Viewpoints

Mehrdad Sabetzadeh
Simula Research Laboratory
Oslo, Norway
mehrdad@simula.no

Anthony Finkelstein
University College London
London, UK
a.finkelstein@cs.ucl.ac.uk

Michael Goedicke
University of Duisburg-Essen, Campus Essen
Essen, Germany
michael.goedicke@s3.uni-due.de

November 8, 2009

Abstract

The construction of any sizable software system involves many agents, each with their own perspective of the system being built. *Viewpoints* provide a framework for guiding and managing development in a multiple-perspective setting, where a variety of agents with different areas of concern collaborate towards building a system. In this article, we explain the main concepts and techniques underlying viewpoint-based development and illustrate them using a number of examples.

Keywords. Perspectives, Separation of Concerns, Integration, Inconsistency Management, Merging, Parallel Composition, Weaving.

1 Introduction

Large-scale software development is necessarily a collaborative effort, involving multiple agents (also called, participants or actors) with different perspectives of the system they are trying to develop. Individual perspectives are partial descriptions of the overall system, arising in response to the specific responsibilities or roles assigned to the agents. These responsibilities or roles may be organizationally-defined, be governed by the physical distribution of agents across development sites, follow some pre-specified structuring of the system at hand, or reflect the agents' knowledge, experience, and descriptive capabilities.

Inevitably, the perspectives of those involved in the development process overlap, interact or compete with one another, giving rise to a need for coordination. The relationships between the perspectives, however, may be far from obvious because the agents may use different vocabularies and representation notations to express their concerns. Further, since development may be done concurrently, different perspectives may be at different stages of elaboration and hence subject to different development strategies.

The problem of how to guide and manage development in this setting - multiple agents, diverse knowledge and expertise, heterogeneous representation notations, differing development strategies - is known as the "multiple

perspectives problem". In this article, we introduce *viewpoints* – a conceptual framework for addressing the multiple perspectives problem.

Intuitively, a viewpoint is an independent and locally-managed object encapsulating a particular perspective. The main feature of viewpoint-based development is its recognition of perspectives and the relationships between them as first-class artifacts, so that perspectives can be directly defined by users, and the relationships between them can be specified, modified, and analyzed explicitly.

Viewpoints have often been studied as part of Requirements Engineering. Although the multiple-perspective problem is particularly acute during the requirements gathering stage where a diverse group of stakeholers with different goals and needs are involved, the problem is not restricted to it. In fact, viewpoints arise with increasing frequency in software design and implementation as well, due to such factors as evolution, functional variability, globalization, etc.

It is not our intention to exhaustively examine all ramifications of viewpoints in software engineering. Our main goal is instead to describe the key ideas that motivate the use of viewpoints and to outline the integration activities that go hand in hand with them. To achieve our goal, we are going to draw on a number of examples throughout the article. While

we have tried to ensure reasonable breadth in our choice of examples, we acknowledge that these examples are not reflective of the full range of the applications of viewpoints reported in the literature. General references for further details about viewpoints and their broader set of applications are provided later in the article (see Section 7).

We use software models as the primary context for our examples. This enables us to build on the growing familiarity of the general software engineering community with modelling languages such as the UML (Unified Modeling Language, 2003), hence alleviating the need to provide extensive background on the notations used.

The remainder of this article is structured as follows:

We begin in Section 2 with an example of a multiple-perspective setting. In Section 3, we introduce viewpoints as a vehicle for capturing perspectives. In Section 4, we discuss how different viewpoints can be related using explicit relationships. We continue in Section 5 with reviewing the core design principles underpinning viewpoints. In Section 6, we concentrate on the most essential activity in viewpoint-based development, called *integration*, and highlight the different facets of the problem. We conclude the article in Section 7 with a summary and references for further reading.

2 An Example of Multiple Perspectives

To illustrate multiple perspectives, we use a simple example capturing some basic aspects of a Hospital Information System (HIS). A HIS is a a specialized kind of information system designed to manage the clinical and administrative functions of a hospital.

An HIS has to meet the needs of a variety of stakeholders. This includes people at the management levels of the hospital, medical staff, technicians, administrators, and so on. These people clearly have different areas of concern, and hence different perspectives. The perspectives are usually expressed in different ways, and capturing them often necessitates the use of different representation notations, chosen to be particularly appropriate to the descriptions provided by each stakeholder.

Figure 1 shows a few representative perspectives in an HIS. The formal specification of perspectives as structured models is seldom done directly by the stakeholders, but rather by the requirements and design analysts. For simplicity, we ignore this technicality and directly associate the models to the stakeholders.

The perspectives in Figure 1 describe an HIS along different dimensions and at different levels of abstraction. The hospital head focuses on the high-level objectives of the system and expresses her view as a goal model in

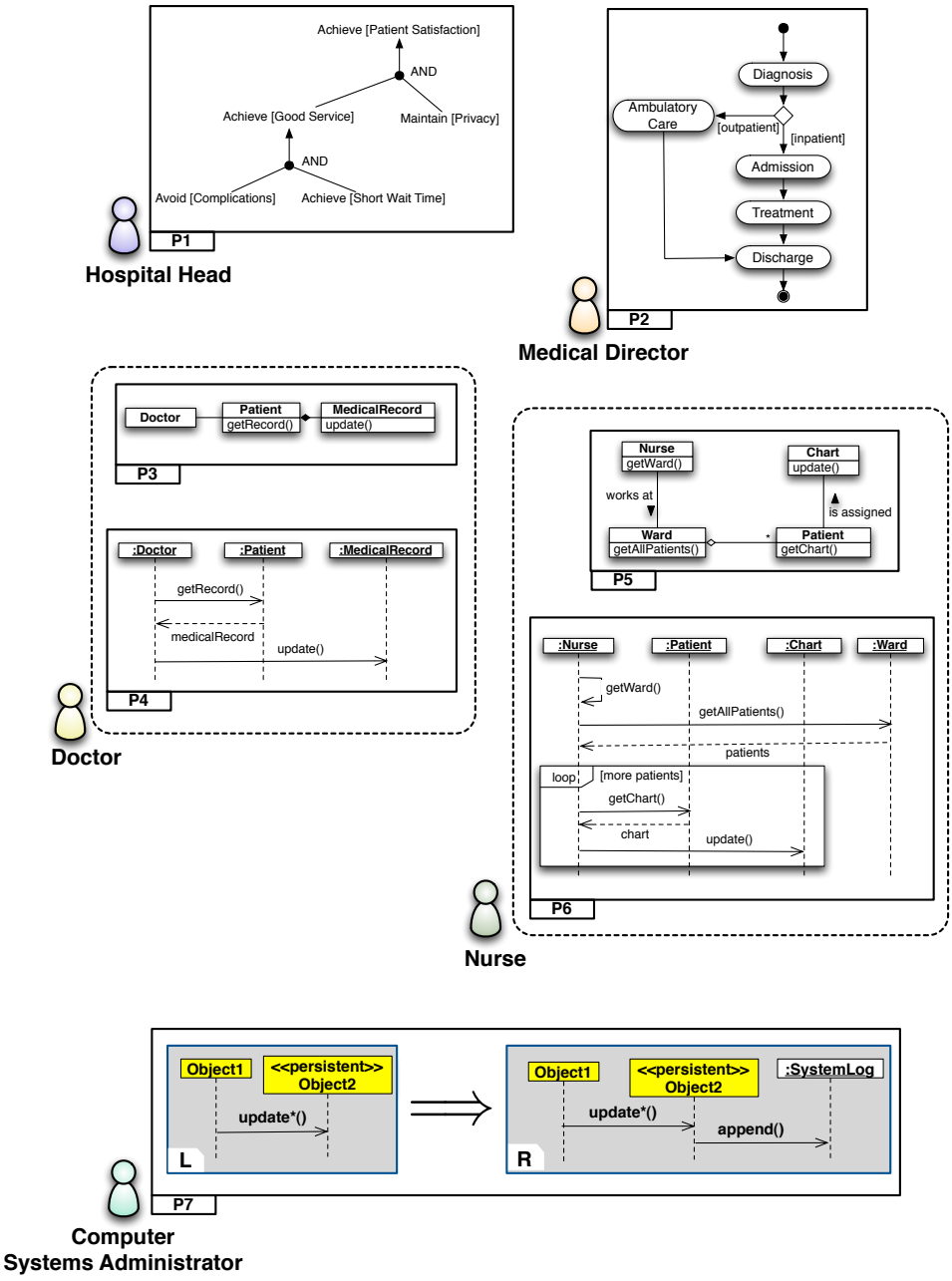

Figure 1: Example Perspectives in a Hospital Information System

KAOS (van Lamsweerde, 2009) notation. In her model (P1 in the figure), achieving patient satisfaction is declared as a top goal, and a decomposition of this goal into more concrete and lower-level goals is provided.

The medical director uses a UML activity diagram (P2) to capture the normal workflow for patient care at the hospital. The process begins with diagnosis, after which a patient is classified as either an inpatient or an outpatient. The patient then gets the proper care based on this classification and is subsequently discharged.

The doctor and the nurse each contribute two perspectives, a UML class diagram and a UML sequence diagram. The class diagrams (P3 and P5) capture the concepts and associations relevant to each stakeholder, and the sequence diagrams (P4 and P6) describe usage scenarios. The usage scenario for the doctor (P4) concerns the routine visiting of a patient and updating her medical record. The usage scenario given by the nurse (P6) captures the daily check up conducted in a hospital ward, where a nurse evaluates each patient individually, and notes her observations in the patient's medical chart.

Finally, the computer systems administrator, who is in charge of ensuring the integrity and performance of the HIS, expresses a requirement that any update made to the persistent data of the system should be logged in a file,

so that problems can be tracked in case of a system failure. This requirement is modelled as a rewrite rule for sequence diagrams (P7). If the left hand side of the rule (denote L) matches a fragment of a sequence diagram, then that fragment is rewritten with the right hand side of the rule (denoted R). The left hand side would be deemed a match if update*() (i.e. a method whose name begins with the "update" prefix) is called on a persistent data object. If a match is found, the sequence diagram in question will be modified as prescribed by R (the right hand side). That is, SystemLog.append() is called after the update to append an entry to the system log.

The models in our example relate to one another in many ways. Some of these relationships are readily identifiable. For example, the objects in P4 are instances of the classes declared in P3, and the messages in P4 are occurrences of the class methods in P3. The relationship between P5 and P6 is similar. Another obvious pair of relationships are P7,P4 and P7,P6. The match between the left hand side of P7 and P4 is $\{$Object1 $\rightarrow$:Doctor, Object2 $\rightarrow$:MedicalRecord, update*() $\rightarrow$ update()$\}$. The P7,P6 relationship is defined similarly.

There are also relationships whose existence is almost certain, but whose details cannot be established with full certainty because of terminological and structural differences between the perspectives. For example, an anal-

ysis of P3 and P5 would hint at some possible overlaps between the two perspectives: the Patient class in P3 is likely to be the same as that in P5; MedicalRecord in P3 is likely to correspond to Chart in P5; and the aggregation between Patient and MedicalRecord in P3 may be an alternative representation of the assigned to link between Patient and Chart in P5. Unless these correspondences are explicitly validated with the stakeholders, one can never be entirely sure about their correctness. For example, it may turn out that the doctor's notion of Patient encompasses both inpatients and outpatients, while the nurse may mean only inpatients when she refers to Patient in her models. Similarly, MedicalRecord and Chart may be distinct entities.

There are several other relationships between the perspectives that are less obvious:

- Since P3 and P5 most likely have overlaps, P4 and P6 will be related as well, in the sense that they can access and modify shared objects. For example, it is possible for a doctor and a nurse to both want to update an individual patient's chart simultaneously. Hence, P4 and P6 may need to synchronize on accesses to shared objects.

- P2 implicitly characterizes the different states that a patient goes through at the hospital. Hence P2 would be related to the design of the Patient class in P3 and P5.

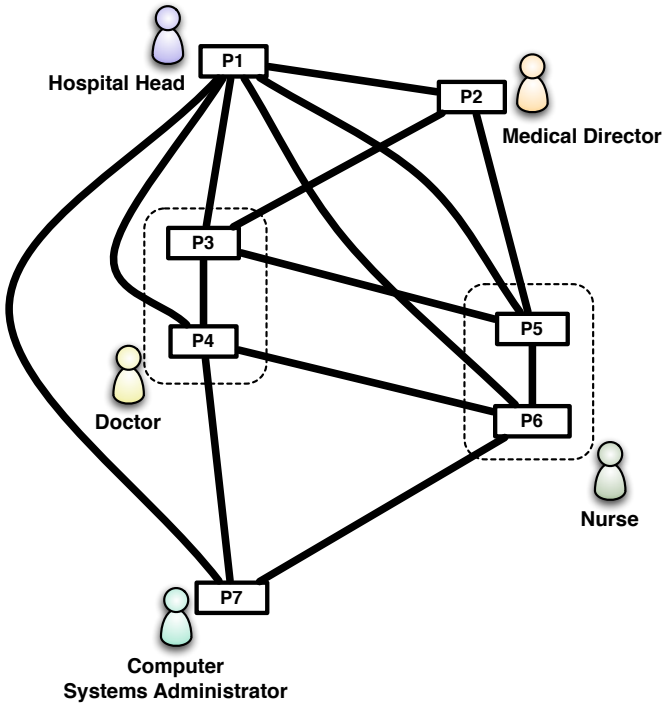

Figure 2: Relationships between the Perspectives of Figure 1

- Finally, at a higher level of abstraction, all P2–P7 contribute to the satisfaction of the system's main objectives, and are, directly or indirectly, traceable to the goals expressed P1.

In Figure 2, we have shown a bird's eye view of the perspectives in our example along with the relationships that capture the known or hypothesized interlocking constraints on the structure and behaviour of the perspectives. The use of perspectives made it a lot easier to ensure that all the stakeholders are properly represented, and much less likely to miss information that is

critical to the success of the system. At the same time, the flexibility to have perspectives results in several inter-related models that need to be maintained independently and may be inconsistent with one another.

Effective use of perspectives in system development requires a systematic framework for management and integration of perspectives and resolving their inconsistencies. Viewpoints, as we describe in the subsequent sections, provide the foundation for such a framework.

3 Viewpoints

The basic building block for organizing and supporting multiple perspectives is a *viewpoint*. A viewpoint can be thought of as a combination of the idea of an agent, role, or knowledge source and the idea of a perspective. In software terms, a viewpoint is a loosely coupled, locally managed object which encapsulates a perspective about the system and domain, specified in a particular representation notation. Each viewpoint is composed of the following components, called slots:

- a **domain** delineating the part of the "world" the viewpoint is concerned with;
- a **representation style** defining the notation used by the specification

slot (below);

- a **specification** expressing the perspective of interest, represented in the viewpoint's style

Additionally, a viewpoint may contain knowledge of the process of its design. This knowledge is captured using the following slots:

- a **work plan** describing the process by which the specification can be built;

- a **work record** giving an account of the history and current state of development

As an example, let us consider the specification provided by the medical director (P2) in the motivating example of the previous Section. In Figure 3, we have shown the complete viewpoint for this specification. The representation style of the viewpoint is UML activity diagrams, and the domain of concern is patient care. The specification is the medical director's current knowledge about the procedure for processing patients at the hospital, expressed as a UML activity diagram. The work plan explains how to build an activity diagram and how, and in what circumstances, to check consistency with the other viewpoints. The work record gives the current state of the specification and an account of its development in terms of the work plan.

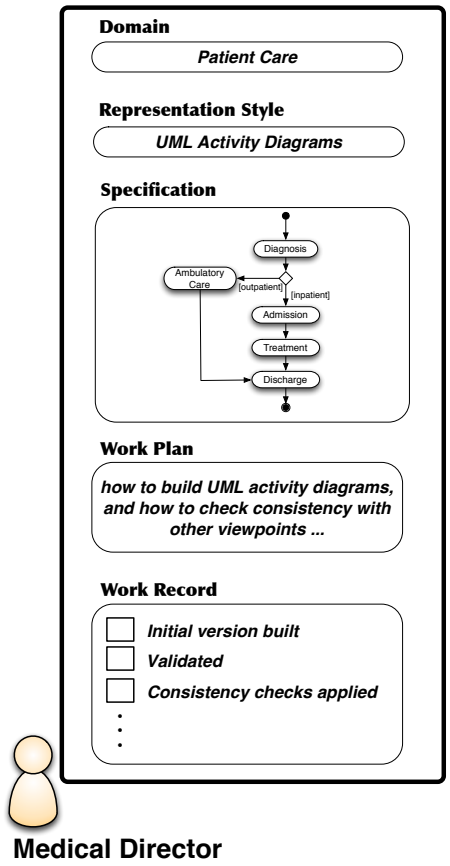

Figure 3: Complete Viewpoint for Perspective P2 of Figure 1

4 Relationships Between Viewpoints

The key feature that distinguishes viewpoint-based development from conventional development methods is that viewpoints are built and maintained independently of one another. This means that viewpoints are not just projections of a common underlying specification, but rather distinct and separately-evolving artifacts with potentially different vocabularies and frames

of reference. As a result, one cannot merely rely on conventions (e.g., name or identifier equivalence between elements of different viewpoints) to give the desired relationships between the viewpoints. Instead, one needs to explicitly specify these relationships. Doing so is a critical prerequisite for any meaningful integration of the viewpoints, and for ensuring that the impact of changes made to the viewpoints can be properly analyzed, scoped, and propagated.

A relationship typically refers to an (explicit) mapping between the contents or interfaces of the specification slots in different viewpoints. In a broader sense, a relationship may also encompass the "organizational structure" of the development team (i.e., the owners of the viewpoints). In this article, we shall deal only with the former type of relationships, i.e., those between specifications.

There is no general rule as to what constitutes an inter-viewpoint relationship. This depends on a variety of factors, most importantly the semantics of the viewpoints involved, the stage of development the viewpoints are in, and the degree of detail one wishes to capture in a relationship.

In the remainder of this section, we illustrate some common relationships between specifications expressed as models. There is no restriction on the number of specifications an individual relationship can connect, but usually,

relationships are defined pairwise, i.e., they are between two specifications. Our example relationships, shown in Figure 4, are all pairwise.

In Figure 4(a), we depict the relationship (denoted R) between the perspectives contributed by the doctor (previously shown in Figure 1). The relationship specifies how the objects in the sequence diagram (V_2) correspond to the classes in the class diagram (V_1) through "instance of" mappings, and how the interaction messages in V_2 correspond to the methods in V_1 through "occurrence of" mappings. If the two diagrams are assumed to be views on a monolithic model of the system, this relationship is derivable from the underlying model and can hence be left implicit. But as we stated earlier, such an assumption is typically not made in viewpoint-based approaches, and as a result, the relationships are always specified explicitly.

In Figure 4(b), the specifications involved are in different abstraction layers. V_1 is the extends–implements fragment of the meta-model for Java class diagrams, and V_2 – an instance class diagram conforming to the meta-model. The relationship between these specifications is a type function mapping each element of V_2 to an element of V_1. The relationship respects the structure of the specifications, in the sense that if it maps an edge e of V_2 to an edge u of V_1, it will also map the endpoints of e to those of u. Such preservation of structure ensures that the edges in the V_2 are well-typed.

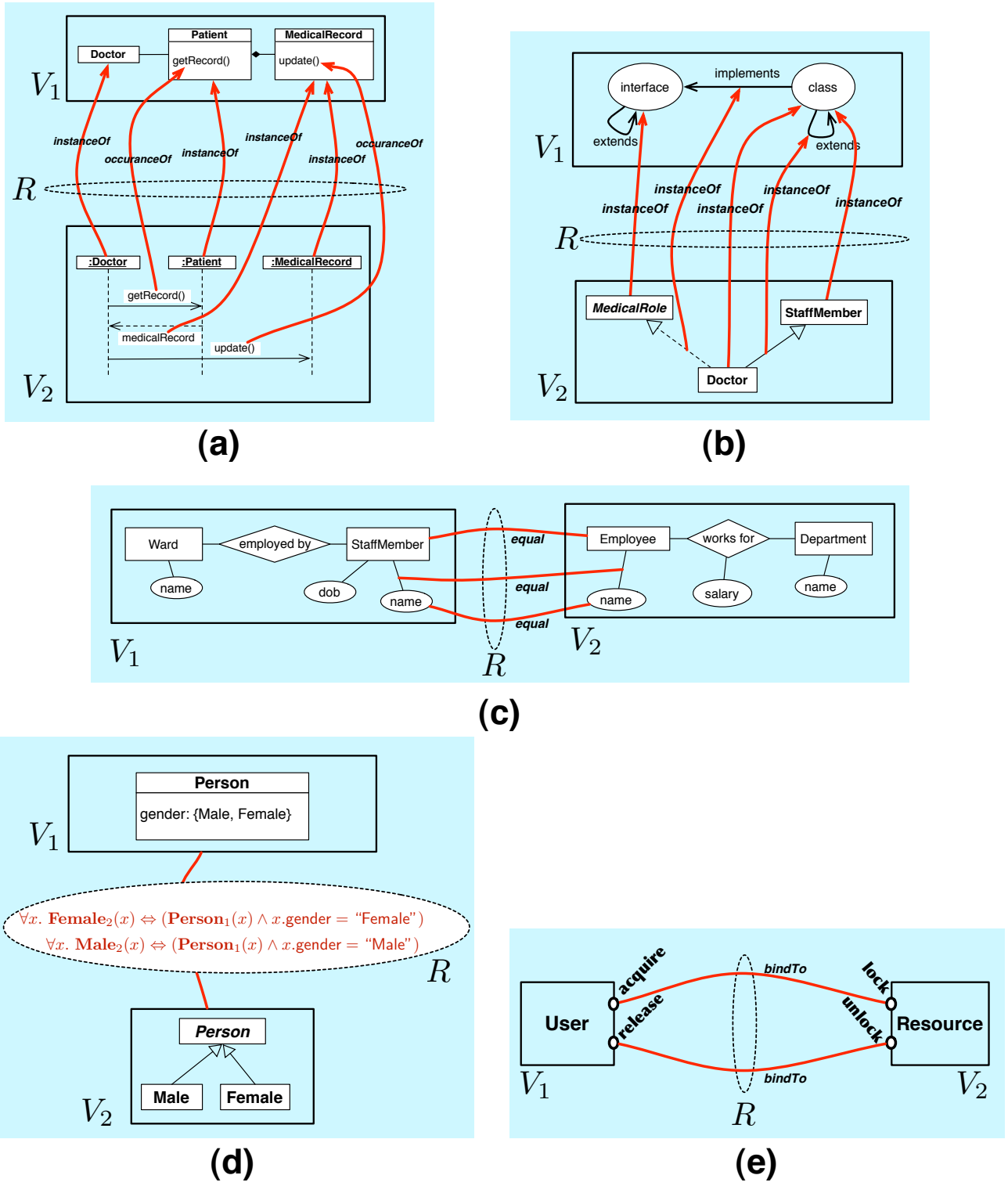

Figure 4: Different Types of Relationships between Viewpoints

Our next example, in Figure 4(c), shows the relationship between two perspectives on the data schema of the hospital's payroll subsystem. Both perspectives are described as entity-relationship diagrams. The vocabularies used by the perspectives differ, but there are conceptual overlaps between the perspectives. These overlaps are specified by a mapping that equates the corresponding elements. The relationship in Figure 4(c) states that the entity referred to as StaffMember in V_1 is the same as Employee in V_2. Further the relationship declares the name attribute of these two entities (as well as the edges linking name to the respective entities) as being equivalent. For a problem of this size, the relationship may be created by hand, but larger problems require automation, usually achieved through heuristic techniques for finding concept matches. These techniques are almost always inexact, i.e., they may yield matches that are incorrect, or they may miss correct matches. Despite this, the techniques can be of significant assistance to a human expert for exploring the relationships between different stakeholder vocabularies.

The relationships we talked about so far were expressed as collections of tuples of the form (e_1, e_2) where e_1 and e_2 are elements of different specifications. While tuple-based techniques are very common and widely used, they are not the only possible way for specifying inter-viewpoint relation-

ships. For example, a relationship may be expressed declaratively using logical formulas. An example is given below:

In Figure 4(d), V_1 and V_2 are two alternative ways of expressing the concept of a person along with their gender. V_1 models gender as an attribute of the Person class, whereas V_2 models gender through sub-classing. This structural discrepancy between the viewpoints is more conveniently expressed using first-order logic formulas than tuples of mapped elements. The formulas shown in the figure specify how Female and Male in V_1 relate to Person in V_2. To avoid ambiguity, predicate names in the formulas are subscripted with a number denoting the model of origin. For example, Person_1 refers to Person in V_1.

As a final illustration, we consider, in Figure 4(e), the relationship between viewpoints capturing different components of a system. In contrast to all our previous examples, the relationship in Figure 4(e) is not between the contents of the viewpoints involved, but rather between their interfaces (viewpoint contents are not shown). Here, V_1 denotes a user process that accesses shared resources during its execution, and V_2 is one such resource. The relationship between V_1 and V_2 describes how the two components are supposed to bind together, hence establishing a communication channel between them. More specifically, the relationship requires that the acquire (resp.

release) action from V_1 should synchronize with lock (resp. unlock) from V_2.

5 Principles Underlying Viewpoints

The use of viewpoints in software engineering is motivated primarily by the *separation of concerns* principle – the observation that, for a large and complex system, it is more effective to build several partial specifications that focus on individual concerns, rather than to attempt to construct a single monolithic specification for the whole system.

Separation of concerns may be carried out along different dimensions. For example, in the motivating problem described in Section 2, the areas of concern were defined along the organizational roles of the involved participants (hospital head, doctors, nurses, etc.). A finer-grained separation could result in developing independent viewpoints for every relevant participant, e.g., individual doctors and nurses.

The factors influencing the choice of dimensions along which viewpoints are built are many. They include, among others, the nature of the problem at hand, organizational considerations, the degree of involvement of the stakeholders, component architecture of the system, the development teams' dynamics, and policies for future maintenance of the system.

Separation of concerns naturally leads to the related principle of *het-*

erogeneity of representation. Viewpoints reinforce this principle by allowing participants to choose the notation in which they want to describe their perspectives. The choice of notation for each viewpoint is recorded explicitly in the representation style slot of that viewpoint.

The next major principle underlying viewpoints is *decentralization* – the idea that the knowledge within each viewpoint is maintained and manipulated locally. Decentralization moves away from any notion of a monolithic system specification that can be managed globally. Such flexibility becomes particularly crucial for development teams that are spread across multiple geographical sites. Decentralization makes it possible for the members of these teams to function independently without having to constantly coordinate their work with one another.

A consequence of having multiple viewpoints in a project is the inevitable inconsistencies that arise between them. The approach taken by viewpoints is that of *living with inconsistency.* In other words, there is no requirement that a set of viewpoints should always be consistent. This stance towards inconsistencies draws on two important observations: Firstly, immediate resolution of inconsistencies can be highly intrusive, particularly in early stages of development, e.g., the requirements elicitation phase, where ambiguities and conflicts tend to occur too frequently. Having to address every

inconsistency as soon as it arises can adversely affect productivity.

Secondly, maintaining consistency at all times can lead to premature commitment to decisions that have not yet been sufficiently analyzed. For example, resolving an inconsistency identified at the requirements stage may require information from the detailed design stage which might not have yet even started. Hence, any attempt to fix the inconsistency early on may fail to amend the problem, and can in fact even worsen it.

Obviously, living with inconsistency should not be viewed as suggesting living with defects in the final system. What this principle does promote is the flexibility to deal with each inconsistency at the right time.

6 Viewpoint Integration

Viewpoint integration is primarily concerned with ensuring that a set of viewpoints fit together in a consistent way while retaining their original structure. In this sense, viewpoint integration is often synonymous with *inconsistency management*. Despite the general desire to maintain viewpoints as independent objects, there are situations where a collection of viewpoints need to be combined into a single larger specification, e.g., to gain a unified perspective, to build a complete blueprint for the system, or to better explore the interactions among viewpoints. To this end, viewpoint integration

may further encompass three other activities, called *merging*, *parallel composition* and *weaving*. We briefly explain and exemplify each of these four facets of integration.

6.1 Inconsistency Management

Consistency is usually defined using inter-viewpoint rules expressing the desired constraints that must hold between viewpoints. As an example, a consistency rule that one might want to express over the viewpoints in Figure 4(a) is that each message in the sequence diagram has a corresponding method in the class diagram. This is captured by the following rule:

$$\begin{aligned} \varphi \quad = \quad & \forall\, msg, obj, cls \cdot \mathsf{Target}(msg, obj) \wedge R(obj, cls) \Longrightarrow \\ & \exists\, mtd \cdot \mathsf{MethodOf}(mtd, cls) \wedge R(msg, mtd). \end{aligned}$$

In the above formula, $R(x, y)$ holds if x and y are related by an inter-viewpoint relationship (see the relationship R in Figure 4(a)); $\mathsf{Target}(x, y)$ holds if message x is invoked on object y in the sequence diagram; and $\mathsf{MethodOf}(x, y)$ holds if x is a method of class y in the class diagram. In this example, the viewpoints satisfy φ; but if we, say, remove the getRecord() method from the Patient class in V_1, the rule would no longer hold.

Inconsistency management is the process of handling the potential plethora of consistency rules, an example of which was shown above, and dealing with

the viewpoints when the rules fail to hold. Inconsistency management involves several steps:

- Consistency *checking*, focusing on monitoring, detecting, and reporting of consistency rule violations.
- Inconsistency *classification*, focusing on identifying the kinds of inconsistencies detected. Inconsistencies may be classified according to their causes, or according to pre-defined kinds prescribed by developers.
- Inconsistency *handling*, focusing on acting in the presence of inconsistencies. For example, when an inconsistency is detected, the appropriate response may be to ignore, tolerate, resolve, or circumvent it.
- Inconsistency *measurement*, focusing on calculating the impact of inconsistencies on different aspects of development, and prioritizing the inconsistencies according to the severity of their impact. The actions taken to handle an inconsistency typically depend on the results of inconsistency measurement.

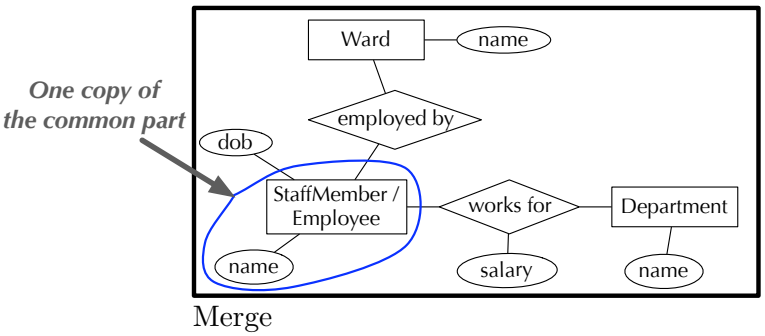

Figure 5: Merge Result for the Viewpoints of Figure 4(c)

6.2 Merging of Viewpoints

The main goal of merging is unification of overlaps between viewpoints. More precisely, given a set of viewpoints, merge combines the viewpoints together in a way that only one copy of the overlapping parts is included in the result. This property is known as *non-redundancy*.

To illustrate merge, consider the two viewpoints in Figure 4(c) and the relationship that specifies their overlaps. Figure 5 shows the merge of these viewpoints (with respect to the relationship between them). This merge is redundancy-free because it has only one copy of the common parts. This ensures that database instances created over the merged schema will not have data redundancies.

Non-redundancy is only the most basic requirement for merge. Viewpoint merging is often subject to additional correctness criteria. The most

notable of these criteria are:

- *Completeness*: Merge should not lose information, i.e., it should represent all the source viewpoints completely.

- *Minimality*: Merge should not introduce information that is not present in or implied by the source viewpoints.

- *Semantic Preservation*: Merge should support the expression and preservation of semantic properties. For example, if viewpoints are expressed as state machines, one may want to preserve their temporal behaviours to ensure that the merge properly captures the intended meaning of the source viewpoints.

- *Totality*: Merge should be well-defined for *any* set of viewpoints, whether or not they are consistent. This property is of particular importance if one wants to tolerate inconsistency between the source viewpoints.

These additional criteria are not universal to all model merging problems. For example, completeness and minimality, in the strong sense described above, may be undesirable if viewpoint merging involves conflict resolution as well, in which case the final merge can potentially add or delete information. Semantic preservation may be undesirable when one wants to induce a

design drift or perform an abstraction during merge. Such manipulations are usually not semantics-preserving. And, totality may be undesirable when the source viewpoints are expected to seamlessly fit together. In such a case, the source viewpoints should be made consistent before they are merged.

6.3 Parallel Composition of Behavioural Viewpoints

Parallel composition refers to the process of assembling a set of behavioural viewpoints that capture different components of a system, and verifying that the result fulfills the properties of interest.

In contrast to merge, the inter-viewpoint relationships built for parallel composition are between the interfaces of the viewpoints, and not between their internal contents. As such, the line of sight of the relationships into the viewpoints is limited to the interfaces that the viewpoints expose to the outside world. A second difference between parallel composition and merge is that, in parallel composition, multiple copies of the same viewpoint may be participating to denote the fact that the system has multiple components of the same type.

Figure 6 shows a simple component diagram, expressed in the visual modelling notation of the Darwin architectural language (Magee *et al.*, 1995). The diagram describes the interconnections between three concur-

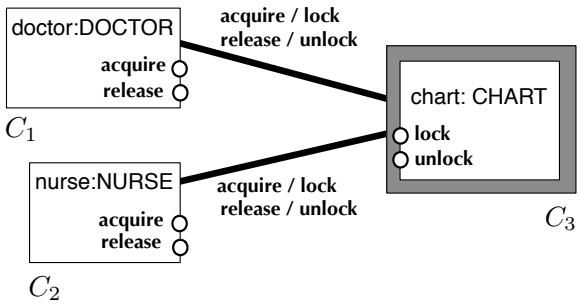

Figure 6: Interconnections between Concurrent Processes

rent processes in the hospital system: C_1, the process taken by a doctor to add a prescription to a patient's chart; C_2, the process taken by a nurse to update a patient's chart after a checkup; and C_3, the process representing a patient's chart as a resource. The semantics of the relationships between the processes in Figure 6 is the same as that of the relationship R in Figure 4(e), discussed earlier.

The behaviours of C_1–C_3 are shown in Figures 7(a)–(c), respectively. The notation used to describe these behaviours is Labelled Transition Systems (LTSs) (Milner, 1989). To make sure that patients' charts are protected against concurrent changes, accesses to C_3 by C_1 and C_2 need to be mutually exclusive.

The parallel composition of C_1–C_3 is shown in Figure 7(d). We have used the LTS Analyzer (LTSA) tool (Magee & Kramer, 2006) for expressing and computing the parallel composition. In Figure 7(e), we show the speci-

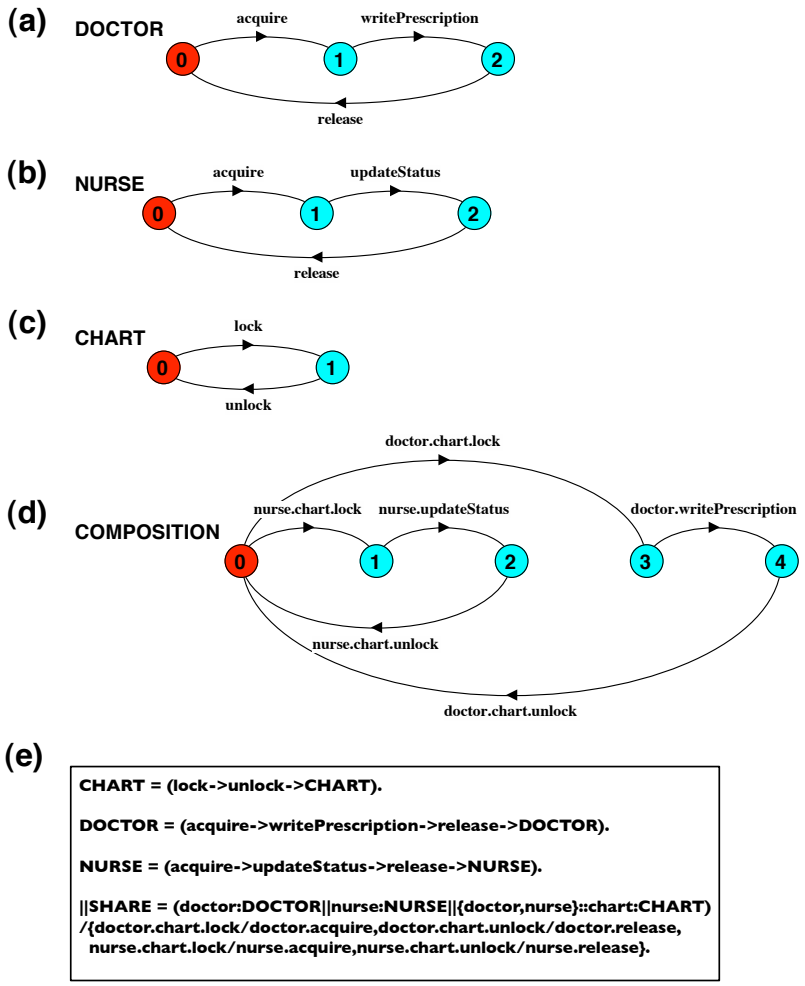

Figure 7: Specification of Processes and Their Parallel Composition

fication of the processes and their parallel composition in the input language of the LTSA tool.

As seen from Figure 7(d), an update made by the doctor is preceded by doctor.chart.lock (i.e., doctor's request to acquire access to the chart), and

followed by doctor.chart.unlock (i.e., doctor's request to release the chart). The same is true for the nurse: her update is preceded by nurse.chart.lock and followed by nurse.chart.unlock. By computing a parallel composition like this, one can verify that the updates are mutually exclusive.

6.4 Weaving of Aspectual Viewpoints

Weaving is the predominant notion of integration in Aspect-Oriented Software Development (AOSD) (Filman *et al.*, 2004). AOSD is an attempt to provide better functional decomposition in complex systems. This can in part be achieved by means of encapsulating different concerns into distinct system components. However, some concerns defy encapsulation as they cut across many parts of the system. A classic example of a cross-cutting concern is logging which affects all logged activities in a system. The perspective of the Computer Systems Administrator (P7) in Figure 1 illustrated this concern for sequence diagrams.

Given a set of cross-cutting concerns and a base system, weaving is the process of incorporating the concerns into the base system. If no base system is provided, weaving would refer to the process of incorporating the given concerns into one another. Weaving operators may be implemented in various ways depending on the nature of the base system and the concerns

involved, and whether weaving is performed statically (at compile time) or dynamically (at runtime).

Aspect-oriented languages usually come equipped with built-in constructs for defining the weaving operator. For example, aspect-oriented programming languages provide *pointcut* constructs by which programmers specify where and when additional code (i.e., an aspect) should be executed in place of or in addition to an already-defined behaviour (i.e., the base program). In aspect-oriented modelling, weaving is usually defined by patterns, to be chosen either manually or automatically using pattern matching techniques. Similar to other integration activities, weaving may result in undesirable side effects. Thus, automated analysis techniques may be required to ensure that the result of weaving satisfies the desired correctness properties.

As an example, Figure 8 shows the result of weaving P7 in Figure 1 into P4 and P6 in the same figure. For this problem, weaving is performed statically, and the weaving operator can be most conveniently implemented using graph rewriting.

7 Conclusion and Further Reading

In this article, we gave a brief introduction to viewpoints as a vehicle for addressing the multiple perspectives problem. We discussed the basic con-

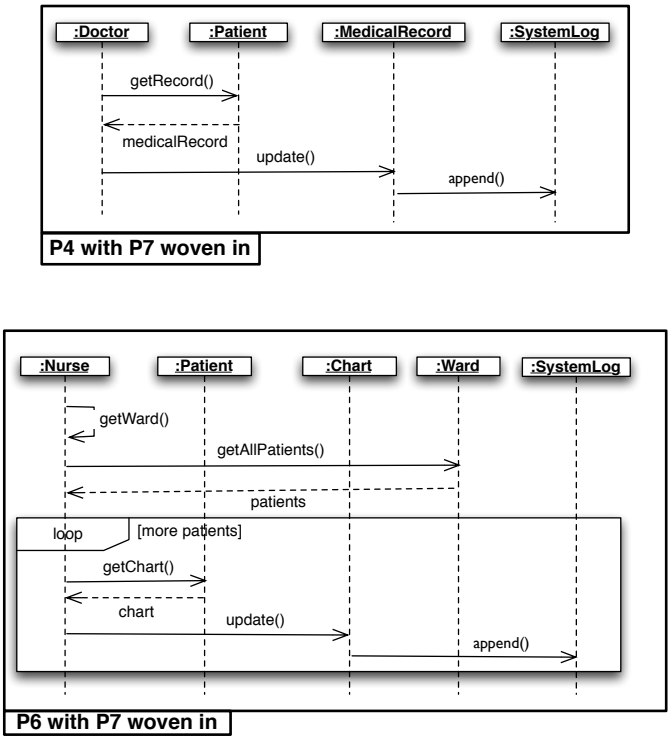

Figure 8: Weaving of P7 (Figure 1) into P4 and P6 (same figure)

cepts involved in building software systems as a collection of inter-related viewpoints, and explained the core integration activities associated with this kind of development.

The principles upon which viewpoints are founded continue to be valuable and relevant in many areas of software engineering today. In particular, several major challenges faced in the areas of global software engineering (Herbsleb, 2007), and model-based development (France & Rumpe, 2007) are intimately related to the problems that have been studied under the

umbrella of viewpoints for the past two decades. The thread that ties these areas to viewpoints is the pervasive need to manipulate systems made up of many tangled artifacts, with potential inconsistencies in their design and usage. This is indeed what viewpoints research has been aimed at all along.

In the light of the arguments presented in this article, viewpoint concepts seem to be naturally applicable for expressing many properties and structures in software development processes. Thus, the question arises why no production-quality tools have been developed to date, incorporating all the core viewpoint principles. We can see a number of tools with support for multiple development views, e.g. UML-based tools; however, these tools do not implement viewpoints in the sense we described here.

There are inherent properties to viewpoints that complicate their implementation. In particular, implementing the various operators for matching partial structures and creating and maintaining connections between viewpoints is a challenging task. This is mainly due to the complexity of the graph algorithms involved in the operators, and also to the potential existence of structural and terminological mismatches between viewpoints, which makes establishing precise inter-viewpoint relationships difficult.

In addition, it appears that the decentralized nature of viewpoints is not fully compatible with the existing project management practices. Par-

ticularly, for project management purposes, one would often like to define milestones and demand progress reports from the analysts and developers. Distributing the work across multiple viewpoints means that a distributed consensus has to be reached if something like a milestone is required, but arriving at a consensus in a distributed setting is not always straight-forward.

Another property of distributed systems applies here as well: the difficulty to build a global state. In the case of viewpoints, the notion of an overarching document containing all specification information in a linear form is lacking. Of course, the local view expressed by a viewpoint and the connections to other viewpoints is a perfect basis for a hypertext document, but this does not automatically ensure that the viewpoints fit together properly and that they are consistent with one another. Developing a more comprehensive and usable suite of tools based on viewpoints requires first addressing the challenges discussed above. These challenges map out a vision and an agenda for future research on viewpoints.

Numerous papers are available on viewpoints for further reading. Providing links to all these papers is not possible here due to space limitations. Below, we outline the expository and technical references used as the basis for this article. These references together provide an extensive body of bibliographic information on viewpoints.

Specification of viewpoints and their relationships. Several frameworks for specifying and inter-relating viewpoints in requirements engineering are surveyed in (Darke & Shanks, 1996). A more up to date bibliography of viewpoints in requirements engineering is maintained by J. Leite at `http://www.requirementsviewpoints.blogspot.com/`.

To read more about the applications of viewpoints in the broader context of software engineering, one can refer to (Spanoudakis *et al.*, 1996) where a diverse agenda on viewpoints is provided.

The detailed anatomy of viewpoints and inter-viewpoint relationships, as described in this article, is based on work on the *ViewPoints* framework (Finkelstein *et al.*, 1992; Nuseibeh *et al.*, 1994; Nuseibeh *et al.*, 2003). This framework is largely attributed as having set forth the principal ideas of viewpoint-based development as they exist today.

Inconsistency management. Several inconsistency management approaches have been proposed, in general based on the success of the *ViewPoints* framework. The main questions in this work centre on appropriate notations for expressing consistency rules, and automated support for monitoring, diagnosis, and resolution of inconsistencies. A direct successor to the *ViewPoints* framework is xlinkit – a lightweight application service that provides rule-based consistency checking and diagnostics generation for distributed web

content. This tool has been described in (Nentwich *et al.*, 2003).

An overview of inconsistency management activities for viewpoints is given in (Nuseibeh *et al.*, 2000). Our discussion in Section 6.1 was based on this reference. A detailed survey of existing inconsistency management techniques is available in (Spanoudakis & Zisman, 2001).

Viewpoint merging. Several papers study merging of partial viewpoints in specific domains including database schemata, requirements models, state machines, and web ontologies. A survey of recent approaches to viewpoint merging is available in (Sabetzadeh, 2008). The description in Section 6.2 was based on this reference.

Parallel composition. Parallel composition is a well-studied notion for combining the behaviours of distributed system components, and has been formally characterized for a variety of behavioural formalisms. Further detail on analysis tools for concurrent distributed systems is available in (Magee & Kramer, 2006).

Aspect weaving. Aspect-oriented concepts were originally coined for programming languages, e.g. see AspectJ (Kiczales *et al.*, 2001). Recently, these concepts have been adapted to software models and applied to software requirements and design viewpoints. The treatment of cross-cutting

viewpoints in Section 6.4 is similar to the approach of (Whittle *et al.*, 2007). For further references on aspect-oriented modelling and aspect weaving, see (France *et al.*, 2007; Morin *et al.*, 2009).

Acknowledgments. The description of viewpoints in this article is based on the ideas developed in the *ViewPoints* framework (Finkelstein *et al.*, 1992), to which many colleagues have contributed over the years. In particular, we wish to acknowledge Bashar Nuseibeh and Jeff Kramer for their significant contributions. We thank Steve Easterbrook, Shiva Nejati, and Marsha Chechik for providing valuable comments on earlier versions of this work. The first author acknowledges financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC), and is grateful to University College London for hosting a visit during which this article was shaped.

References

Darke, P., & Shanks, G. 1996. Stakeholder viewpoints in requirements definition: A framework for understanding viewpoint development approaches. *Requirements Engineering*, **1**(2), 88–105.

Filman, R., Elrad, T., Clarke, S., & Aksit, M. 2004. *Aspect-Oriented Soft-*

ware Development. Reading, USA: Addison-Wesley.

Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., & Goedicke, M. 1992. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering*, **2**(1), 31–58.

France, R., & Rumpe, B. 2007. Model-Driven Development of Complex Software: A Research Roadmap. *Pages 37–55 of: Future of Software Engineering Track of the 29th International Conference on Software Engineering*.

France, R., Fleurey, F., Reddy, R., Baudry, B., & Ghosh, S. 2007. Providing Support for Model Composition in Metamodels. *Pages 253–266 of: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*.

Herbsleb, J. 2007. Global Software Engineering: The Future of Socio-Technical Coordination. *Pages 188–198 of: Future of Software Engineering Track of the 29th International Conference on Software Engineering*.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold,

W. 2001. An Overview of AspectJ. *Pages 327–353 of: Proceedings of the 15th European Conference on Object-Oriented Programming.*

Magee, J., & Kramer, J. 2006. *Concurrency: State models and Java Programming: 2nd Edition.* Wiley.

Magee, J., Dulay, N., Eisenbach, S., & Kramer, J. 1995. Specifying Distributed Software Architectures. *Pages 137–153 of: Proceedings of the 5th European Software Engineering Conference.*

Milner, R. 1989. *Communication and Concurrency.* New York, USA: Prentice-Hall.

Morin, B., Barais, O., Nain, G., & Jézéquel, J. 2009. Taming Dynamically Adaptive Systems using Models and Aspects. *Pages 122–132 of: Proceedings of the 31st International Conference on Software Engineering.*

Nentwich, C., Emmerich, W., Finkelstein, A., & Ellmer, E. 2003. Flexible Consistency Checking. *ACM Transactions on Software Engineering and Methodology*, **12**(1), 28–63.

Nuseibeh, B., Kramer, J., & Finkelstein, A. 1994. A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *IEEE Transactions on Software Engineering*, **20**(10), 760–773.

Nuseibeh, B., Easterbrook, S., & Russo, A. 2000. Leveraging Inconsistency in Software Development. *IEEE Computer*, **33**(4), 24–29.

Nuseibeh, B., Kramer, J., & Finkelstein, A. 2003. ViewPoints: Meaningful Relationships Are Difficult! *Pages 676–683 of: Proceedings of the 25th International Conference on Software Engineering.*

Sabetzadeh, M. 2008. *Merging and Consistency Checking of Distributed Models.* Ph.D. thesis, University of Toronto.

Spanoudakis, G., & Zisman, A. 2001. *Handbook of Software Engineering and Knowledge Engineering.* World scientific. Chap. Inconsistency management in software engineering: Survey and open research issues, pages 329–380.

Spanoudakis, G., Finkelstein, A., & Emmerich, W. (eds). 1996. *Viewpoints 96: International Workshop on Multiple Perspectives in Software Development.*

Unified Modeling Language. 2003. *The Unified Modeling Language.* `http://www.uml.org/`.

van Lamsweerde, A. 2009. *Requirements Engineering: From System Goals to UML Models to Software Specifications.* Wiley.

Whittle, J., Moreira, A., Araújo, J., Jayaraman, P., Elkhodary, A., & Rabbi, R. 2007. An Expressive Aspect Composition Language for UML State Diagrams. *Pages 514–528 of: Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems.*