failureAdapt[] //Adaptation due to external failure
        r_master_congestionAdapt[] //Adaptation due to congestion
    endseq

//Agent's program initialization
rule r_OrganizationController = r_masterBevavior[]
agent OrganizationController : r_organizationController[]
```

Every camera (referred by the function *cam*) has four basic states (specified by means of a function *state*). In normal operation, the camera can be master with no slaves (MASTER), master of an organization with slaves (MASTERWITHSLAVES), or slave (SLAVE). Additionally, the camera can be in a failed state (FAILED). Initially, all cameras are masters with no slaves. As part of the adaptation logic, the organization controller adapts the camera's state and its program dynamically. Therefore, in the initial state, the organization controller's program consists of the rule `r_masterBevavior` (see Listing 1[7]) that represents the behavior of the organization controller in the role of MASTER (i.e., when the state of the camera is MASTER). The rule `r_masterBevavior` executes sequentially three rules as behavioral contributions of the agent to the three MAPE-K control loops identified before. It can be intended as concrete instance of the rule schema presented in Sect. 5:

$$program(\text{OrganizationController}) = \otimes(\text{r\_selfFailureAdapt}, \text{r\_failureAdapt}, \text{r\_master\_congestionAdapt})$$

where the operator $\otimes$ is **seq**.

The first rule `r_selfFailureAdapt` handles adaptation in case of internal failure signaled through the service *StartStopService*. The second rule `r_failureAdapt`) deals with adaptation due to external failures (silent cameras) notified by the self-healing controller through the service *OrgService*. The third rule `r_master_congestionAdapt` deals with adaptation in the role of MASTER due to traffic congestion notified by the traffic monitor of the local camera (through the service `TrafficNotifyService`).

Details on the rule definitions can be found in the specification available on line. As exemplification, we here explain the rule `r_master_congestionAdapt` of the organization controller for adaptation due to congestion. The rule contains instances of the (centralized) pattern (2) described in Sect. 5 for monitoring computations and it is shown in Listing 2.

In the role of master, when a congestion is detected (by receiving the event *cong*) the organization controller sends

[7]Note that the concrete syntax `agent agent_type : rule[]` denotes in AsmetaL the initialization of the component agent's function *program*.

Listing 2: Excerpt of rule `r_master_congestionAdapt`

```
//@P
macro rule r_turnMasterWithSlaves =
    par
        r_send[knowledge(self),"r_addNewSlave(Camera)",cam(self)]
        r_send[cam(self),"r_setState(CameraState)",MASTERWITHSLAVES]
        program(self):= <<r_masterWithSlavesBehavior>>
    endpar

//@P
macro rule r_turnSlave($master in Camera) =
    par
        r_send[knowledge(self),"r_setMaster",(cam(self),$master)]
        r_removeSlavesTurningSlave[]
        program(self):= <<r_slaveBevavior>> //@E
        r_send[extOrg(self),"r_m_offer",($master,cam(self))]
    endpar

//@P
macro rule r_send_s_offer =
    seq
        congested(self) := true
        r_send[extOrg(self),"r_s_offer",("s_offer",next(cam(self)))]
    endseq

//@A
macro rule r_analyzeCongestion =
    if event(self)="m_offer" then
        r_turnMasterWithSlaves[]
    else
        if event(self)="s_offer" then
            seq
                r_sendreceive[knowledge(self),"r_getPrev",
                              cam(self),prev(cam(self))]
                if isDef(prev(cam(self))) then
                    r_turnSlave[prev(cam(self))]
                endif
            endseq
        endif
    endif

macro rule r_master_congestionAdapt =
    seq
        r_receive[requester(self),"r_event",event(self)]
        if isDef(event(self)) then
            if ( event(self)="cong" and not congested(self) ) //@M_c @M_s
            then //Congestion detected!
                seq
                    r_sendreceive[knowledge(self),"r_getNext",
                                  cam(self),next(cam(self))]
                    if isDef(next(cam(self))) then
                        r_send_s_offer[]
                    endif//@A
                endseq
            else
                if congested(self) then
                    r_analyzeCongestion[]
                endif //@M_s
            endif
        endif
    endseq
```

the event *s_offer* (see the plan rule `r_send_s_offer` in Listing 2) to the next alive camera (if any) in the direction of the traffic flow as request to join, as slave, the organization. Depending on the traffic condition of the next camera and its current role, the organizations may be restructured depending on the analysis computation represented by the rule `r_analyzeCongestion` (see Listing 2). If traffic is not jammed (the controlled predicate *congested* is false) in the viewing range of the next camera, organizations are not changed. Otherwise, organizations are joined:

Listing 3: Organization Controller's behavior in the roles slave and master with slaves

```
macro rule r_slave_congestionAdapt =
    seq
        r_receive[requester(self),"r_event",event(self)]
        if isDef(event(self)) then
            if event(self)="no_cong" then //@M_c No longer congested!
                seq
                    congested(self) := false
                    r_sendreceive[knowledge(self),"r_getMaster(Camera)",
                                  cam(self), master(cam(self))]
                    r_send[extOrg(self),"r_slaveGone",(master(cam(self)),
                                                       cam(self))]
                    r_turnMaster[]
                endseq
            else r_receiveOrgSignals[] endif endif endseq

macro rule r_masterWithSlaves_congestionAdapt =
    seq
        r_receive[requester(self),"r_event",event(self)]
        if isDef(event(self)) then
        if event(self)="no_cong" then //@M_c No longer congested!
            par
                congested(self):= false
                r_removeSlavesTurningMaster[]
            endpar
        else r_analyzeOrganization[] endif endif
    endseq
```

the next camera becomes slave of the requester camera by executing the plan rule `r_turnSlave` (shown in Listing 2 – the rule contains instances of the (centralized) pattern (4) described in Sect. 5) that sets the requester camera as new master (knowledge adaptation), changes the camera state (by the rule `r_removeSlavesTurningSlave`), adapts the organization controller's program to the rule `r_slaveBevavior`, and informs the requester by sending the event *m_offer*. When the requester camera receives *m_offer*, it becomes master of the joined organization by executing the rule `r_turnMasterWithSlaves` (shown in Listing 2) to add the new slave to its list, change the camera state and adapt its program to `r_masterWithSlavesBehavior`.

Note that atomic adaptation actions to restructure master/slave organizations (i.e., add a camera as slave of a master camera, set the new master of a slave camera, etc.) are services provided by the knowledge component (not reported here) and modeled as ASM rules annotated with @E.

The rules `r_slave_behavior` and `r_masterWithSlaves_behavior` for the roles SLAVE and MASTERWITHSLAVES, respectively, are defined similarly. They only differ in executing the rules `r_slave_congestionAdapt` and `r_masterWithSlaves_congestionAdapt`, respectively, as contribution to the third MAPE loop. These rules are reported in Listing 3. In the role of slave (rule `r_slave_congestionAdapt` in Listing 3), if the traffic is no longer jammed (event *no_cong*), the organization controller leaves the organization it belongs to by sending the event *slaveGone* to its master and becomes master of a single member organization by the rule `r_turnMaster`. Otherwise (still congested), the organization controller waits (by the rule `r_receiveOrgSignals`) for a restructuring event *change_master* or *masterGone*.

In the role of master with slaves (rule `r_masterWithSlaves_congestionAdapt`), when the traffic is no longer jammed, the

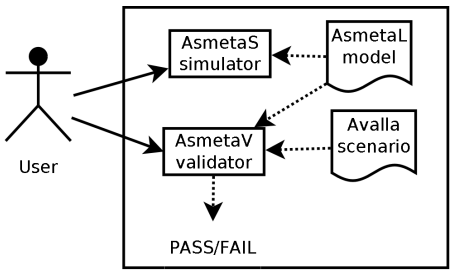


Figure 5: Validation in the ASMETA framework

organization controller notifies all its slaves that the master is gone (the event *masterGone*) and leaves the organization becoming master of a single member organization (by the rule `r_removeSlavesTurningMaster`). Otherwise (still congested), the organization controller analyzes the organization executing the rule `r_analyzeOrganization` to add and remove slaves dynamically. When no slaves remains, it becomes master again. During analysis of its organization, in case it receives an event *s_offer*, it becomes slave of the requester camera (by the rule `r_turnSlave`).

7. TMA VALIDATION

Model validation is a model analysis activity, less demanding than other analysis techniques as formal verification, to execute starting from the earlier stages of the model development. Validation is the process of investigating a model with respect to the user perceptions in order to ensure that the model really reflects the user needs and statements about the application, and usually permits to detect faults with limited effort.

Among different existing model validation approaches, in our work we have carried out *simulation* and *scenario-based validation* through an execution platform for SCA-ASM [26]. This execution environment is based on the SCA `Tuscany` runtime platform [32], and the ASM simulator `AsmetaS` [17] and validator `AsmetaV` [11] provided by the ASMETA framework [4]. Fig. 5 shows the validation process: the user can directly simulate an ASM-based specification in an interactive way (see Sect. 7.1) or write a scenario that automatize the simulation and the checking of the produced output (see Sect. 7.2).

These two validation techniques allowed us to simulate different adaptation mechanisms described previously and be confident that the TMA model behaved as expected in the adaptation requirements. As better described in the next subsections, we simulated the TMA model with an increasing number of cameras and we reproduced different adaptation scenarios such as those shown in Fig. 2 from T0 to T2 for flexibility (i.e., the ability of the system to adapt dynamically with changing conditions in the environment), and from T2 to T3 for robustness (i.e., the ability of the system to cope autonomously with errors during execution).

7.1 Simulation

`AsmetaS` permits to perform either *interactive simulation*, where required inputs are provided interactively by the user

```
Insert a boolean constant for stopCam(c2):
true
Insert a boolean constant for startCam(c2):
false
...
<State 1 (controlled)>
state(c1)=MASTER
state(c2)=FAILED
state(c3)=MASTER
state(c4)=MASTER
</State 1 (controlled)>
Insert a boolean constant for stopCam(c2):
true
Insert a boolean constant for startCam(c2):
true
INCONSISTENT UPDATE FOUND:
location state(c2) updated to FAILED != MASTER
```

Figure 6: Example of simulation – Detection of an inconsistent update

during simulation, and *random simulation*, where inputs values are chosen randomly by the simulator itself. The simulator, at each step, performs *consistent updates checking* to check that all the updates are consistent: in an ASM, two updates are inconsistent if they update the same location to two different values at the same time [8]. In our case, by simulation we found (in a preliminary version of our specification) that a self-healing subsystem could, at the same time, turn the status of its camera both to `MASTER` and to `FAILED` (see the simulation trace in Fig. 6); that particular situation could occur when a camera `c` was already `FAILED` and the system received the events to both turn on and turn off the camera (i.e., both monitored locations `startCam(c)` and `stopCam(c)` were true). This consistency violation was due to a wrong scheduling of the operations of the self-healing subsystem.

Moreover, the `AsmetaS` simulator also permits to check if some invariants are satisfied during simulation. For example, we added to the specification the following invariants

$$(\mathsf{state}(c_i) = \mathsf{FAILED} \text{ and } \mathsf{state}(c_{i-1}) \mathrel{!}= \mathsf{FAILED}) \text{ implies } \mathsf{next}(c_{i-1}) = c_{i+1}$$
$$(\mathsf{state}(c_i) = \mathsf{FAILED} \text{ and } \mathsf{state}(c_{i+1}) \mathrel{!}= \mathsf{FAILED}) \text{ implies } \mathsf{prev}(c_{i+1}) = c_{i-1}$$

checking that the neighboring camera relations are correctly arranged after a failure: whenever a camera c_i fails (with $i = 2, \ldots, n-1$), camera c_{i-1} updates its next camera to c_{i+1}, and camera c_{i+1} updates its previous camera to c_{i-1}.

7.2 Scenario-based validation

Scenario-based validation is a more advanced way to simulate and inspect ASMs, by specifying a scenario representing a description of the actions of an external actor and the corresponding reactions of the system. There are two kinds of external actors:

- a *user* interacts with the system in a black box manner, by setting the values of the external environment (e.g., asking for a particular service), waiting for a step of the machine as reaction to his/her request, and checking the output values;
- an *observer*, instead, can also inspect the internal state of the system (i.e., values of machine functions) and check the validity of possible invariants of a certain scenario.

Scenarios are described in an algorithmic way using the tex-

Listing 4: Flexibility validation scenario from T0 to T1 in Avalla

```
scenario Flexibility_T0_T1
load main.asm

set stopCam(c1) := false; set stopCam(c2) := false; set stopCam(c3) := false; set stopCam(c4) := false;
set startCam(c1) := false; set startCam(c2) := false; set startCam(c3) := false; set startCam(c4) := false;
set congestion(c1) := false; set congestion(c2) := false; set congestion(c3) := true;
set congestion(c4) := true; set elapsed_wait_time(selfHealingController3) := false;
set elapsed_wait_time_plus_delta(selfHealingController4) := false;
exec par
        state(c3) := MASTERWITHSLAVES
        state(c4) := SLAVE
        slaves(c3, c4) := true
        getMaster(c4) := c3
        congested(organizationController3) := true
        congested(organizationController4) := true
    endpar;
step

set congestion(c2) := true;
step
check getMaster(c4)=c3 and s_offer(c3)=true and s_offer(c4)=false and slaves(c3,c4)=true and
      state(c1)=MASTER and state(c2) = MASTER and
      state(c3) = MASTERWITHSLAVES and state(c4)=SLAVE;

step
check isAlive(c4)=false and newSlave(c2,c3)=true and getMaster(c4)=c3 and s_offer(c3)=true and
      s_offer(c4)=false and slaves(c3,c4)=false and state(c1)=MASTER and
      state(c2) = MASTER and state(c3) = SLAVE and state(c4)=SLAVE;

step
check isAlive(c4) = false and newSlave(c2,c3) = false and getMaster(c4) = c3 and s_offer(c3) = true and
      s_offer(c4) = false and slaves(c2,c3) = true and slaves(c2,c4) = true and state(c1) = MASTER
      and state(c2) = MASTERWITHSLAVES and state(c3) = SLAVE and state(c4) = SLAVE;
```

tual language `Avalla` [11]. An scenario is as interaction sequence consisting of actions of the external actor (user or observer) and activities of the machine as reaction to the actor actions. The `Avalla` language provides constructs to **set** the environment (i.e., the values of monitored/shared functions), to **check** the machine state, to ask for the **exec**ution of certain transition rules, and to force the machine itself to make one **step** (or a sequence of steps by **step until**).

The tool `AsmetaV` reads scenarios written in `Avalla` and executes them using the simulator `AsmetaS`; during simulation, `AsmetaV` captures any check violation and, if none occurs, it finishes with a *PASS* verdict (see Fig. 5).

We simulated different scenarios with increasing number of cameras. In particular, we created and validated the adaptation scenarios shown in Fig. 2 from T0 to T1 for flexibility, and from T2 to T3 for robustness. The scenario reported in Listing 4 describes the adaptation from T0 to T1. At the beginning, `c3` and `c4` form an organization; when `c2` detects congestion, it joins the organization as MASTER. Appropriate assertions control that the right messages are sent and that the correct slavery relations are established.

8. RELATED WORK

SA has been widely studied in the software architecture community [2]. Various mechanisms and frameworks for handling adaptation have been proposed, such as (to name a few): SA with aspect-orientation, Dynamic Reconfiguration, Model-Driven Development frameworks for SA, and frameworks for self-optimization (including the adaptation cost itself) [12, 14, 22, 25, 10, 24].

However, as shown in the study [34], little attention has been given to formal modeling and analysis of self-adaptive service-oriented applications. Here we review, in a non-exhaustive manner, the main contributions.

The work in [21] compares JOLIE, PiDuce and COWS, as candidate formalisms for modeling dynamically adaptable services. The analysis is, however, focused on time-constrained dynamic adaptation concerns. They illustrate strengths and limitations of each formalism from the point of view of the quality of service properties, such as availability and responsiveness, and justify the selection of COWS as the best-fit, though limited, language for studying compliance to time bounds.

The PLASTIC approach [6] uses the formal framework Chameleon to support context-aware adaptive services in Java applications. The approach is targeted to Java.

[23] concerns the formal specification and verification of dynamic reconfigurations of component-based systems using the B formal method for the specification of component architectures and FTPL – a logic based on architectural constraints and on event properties, translatable into LTL – to express temporal properties over (re-)configuration sequences to model-check. Their approach, though not conceived for service-oriented architectures, is similar to ours, but they support only architectural adaptation.

[18] uses architectural constraints specified in Alloy for the specification, design and implementation of self-adaptive architectures for distributed systems.

[13] proposes a model-based approach to combine self-adaptive composition and error recovery techniques to perform adaptation of BPEL or WF services. The authors present techniques to provide semi-automated support for identification and resolution of mismatches between service interfaces and protocols, and generate adaptation behavioural specifications.

[15] presents a model for dynamic reconfiguration of services using the service-oriented ADL SRMLight (a language derived from the more complex modeling language SRML developed within the EU project SENSORIA). Though they address only the specification of dynamic architectural characteristics of service-oriented applications, their mathematical model is general enough and useful to develop general assembly and binding mechanisms for service components in runtime middleware/platforms.

Timed Automata and Z are used in, respectively, [20] and [35], to model the Traffic Monitoring case study (the same presented here), and verify flexibility and robustness properties. These are (to the best of our knowledge) the first works presenting a formal approach to specify and verify behavioral properties of decentralized self-adaptive systems – most existing formal approaches to SA assume a centralized point of control –, and this is the reason why we were inspired mainly by them. However, Timed Automata and Z are used as they are without any extension for service-oriented applications, while SCA-ASM is a language for service-modeling. Moreover, the simplicity of the ASMs in expressing fundamental computing concepts allows one to avoid over-specification due to the rigidity of the formalism. That is, indeed, the main limitation of the approaches in [20, 35]. Such expressiveness makes, instead, the SCA-ASM method comprehensible to practitioners and feasible for large-scale applications.

Similarly to our work, the goal-oriented framework Sim-

SOTA is proposed in [1] for modeling, simulating and validating MAPE-K feedback loop models of self-adaptive systems. However, SimSOTA adopts a semi-formal notation, namely UML activity diagrams, to model feedback loops.

Concerning lightweight modeling languages, in [19] an UML profile, called the *control loop* UML profile, is presented. It extends UML modeling concepts such that control loops become first class elements of the architecture. Another graphical notation to explicitly capture interacting MAPE loops is presented in [36]. The *control loop* UML profile supports modeling and reasoning about interactions between coarse-grained "controllers", while the notation in [36] aim to model finer-grained interactions between the components of control loops and to define MAPE patterns (i.e., recurring structure of interacting MAPE components).

9. LESSONS LEARNED

Enhancing SCA-ASM with self-adaptation features was facilitated from the availability in SCA-ASM of modeling constructs for service-oriented applications, and of operators to model distributed computation and coordination/communication among components. By explicitly modeling MAPE-K control loops in SCA-ASM, we achieve a clear separation between adaptation logic and functional logic. This is possible since the formal approach allows us:

(i) To separate, by modeling them as separated agents, managing components from managed ones (e.g., the agent `OrganizationController` manages the local camera).

(ii) To separate, inside the behavior of a managing component, different adaptation concerns, each modeled as a separated transition rule (e.g., in the module `OrganizationController`, rule `r_failureAdapt[]` models the robustness scenario, while rule `r_master_congestionAdapt[]` models the flexibility scenario).

(iii) To keep separated, as invocations of separate rules, the four computations of a MAPE loop (e.g., rule `r_master_congestionAdapt[]` corresponds to a monitoring computation, while rule `r_analyzeCongestion[]` is an analyzing computation, both rules of the same adaptation loop).

(iv) To distribute a given MAPE loop among different agents (e.g., the computation M to detect camera failures in case of robustness loop is in the `SelfHealingController`, while the other A,P,E are in the `OrganizationController`).

This separation of concerns helps the designer to focus on one adaptation activity at a time, and, for each adaptation aspect, separate the adapting parts from the adapted ones. This also facilities reasoning about component behavior and avoid over-specification, keeping models concise.

The availability of an execution platform for SCA-ASM, made possible to validate adaptation scenarios. Initially, a number of *inconsistent updates* arose, during the scenario simulation, due to the simultaneous execution of different adaptive behaviors performed by the same component. Validation activity helped us to reason about *priorities* of the different adaptation concerns and to fix the component models accordingly. The formalization of the different adaptive behaviors in terms of separate MAPE loop rules and a priority-based sequential scheme to coordinate their execution made it easy to set the correct component behavior thus avoiding unexpected interference among MAPE loops.

10. CONCLUSION AND FUTURE WORK

This work is part of our ongoing effort in defining a formal framework to model self-adaptive service-oriented applications, and to validate and verify abstract models in order to assure properties (e.g., flexibility, robustness, etc.) of self-adaptive applications already at design time.

By showing how to model self-adaptation in an SCA-ASM assembly of service components, we provide a framework for rigorous development of service-oriented applications exposing adaptive behavior in a decentralized manner and with partial shared knowledge. We applied this modeling approach to the Traffic Monitoring case study.

The mathematical rigor of the ASM formal method and the execution nature of ASM models make SCA-ASM not only a formal framework to develop *design* models, covering both structural and behavioral aspects, of self-adaptive service-oriented applications, but also a means to perform rigorous model analysis. In our work we have carried out model validation by means of the SCA-ASM simulation environment [30]. This model validation allowed us to reproduce the different adaptation behaviors as implemented by the managing components over the managed ones, and be confident that the model behaved as expected.

Model validation is a model analysis activity to be executed from the earlier stages of the model development before other more demanding but more sophisticated and complex analysis techniques such as formal verification. Currently, we are studying how to apply ASM model slicing techniques to tackle the verification of self adaptive SCA-ASMs and guarantee *required properties* (robustness, flexibility, safety, liveness, etc.) by model checking [3].

In the future, we plan to improve SCA-ASM to provide the designer with distributed coordination patterns for MAPE-K loop interactions.

11. ACKNOWLEDGMENTS

The work was partially supported by Charles University research funds PRVOUK.

12. REFERENCES

[1] D. B. Abeywickrama, N. Hoch, and F. Zambonelli. SimSOTA: engineering and simulating feedback loops for self-adaptive systems. In B. C. Desai, A. M. de Almeida, J. Bernardino, and S. P. Mudur, editors, *International C* Conference on Computer Science & Software Engineering, C3S2E13, Porto, Portugal - July 10 - 12, 2013*, pages 67–76. ACM, 2013.

[2] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In E. Astesiano, editor, *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 21–37. Springer Berlin Heidelberg, 1998.

[3] P. Arcaini, A. Gargantini, and E. Riccobene. AsmetaSMV: A Way to Link High-Level ASM Models to Low-Level NuSMV Specifications. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *Abstract State Machines, Alloy, B and Z*, volume 5977 of *Lecture Notes in Computer Science*, pages 61–74. Springer Berlin Heidelberg, 2010.

[4] P. Arcaini, A. Gargantini, E. Riccobene, and P. Scandurra. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41(2):155–166, 2011.

[5] The ASMETA toolset website. http://asmeta.sourceforge.net/, 2015.

[6] M. Autili, P. Di Benedetto, and P. Inverardi. Context-aware adaptive services: The PLASTIC approach. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering*, volume 5503 of *Lecture Notes in Computer Science*, pages 124–139. Springer Berlin Heidelberg, 2009.

[7] E. Börger. Modeling workflow patterns from first principles. In C. Parent, K.-D. Schewe, V. Storey, and B. Thalheim, editors, *Conceptual Modeling - ER 2007*, volume 4801 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2007.

[8] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

[9] D. Brugali, L. Gherardi, E. Riccobene, and P. Scandurra. Coordinated Execution of Heterogeneous Service-Oriented Components by Abstract State Machines. In F. Arbab and P. C. Ölveczky, editors, *Formal Aspects of Component Software*, volume 7253 of *Lecture Notes in Computer Science*, pages 331–349. Springer Berlin Heidelberg, 2012.

[10] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. Lo Presti, and R. Mirandola. MOSES: A framework for QoS driven runtime adaptation of service-oriented systems. *IEEE Trans. Software Eng.*, 38(5):1138–1159, 2012.

[11] A. Carioni, A. Gargantini, E. Riccobene, and P. Scandurra. A Scenario-Based Validation Language for ASMs. In E. Börger, M. Butler, J. Bowen, and P. Boca, editors, *Abstract State Machines, B and Z*, volume 5238 of *Lecture Notes in Computer Science*, pages 71–84. Springer Berlin Heidelberg, 2008.

[12] B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, et al. *Software engineering for self-adaptive systems: A research roadmap*. Springer, 2009.

[13] J. Cubo, C. Canal, and E. Pimentel. Model-based dependable composition of self-adaptive systems. *Informatica (Slovenia)*, 35(1):51–62, 2011.

[14] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[15] J. L. Fiadeiro and A. Lopes. A model for dynamic reconfiguration in service-oriented architectures. *Software and System Modeling*, 12(2):349–367, 2013.

[16] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, 31(1):48–59, 1982.

[17] A. Gargantini, E. Riccobene, and P. Scandurra. A Metamodel-based Language and a Simulation Engine for Abstract State Machines. *Journal of Universal Computer Science*, 14(12):1949–1983, 2008.

[18] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In D. Garlan, J. Kramer, and A. L. Wolf, editors, *WOSS*, pages 33–38. ACM, 2002.

[19] R. Hebig, H. Giese, and B. Becker. Making control loops explicit when architecting self-adaptive systems. In *Proceedings of the Second International Workshop on Self-organizing Architectures*, SOAR '10, pages 21–28, New York, NY, USA, 2010. ACM.

[20] M. U. Iftikhar and D. Weyns. A case study on formal verification of self-adaptive behaviors in a decentralized system. In N. Kokash and A. Ravara, editors, *FOCLASA*, volume 91 of *EPTCS*, pages 45–62, 2012.

[21] S. C. J. Fox. An analysis of formal languages for dynamic adaptation. In *Proceedings of the 15th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS 2010, pages 3–13. IEEE, 2010.

[22] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[23] A. Lanoix, J. Dormoy, and O. Kouchnarenko. Combining proof and model-checking to validate reconfigurable architectures. *Electronic Notes in Theoretical Computer Science*, 279(2):43 – 57, 2011.

[24] R. Mirandola, P. Potena, and P. Scandurra. Adaptation space exploration for service-oriented applications. *Science of Computer Programming*, 80:356–384, 2014.

[25] B. Morin, O. Barais, G. Nain, and J.-M. Jezequel. Taming Dynamically Adaptive Systems Using Models and Aspects. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 122–132, Washington, DC, USA, 2009. IEEE Computer Society.

[26] E. Riccobene and P. Scandurra. A formal framework for service modeling and prototyping. *Formal Aspects of Computing*, 26(6):1077–1113, 2014.

[27] E. Riccobene and P. Scandurra. Formal modeling self-adaptive service-oriented applications. In R. L. Wainwright, J. M. Corchado, A. Bechini, and J. Hong, editors, *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1704–1710. ACM, 2015.

[28] E. Riccobene, P. Scandurra, and F. Albani. A modeling and executable language for designing and prototyping service-oriented applications. In *EUROMICRO-SEAA*, pages 4–11. IEEE, 2011.

[29] OASIS/OSOA. Service Component Architecture

(SCA). http://www.oasis-opencsa.org/sca.

[30] The SCA-ASM design framework. See directory `/code/experimental/SCAASM` in the svn repository `http://sourceforge.net/p/asmeta/code/`.

[31] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience*, 42(5):559–583, May 2012.

[32] Apache Tuscany. `http://tuscany.apache.org/`.

[33] P. Vromant, D. Weyns, S. Malek, and J. Andersson. On interacting control loops in self-adaptive systems. In H. Giese and B. H. C. Cheng, editors, *SEAMS*, pages 202–207. ACM, 2011.

[34] D. Weyns, M. U. Iftikhar, D. G. de la Iglesia, and T. Ahmad. A survey of formal methods in self-adaptive systems. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, C3S2E '12, pages 67–79, New York, NY, USA, 2012. ACM.

[35] D. Weyns, S. Malek, and J. Andersson. FORMS: A Formal Reference Model for Self-adaptation. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 205–214, New York, NY, USA, 2010. ACM.

[36] D. Weyns, B. R. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka. On patterns for decentralized control in self-adaptive systems. In R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems*, volume 7475 of *Lecture Notes in Computer Science*, pages 76–107. Springer, 2010.

ABOUT THE AUTHORS:

Paolo Arcaini is assistant professor at the Faculty of Mathematics and Physics, Charles University in Prague (Czech Republic). He is member of the Formal Methods Group of the Department of Distributed and Dependable Systems. He graduated and received a PhD in Computer Science at the University of Milan, Italy. His research topics include specification, verification and runtime verification using Abstract State Machines, and model-based testing. He serves as member of the program committee of international conferences. He published papers in international journals and in proceedings of international conferences.

Elvinia Riccobene is associate professor in Computer Science at the Computer Science Department of the University of Milan, Italy. He teaches courses on Software Engineering and on System Modeling and Analysis. She is the director of the FALSE Lab (Formal Methods and Algorithms for Large Scale Systems Laboratory). She received master and PhD degree in Mathematics. Her research interests include formal methods, with particular expertise in the Abstract State Machines, integration between formal modelling and model-driven engineering, model-driven design of service-oriented applications and self-adaptive systems, model analyses techniques for software systems. She has been the scientific coordinator of national research projects and she participated to various European and Italian research projects. She serves as member of the program committee of international conferences. She published several papers in international journals and in proceedings of international conferences.

Patrizia Scandurra is Assistant Professor at the Dep. of Management, Information and Production Engineering of the University of Bergamo (Italy). She teaches courses of software design/programming and operating systems. She received master and PhD degree in Computer Science. Her research interests include: integration of formal and semi-formal modelling languages; formal methods and functional analysis techniques; Software Architecture; Component-based Development; Service-oriented Computing; Model-driven Engineering; design and analysis techniques for distributed and self-adaptive systems, mobile/cloud systems, and embedded systems. She is a member of the Abstract State Machines (ASM) formal method community. She participated to several Italian and European research projects. She collaborates with companies and research centers. She serves as member of the program committee of international conferences. She published several papers in international journals and in proceedings of international conferences.

When Temporal Expressions Help to Detect Vital Documents Related to An Entity

Rafik Abbes
IRIT, University of Toulouse
rafik.abbes@irit.fr

Karen Pinel-Sauvagnat
IRIT, University of Toulouse
karen.sauvagnat@irit.fr

Nathalie Hernandez
IRIT, University of Toulouse
nathalie.hernandez@irit.fr

Mohand Boughanem
IRIT, University of Toulouse
mohand.boughanem@irit.fr

ABSTRACT

In this paper we aim at filtering documents containing timely relevant information about an entity (e.g., a person, a place, an organization) from a document stream. These documents that we call vital documents provide relevant and fresh information about the entity. The approach we propose leverages the temporal information reflected by the temporal expressions in the document in order to infer its vitality. Experiments carried out on the 2013 TREC Knowledge Base Acceleration (KBA) collection show the effectiveness of our approach compared to state-of-the-art ones.

CCS Concepts

•Information systems → Information retrieval; Document filtering;

Keywords

Entity, vital documents, temporal expressions, document filtering, TREC Knowledge Base Acceleration

1. INTRODUCTION

Knowledge bases such as Wikipedia and Freebase are among the main sources visited by users to access knowledge on a wide variety of entities [12]. Accessing this information might seem easy, but it must be tempered by the freshness of information that can be found in the knowledge bases. With the growing amount of information available on the web, it becomes more and more difficult to detect relevant and new information that can be used to update knowledge base entities. Frank et al. [10] showed that the median delay of update can reach 365 days for wikipedia articles related to a sample of non-popular entities, which makes many knowledge bases entries out of date. This gap could be reduced if timely-relevant information could be automatically detected as soon as it is published and then recommended to the editors.

Let t_0 be a reference date corresponding to the date of the last update of the knowledge base entity (e.g., the Wikipedia page of the entity). A system that analyses a stream of documents to filter those providing new entity related information that was not known at t_0 can be very useful for knowledge base editors. We call these documents *vital* documents.

The most challenging aspect is to draw a distinction between *old-relevant* documents (also called *useful*) and *vital* ones. The former provide entity related information that was known at t_0, whereas the latter reveal new relevant information about the entity that was not known at t_0 and they are more likely to be helpful to maintain an already up-to-date knowledge page entity.

In Figure 1, we consider the entity *Michael Schumacher* and the reference date t_0= *December* 2013. Document D does not provide any information about the entity, it is therefore *non-relevant*. Document A is *old-relevant* as it reports only old relevant information that is known at t_0. Document C contains timely relevant information about *Michael Schumacher*, and is thus considered as *vital*. Obviously, a previously *vital* document for the entity will become *old-relevant* in the future, for example document B is vital if t_0 = *December* 2013, but probably not after a couple of months.

Most of the state-of-the-art approaches [2, 7, 8, 15] focused on detecting documents that are relevant for the entity without distinguishing between *old-relevant* and *vital* ones. In this paper, we attempt to make this distinction and we aim at detecting only vital documents for the entity.

The approach we propose is based on leveraging the delay between the *publication date* of the document and the dates referred to by the *temporal expressions* (dates, days, times, etc.) that can exist in the document text. We think that when this delay is short, the document matching the entity is more likely to be *vital*. For example, in Figure 1, by analysing the temporal expressions mentioned in the text of the documents A and B, we can guess that the second document is more likely to be vital for the entity as it mentions temporal expressions (*December 31,2013*, *Tuesday*) that refer to a date close to the publication date of the document, whereas document A contains old dates (*2006*, *2012*) compared to the publication date (*2014*).

To the best of our knowledge, this is the first work that uses temporal expressions expressed in the document to infer its

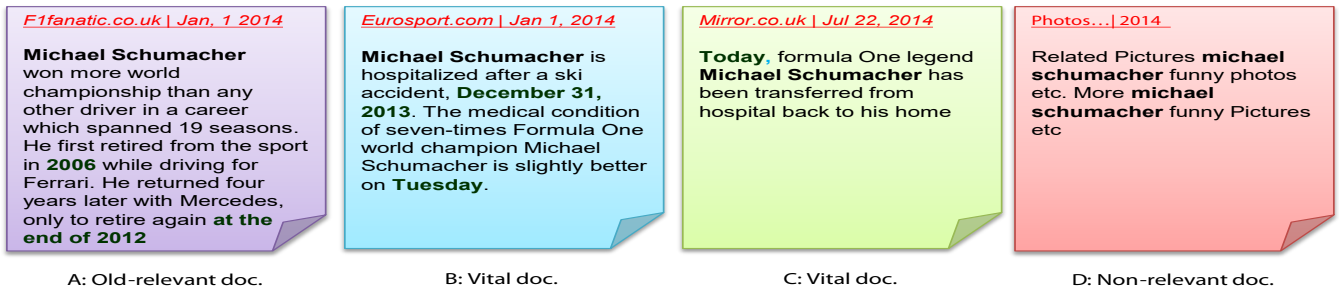


Figure 1: Distinction between *non-relevant*, *old-relevant* and *vital* documents assuming that the reference date t_0= *December* 2013

vitality for a given entity. Experiments carried out on the 2013 TREC KBA collection show the effectiveness of our approach against state-of-the-art ones.

The remaining of the paper is organized as follows. Section 2 reviews some related work. Section 3 describes the approach we propose. In Section 4, we report and discuss the experimental results carried out on the 2013 CCR collection of the TREC KBA track. We conclude and suggest some future work in Section 5.

2. RELATED WORK

In recent years, more and more attention has focused on filtering entity-centric documents. For instance, the TREC *Knowledge Base Acceleration* (KBA) has investigated the challenge of detecting relevant documents about a specific entity (e.g., a person, an organization, a place) since 2012. Particularly, the *Cumulative Citation Recommendation* (CCR) task strives to recommend to the editors of a knowledge base, relevant documents from an incoming document stream [9, 10, 11]. Since 2013, the CCR task asked participants to distinguish between *old-relevant* documents (called *useful* in the terminology of TREC KBA) and *vital* documents (containing timely relevant information about the entity). The proposed approaches can be classified into three categories.

The first category of approaches tackled the task as *a ranking problem* [7, 13]. Dietz and Dalton [7] used query expansion with related entity names to retrieve relevant documents. Liu et al. [13] ranked documents that match the entity by leveraging the number of occurrences and weights of related entities collected by parsing the Wikipedia page of the entity. The previously described works perform well when filtering relevant documents for the entity. However they do not attempt to distinguish *vital* documents from *old-relevant* ones. One possible reason is that these relevance models attempt to capture topicality rather than temporal characteristics.

The second category tackled the task as a *classification problem* [3, 4, 14]. Bonnefoy et al. [4] as well as Balog et al. [3] proposed one-step and multi-step classification approaches that attempt to learn the relevance of a document based on four families of features: (1) *document features* such as document length and document source; (2) *entity features* such as the number of related entities from DBpedia; (3) *document-entity features* that describe the relation between a document and the entity such as the number of occurrences of the entity in the document; and (4) *temporal features* which attempt to capture if something is happening around the entity at a given point in time by analysing the changes in the stream volume and in the number of views in the Wikipedia page of the entity. Wang et al. [14] used the Random Forest classifier trained on human-annotated documents. The same four families of features as in [3] were used, with in addition, *citation features* reflecting the similarity between a new document and cited documents in the Wikipedia page of the entity.

The previously reported works leverage *temporal* features that attempt to capture the entity related "bursts" (changes in the entity Wikipedia page views, or in the stream). These features have been shown to perform well [3]. In this work, we use another kind of temporal evidence to detect vital documents. We exploit *temporal expressions* in the document which, we believe, have not been previously used for this particular problem.

The first two categories of approaches assign a score for each document based on the probability of it belonging to a relevant class, or on a scoring function. Efron et al. [8] proposed a third kind of approach based on learning boolean queries that can be applied deterministically to filter relevant documents for the entity. In this work, we apply some heuristic boolean filtering rules that help to reject many non-vital documents (Section 4).

3. LEVERAGING TEMPORAL EXPRESSIONS FOR FILTERING VITAL DOCUMENTS

This paper is concerned with the task of filtering vital documents related to an entity: Given an entity E and a reference date t_0, we aim at identifying from a stream of documents those that are vital to the entity (i.e. containing new relevant information not known at t_0).

Generally, tackling this task involves two main steps: *filtering* then *scoring*. The filtering step can be seen as a way to eliminate many non-relevant documents (for example documents not mentioning the target entity). We also used this step and we detail some filtering rules in Section 4. In this section, we focus on the scoring step and we assume having a set of candidate documents that were filtered.

Our idea is to consider that a vital document related to an entity should report information about this entity and also should be fresh. *Freshness* can be determined by checking the publication date of the document and the temporal expressions used in its text. We assume that a vital document should be recent (published after the reference date t_0), and report a date greater than t_0 and close to its publication date.

In Figure 1, documents A and B are topically relevant to the entity *Michael Schumacher* and both were published on *Jan 1, 2014*. Document B is fresher than A because it mentions temporal expressions that refer to a date close (*Tuesday December 31,2013*) to the publication date of the document, whereas document A contains old dates (*2006*, *end of 2012*) compared to the publication date (*2014*).

Given an entity E and a new candidate document d published on $Date_p(d)$. Let $Date_t^*$ be the closest date (to $Date_p(d)$) recognized from the part of text mentioning the entity E in d. We assume that the shorter the period between $Date_p(d)$ and $Date_t^*$, the higher the probability of document d to be vital for E will be. Formally, we evaluate a freshness score of the document d with regard to the entity E as follows:

$$Freshness(d, E) = e^{-\frac{1}{\sigma^2}\Delta(d,E)} \quad (1)$$

$$\Delta(d, E) = \min_{x \in X(d,E)} (|Date_p(d) - Date_t(x, d)|^2) \quad (2)$$

Where

- $Date_p(d)$ is the publication date of d.
- $X(d, E)$ is the set of temporal expressions detected from the parts of d (sentences, paragraphs, etc.) that mention entity E.
- $Date_t(x, d)$ is the date indicated by the expression x.
- $\Delta(d, E)$ is the optimal (minimum) delay between $Date_p(d)$ and $Date_t(x, d)$ $\forall x \in X(d, E)$.
- $Date_t^*$ corresponds to $Date_t(x, d)$ where $\Delta(d, E)$ is minimal. $Date_t^*$ should be greater than the reference date t_0, otherwise the document is rejected as it is more likely to be *non-vital*.
- The maximum value of $Freshness(d, E)$ is equal 1 when document d mention a temporal expression referring to a fresh date that is equal to the publication date. When the delay $\Delta(d, E)$ increase, the freshness score tends to decrease until it reaches 0.
- Note that when the considered part of document d does not contain any temporal expression close to the entity E, its Freshness score is set to 0.
- The delay between $Date_p(d)$ and $Date_t(x, d)$ is measured in number of days.

As a candidate document may be fresh but not relevant to the entity E, we evaluate a relevance score in order to prioritize fresh relevant documents. *Relevance* score is evaluated as follows:

$$Relevance(d, E) = \prod_{t \in top_k(P_E)} P(t|\theta_d)^{P(t|\theta_{P_E})} \quad (3)$$

P_E is a known relevant page for the entity (for example, the Wikipedia page of the entity).

$top_k(P_E)$ is the set of top k frequent terms in P_E. It can be determined experimentally.

$P(t|\theta_d)$ and $P(t|\theta_{P_E})$ are estimated using a Dirichlet Smoothing as described in Eq. 4

$$P(t|\theta_d) = \frac{tf(t, d) + \mu \frac{tf(t,C)}{\sum_{t' \in C} tf(t',C)}}{|d| + \mu} \quad (4)$$

$tf(t, d)$ is the term frequency of term t in document d.

$tf(t, C)$ is the term frequency of term t in collection C.

C is the reference collection composed from early stream documents before t_0.

μ is a smoothing parameter used to avoid null probabilities.

Finally, we evaluate the vitality score of a document (eq. 5) as the product combination of the *relevance* and *freshness* scores with a $+\epsilon$ smoothing for the freshness score to avoid a null vitality score when the freshness score is equal to zero caused by the absence of temporal expressions.

$$Vitality(d, E) = Relevance(d, E) * (Freshness(d, E) + \epsilon) \quad (5)$$

4. EXPERIMENTS

4.1 Collection

We evaluated our approach within the 2013 CCR task of the TREC KBA track. The task is defined as follows: given an entity identified by a URI (Twitter/Wikipedia), a CCR system strives to recommend to the contributors of a knowledge base relevant documents that are worth citing in a profile of the entity (e.g. its Wikipedia article). The stream corpus contains more than 500 million web documents from several sources (News, Social, Forum, Blog, etc.). It has a size of 4.5 Tera Bytes (compressed), and documents were published in the time range of October 2011 through February 2013. The stream corpus is divided into a training part and an evaluation part. In the latter, we conducted our experiments with **122** entities (persons, organizations, locations). In Table 1, we present some statistics about the KBA 2013 collection. We note that among the 122 entities, only 108 have at least one vital document. Moreover, the higher value of the mean compared to the median indicates that there are some entities that have many more vital documents than others.

Table 1: Some statistics about the KBA 2013 collection

	Training	Evaluation
Time range	Oct.2011 - Feb.2012	Mar.2012 - Feb.2013
Entities with vital(s)	88	108
Total vital	1619	3922
Median of vital	2	6
Mean of vital	13	32

Document annotations were done by the KBA organizers. A document is considered as *vital* if it contains a timely relevant information about the entity (not known before the reference date $t_0 = January,\ 2012$), *useful* (*old-relevant*) if it contains relevant but not timely information about the entity, and *non-relevant* otherwise. We recall that in this work we are interested only in *vital* documents. The official metric for the KBA CCR task is the ***hF1***, i.e. the maximum macro-averaged *F1* measure (Eq. 6). $F1$ is evaluated for each confidence cutoff $i \in]0, 1000]$ (Eq. 7), where 1000 corresponds to the highest level of confidence and 1 corresponds to the level in which all documents are kept.

$$hF1 = Max_i(F1@i) \tag{6}$$

$$F1@i = \frac{2 * mPrecision@i * mRecall@i}{mPrecision@i\ +\ mRecall@i} \tag{7}$$

$$mPrecision@i = \frac{1}{n} * \sum_{E \in \Omega} Precision@i(E) \tag{8}$$

$$mRecall@i = \frac{1}{n} * \sum_{E \in \Omega} Recall@i(E) \tag{9}$$

Ω: set of all evaluated entities

n: number of evaluated entities

$Precision@i(E)$: Precision of E at the confidence cutoff i

$Recall@i(E)$: Recall of E at the confidence cutoff i

4.2 Our proposed method for the CCR task

As indicated in section 3, an entity-vital document filtering system involves two main steps, *filtering* then *scoring*.

Filtering step:.

To reduce the number of documents that are more likely to be non-vital for the entity, we define two sub-steps :

- **Entity Matching** (***E-Matching***):
  A vital document should mention the target entity. As an entity can be mentioned in a document with different variants (surface forms), we collect for each entity a list of variants. For a Wikipedia entity, we use its Wikipedia page title and the bold texts in the first paragraph as variants [6]. For a Twitter entity, we use the display name in the Twitter page as variant. In this level, we retain all documents matching at least one variant of the entity. We alleviate queries in order to capture the maximum number of potential relevant documents. For example, for the Wikipedia entity *Phyllis Lambert*, we can capture documents mentioning the entity by *Phyllis Lambert*, *Phyllis Barbara Lambert* or *Phyllis B Lambert*, etc.

- **Filtering spam documents** (***Filters***):
  Excluding spam documents could improve system performance. Therefore, we define three filters in order to reject documents matching the entity but more likely to be spam:

  - *A language filter* that removes all documents recognized as Non-English-documents using a Java language detector[1].
  - *An enumeration filter* that removes documents that mention the entity only in an abusive list of more than n entities. We set n to 30 based on some observations in the training time range.
  - *A link filter* that removes documents that contain more than *20* hyper-links (based on some observations in the training time range).

Scoring step:.

We evaluate the vitality score of documents that passed the filtering step based on *topicality and freshness* as described in Eq. 1. *Freshness* is reflected by the dates recognized from temporal expressions contained in the document text. Temporal expressions such as *Last week*, *At the end of 2012*, *December 31, 2013*, etc. are identified and normalized using SUTIME [5]. This library uses the document publication date as reference. For example, for a document from 2014-01-01, SUTIME would resolve the date referred to by *Tuesday* as 2013-12-31.

We considered three configurations of our approach:

- **T&F$_{\mathbf{Sen}}$**: The freshness is estimated considering the temporal expressions recognised from the *sentences* mentioning the entity.
- **T&F$_{\mathbf{Pg}}$**: The freshness is estimated considering the temporal expressions recognised from the *paragraphs* mentioning the entity.
- **T&F$_{\mathbf{W}}$**: The freshness is estimated considering the temporal expressions recognised from the *whole document*.

To evaluate the impact of *freshness*, we run another configuration that considers only the *topicality* of documents (Eq. 3); we denote it by **onlyT**.

[1] www.jroller.com/melix/entry/nlp_in_java_a_language

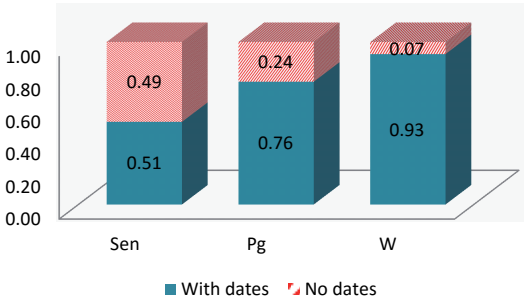


Figure 2: Availability of dates in the different parts of all documents that pass the filtering step

The above configurations are evaluated with and without using spam filters. When spam filters are used, we add +**S** to the configuration label.

To estimate the topical relevance of the document (Eq. 3), we use a known relevant page (P_E) to get information about the entity. For a Wikipedia entity, P_E consists of the Wikipedia article of the entity (on January 1st, 2012), and for a Twitter entity, P_E contains only the display name in the entity Twitter page.

Our approach uses 3 parameters σ, μ and $top_k(P_E)$. To estimate their optimal values, we use a 3-fold cross-validation method. We vary $\sigma \in [1, 360]$ (step=30), $\mu \in [50, 1000]$ (step=50) and $top_k(P_E) \in [5, 30]$ (step=5). Optimal values are: $\sigma = 30$, $\mu = 200$ and $top_k(P_E) = 20$. We set ϵ in eq. 5 to 10^{-4}.

The CCR task requires systems to assign a confidence score $\in [1, 1000]$ to each document. We adopted the following strategy; rank 1 gets a confidence value 1000, rank 2 gets a confidence value 999, etc.

4.3 Results

4.3.1 *Recognized dates in documents*

The freshness score calculated in equation 1 is based on the recognition of temporal expressions mentioning the entity. However, in some cases, we can fail to detect a temporal expression in one or some parts (Sen, Pg, W) of the document. Figure 2 shows the availability of dates in the different parts of all documents matching the studied entities.

We can see that most of the documents (93%) contain at least one temporal expression in their bodies. Considering only smaller parts that mention the entity, we can recognize at least one date in a large part of paragraphs (76%) and in the half of sentences.

In our approach, we hope that in vital documents we can find more fresh dates than in other documents classes which means that the optimal temporal distance in vital documents ($\Delta(d, E)$) are expected to be low. We further analyse documents by classifying $\Delta(d, E)$ into two ranges : "one-year-fresh" range when the distance is less than one year and "old" otherwise. Figures 3, 4 and 5 plot the results by distinguishing the three different classes : vital, useful and non-relevant respectively.

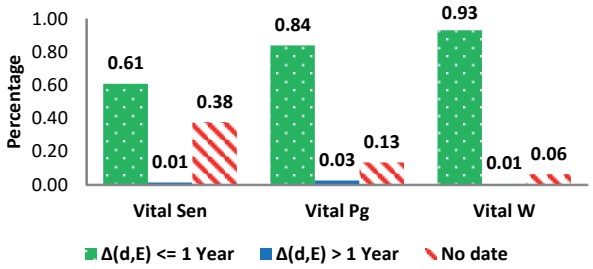


Figure 3: Freshness of dates in vital documents

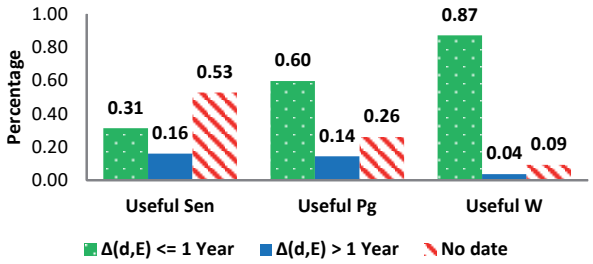


Figure 4: Freshness of dates in useful documents

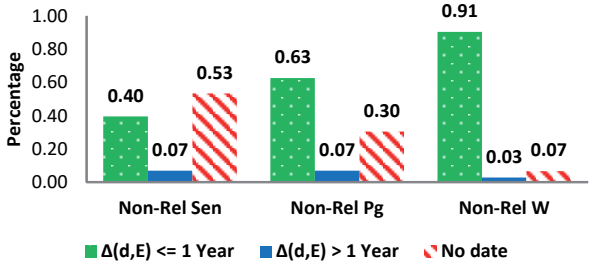


Figure 5: Freshness of dates in non-relevant documents

We can see that the percentage of one-year-fresh dates are greater in vital documents than in useful or non-relevant documents. The difference is notable, especially when focusing on the sentences or the paragraphs. 84% of vital paragraphs contain a temporal expression referring to a fresh date comparing to the publication date of the document (i.e., $\Delta(d, E) < 1year$). In addition, we notice also that old dates are more likely to be mentioned in useful documents that describe old-relevant information about the entity.

In the following section we compare our system configurations that leverage temporal expressions recognized in the different parts of documents.

Table 2: Comparison of the different configurations of our approach. i* corresponds to the confidence cut-off in which F1 is maximum.

	i*	mPrec.@i*	mRec.@i*	hF1
$T\&F_{Sen}+S$	910	**0.281**	0.637	0.390
$T\&F_{Pg}+S$	910	**0.281**	0.671	**0.396**
$T\&F_{W}+S$	370	0.249	**0.783**	0.378
$onlyT+S$	340	0.248	**0.783**	0.377
$T\&F_{Sen}$	590	0.229	0.765	0.352
$T\&F_{Pg}$	920	0.256	0.617	0.362
$T\&F_{W}$	880	0.232	0.673	0.345
$onlyT$	870	0.229	0.675	0.342

4.3.2 *Comparison of the different configurations*

Table 2 compares the different configurations of our approach. We observe that exploiting the freshness and topicality ($T\&F_*$) improves results compared to using only the topicality without leveraging temporal expressions ($onlyT$). This confirms that freshness represents an important factor to detect vital documents. Estimating freshness using only parts of text describing the entity (Sen or Pg) performs better than using the whole document content. One reason could be that considering some parts that are not in the proximity of references to the entity could bring fresh dates (closer to the publication date) which are not related to the target entity. Searching temporal expressions only in sentences mentioning the entity ($T\&F_{Sen}*$) may be insufficient in some cases to detect dates related to the entity (as shown in figure 3, 61% of sentences mentioning the entity in vital documents contain a fresh date), which can explain the slight improvement when considering the paragraphs mentioning the entity ($T\&F_{Pg}*$) where fresh dates are present in 84% of them. The high values of the optimal confidence cutoff ($i^* = 910$) indicates that $T\&F_{Sen}+S$ and $T\&F_{Pg}+S$ rank well vital documents.

We also notice that all configurations perform better when applying spam filters. Figure 6 shows the impact of each of the filters in hF1 when added to $T\&F_{Pg}$. We can observe that all of them have a good precision (97%) which means that they do not reject many vital documents. $hF1$ is well improved especially when using *link* and *enumeration* filters which indicates that the test collection contains many documents mentioning the target entity in a spam way without providing any relevant information about it. Using all filters rejects many non-vital documents (about one-fourth) which improves the filtering step performance in terms of $hF1$ (+0.032). This observation supports the hypothesis made in [8]: *well-crafted Boolean queries can be effective filters for CCR task.*

4.3.3 *Vitality probability given the value and the position of the optimal date*

In figure 7 we analyse the probability of document d be vital given the value of the optimal delay $\Delta(d, E)$ and the considered part in the document (Sen, Pg or W).

The first remark worth making is the presence of a fresh

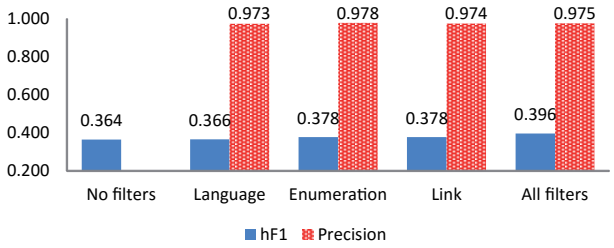


Figure 6: Impact of each spam filter on hF1 when added to $T\&F_{Pg}$. $Precision = \frac{\#non\ vital\ rejected\ docs}{\#all\ rejected\ docs}$

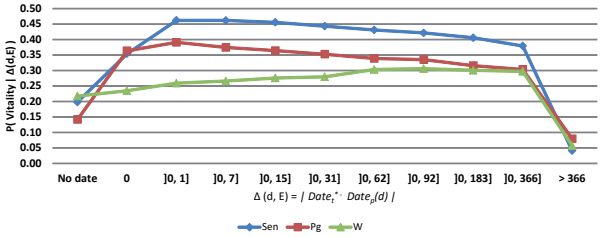


Figure 7: Vitality probability of a document given of the optimal delay and the considered part in the document (Sen, Pg, W)

date (what ever its value) in the sentence mentioning the entity gives a better indication of the document vitality (up to 46%) than its presence in a farther position (i.e., Pg or W). In addition, the fresher this date is, the higher the vitality probability is, except when the optimal delay $\Delta(d, E)$ is zero. The reason of this exception is that many non-vital documents mention their publication date in the beginning or in the end of the body. We can also notice that when $\Delta(d, E)$ is greater that one year, the vitality probability of the document becomes very low. This observation can be helpful to discard many non-vital documents from stream. Finally, when no date is recognized in the document, the probability of vitality is low but not too low, which requires to exploit other features to decide whether the document is vital or not.

4.3.4 *Our approach vs. state-of-the-art approaches*

In this section, we compare our best configuration ($T\&F_{Pg}+S$)[2] with the top 3 CCR systems: **BIT**[14], **Umass**[7] and **Udel**[13].

Table 3 compares system performances in the *filtering* step considering all returned documents (i.e., *confidence cutoff =1*). We denote our approach in the filtering step by **E-Matching** when no filter is used and **E-Matching+filters** when filters are applied. *Ranking strategies* (our approach, Umass, Udel) perform generally better in recall than the *classification strategy* (BIT) that can be penalized if it fails to properly classify many vital documents. Our matching

[2]Comparison is done using the official scorer of the task. Our best run is available at this URL: http://www.irit.fr/~Rafik.Abbes/SAC15Runs/

Table 3: Comparison of our approach with 2013 CCR systems in the filtering step.

	mPrec.@1	mRec@1	F1@1
Udel	0.199	0.695	0.309
Umass	0.201	0.662	0.309
BIT	0.244	0.650	0.355
E-Matching	0.217	**0.794**	0.354
E-Matching + Filters	**0.248**	0.782	**0.377**

method performs best among the proposed methods in terms of recall which can be explained by the use of alleviated queries in the entity matching sub-step.

Table 4 compares systems using the official metric of the task (hF1). Our approach outperforms the best proposed system in the task (BIT). Significance test is not suitable in the case of $mPrecision@i^*$, $mRecall@i^*$ and $hF1@i^*$ because the optimal confidence cutoff i^* and the confidence scoring strategy are not the same for each system.

Table 4: Comparison of our approach with the 2013 CCR systems. i* corresponds to the confidence cutoff in which F1 is maximum.

	i*	mPrec.@i*	mRec.@i*	hF1
Udel	40	0.199	0.695	0.309
Umass	660	0.216	0.591	0.316
BIT	140	0.257	0.601	0.360
$T\&F_{Pg}+S$	910	**0.281**	**0.671**	**0.396**

To evaluate in depth the performance of our system compared to the best proposed system (BIT) in the task, we consider the top-30 returned documents of each system. We choose 30 as it represents the mean of vital documents per entity. Table 5 shows the results using the macro averaged precision ($mPrec.@T30$), recall ($mRec@T30$) and F-measure ($mF1@T30$). Results show that our approach ($T\&F_{Pg}+S$) significantly improves the ranking of vital documents compared to BIT. Figure 8 illustrates the difference in $F1@T30$ of each entity between $T\&F_{Pg}+S$ and BIT. We can observe that $T\&F_{Pg}+S$ performs better than BIT in 69 topics and worse in 25.

Table 5: Comparison of our best configuration ($T\&F_{Pg}+S$) with the best proposed system (BIT) in the task . † denotes a significant improvement (we used the paired t-test with p <0.05). *@T30* means considering the top-30 documents

	mPrec.@T30	mRec.@T30	mF1@T30
BIT	0.211	0.297	0.170
$T\&F_{Pg}+S$	**0.286** †	**0.489** †	**0.262** †

Let us take for example the entity *Hoboken Volunteer Ambulance Corps*. This topic has 32 vital documents in the evaluation time range. Regardless of the ranking (considering all the returned documents, i.e., *confidence cutoff=1*), our approach ($T\&F_{Pg}+S$) returns 66 documents containing 31 vital ones, whereas BIT returns 117 documents containing 29 vital ones. We can see that both methods perform well in recall (> 90%). However, our method rejects more non-vital documents than BIT due to the use of spam filters which allow rejecting many spam documents such as the third document in Figure 9. In terms of ranking, considering the top-30 returned documents of each system, BIT gets only one vital document, whereas, our approach ($T\&F_{Pg}+S$) detects 16 vital documents which corresponds to 50% of recall. These documents contain fresh dates in the proximity of the entity (such as the first two documents in Figure 9), which can explain the good performance.

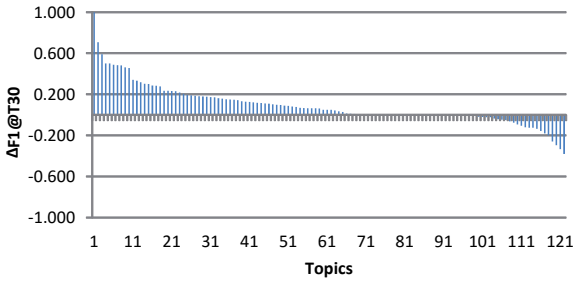


Figure 8: Difference in $F1@T30$ between $T\&F_{Pg}+S$ and BIT for each topic

4.3.5 Real cases from the TREC KBA 2013 corpus

Table 6 shows some real case examples from the TREC KBA 2013 corpus. In the first part, we show examples in which temporal expressions recognized either implicitly (line 1) or explicitly (line 2) from the documents are helpful to infer their vitality as the delay of the inferred dates to the publication dates are low. The large delay in the third example (line 3) infer correctly the right class of the document.

In the second part of the table, we give some examples in which using only the temporal expressions is insufficient for many possible reasons :

- some vital documents do not mention a date near to the entity (lines 4 and 5),
- the delay is high because the vital document mention an upcoming event in the far future (line 6),
- the normalisation of the recognized date is wrong like in line 7 where the expression *'to this day'* refers to a past fact whereas the tool that we used to recognize dates considered that this expression refers to the present and therefore the delay is estimated to 0,
- some non-relevant documents (lines 8, 9 and 10) can mention a new date, so the freshness score should be combined by a relevance score. We remark that the relevance score is better estimated for wikipedia entities for which we have some context information (the wikipedia page of the entity), whereas for twitter entities only the entity name is used.

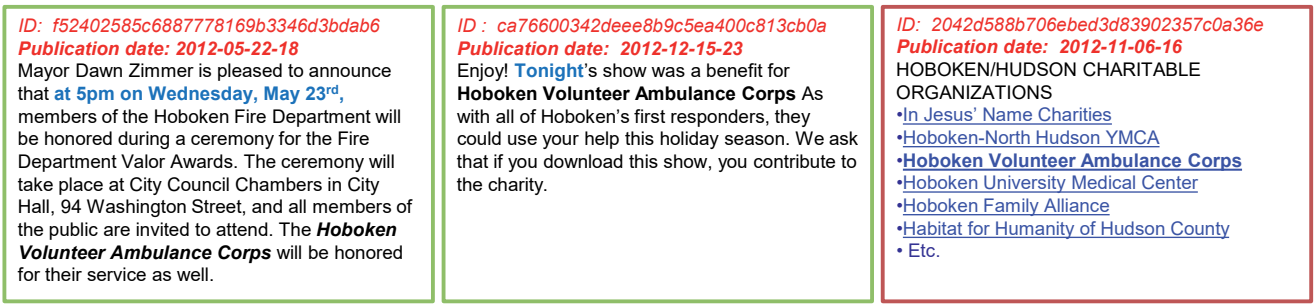


Figure 9: Examples of documents for the topic *Hoboken Volunteer Ambulance Corps*, the first two documents are vital and the last one is not relevant rejected by our spam filters

Table 6: Real cases from the TREC KBA 2013 corpus when the temporal expressions are helpful and when they are insufficient

#	Entity (E)	Sentence (Sen)	$Date_p(d)$	$\Delta(d, E)$	class
		cases where temporal expressions are helpful			
1	Atacocha	On ***Friday***, silver miner Minera *Atacocha*, the Lima-based zinc and silver mining company, fell by 5.4%	2012-03-02-04	1 day	vital
2	Barbara Liskov	***Monday, 05 March 2012*** *Barbara Liskov* is among the 2012 inductees to the National Inventors Hall of Fame in recognition of her contributions to programming languages and system design.	2012-03-05-12	0 day	vital
3	Brenda Weiler	The local chapter's community walk was started in 2006 by *Brenda Weiler*, of Fargo, after she lost her older sister to suicide ***the year prior***.	2012-09-22-05	265	old-relevant
		cases where temporal expressions are insufficient			
4	Angelo Savoldi	The NWA is pleased and proud to induct *Angelo Savoldi* into the Hall of Fame!	2012-11-14-13	No date!	vital
5	Barbara Liskov	Murray on mathematical biology, *Barbara Liskov* of MIT on in modern programming languages, Ronald Rivest of MIT on cryptography, Leslie G.	2012-04-30-20	No date!	vital
6	evvnt	The 13th Annual European Shared Services & Outsourcing Week The 13th Annual European Shared Services & Outsourcing Week The 13th Annual European Shared Services & Outsourcing Week offered by *evvnt* will take place in Prague on ***21 May 2013***	2012-11-14-06	188 days	vital
7	Bob Bert	Drummer *Bob Bert* played on the album but departed before the tour, and his replacement Steve Shelley remains to ***this day***.	2012-11-08-12	0 day	old-relevant
8	Tony Gray	Port continued their pre-season schedule with a 4-0 win at Radcliffe Borough on ***Tuesday night*** with goals from Steven Tames (two), Shaun Whalley and *Tony Gray*	2012-08-02-09	0 day	non-relevant
9	Alexandra Hamilton	By *Alexandra Hamilton* Email the author ***March 6, 2012*** Tweet Email Print 1 Comment ? Back to Article new Embed | Share	2012-03-06-12	0 day	non-relevant
10	Blair Thoreson	*Blair Thoreson* has served in the North Dakota House of Representatives since 1998, representing District 44. Tags: ALEC, American Legislative Economic Council, *Blair Thoreson*, van jones This entry was posted on Wednesday, ***April 18th, 2012 at 12:19 pm*** and is filed under Blog	2012-04-18-19	0.5 day	non-relevant

5. CONCLUSION AND FUTURE WORK

In this paper, we are interested in filtering vital documents related to an entity from a document stream. We propose an approach that evaluates the freshness of temporal expressions that mention the entity with regard to the publication date of the document to infer the vitality. Experiments carried out over the 2013 TREC KBA collection confirm the usefulness of leveraging temporal expressions to detect vital documents. We gave some real examples in which leveraging temporal expressions can be helpful is some cases or insufficient in others. The absence of temporal expressions in some documents, or in a specific considered part like the sentences mentioning the entity, requires to exploit other factors in order to estimate the vitality. In addition, we show that applying some boolean filters leads to substantial improvement in the system's performance. Further work will concern a way to combine the freshness of temporal expressions with other factors that can infer vitality such as detecting bursts in the stream or exploiting some action patterns. In addition, we would like to investigate how to automatically extract vital information from the detected vital documents. In this context, we made a preliminary work described in [1] that extracts only the interesting sentences from the document stream. In the short term, we intend to extend our system by using knowledge extraction tools in order to automatically update knowledge bases.

6. REFERENCES

[1] R. Abbes, K. Pinel-Sauvagnat, N. Hernandez, and M. Boughanem. Accelerating the update of knowledge base instances by detecting vital information from a document stream. In *2015 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT), Singapore, December 06-09, 2015*, 2015. (to appear).

[2] K. Balog and H. Ramampiaro. Cumulative citation recommendation: Classification vs. ranking. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, pages 941–944, New York, NY, USA, 2013. ACM.

[3] K. Balog, H. Ramampiaro, N. Takhirov, and K. Nørvåg. Multi-step classification approaches to cumulative citation recommendation. In *Proceedings of the 10th Conference on Open Research Areas in Information Retrieval*, OAIR '13, pages 121–128, Paris, France, 2013.

[4] L. Bonnefoy, V. Bouvier, and P. Bellot. A weakly-supervised detection of entity central documents in a stream. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, pages 769–772, New York, NY, USA, 2013. ACM.

[5] A. X. Chang and C. Manning. Sutime: A library for recognizing and normalizing time expressions. In *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, Istanbul, Turkey, may 2012. European Language Resources Association (ELRA).

[6] S. Cucerzan. Large-scale named entity disambiguation based on Wikipedia data. In *Proceedings of the Joint Conference on EMNLP-CoNLL*, EMNLP-CoNLL'07, pages 708–716, Prague, Czech Republic, June 2007. Association for Computational Linguistics.

[7] L. Dietz and J. Dalton. Umass at TREC 2013 knowledge base acceleration track: Bi-directional entity linking and time-aware evaluation. In *Proceedings of The Twenty-Second Text REtrieval Conference, TREC 2013, Gaithersburg, Maryland, USA, November 19-22, 2013*, 2013.

[8] M. Efron, C. Willis, and G. Sherman. Learning sufficient queries for entity filtering. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '14, pages 1091–1094, New York, NY, USA, 2014. ACM.

[9] J. R. Frank, S. J. Bauer, M. Kleiman-Weiner, D. A. Roberts, N. Tripuraneni, C. Zhang, C. Ré, E. M. Voorhees, and I. Soboroff. Evaluating stream filtering for entity profile updates for TREC 2013. In *Proceedings of The Twenty-Second Text REtrieval Conference, TREC 2013, Gaithersburg, Maryland, USA, November 19-22, 2013*, 2013.

[10] J. R. Frank, M. Kleiman-Weiner, D. A. Roberts, F. Niu, C. Zhang, C. Ré, and I. Soboroff. Building an entity-centric stream filtering test collection for TREC 2012. In *Proceedings of The Twenty-First Text REtrieval Conference, TREC 2012, Gaithersburg, Maryland, USA, November 6-9, 2012*, 2012.

[11] J. R. Frank, M. Kleiman-Weiner, D. A. Roberts, E. M. Voorhees, and I. Soboroff. Evaluating stream filtering for entity profile updates in TREC 2012, 2013, and 2014. In *Proceedings of The Twenty-Third Text REtrieval Conference, TREC 2014, Gaithersburg, Maryland, USA, November 19-21, 2014*, 2014.

[12] X. Li, C. Li, and C. Yu. Entity-relationship queries over wikipedia. *ACM Transactions on Intelligent Systems and Technology*, pages 70:1–70:20, 2012.

[13] X. Liu, H. Fang, and J. Darko. A related entity based approach for knowledge base acceleration. In *Proceedings of The Twenty-Second Text REtrieval Conference, TREC 2013, Gaithersburg, Maryland, USA, November 19-22, 2013*, TREC'13, Gaithersburg, USA, 2013.

[14] J. Wang, D. Song, C.-Y. Lin, and L. Liao. BIT and MSRA at TREC KBA CCR Track 2013. In *Proceedings of The Twenty-Second Text REtrieval Conference, TREC 2013, Gaithersburg, Maryland, USA, November 19-22, 2013*, TREC'13, Gaithersburg, USA, 2013.

[15] M. Zhou and K. C.-C. Chang. Entity-centric document filtering: Boosting feature mapping through meta-features. In *Proceedings of the 22Nd ACM International Conference on Conference on Information & Knowledge Management*, CIKM '13, pages 119–128, New York, NY, USA, 2013. ACM.

ABOUT THE AUTHORS:

Rafik Abbes received the Bachelor's Degree in computer science in June 2011 from the Higher Institute of Computer Science and Multimedia of Sfax (University of Sfax, Tunisia) and the M.S degree in computer science in June 2012 from Paul Sabatier University (University of Toulouse, France). Currently, he is a Ph.D Student in the University of Toulouse since September 2012. His research interests mainly focus on Entity-centric Information retrieval and Knowledge Base population.

Karen Pinel-Sauvagnat is an Associate Professor in the University of Toulouse. She received her Ph.D degree in Computer Science from Paul Sabatier University in June 2005. Her research interests mainly fall in the area of Information Retrieval. She is currently working on the following subjects: Aggregated search, Evaluation of Information retrieval, Contextual Information retrieval. In the past, she also worked on Information retrieval on semi-structured documents.

Nathalie Hernandez is an Associate Professor in the University of Toulouse. She received her Ph.D degree in Computer Science from Paul Sabatier University in December 2005. Her research interests fall in the area of Semantic Web, Knowledge Engineering and Information Retrieval. Her current research activities include Ontology construction and enrichment, Document annotation and Semantic querying.

Mohand Boughanem is a Professor in the University of Toulouse. He received his Ph.D degree in Computer Science from Paul Sabatier University in 1992. His research interests mainly fall in the area of information retrieval. He is particularly interested in the following topics: Information Retrieval Models, Contextual and Social information retrieval, Blog search, opinion detection, aggregated search, XML information retrieval, Semantic-based IR and IR evaluation.

A Wait-Free Multi-Producer Multi-Consumer Ring Buffer

Steven Feldman
University of Central Florida
Feldman@knights.ucf.edu

Damian Dechev
University of Central Florida
dechev@eecs.ucf.edu

ABSTRACT

The ring buffer is a staple data structure used in many algorithms and applications. It is highly desirable in high-demand use cases such as multimedia, network routing, and trading systems. This work presents a new ring buffer design that is, to the best of our knowledge, the only array-based first-in-first-out ring buffer to provide wait-freedom. Wait-freedom guarantees that each thread completes its operation within a finite number of steps. This property is desirable for real-time and mission critical systems.

This work is an extension and refinement of our earlier work. We have improved and expanded algorithm descriptions and pseudo-code, and we have performed all new performance evaluations.

In contrast to other concurrent ring buffer designs, our implementation includes new methodology to prevent thread starvation and livelock from occurring.

CCS Concepts

•Theory of computation → Shared memory algorithms;

Keywords

concurrent, parallel, non-blocking, wait-free, ring buffer

1. INTRODUCTION

A ring buffer, or cyclical queue, is a first-in-first-out(FIFO) queue that stores a finite number of elements, usually on a fixed-length array. This design enables operations to complete in O(1) complexity and allows for cache-aware optimizations. Additionally, its low memory utilization makes it highly desirable for systems with limited memory.

The rise in many-core architecture has resulted in an increase in applications that are parallelizing their workloads (e.g., cloud-based services). Often buffers are used to pass work from one thread to another, and increasingly these buffers have become the source of bottlenecks.

Existing research has forgone the use of coarse-grained lock, and it has achieved greater scalability and core utilization by using fine-grained and/or non-blocking designs. Non-blocking algorithms are a class of algorithms that focus on providing guarantees of progress, which imply freedom from different concurrency dangers. The strongest form of non-blocking algorithms are wait-free, which guarantees that each operation completes at a finite number of steps. Wait-free algorithms are immune to deadlock, livelock, and thread starvation [6].

Deadlock occurs when there is a cyclic dependency between one or more operations. This dependency causes them to wait for one another to finish, thus blocking all operations involved. In livelock, operations are continually yielding to one another instead of making progress with their own operations. Thread starvation occurs when an operation waits indefinitely for a resource or is continually denied a resource.

Weaker forms include lock-free and obstruction-free, which are only free from a subset of those dangers. Lock-free algorithms guarantee that some thread is always making progress in its operation. Obstruction-free guarantees that if all threads are suspended, there exists a thread that if resumed and executed in isolation, then it can complete its operation.

We are aware of two other non-blocking ring buffers in literature. Tsigas et al. presented a lock-free approach in which threads compete to apply their operation [11]. Section 7 shows this approach suffers from thread congestion and poor scaling. Krizhanovsky presented a non-blocking approach which improves scalability through the use of the *fetch-and-add* (*faa*) operation [8], but unfortunately the design is susceptible to thread starvation and is not thread death safe.

To address these limitations we propose a new wait-free ring buffer algorithm based on the atomic *faa* operation. A thread performing an enqueue or dequeue operation will perform a *faa* on the tail or head sequence counter, respectively. The returned sequence identifier (*seqid*) is used to determine the position to enqueue or dequeue an element from the buffer. Specifically, the position is determined by the *seqid* modulo the buffer's length. This *seqid* is also stored within the value type that is placed on the buffer. To address the case where the *seqid* of two different threads refer to the same position, we employ a novel method of determining which thread is assigned the position and which

thread must get a new position. To prevent the case in which a thread is continually prevented from completing its operation, we integrated a progress assurance scheme [2].

Previous research has shown that the use of the *faa* operation can provide significant increase in performance when compared to similar designs implemented using the atomic compare-and-swap (*cas*). For example, Feldman et al. [3] compared the performance of a *faa*-based vector push back operation and a *cas*-based approach and found that the *faa* design outperforms the *cas* design by a factor of 2.3.

This work includes a naive wait-free ring buffer design that uses a multiword-compare-and-swap (mcas) [1] to enqueue and dequeue values. This allows us to compare the performance difference between our *faa*-based approach to that of a *cas*-based approach. Section 7 provides a detailed analysis of the performance difference between these approaches.

This work presents the following novel contributions:

- This is the first array-based wait-free ring buffer. Other known approaches are susceptible to hazards such as livelock and thread starvation.
- Our design presents a new way of applying sequence numbers and bitmarking to maintain the FIFO property. This allows for a simple way to mark and correct out-of-sync locations.
- Our design maintains throughput in scenarios of high thread contention. Other known approaches degrade as the thread contention increases, making our implementation more suitable for highly parallel environments.

2. ALGORITHM DESIGN

This section describes how threads perform the *enqueue* and *dequeue* operations. These operations were implemented using the open-source *Tervel* wait-free algorithm framework [2]. This framework provides memory management constructs, inter-thread helping techniques, and a progress assurance scheme. Progress assurance enables a thread to detect when it has been delayed by other threads and recruit those threads to aid in its operations.

Figure 1 presents the ring buffer class, its public member functions, and its private member variables.

Table 1 provides an overview of several utility functions used in the presented algorithms. Their full implementations have been omitted for brevity, but their names are representative of their behavior and we clarify their behavior in text. A full and annotated implementation is available in the latest release of the *Tervel* library [2].

The rest of this section is organized as follows: Section 2.1 presents the restrictions and limitations of our design. Section 2.2 presents an overview of our approach and the ring buffer class object. Sections 2.3 and 2.4 present high-level descriptions of the enqueue and dequeue opetations. Sections 2.5 and 2.6 expand upon the high-level descriptions using implementation examples.

```
1: template<typename T>
2: Class RingBuffer
3:     //member functions
4:     bool isFull();
5:     bool isEmpty();
6:     bool enqueue(T v);
7:     bool dequeue(T &v);
8:     //member variables
9:     const int64_t capacity_;
10:    std::atomic<int64_t> head_ 0;
11:    std::atomic<int64_t> tail_ 0;
12:    std::unique_ptr<std::atomic<uintptr_t>[]> array_;
13: end Class
```

Figure 1: Ring Buffer Class.

- valueType: a pointer to a type T object
- emptyType: a pointer sized integer indicating that a position does not hold a value.
- nextHead(): returns buffer.tail_.fetch_add()
- nextTail(): returns buffer.head_.fetch_add()
- getPos(int v): returns v%buffer.capacity_
- backoff(int pos, T& val): performs a user-defined backoff procedure, then loads the value held at 'pos' and returns whether or not it equals val
- DelayMarkValue(T *val): returns 'val' with a delay mark
- getInfo(T& val, int64_t& val_seqid, bool& val_isValueType, bool& val_isDelayedMarked): based on 'val', this function assigns values to the other passed variables.

Table 1: Utility Functions

2.1 Design Restrictions

Our approach requires support for the following atomic primitives: *Compare-and-Swap* (*CAS*), *Fetch and Add* (*FAA*), *Load*, and *Store*. These atomic primitives are supported by the majority of modern hardware architectures and provided in C++11 [7].

It also reserves the three least significant bits of a value for encoding state and type information.

Another constraint is that elements stored within the buffer each contain an integer for storing a sequence identifier. This sequence identifier is used to ensure that the buffer provides a first-in-first-out ordering of elements. In the presented implementation, a pointer to the data is stored in the buffer instead of the data itself. Storing the data on the buffer is possible using techniques described in [4].

2.2 Design Overview

The design of our ring buffer diffuses contention by assigning sequence numbers (seqid) to each thread to use to complete its operation. The seqid is assigned by performing a *faa* on

the head counter for the *dequeue* operation and on the tail counter for the *enqueue* operation.

After receiving a seqid, a thread examines the value held at a position determined by the seqid and acts based on the value's seqid, type, and whether or not it is marked as delayed. If the least significant bit is set to zero, its type is a pointer to a *valueType*, otherwise its type is an *emptyType*. A *valueType* is a pointer to an object containing a value that has been enqueued, and an *emptyType* indicates a value is not presently enqueued at the position. The second least significant bit indicates whether or not the value is marked as delayed. The seqid of an *emptyType* is represented by the unreserved bits of the value, and the seqid of a *valueType* is stored within the object. The *getInfo* function is used to extract this information from the value.

We make the restriction that a *valueType* can only be removed by a dequeuer that is assigned a seqid that is equal to that *valueType*'s seqid.

Due to the nature of the ring buffer, the actions or inactions of other threads may cause a thread to livelock by indefinitely reattempting its operation. We prevent this scenario from occurring by using the progress assurance scheme described in [2].

2.3 Enqueue Overview

A thread performing enqueue loads the value at the specified position and acts based on the information returned by the *getInfo* operation. Ideally, this value will be an *emptyType* that has a seqid that matches the thread's seqid and is not marked as delayed. In this case, the thread attempts to replace the value with a *valueType* that contains the thread's seqid. If the thread successfully replaces the value it returns true, otherwise the thread re-loads the value at the specified position and acts based on that value.

If the current value's seqid is less than the thread's seqid, but still an *emptyType* and not marked as delayed, the thread will perform a back-off routine to allow a delayed thread to complete its operation, and if the value has not changed, the thread will attempt to replace it.

If the current value is marked as delayed or it is a *valueType*, it will also cause the thread to perform a back-off routine, but in this case, if the value has not changed, the thread will get a new seqid and restart its operation. Similarly, if the current value's seqid is greater than the thread's seqid, this will also cause the thread to get a new seqid and start over.

If the thread successfully replaced a value, it will return, otherwise it will retry part or all of its operation.

2.4 Dequeue Overview

A thread performing dequeue loads the value at the specified position and acts based on the information returned by the *getInfo* operation. In the ideal case, this value will be a *valueType* whose seqid matches the thread's seqid. In this case the thread will attempt to replace it with an *emptyType* containing the thread's seqid plus the capacity of the buffer. If the current value was marked as delayed, the replaced value will also be marked as delayed. If the replacement was not successful, it implies that it failed because the current value became marked as delayed. The thread will re-attempt its replacement and this time it is guaranteed to be successful because we make the restriction that only the thread assigned the seqid of a value may remove that value. Since no other threads will attempt to remove it and it is already marked as delayed, the value can not be changed by another thread.

If the current value's seqid is greater than the thread's seqid, this means no value was enqueued with this seqid and the thread needs to get a new seqid and start over.

In the event of thread delay, the current value's seqid could be less than the thread's seqid or the current value maybe an *emptyType*. In this case the back-off routine is used to give the delayed thread a chance to complete its operation. If it has not made progress upon the routine's return then one of the following steps is taken to ensure progress of other threads.

- If the value is an *emptyType* with a delay mark, it is replaced by an *emptyType* that has a seqid larger than the current tail counter.
- If the value is an *emptyType*, it is replaced by the same *emptyType* described in the ideal case at the start of this section.
- If the value is a *valueType*, then it is marked as delayed.

The specific reasoning for this logic is explored in the following two sections. In short, it is designed to allow threads to skip positions without losing the first-in-first-out property of the buffer.

2.5 Enqueue Implementation

Figure 2 presents the enqueue operation as implemented in the Tervel library.

Lines 2 to 4 and 25 to 28 are part of the progress assurance scheme used to provide wait-freedom. To prevent a thread from perpetually delaying another thread, we include a call to the *check_for_announcement* function, which may cause the calling thread to help one other thread. The *Limit* class is used to detect when a thread has been delayed, calling *notDelayed* increments a counter by the passed value, and it returns false when a user-specified limit has been reached. In the event a thread determines it has been delayed, the logic starting at line 25 enables the thread to recruit other threads to complete its operation. This code is further discussed in Section 4.

After getting a seqid and position to operate on, a thread loads the value currently stored at position(Line 10) and then acts based on that value. The *readValue* function may return false in the event it is unsafe to dereference the current value(Section 4). It extracts the necessary information form the value by calling the *getInfo* function(Line 13).

First, it checks to see if its seqid has been skipped(Line 14) and if it has, the thread gets a new seqid. Then it checks to see if the value is marked as delayed and if so the thread performs a back-off routine(Line 17). If the back-off was successful, the value at the specified position has changed and a

```
1: function bool enqueue(T v)
2:     check_for_announcement();
3:     Limit progAssur;
4:     while progAssur.notDelayed(0) do
5:         if isFull() then return false;
6:         int64_t seqid = nextTail();
7:         uint64_t pos = getPos(seqid);
8:         uintptr_t val;
9:         while progAssur.notDelayed(1) do
10:            if !readValue(pos, val)) then continue;
11:            int64_t val_seqid;
12:            bool val_isValueType, val_isDelayedMarked;
13:            getInfo(val, val_seqid, val_isValueType,
val_isDelayedMarked);
14:            if val_seqid > seqid then
15:                break;
16:            if val_isDelayedMarked then
17:                if backoff(pos, val) then continue;
18:                else break;
19:            else if !val_isValueType then
20:                if val_seqid < seqid && backoff(pos, val)
then continue;
21:                uintptr_t new_value = ValueType(value,
seqid);
22:                if array_[pos].cas(val, new_value) then
23:                    return true;
24:            else if !backoff(pos, val) then break;
25:    EnqueueOp *op = new EnqueueOp(this, value);
26:    make_announcement(op);
27:    bool res = op->result();
28:    op->safe_delete();
29:    return res;
```

Figure 2: Enqueue Operation

thread will reprocess it, otherwise the thread will get a new seqid. It is safe to get a new seqid as we allow for both enqueue and dequeue operations to replace *emptyType* objects that have seqid less than expected. A dequeuer assigned a skipped seqid would simply get a new seqid.

The next check determines if the value is an *emptyType* or *valueType*(Line 19). If it is an *emptyType* with a matching seqid, the thread attempts to replace it(Line 22). If it is an *emptyType* with a seqid less than the thread's seqid, the thread performs a back-off routine before attempting to replace the value. By performing a back-off routine, it provides a slow executing thread with the opportunity to complete its operation. If the thread was successful at replacing the value it will return true, otherwise, it will get the new current value.

If the value is a *valueType*, the thread will perform a back-off routine, which returns whether or not the value at passed position has changed. If it has not changed, the thread will get a new seqid, otherwise the thread acts based on that value.

2.6 Dequeue Implementation

Figure 3 presents the dequeue operation as implemented in the Tervel library.

```
1: function bool dequeue(T &v)
2:     check_for_announcement();
3:     Limit progAssur;
4:     while progAssur.isDelayed(0) do
5:         if isEmpty() then return false;
6:         int64_t seqid = nextHead();
7:         uint64_t pos = getPos(seqid);
8:         uintptr_t val, new_value =
EmptyType(nextSeqId(seqid));
9:         while progAssur.isDelayed(1) do
10:            if !readValue(pos, val) then continue;
11:            int64_t val_seqid;
12:            bool val_isValueType, val_isDelayedMarked;
13:            getInfo(val, val_seqid, val_isValueType,
val_isDelayedMarked);
14:            if val_seqid > seqid then break;
15:            if val_isValueType then
16:                if val_seqid == seqid then
17:                    if val_isDelayedMarked then
18:                        new_value =
DelayMarkValue(new_value);
19:                    value = getValueType(val);
20:                    if !array_[pos].cas(val, new_value)
then
21:                        new_value =
DelayMarkValue(new_value);
22:                    array_[pos].cas(val, new_value);
23:                return true;
24:            else
25:                if !backoff(pos, val) then
26:                    if val_isDelayedMarked then
27:                        break;
28:                    else atomic_delay_mark(pos);
29:            else
30:                if val_isDelayedMarked then
31:                    int64_t cur_head = getHead();
32:                    int64_t temp_pos = getPos(cur_head);
33:                    cur_head += 2*capacity_;
34:                    cur_head += pos - temp_pos;
35:                    array_[pos].cas(val,
EmptyType(cur_head));
36:                else if !backoff(pos, val) &&
array_[pos].cas(val, new_value) then break;
37:    DequeueOp *op = new DequeueOp(this);
38:    make_announcement(op);
39:    bool res = op->result(value);
40:    op->safe_delete();
41:    return res;
```

Figure 3: Dequeue Operation

After getting a seqid, the thread determines the position to operate on and loads the value currently stored at that position(Line 10). The *readValue* function may return false in the event it is unsafe dereference the current value(Section 4). The thread will then extract the necessary information form the value(Line 13) and act accordingly.

First, it ensures that its seqid has not been skipped(Line 14) and if it has, the thread will get a new seqid. Next it checks if it is a *valueType* and if so compares its seqid to the current value's seqid. If the seqid matches, then the thread replaces that value with an *emptyType* containing the next seqid for that position(Line 20 or 22). If the current value was marked as delayed, the value replacing it will also have that mark. Section 3 discusses the importance of this step.

If the value is a *valueType* whose seqid is less than the thread's seqid, the back-off routine will be used to give a delayed thread a chance to complete its operation. If the back-off routine was unsuccessful, the thread will get a new seqid if the value is marked as delayed. Otherwise, the thread will place a delay mark on the value and then re-process the value(Lines 25- 28). This reprocessing is important to handle the case where the value changes just before placing the delay mark.

If the current value is an *emptyType* that is marked as delayed, the thread will attempt to replace it with an *emptyType* that has a seqid that has not been assigned(Lines 30-35). If the *emptyType* is not marked as delayed, the thread will perform the back-off routine and if the value has not changed it will try to skip the position(Line 36).

3. CORRECTNESS

In exploring the correctness of the *enqueue* and *dequeue* operations, we want to ensure that the effects of an operation occur exactly once, that the return result of the operation accurately describes how the buffer was modified, and that the operations are linearizable to one another. To support this, we show that a valid sequential history can be constructed from any valid concurrent history(*Linearizability*).

When constructing this valid sequential history, concurrent operations are ordered by their linearization points. The enqueue operation's linearization point occurs when the cas at Line 22 returns true. The dequeue operation's linearization point occurs when the cas at Line 20 or 22 returns true.

Showing that the effects of an operation occur once and that they return the correct value can be proven by examining the two algorithms. The enqueue operation returns false when the buffer is full and similarly the dequeue operation returns false when the buffer is empty. They return true after successfully replacing a value in the buffer with the effect of their operation. In this regard they can not return incorrectly.

To show that each concurrent history corresponds to a valid sequential history we have to ensure that elements are removed in a first-in-first-out order. We first show that this holds for buffers of infinite length and then we show it also holds for buffers of finite length.

For an infinite length buffer, concurrent enqueues are ordered by a monotonically increasing counter and concurrent dequeue operations are ordered by a separate but also monotonically increasing counter. We use the seqid returned from these two counters to impose a FIFO ordering on the operations. This is accomplished by initializing the positions on the buffer with *emptyType* values and upon enqueue replacing the value with a *valueType* containing the same seqid. By the nature of two monotonically increasing counters, elements must be removed in FIFO order.

If the capacity of the buffer is finite, positions on the buffer must be reused, which allows for two or more threads to attempt to apply their operations at the same position. Further, the algorithm must also correctly handle the case where an enqueuer is delayed and has not written its value before a dequeuer attempts to remove that value.

We are able to prevent these scenarios from leading to incorrect behavior by leveraging the seqid to enforce first-in-first-out ordering of actions.

From Figures 2 and 3 we extracted the following valid changes to a value at a position on the ring buffer's internal array.

- Any value may become delay marked.
- A thread will only attempt to replace a *valueType* that does not have a delay mark with an *emptyType* with a seqid equal to the seqid of the *valueType* plus the capacity of the buffer.
- A *valueType* that has a delay mark can only be replaced by an *emptyType* with a delay mark and a seqid equal to the seqid of the *valueType* plus the capacity of the buffer.
- A thread will only attempt to replace an *emptyType* that does not have a delay mark with either an *emptyType* with a higher seqid or a *valueType* with a seqid greater than or equal to its seqid.
- A thread will only attempt to replace an *emptyType* that has a delay mark with an *emptyType* with a seqid greater than the value of the current tail counter.

Using these valid transitions, we derived the following rules, that ensure FIFO behavior of the buffer.

- A value with a lower seqid can not replace a value with a higher seqid.
- An element is enqueued by replacing an *emptyType* whose seqid is less than or equal to the elements seqid.
- An element can only be removed from the buffer by a thread performing a dequeue operation that has been assigned a seqid that matches the seqid stored within the element.
- A dequeue thread can only get a new seqid if it can guarantee that an element has not and will not be enqueued with that seqid.

Together these valid transitions and rules show that our algorithms behave as expected and operate correctly. Our implementation contains numerous state checks designed to detect incorrect behavior and our test handler contains logic to detect if the values are dequeued multiple times or out of order. In the numerous tests and experiments performed by our implementation, we were unable to detect any errors or anomalousness behavior that was not a result of correctable implementation error.

4. PROGRESS ASSURANCE

This section describes how we integrated *Tervel's* progress assurance scheme [2] into our design to guard against possible scenarios of livelock and thread starvation. We used the following *Tervel* constructs explicitly within the aforementioned algorithm:

- Limit: This object encapsulates logic related to detecting when a thread has been delayed beyond a number of attempts defined by the user.
- check_for_announcement: Every $checkDelay$[1] number of operations, this function tries to help one thread, if that thread has made an announcement. After making $checkDelay * numThreads$ calls to this function, a thread will have attempted to help every other thread.
- make_announcement: This function is used to recruit threads to help the calling thread complete an operation. After making an announcement, only $checkDelay * numThreads$[2] more operations may complete before it is guaranteed for the operation described in the announcement to be completed.

To enable a thread to safely recruit other threads to help complete its operation, we implemented specialized versions of the enqueue and dequeue operations encapsulated within *EnqueueOp* and *DequeueOp* class objects. The following two sections present a brief description of these classes and the differences between them and the enqueue and dequeue operations presented in Section 2. Full implementation details are available in the documentation of Tervel [10].

4.1 Overview of Operation Record

The ring buffer's operation record uses Tervel's association model to enable a thread to temporarily place a reference to a helper descriptor object (*Helper*) at a position on the buffer. If the *Helper* object is associated with the operation record, it is replaced replaced by the result of the operation. Otherwise, it is replaced by the value that the *Helper* had replaced.

An operation record is associated with a *Helper* object if that operation record's atomic reference was changed from *nullptr* to the address of the *Helper* object. By design once this reference has been set, it can not be changed again.

If the *readValue* function loads a reference to a *Helper* object, the thread executing the function will attempt to acquire hazard pointer protection[2] on that object, which causes the object's *on_watch* function to be called. We implemented the *Helper's on_watch* function to cause the *Helper* object to be replaced by its logical value and return false. This prevents the needed to add additional logic to earlier algorithms. As a result, when the *readValue* function loads a *Helper* value, it will fail to acquire memory protection and return false.

Tervel's operation record classes require the implementation of the function *help_complete* which is called by threads recruited to help complete an operation. Our implementation of this function mirrors the implementation of the enqueue and dequeue operations, with a few key changes. First, the looping structure terminates when the operation record becomes associated with a *Helper* object. In the event the buffer is full (for enqueue) or empty (for dequeue), the operation record will become associated with a constant value indicating such.

The following sections present summaries of the design of these operation records. For complete implementation details with in-line comments, please see the latest release of the Tervel library.

4.2 Enqueue Operation Record

The enqueue operation record's *help_complete* function begins by setting its seqid to the current value of the tail counter, and then examining the value stored at the position indicated by that value. If the value is not a non-delay marked *emptyType* with a *seqid* less than or equal to its seqid, the thread will increment its seqid and try again. Otherwise, it will attempt to place a reference to a *Helper* object, and if unsuccessful it will re-examine the value at the position.

After successfully placing the reference to a *Helper* object, the *Helper's finish* function is called to remove it and replace it with its logical value. This function is also called by the *Helper's on_watch* function, which always returns false because the *finish* function removes the reference to the helper object. This design pattern removes the need for other operations to include logic to resolve *Helper* objects.

After removing the *Helper* object, the last step is to repeatedly try to update the tail counter until it is greater than or equal to seqid used to enqueue the element. This ensures that *isEmpty* does not report falsely. The loop used to implement this will terminate after a finite number of iterations, since each failed attempt implies the counter has increased in value and has gotten closer to the seqid.

To ensure that the enqueued element has the correct seqid, before beginning the operation the element's seqid is set to a negative value. Then, before the *Helper* object is removed a *cas* operation is used to change it to the seqid stored within the *Helper* object.

4.3 Dequeue Operation Record

This dequeue operation permits threads to dequeue a value without modifying the head counter. A thread assigned the seqid of a value that was dequeued in this manner will act as if the seqid was never used to enqueue a value. The dequeue operation record's *help_complete* function begins by setting its seqid to the current value of the head counter and then examining the value stored at the position indicated by that

[1] A user defined constant

[2] Hazard pointer protection is a memory protection technique used to prevent objects from being freed by one thread whilst being used by another. If memory protection was not used, then the algorithm would be forced waste memory or risk entering a corrupted state. A thread acquires hazard pointer protection on an object by writing the address of an object to a shared table and then examining the address the reference to the object was loaded from. If the value of the address has changed, the thread will often re-attempt part of its operation. If it has not changed, the thread has successfully acquired memory protection on the object [5].

value.

If the value's seqid is greater than thread's seqid, the thread will increment its seqid and examine the next position.

If the value is an *emptyType* and delay marked, it will be handled as described in Section 2.6. If it is not delay marked and it has a seqid less than the thread's seqid, the thread will attempt to change the position to an *emptyType* with the next seqid. If the thread is successful, it will increment its seqid and examine the next position, otherwise it will re-examine the position. This ensures that values will not be enqueued after we passed this position, which may affect the FIFO property.

If the value is a *valueType* with a seqid less than or equal to the thread's seqid and greater than or equal to a minimum seqid specified in the operation record, the thread will use a *Helper* object to attempt to dequeue the value, re-examining the position if it is unsuccessful. The minimum value is important to ensure the FIFO property, otherwise a value assigned to a thread that has been delayed for an extended period maybe returned. If the seqid was not within the proper range, the thread will increment its seqid and examine the next position.

After successfully placing the reference to a *Helper* object, the *Helper's finish* function is called to remove it and replace it with its logical value. This function is also called by the *Helper's on_watch* function, which always returns false because the *finish* function removes the reference to the *Helper* object. This design pattern prevents other algorithms from perceiving the *Helper* object.

After removing the *Helper* object, the last step is to repeatedly try to update the head counter until its value is greater than or equal to the seqid used to dequeue the element. This ensures that *isFull* does not report falsely. The loop used to implement this will terminate after a finite number of iterations, since each failed attempt implies the counter has increased in value and has gotten closer to the seqid.

5. WAIT-FREEDOM

To show that the *enqueue* and *dequeue* operations are wait-free we examine the function calls and the looping structures used in constructing them.

Each function called by these operations, except for the two *Tervel* operations, *check_for_announcement* and *make_announcement*, are simple utility functions that contain no loops or calls to other functions. The looping structures used in both algorithms terminate when the call to *notDelayed* returns false. The *Tervel* library specifies that this function returns false when the a counter in the *progAssur* reaches 0. This counter, which is initially set equal to a compile-time constant (*delayLimit*), is decremented by an amount specified in the call to *notDelayed*. As a result these loops terminate after *delayLimit* iterations of the inner loop, which is called at least once for each iteration of the outer loop.

If *check_for_announcement* and *make_announcement* are wait-free, then the *Enqueue* and *Dequeue* operations are also wait-free since it can be shown that they terminate after a finite number of instructions. Both of these functions may call the *help_complete* function of an operation record class. Because of this, if each operation record's *help_complete* function is lock-free then these operation's are also wait-free.

The reason the requirement is lock-free stems from how *Tervel* recruits threads to help complete delayed operations. For example, if an operation is delayed long enough, then all threads will eventually try to complete it, and no new operations can be created until it is complete. In this case, the lock-free property ensures that one of the thread's will complete the operation, causing all threads to return.

In Section 4, we discuss the design of the operation records used to help complete a delayed thread's enqueue or dequeue operation. The *help_complete* function in the two objects is a modified version of the algorithms presented in Figures 2 and 3. Specifically, we changed the terminating condition of the while loops to terminate when the operation record becomes associated with a helper object. Since these are lock-free functions and there is a finite number of new operations before all threads are executing the *help_complete* function of a particular operation record, the *help_complete* function of the *EnqueueOp* and *DequeueOp* classes must be wait-free.

6. RELATED WORKS

In this section we describe the implementation of other concurrent ring buffers capable of supporting multiple producers and consumers.

6.1 Lock-free Ring Buffer by Tsigas et al.

Tsigas et al. [11] presents a lock-free ring buffer in which threads compete to update the head and tail locations. An enqueue is performed by determining the tail of the buffer and then enqueueing an element using a *cas* operation. A dequeue is performed by determining the head of the buffer and then dequeueing the current element using a *cas* operation. This design achieves dead-lock and live-lock freedom; if a thread is unable to perform a successful *cas* operation, the thread will starve. Unlike other designs, which are capable of diffusing contention, the competitive nature of this design leads to increased contention and poor scaling.

6.2 Krizhanovsky Ring Buffer

Krizhanovsky [8] describes a ring buffer that he claims to be lock-free. It relies on the *fetch-and-add* operation to increment head and tail counters, which determine the index to perform an operation.

Each thread maintains a pair of local variables, local_head and local_tail, to store the indexes where the thread last completed a enqueue or dequeue operation. The smallest of all the threads' local_head and local_tail values is used to determine the head and tail value at which all threads have completed their operations. These values prevent an attempt to enqueue or dequeue at a location where a previously invoked thread has yet to complete it's operation.

Though the author claims it to be lock-free, if a thread fails to update its local variables, it will prevent other threads form making progress, even if the buffer is neither full or

empty. As a result, this design is actually obstruction-free because it can enter a state where all threads are waiting on a single thread to perform an action.

6.3 Intel Thread Building Blocks

Intel Thread Building Blocks (TBB) [9] provides a concurrent bounded queue, which utilizes a fine-grained locking scheme. The algorithm uses an array of micro queues to alleviate contention on individual indices. Upon starting an operation, threads are assigned a ticket value, which is used to determine the sequence of operations in each micro queue. If a thread's ticket is not expected, it will wait until the delayed threads have completed their operations.

6.4 MCAS Ring Buffer

Since there are no other known wait-free ring buffers to compare the performance of our approach against, we implemented one using a wait-free software-based Multi-Word Compare-and-Swap (MCAS) [1]. MCAS is an algorithm used to conditionally replaces the value at each address in a set of address. An MCAS operation is successful and subsequently performs this replacement if the value at each address matches an expected value for that address. To provide correctness this must appear to happen atomically, such that overlapping operations cannot read a new value at one address and then read an old value at another address.

In our MCAS-based ring buffer design, we use MCAS to move a head and tail markers along the buffer's internal array. The act of moving a marker requires the completion of six hardware compare-and-swap operations. Though this implementation is simple to implement and reason about, the number of compare-and-swap operations necessary to complete a single operation results in poor permanence and scalability.

The specific procedure to enqueue an element is as follows: after identifying the position holding the head marker, a thread enqueues an element by performing a successful MCAS operation that replaces the head marker with the value being enqueued and replaces an empty value at the next position with the head marker. If the next position holds the tail marker, it indicates that the buffer is full.

Similarly, a value is dequeued by a thread using MCAS to replace the tail marker with an empty value and to replace the value at the next position with the tail marker. If the tail marker is followed by the head marker, this indicates the buffer is empty.

6.5 Comparison

In contrast to the design presented by Tsigas et al., where threads compete to update a memory location, our design diffuses thread congestion by incrementing the head or tail value and using the pre-incremented value to determine where to apply an operation. By reducing the amount of threads attempting a *cas* operation at a given buffer location, our approach is able to reduce the amount of failed *cas* operations. This is similar to the approach presented by Krizhanovsky, however, in contrast Krizhanovsky's design, we have devised methodology to prevent threads from becoming livelocked or starved.

7. RESULTS

This section presents a series of experiments that compare the presented ring buffer design (WaitFree) to the alternative designs described in Section 6. Tests include performance results of TBB's concurrent bounded queue (TBB), Krizhanovsky's ring buffer (Krizhanovsky), Tsigas et al.'s cycle queue (Tsigas), and the MCAS-based design (MCAS). For each experiment we present the configuration of the test, the collected results, and performance analysis.

7.1 Testing Hardware

Tests were conducted on a 64-core ThinkMate RAX QS5-4410 server running the Ubuntu 12.04 LTS operating system. It is a NUMA architecture with 4 AMD Opteron 6272 CPUs, which contain 16 cores per chip with clockrate 2.1 GHz and 314 GB of shared memory. Executables were compiled with GNU's C++ compiler, g++-4.8.1 with level three optimizations and *NDEBUG* set. The algorithm implementations used for testing where the latest known versions from their respective authors.

7.2 Experimental Methodology

Our analysis examines how the number of threads and the type of operation executed by each thread impacts the total number of operations completed.

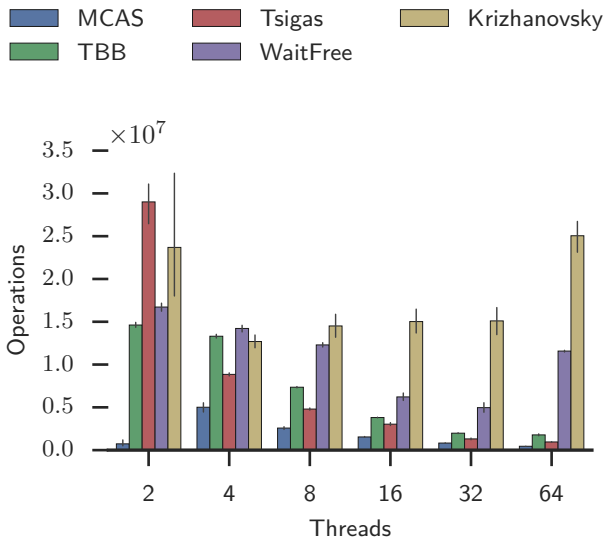
We use the *Tervel* library's synthetic application tester to simulate various use cases and configurations of the ring buffer implementations. The testing procedure consists of a main thread, which configures the testing environment and spawns the worker threads. For these tests, the queue was constructed with a fixed capacity of $32,768$ elements and then initialized by performing $16,384$ enqueues. The main thread signals the worker threads to begin execution, sleeps for 5 seconds, and then signals the end of the execution. Information gathered during the test is outputted to a log file.

The following graphs depict the total number of operations of all threads completed while the main thread was asleep. Each test configuration was executed twenty times, the standard deviation of these results is included within the graphs.

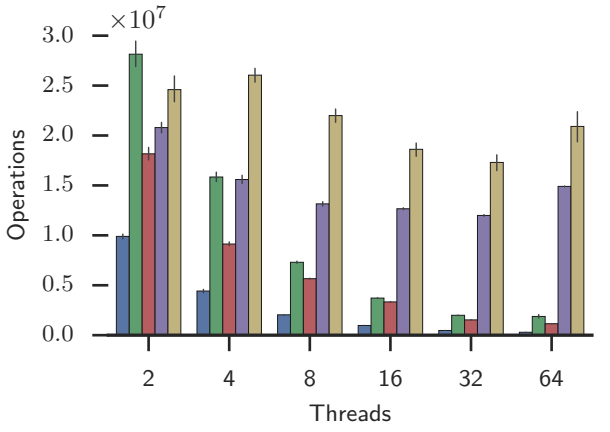
7.3 Even Enqueue and Dequeues

We first discuss experiments where the number of enqueue and dequeue operations are relatively equal. This can be accomplished in three different ways, each of which results in slightly different performance. Performance of a data structure, in this case, is the number of successful operations applied to the ring buffer during the test, unsuccessful operations occur when attempting to enqueue on a full buffer or dequeue on an empty buffer.

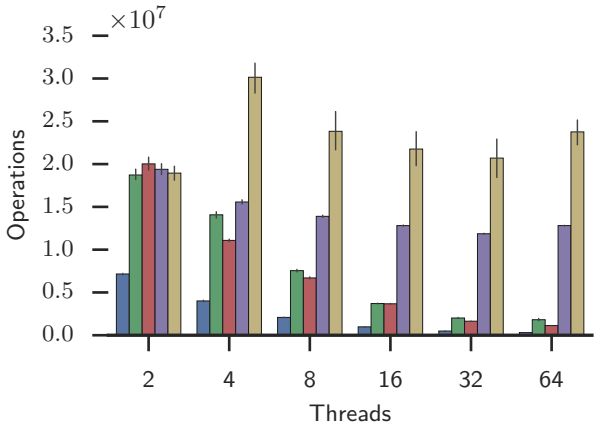
Figure 4a shows the performance of the first case, where half the threads perform enqueue and the other half perform dequeue. In this graph, the MCAS-based buffer (*MCAS*) performs significantly worse than the other designs at all thread levels. The presented wait-free ring buffer (*WaitFree*) performs better than the all but Tsigas et al.'s non-blocking buffer (*Tsigas*) at two threads and at 64 threads it outperforms all except for Krizhanovsky's buffer (*Krizhanovsky*).



(a) 50% of Threads Enqueue and 50% Dequeue



(b) Threads Enqueue then Dequeue



(c) Threads pick randomly, with even chance

Figure 4: Equal Enqueue and Dequeue

Developers using the Krizhanovsky buffer accept the risk that under certain conditions threads executing operations on the buffer may become blocked indefinably. The logic we include in our design to prevent this, however, it is the likely reason for this performance difference.

Figure 4b shows the performance in the case where threads alternate between enqueue and dequeue operations. In this figure, Krizhanovsky, and TBB perform significantly better at low thread levels, while the WaitFree version performs slightly worse. However, at high thread levels, the performance differences are similar to the previous figure.

Figure 4c shows the performance of the last case, where threads select randomly between the two operations, with each option having equal probability. The cost of calculating the random value creates slightly more work between calls to buffer operations. At low thread levels, each design (except for MCAS) performs equal well, however, at higher thread levels, the Krizhanovsky and WaitFree buffers still perform the best.

On average, at 64 threads our design performs 3825.84% more operations than the MCAS, 1117.52% more than Tsigas, 631.52% more than TBB, but 43.07% less than the Krizhanovsky buffer. Overall even enqueue and dequeue test scenarios, our design performs 1329.61% more operations than the MCAS, 339.51% more than Tsigas, 209.90% more than TBB, but 34.47% less than the Krizhanovsky buffer.

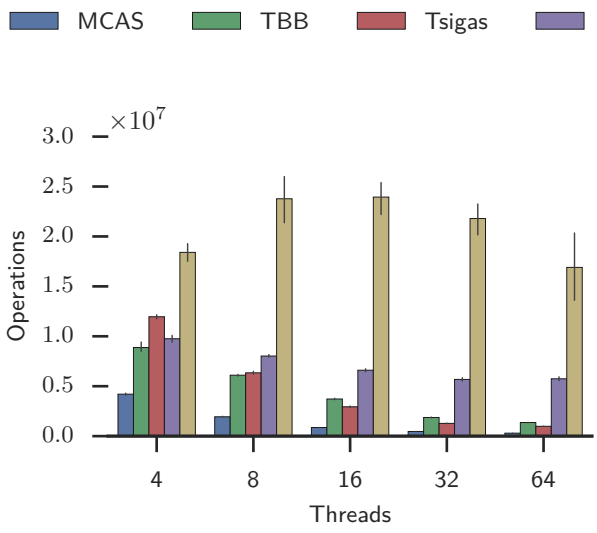
7.4 Higher Enqueue Rate

Figure 5 shows the performance of the algorithms when 75% of the threads perform enqueue and 25% of the threads perform dequeue. Figure 5a graphs the number of successful queue operations, Figure 5b graphs the number of failed queue operations, and Figure 5c graphs the total number of queue operations. Failed queue operations are the result of enqueue operations returning that the queue is full. Unlike other designs, threads executing on *Krizhanovsky* buffer blocks until they are able to apply their operations, as such they do not return failure.

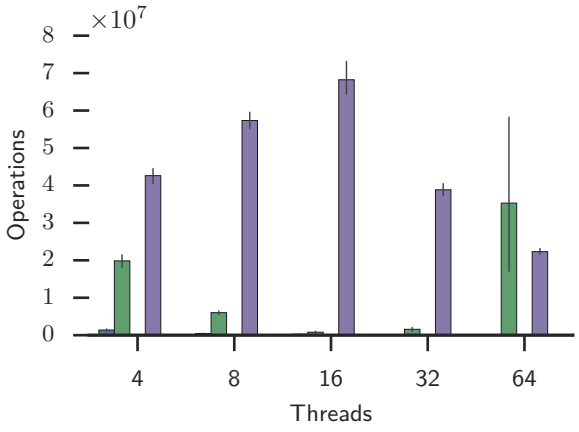
If we only presented the total number of operations completed by each threads, then our results would not accurately reflect the work being completed. For example, *TBB* performs more operations than either *Krizhanovsky* or *WaitFree*, but nearly all of them are failed operations. Similarly about half of the *WaitFree* operations are failed. Having a significant number of failed operations indicates that the way the data structure is being used is not ideal. In this example, there is a three to one ratio of enqueue to dequeue operations and because of this, the queue becomes full rather quickly.

On average, at 64 threads our design performs 1835.45% more successful operations than the MCAS, 483.34% more than Tsigas, 321.58% more than TBB, and 66.02% less than the Krizhanovsky buffer. Overall test in this scenario, our design performs 813.27% more successful operations than the MCAS, 192.28% more than Tsigas, 128.95% more than TBB, and 64.12% less than the Krizhanovsky buffer.

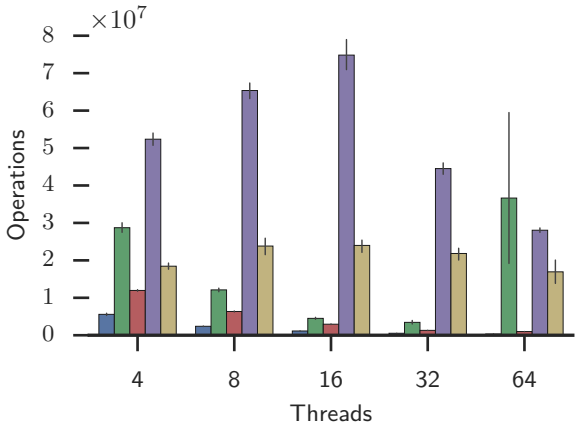
7.5 Higher Dequeue Rate



(a) Successful Operations

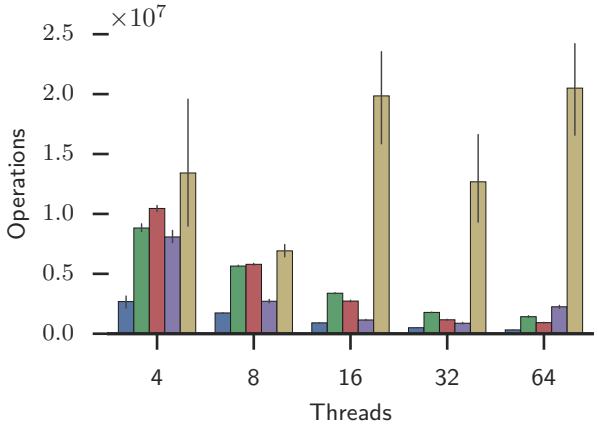


(b) Failed Operations

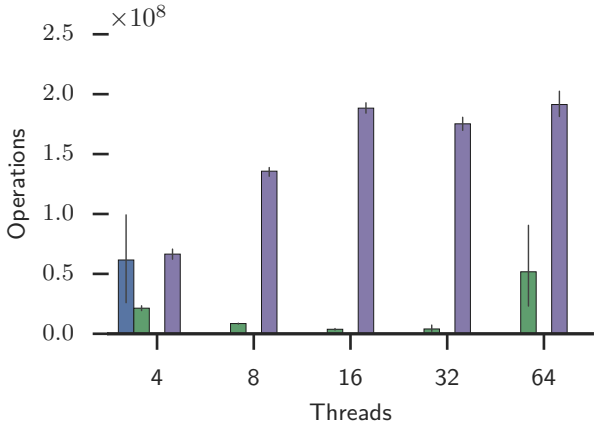


(c) Total Operations

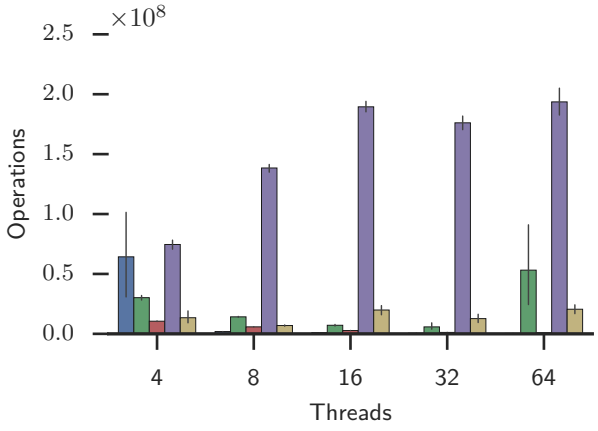
Figure 5: Use Case: High Enqueue



(a) Successful Operations

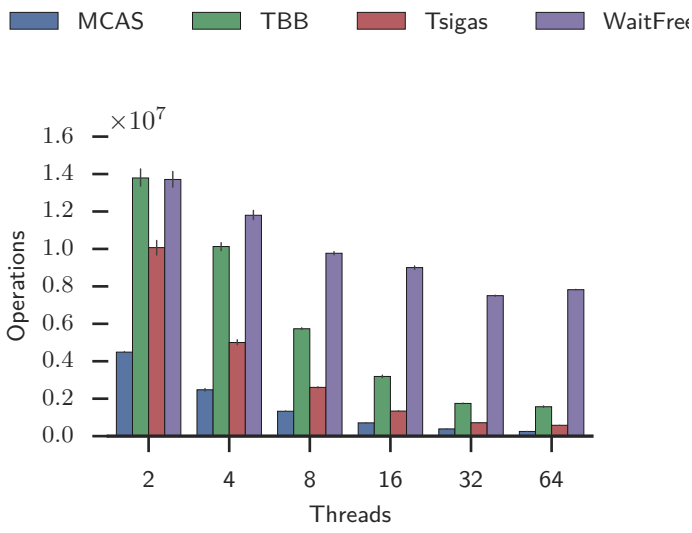


(b) Failed Operations



(c) Total Operations

Figure 6: Use Case: High Dequeue

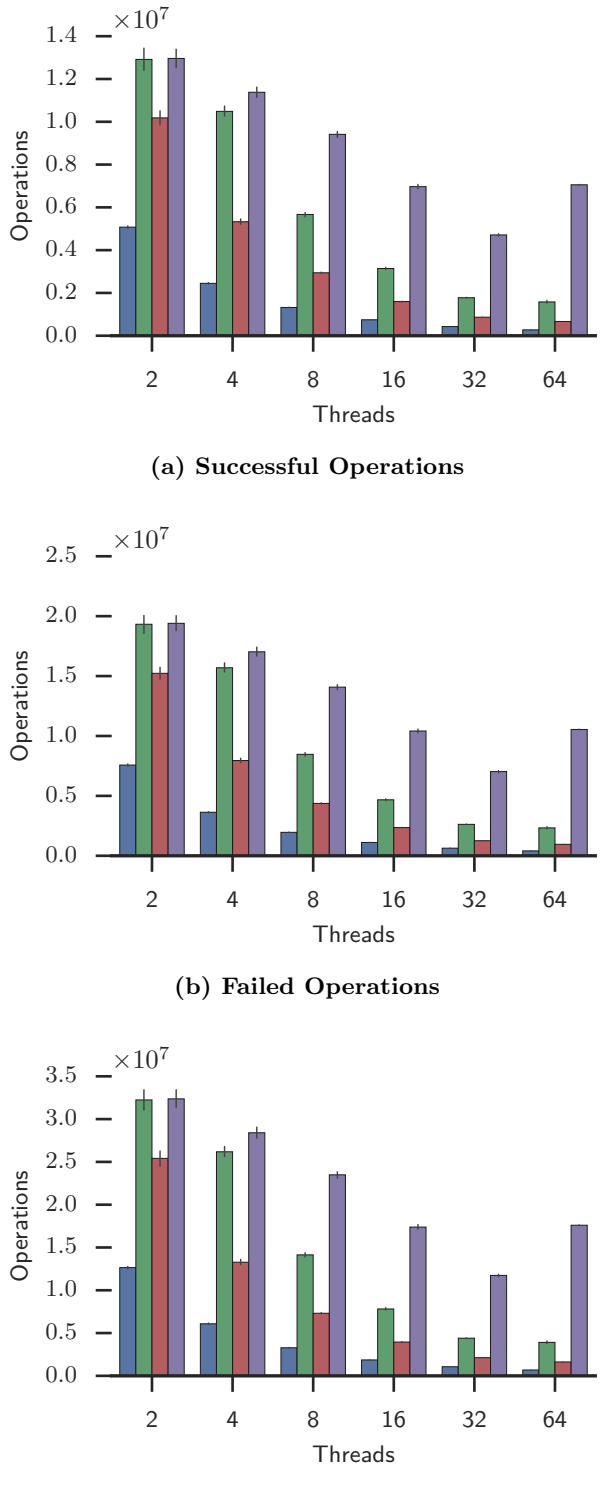


(a) Successful Operations

(b) Failed Operations

(c) Total Operations

Figure 7: Use Case: 80% Enqueue, 20% Dequeue



(a) Successful Operations

(b) Failed Operations

(c) Total Operations

Figure 8: Use Case: 20% Enqueue, 80% Dequeue

Figure 6 shows the performance of the algorithms when 75% of threads perform dequeue and 25% of threads perform enqueue. Like in the previous section, Figure 6a graphs the number of successful queue operations, Figure 6b graphs the number of failed queue operations, and Figure 6c graphs the total number of queue operations. Failed queue operations are the result of dequeue operations returning that the queue is empty. As stated earlier, threads executing in the Krizhanovsky buffer block until they are able to apply their operation, as such they do not return failure.

In this example, the amount of failed operations is much higher than in the previous example. This indicates that the dequeue operation takes less time to complete then the enqueue operation, which causes the queue to become empty faster than the queue in the previous example to become full.

On average, at 64 threads our design performs 605.11% more successful operations than the MCAS, 143.28% more than Tsigas, 58.18% more than TBB, but performs 89.03% less than the Krizhanovsky buffer. Overall test in this scenario, our design performs 192.85% more operations than the MCAS, however it performs on average 3.05% less than Tsigas, 23.81% less than TBB, and 75.39% less than the Krizhanovsky buffer.

7.6 Further Analysis

We further analyze the performance by examining the performance of the data structures when each thread is capable of performing both enqueue and dequeue operations. Figure 7 presents the results from tests that have threads select enqueue with a 80% chance and dequeue with a 20% chance and Figure 8 presents results when there is a 20% chance of enqueue and 80% chance of dequeue. Section 7.3 presents the results when there is an even ratio of enqueue and dequeue operations. The Krizhanovsky buffer was not included in these results as it would become deadlocked when the buffer became empty and all threads are trying to dequeue or when the buffer became full and all threads are trying to enqueue.

Comparing these graphs to one another and to Figure 4c, we see that performance behavior between the different use cases is very similar. On average, at 64 threads our design performs 2746.50% more successful operations than MCAS, 372.47% more than Tsigas, and 1105.065% more than TBB. Overall test in this scenario, our design performs 943.05% more successful operations than the MCAS, 378.02% more than Tsigas, and 122.78% more than TBB.

8. CONCLUSION

This paper presents a wait-free ring-buffer suitable for parallel modification by many threads. We defined the linearization point of each operation and used it to argue that the design is linearizable and correct. We have demonstrated that our buffer provides a first-in-first-out ordering on elements and that its API is expressive enough for use in a concurrent environment.

The presented design introduces methodology to relieve thread contention by combining atomic operations, sequence counters, and strategic bitmarking. Though our approach is not the first design to use sequence counters, it is the only design that is able to do so without the risk of livelock and thread starvation. We maintain the FIFO ordering of the ring buffer through the use of these sequence counters. Our design is supported by a strategic bitmarking scheme to correct scenarios that would otherwise invalidate the FIFO property. Lastly, we integrated a progress assurance scheme to guarantee that each thread completes its operation in a finite number of steps, thus achieving wait-freedom.

Our performance analysis showed that our design performs competitively in a variety of uses cases. Though it does not provide significant overall performance improvements, it provides stronger safety properties then other known buffers. Because of this, we believe that our design is practical for a variety of use cases that require strict safety properties and guarantees of progress.

9. ACKNOWLEDGMENT

This work is funded by National Science Foundation grants NSF ACI-1440530 and NSF CCF-1218100.

10. REFERENCES

[1] S. Feldman, P. LaBorde, and D. Dechev. A wait-free multi-word compare-and-swap operation. *International Journal of Parallel Programming*, pages 1–25, 2014.

[2] S. Feldman, P. LaBorde, and D. Dechev. Tervel: A unification of descriptor-based techniques for non-blocking programming. In *Proceedings of the 15th IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XV)*, Samos, Greece, July 2015.

[3] S. Feldman, C. Valera-Leon, and D. Dechev. An efficient wait-free vector. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1–1, 2015.

[4] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Distributed Computing*, pages 265–279. Springer, 2002.

[5] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[6] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[7] ISO/IEC 14882 Standard for Programming Language C++. *Programming languages: C++*. American National Standards Institute, September 2011.

[8] A. Krizhanovsky. Lock-free multi-producer multi-consumer queue on ring buffer. *Linux Journal*, 2013(228):4, 2013.

[9] C. Pheatt. Intel threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, Apr. 2008.

[10] C. S. E. Scalable and S. S. Lab. Tervel: A framework for implementing wait-free algorithms, 2015.

[11] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 134–143. ACM, 2001.

ABOUT THE AUTHORS:

Steven Feldman received his MS degrees in computer science from the University of Central Florida, in 2013. His research interests include concurrency, inter-thread helping techniques, and progress conditions, which has led to the development of several wait-free algorithms.

Damian Dechev is an assistant professor in the EECS Department of the University of Central Florida and the founder of the Computer Software Engineering-Scalable and Secure Systems Lab, UCF. He specializes in the design of scalable multiprocessor algorithms and has applied them to real-time embedded space systems at NASA JPL and HPCdata-intensive applications at Sandia National Labs. His research has been supported by grants from the US National Science Foundation(NSF), Sandia National Laboratories, and the Department of Energy.