Consistency checking for multiple view software architectures*

Pascal Fradet, Daniel Le Métayer, and Michaël Périn

IRISA/INRIA
Campus de Beaulieu, 35042 Rennes Cedex, France
{fradet,lemetayer,mperin}@irisa.fr

Abstract. Consistency is a major issue that must be properly addressed when considering multiple view architectures. In this paper, we provide a formal definition of views expressed graphically using diagrams with multiplicities and propose a simple algorithm to check the consistency of diagrams. We also put forward a simple language of constraints to express more precise (intra-view and inter-view) consistency requirements. We sketch a complete decision procedure to decide whether diagrams satisfy a given constraint expressed in this language. Our framework is illustrated with excerpts of a case study: the specification of the architecture of a train control system.

1 Multiple views: significance and problems

Because software must satisfy a variety of requirements of different natures, most development methods or notations include a notion of software *view*. For example, RM-ODP descriptions [2] include five viewpoints (enterprise, information, computational, engineering, and technology); UML [15] is defined in terms of a collection of diagrams such as static structure, statechart, component and deployment diagrams. However, these methods do not have a mathematical basis. As a consequence, some of their notations may be interpreted in different ways by different people. Another drawback of this lack of formalization is the fact that it is impossible to reason about the consistency of the views. As we will see later, it is quite easy to define views that are in fact inconsistent (either internally or because of contradictory constraints on their relationships).

The relevance of the notion of view for software architectures has already been advocated in [12] but, to the best of our knowledge, no existing formally based architecture description language really supports the notion of multiple views. The key technical issue raised by multiple view architectures is consistency. In order to be able to reason about consistency, it is necessary to provide a formal definition of the views and their relationships. This is precisely the objective of this paper, with the additional challenge to design an automatic method for multiple view consistency checking.

We see a software architecture as a specification of the global organization of software involving components and connections between them. Components and

*To be published in Proc. of ESEC/FSE'99, LNCS series © Springer-Verlag connections are associated with attributes whose nature depends on the property of interest. Most papers published in the area of software architectures have focused on attributes related to communication and synchronization [1, 11, 4]. In this paper, we focus on structural properties and we consider only simple, type-like attributes. This context is sufficient both to express interesting properties and to raise non-trivial consistency issues that must be properly addressed before considering more sophisticated attributes.

In order to avoid introducing a new language, we start in Section 2 from diagrams, a graphical representation which is reminiscent of notations such as UML. We justify the use of diagrams in the context of software architectures and define their semantics as sets of graphs. This notation does not prevent us from defining (either internally or mutually) inconsistent views. A simple algorithm is proposed to check the consistency of diagrams. Section 3 illustrates the framework with a brief account of a case study conducted in collaboration with the Signaal company. The goal of the study was the specification of the architecture of a train control system involving a variety of functional and non functional requirements. We provide excerpts of some of the views that turned out to be relevant in this context. Diagrams themselves express in a natural way structural constraints on the views but it is desirable to be able to express more sophisticated constraints on views. In section 4, we put forward a simple language of constraints to express more precise (intra-view and inter-view) consistency requirements. We sketch a complete decision procedure to decide whether diagrams satisfy the constraints expressed in this language. Section 5 compares our approach with related work and suggests some avenues for further research.

2 Diagrams: notation, semantics and consistency

In this section, we provide a formal definition of diagrams and study their consistency. Their use to describe software architecture views is illustrated in the next section.

2.1 Graphical notation

As mentioned in [1], software architectures have been used by developers for a long time, but in a very informal way, just as "box and line drawings". Typically, such drawings cannot be used to detect potential inconsistencies of the architecture in a systematic way or to enforce the conformance of the constructed software with respect to the architecture. It is the case however that some conventions have emerged for the graphical representation of software views in the area of development methods. These graphical notations were not proposed originally to describe software architectures in the accepted sense of the word (e.g. they do not include provision for defining the behavior of connectors or component interfaces). However, they are rich enough to be considered as a good basis for the specification of the overall organization of a software. So, rather than crafting a language of our own, we decided to use a graphical notation already familiar to many developers. A diagram is a collection of nodes and edges with multiplicities, in the spirit of UML class diagrams. Figure 1 provides an example of a diagram.

$$\begin{tikzcd}
& A \arrow[d, "\alpha"', "1"' near start, "*"' near end] & \\
& B \arrow[dl, "\beta"', "1" near start, "1"' near end] & \\
C \arrow[rr, "\delta"', "1"' near start, "1"' near end] & & D \arrow[ul, "\gamma"', "1..*"' near start, "1"' near end]
\end{tikzcd}$$

Fig. 1. A diagram

Nodes (A, B, C, D in Figure 1) are connected via directed edges. An edge bears a type ($\alpha, \beta, \gamma, \delta$ in Figure 1) and an interval over the natural numbers (called a *multiplicity*) at each end. The intervals $[i, j]$, $[i, i]$, $[i, \infty[$ and $[0, \infty[$ are noted $i..j$, i, $i..*$, and $*$ respectively.

Such a diagram represents in fact a class of graphs (called instance graphs). Each node and edge in a diagram may represent several nodes and edges in an instance graph. The role of multiplicities is to impose constraints on the number of connected instance nodes. More precisely, an edge $A \xrightarrow{I\ \alpha\ J} B$ specifies that each instance a of A is connected via outcoming α-edges to j_a ($j_a \in J$) instances of B, whereas each instance b of B is connected via incoming α-edges to i_b ($i_b \in I$) instances of A. For example, the graphs of Figure 2 are valid instances of the diagram of Figure 1:

$$\begin{tikzcd}
& & & a_1 \arrow[dll, "\alpha"'] \arrow[drr, "\alpha"] & & & \\
& b_1 \arrow[dl, "\beta"'] & & & & b_2 \arrow[dl, "\beta"'] & \\
c_1 \arrow[rr, "\delta"'] & & d_1 \arrow[ul, "\gamma"'] & & c_2 \arrow[rr, "\delta"'] & & d_2 \arrow[ul, "\gamma"']
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
& & & a_1 \arrow[dll, "\alpha"'] \arrow[drr, "\alpha"] & & & \\
& b_1 \arrow[dl, "\beta"'] & & & & b_2 \arrow[dl, "\beta"'] & \\
c_1 \arrow[rrrrrr, bend right, "\delta"'] & & d_1 \arrow[ul, "\gamma"'] & & c_2 \arrow[ll, "\delta"'] & & d_2 \arrow[ul, "\gamma"']
\end{tikzcd}$$

(b)

Fig. 2. Instance graphs

2.2 Semantics

In order to be able to reason about diagrams, we provide a formalization of the intuitive definition suggested above. Formally, a diagram is represented by a structure $\langle Ng, Eg, m_s, m_d \rangle$ called a generic graph where:

- $Ng \subset \mathcal{N}$ is a finite set of typed nodes[1] ($\mathcal{N}$ denotes the domain of nodes).
- $Eg \subset \mathcal{N} \times \mathcal{T}_e \times \mathcal{N}$ is a finite set of typed edges (the set $\mathcal{T}_e$ denotes the domain of edge types). If $(A, \alpha, B) \in Eg$ then $A \in Ng$ and $B \in Ng$; A is called the source node and B the destination node.
- $m_s, m_d : \mathcal{N} \times \mathcal{T}_e \times \mathcal{N} \to \mathcal{I} - [0, 0]$ are functions mapping each edge to the multiplicities associated with the source and destination nodes respectively. $\mathcal{I}$ is the set of (non empty) intervals over *Nat*. Like UML, we disallow null multiplicities ($[0, 0]$).

The instances of a generic graph are graphs $\langle Ni, Ei \rangle$ where $Ni \subset \mathcal{N}$ is the set of instance nodes and $Ei \subset \mathcal{N} \times \mathcal{T}_e \times \mathcal{N}$ is the set of instance edges.

$$\mathit{Sem}(Gg) = \{Gi \mid \exists \mathit{Class} : Ni \mapsto Ng,\ Gg \overset{Class}{\rightleftharpoons} Gi\}$$

$$\text{where} \quad \langle Ng, Eg, m_s, m_d \rangle \overset{Class}{\rightleftharpoons} \langle Ni, Ei \rangle \quad \text{iff:}$$

$$\forall A \in Ng.\ \exists a \in Ni.\ \mathit{Class}(a) = A \tag{1}$$

$$\begin{gathered} \forall E = (A, \alpha, B) \in Eg.\ \forall a \in Ni. \\ \mathit{Class}(a) = A \Rightarrow \mathit{Card}\{b \mid \mathit{Class}(b) = B \wedge (a, \alpha, b) \in Ei\} \in m_d(E) \end{gathered} \tag{2}$$

$$\begin{gathered} \forall E = (A, \alpha, B) \in Eg.\ \forall b \in Ni. \\ \mathit{Class}(b) = B \Rightarrow \mathit{Card}\{a \mid \mathit{Class}(a) = A \wedge (a, \alpha, b) \in Ei\} \in m_s(E) \end{gathered} \tag{3}$$

$$\forall a, b \in Ni.\ \ \mathit{Class}(a) = A \wedge \mathit{Class}(b) = B \wedge (A, \alpha, B) \notin Eg \Rightarrow (a, \alpha, b) \notin Ei \tag{4}$$

Fig. 3. Semantics of generic graphs

Figure 3 describes the semantics of a generic graph as the set of instance graphs respecting the multiplicities and type constraints. A graph Gi is a valid instance of Gg if and only if there exists a total function *Class* mapping instance nodes to their generic node such that four conditions hold. The first condition enforces that each generic node has at least one instance. The two other conditions enforce the multiplicity constraints as described before. The last condition expresses the fact that all the typed edges allowed in instance graphs are those described in the generic graph.

2.3 Consistency checking

It is important to realize that a diagram can represent an empty set of graphs. In this case, we say that the diagram is inconsistent.

Definition 1 $\mathit{Consistent}(Gg) \equiv \mathit{Sem}(Gg) \neq \emptyset$

For example, the diagram $\begin{tikzcd} A \arrow[r, bend left, "\alpha", "2" near start, "1" near end] & B \arrow[l, bend left, "\beta", "1" near start, "1" near end] \end{tikzcd}$ is inconsistent. The reason lies

[1] Node types do not play any role in this section. They are used to distinguish different kinds of entities in diagrams defining software architecture views (Section 3).

in the contradiction in the specification of multiplicities: the multiplicities of the α-edge imply that there must be two instances of A for each instance of B whereas the multiplicities of the β-edge imply that there must be a single instance of A for each instance of B.

In this case, inconsistency may look obvious but it is not always easy to detect contradictions in more complex diagrams. Fortunately, consistency can be reduced to the satisfiability of a system of linear inequalities. This system is derived from the generic graph, using the formula described in Figure 4.

$$\boxed{\begin{array}{l} \mathit{Sat}(\langle Ng, Eg, m_s, m_d\rangle) = \exists\{x_N \geq 1\}_{N\in Ng} \text{ such that} \\ \displaystyle\bigwedge_{E=(A,\alpha,B)\in Eg} \left\{\begin{array}{ll} x_A & \geq i_A \\ x_B & \geq i_B \\ x_A\ j_B & \geq x_B\ i_A \\ x_B\ j_A & \geq x_A\ i_B \end{array}\right. \quad \text{where} \quad \begin{array}{l} [i_A, j_A] = m_s(E) \\ [i_B, j_B] = m_d(E) \end{array} \end{array}}$$

Fig. 4. Consistency as integer constraint solving

In Figure 4, a variable x_N represents the number of instances of a generic node N. The semantics enforces that each node has at least one instance so each variable x_N must satisfy the constraint $x_N \geq 1$. Furthermore, for each generic edge

$$A \xrightarrow{[i_A,j_A] \quad \alpha \quad [i_B,j_B]} B$$

four constraints between the number of instances and the multiplicities are produced. These constraints will be justified in the proof below. For a generic graph with n nodes and e edges this produces a system of $4e+n$ linear inequalities over n variables. Standard and efficient techniques can be applied to decide whether such a system has a solution.

We now apply the consistency check on the simple diagram
$$\begin{tikzcd} A \arrow[r, bend left, "2\quad\alpha\quad 1"] & B \arrow[l, bend left, "1\quad\beta\quad 1"] \end{tikzcd}$$
that was declared inconsistent at the beginning of this section. The edge $A \xrightarrow{2\ \ \alpha\ \ 1} B$ raises constraints $x_A \geq 2\ x_B$ and $x_A \leq 2\ x_B$ which are equivalent to $x_A = 2\ x_B$. The edge $A \xleftarrow{1\ \ \beta\ \ 1} B$ raises contraints $x_A \geq x_B$ and $x_A \leq x_B$ which impose $x_A = x_B$. Thus, the system of linear inequalities derived from the diagram requires that $x_B = 2\ x_B$. Together with the constraint $x_A \geq 1$ and $x_B \geq 1$, this equation has no solution. Hence, *Sat* returns false and we can conclude that the diagram is inconsistent.

It remains to prove that $\mathit{Sat}(Gg)$ provides a necessary and sufficient condition for the consistency of Gg.

Property 1 *Consistent*$(Gg) \Leftrightarrow$ *Sat*(Gg)

Proof.
($\Rightarrow$) If *Sem*$(Gg) \neq \emptyset$, there exists an instance graph Gi and a function *Class* such that conditions (1), (2), and (3) of Figure 3 are satisfied. For each node A of Gg we note X_A the number of instances of A occurring in Gi and show that the X_A's form a solution to the system of constraints.
· Condition (1) implies that $X_A \geq 1$ for each $A \in Ng$.
· For each edge $A \xrightarrow{[i_A,j_A]\quad\alpha\quad[i_B,j_B]} B$ of Gg condition (2) implies that $X_B \geq i_B$ and condition (3) that $X_A \geq i_A$.
· Condition (2) (resp. (3)) implies that each instance of A (resp. B) is connected to at least i_B (resp. i_A) and at most j_B (resp. j_A) instances of B (resp. A). Let $E^{\alpha^{A}_{B}}$ be the total number of α-edges between instances of A and B, we have $X_A\ i_B \leq E^{\alpha^{A}_{B}} \leq X_A\ j_B$ and $X_B\ i_A \leq E^{\alpha^{A}_{B}} \leq X_B\ j_A$ and therefore $X_A\ j_B \geq X_B\ i_A$ and $X_B\ j_A \geq X_A\ i_B$

($\Leftarrow$) We assume that *Sat*(Gg) holds and construct an instance graph $Gi = \langle Ni, Ei\rangle$ such that $Gi \in$ *Sem*(Gg).
From *Sat*(Gg) we can associate with each node A in Ng a number of instances X_A respecting the whole set of constraints. We take a set of nodes Ni and a total function $Class : Ni \mapsto Ng$ such that $\forall N \in Ng,\ Card\{n_i \in Ni \mid Class(n_i) = N\} = X_N$. Since $X_N \geq 1$, condition (1) of Figure 3 is verified.
The number of instance nodes being fixed, we can reason locally and show that for each edge $A \xrightarrow{[i_A,j_A]\quad\alpha\quad[i_B,j_B]} B$ of Gg we can produce instance edges respecting conditions (2) and (3) of Figure 3. From the constraints, the interval $[X_A\ i_B, X_A\ j_B] \cap [X_B\ i_A, X_B\ j_A]$ cannot be empty. We choose a value $E^{\alpha^{A}_{B}}$ of this interval as the number of α-edges between the instances of A and B. We attach to each instance of A (in turn and cyclically) one edge to a new arbitrary instance of B until the $E^{\alpha^{A}_{B}}$ edges are placed. Since $X_A\ i_B \leq E^{\alpha^{A}_{B}} \leq X_A\ j_B$, this process ensures that condition (2) holds. Now, if an instance b_k of B has more than j_A (resp. less than i_A) incoming α-edges we know from $X_B\ i_A \leq E^{\alpha^{A}_{B}} \leq X_B\ j_A$ that there exists another instance b_l of B having less than j_A (resp. more than i_A) incoming α-edges. We switch the destination of one incoming α-edge from b_k to b_l (resp. b_l to b_k). This process can be repeated until condition (3) holds.

2.4 Solid edges

From our experience, the notation described above is too imprecise to define software architecture views. For example, it is not possible to express the constraint that each instance of a node A is doubly linked to an instance of B. Indeed, the diagram

$$\begin{tikzcd} A \arrow[r, bend left, "1\ \alpha\ 1"] & B \arrow[l, bend left, "1\ \beta\ 1"] \end{tikzcd} \qquad \text{accepts} \qquad \begin{tikzcd} & b_1 \arrow[dr, "\beta"] & \\ a_1 \arrow[ur, "\alpha"] & & a_2 \arrow[dl, "\alpha"] \\ & b_2 \arrow[ul, "\beta"] & \end{tikzcd}$$

as a valid instance. Although our graphical notation can specify graphs with properties such as "for all instances of A there is a simple[2] (or length k, α-typed, ...) path to an instance of B", it does not have the ability to enforce properties such as "there is a simple path from each instance of A to itself". This makes some sharing patterns impossible to describe. While this may not be a problem for diagrams describing the organization of data in UML structural views, a greater precision in the specification is often desirable for many typical software architecture views (such as the control or physical views).

We extend our notation by introducing the notion of "solid edges". They are represented as bold arrows as shown in the diagram of Figure 5.

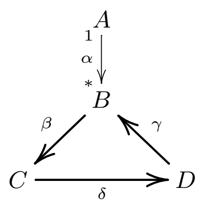

Fig. 5. A diagram with solid edges

Solid edges bear implicitly the multiplicity 1 at both ends. The intention is that the structure of the region delimited by solid edges should be reflected in the instances of the diagram. For example, the diagram of Figure 5 accepts the graph of Figure 2 (a) as a valid instance but not the graph of Figure 2 (b).

The semantics of diagrams must be extended to take into account this new feature. A generic graph is now represented by the structure $\langle Ng, Eg, m_s, m_d, s\rangle$ where the function $s : \mathcal{N} \times \mathcal{T}_e \times \mathcal{N} \to \mathit{Bool}$ is such that $s(E) \Leftrightarrow E$ *is a solid edge.* We assume that $s(E) \Rightarrow m_s(E) = [1, 1] \wedge m_d(E) = [1, 1]$. A new condition (5) is imposed in the semantics of Figure 3.

$$\forall a, b \in Ni.\ \left\{ \begin{array}{l} \mathit{Class}(a) = A \wedge \mathit{Class}(b) = B \\ \wedge\ \mathit{solid}(a, b) \wedge s(A, \alpha, B) \end{array} \right\} \Rightarrow (a, \alpha, b) \in Ei \tag{5}$$

where

$$\begin{array}{rcl} \mathit{solid}(a, b) & \equiv & (a = b) \\ & \vee & \exists\, (a_1, \alpha_1, b_1), \ldots, (a_k, \alpha_k, b_k) \in Ei \text{ such that } a_1 = a, b_k = b, \\ & & \bigwedge_{i=1}^{k-1} s\,(\mathit{Class}(a_i), \alpha_i, \mathit{Class}(b_i)), \\ & & \bigwedge_{i=1}^{k-1} \big((a_i = a_{i+1}) \vee (a_i = b_{i+1}) \vee (b_i = a_{i+1}) \vee (b_i = b_{i+1})\big) \end{array}$$

This condition ensures that for each connected region in the instance graph corresponding to a solid region in the generic graph, the *Class* function is a bijection. So, each solid region must have only isomorphic images in the instance

[2] a path where any nodes x and y are such that $\mathit{Class}(x) \neq \mathit{Class}(y)$

graph. Consistency checking is not affected by this extension. It can be done as before by considering solid edges as standard edges with multiplicities 1 and 1. This is expressed formally as follows:

Property 2

$$\mathit{Sem}(\langle Ng, Eg, m_s, m_d, \lambda x.\mathit{false}\rangle) \neq \emptyset \Rightarrow \mathit{Sem}(\langle Ng, Eg, m_s, m_d, s\rangle) \neq \emptyset$$

Here, $Gg = \langle Ng, Eg, m_s, m_d, \lambda x.\mathit{false}\rangle$ denotes the generic graph of a diagram where all solid edges are considered as standard edges with multiplicities 1 and 1. The proof proceeds by constructing from any valid instance Gi of Gg another valid instance graph satisfying condition (5). First, all edges of Gi corresponding to solid edges are removed. From the constraints of Figure 4, we know that all nodes $\{A_1, \ldots, A_k\}$ connected by edges of multiplicities 1 and 1 in Gg have the same number of instances (say p). For all largest sets of nodes $\{A_1, \ldots, A_k\}$ connected by solid edges in Gg, the corresponding instance nodes can be split into p sets $\{a_1, \ldots, \alpha_k\}$ where $\mathit{Class}(a_i) = A_i$. For each such set, the previously removed edges are replaced isomorphically to the solid sub-graph. The multiplicity constraints are respected and condition (4) holds.

3 Application to the design of a train control system

We illustrate our framework with a case study proposed in [5] concerning the specification of the architecture of a train control system. We propose a multiple view architecture defined as a collection of diagrams – one per view, plus one diagram to describe the correspondences between views. The following is an excerpt of [5] identifying the main challenges of the case study:

> "Generally, a control system performs the following tasks: processing the raw data obtained from the environment through sensing devices, taking corrective action [...]. In addition to the *functional* requirements of these systems, many non-functional requirements, such as *geographical distribution* over a possibly wide variety of different host processors, place constraints on the design freedom that are very difficult to meet. In practice, there are many *interrelated system aspects* that need to be considered. [...] Although solutions are available for many of the problems in isolation, incompatible, or even conflicting premises make it very difficult to cover all design aspects by a *coherent solution*."

We present here a simplified version of the architecture of a train control system focusing on the process of corrective actions. We distinguish two parts in the system: the trains and a control system monitoring the traffic. In our solution, each train computes and performs speed corrections with respect to three parameters: its route (a detailed schedule including the reservation dates of each track section), the railway topology, and its actual state (speed and position) as indicated by its sensing devices. Trains periodically send their states to the control system which extrapolates from the collected states to detect future conflicts. Once identified, conflicts are solved and new routes are sent to trains.

The train control system is described as a multiple view architecture. In the simplified version, the architecture consists of three views: the distributed functional view (DFV), the distributed control view (DCV), and the physical view (PhV). In our framework, each view is described by a diagram, and correspondences between views are established through an additional diagram. From these related views, we show how the requirements listed above can be addressed in our framework.

In each of the three views, we distinguish two parts consisting of nodes connected by solid edges: the train part and the control system part. These two parts themselves are connected by standard edges with multiplicities $*$ and 1. It allows several trains to be connected to the control system. Each view is described by a diagram belonging to a given style. A style is defined here as a set of node and edge types together with type constraints such as: edges of type R (for read) can only be used to connect nodes of type DATA to nodes of type PROCESS. Different views use different types so that a node type indicates without any ambiguity the view to which the node belongs. Types are associated with a graphical notation defined in the caption of each view. For the sake of clarity, we use expressions in typewriter font (such as `process speed correction`) to denote node names.

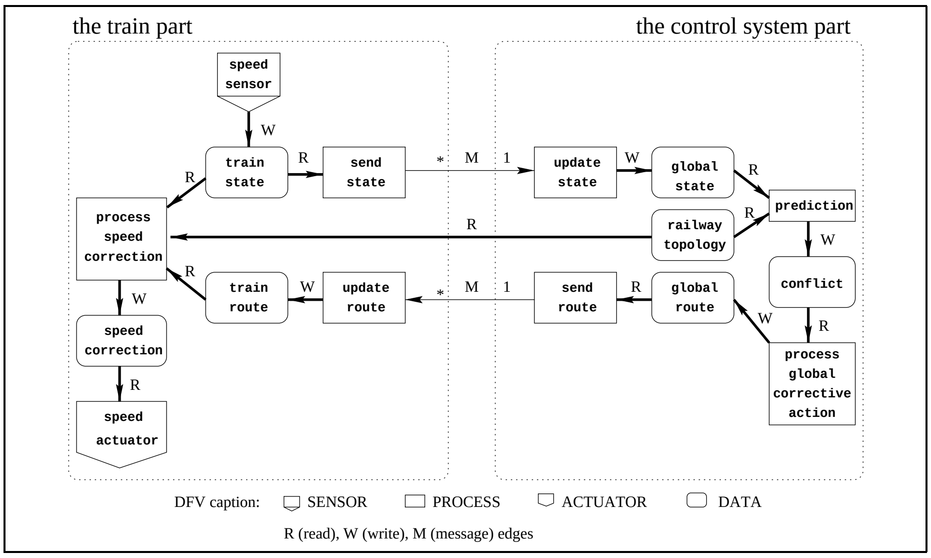

Fig. 6. The distributed functional view

The *distributed functional view* (Figure 6) describes the data flows and data dependencies between processes, using four node types and three edge types. SENSOR and ACTUATOR nodes represent the input and output ports of the system, PROCESS nodes correspond to entities of computation, and DATA nodes correspond to variables. SENSOR, ACTUATOR and PROCESS nodes can be connected to DATA nodes by edges of type R or W denoting respectively the potential to read or write a variable. An edge of type M represents message passing between two PROCESS nodes.

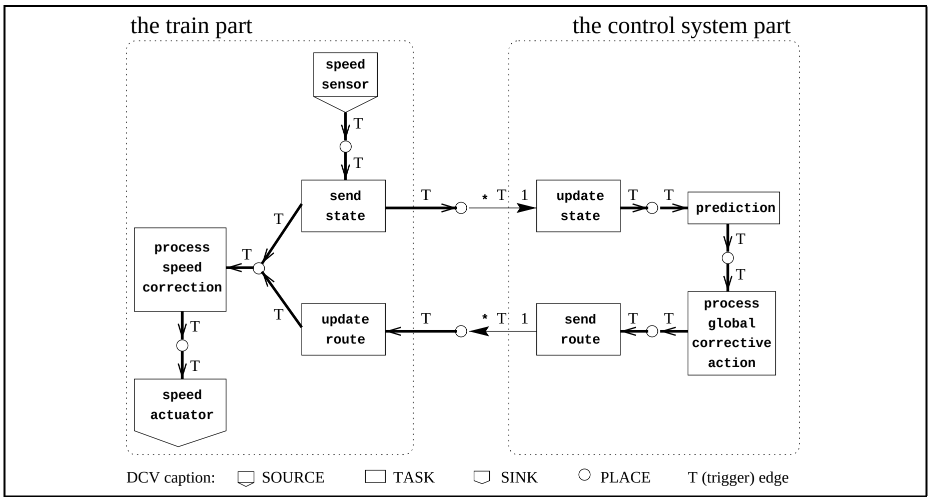

Fig. 7. The distributed control view

The *distributed control view* (Figure 7) describes the scheduling of processes and the control flow in the spirit of Petri nets. It uses four node types and one edge type. Nodes of types SOURCE, SINK, and TASK can be seen as transitions in a Petri net. Our intention is to draw a parallel between those node types and the node types of the distributed functional view (SENSOR with SOURCE, ACTUATOR with SINK, and PROCESS with TASK). PLACE nodes correspond to repository of tokens in Petri nets. They are used for description purpose only and have no corresponding nodes in the other views. Nodes of the distributed control view are connected by triggering edges of type T.

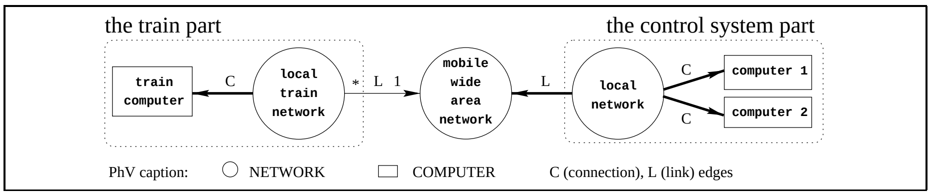

Fig. 8. The physical view

The *physical view* (Figure 8) describes the network connections between computers. It has two types of nodes, COMPUTER and NETWORK, and two types of edges: C which denotes a connection between a computer and a network, and L which represents a connection between networks.

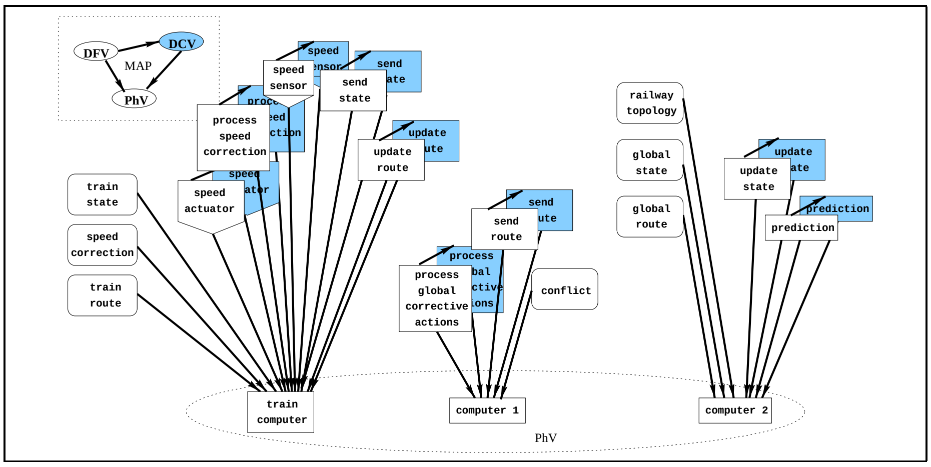

Fig. 9. DFV, DCV, and PhV correspondences

It is easy to check that the multiplicity constraints can be satisfied; so that each view is consistent. The different views of the system are not unrelated, though. In our framework, the inter-view relationships are expressed through edges of a specific type, called MAP. For example, an edge (P, MAP, C) between node P of type PROCESS and node C of type COMPUTER indicates that the process P of the distributed functional view is mapped onto the computer C of the physical view. The MAP relations between the three views are described in an additional diagram (Figure 9) which contains nodes of the three views to be related. Each node of type PROCESS (resp. SENSOR, ACTUATOR) in the distributed functional view is mapped onto a node of type TASK (resp. SOURCE, SINK) with the same name in the distributed control view. DATA nodes do not have any counterpart in the distributed control view. All nodes of types DATA, PROCESS, SENSOR, and ACTUATOR in the train part of the distributed functional view are mapped onto the node named `train computer` in the physical view. The other ones are mapped onto one of the nodes `computer 1` and `computer 2` in the control system part of the physical view. All nodes of types TASK, SOURCE, and SINK in the train part (resp. the control system part) of the distributed control view are mapped onto nodes of type COMPUTER in the same part of the physical view.

The diagram of Figure 9 can be superimposed on the three diagrams of Figure 6, 7 and 8 to form the complete diagram defining the software architecture of the system. *Consistency checking* (Section 2.3) of *the complete diagram* ensures that the family of architectures described by the three related views is not empty.

4 Specification and verification of constraints

The graphical notation introduced in Section 2 is well-suited to the specification of the overall connection pattern of each view and the correspondences between their components. However, this notation generates only simple constraints on the number of occurrences of edges and nodes. In the context of software architectures, it is often desirable to be able to impose more sophisticated (both inter-view and intra-view) consistency constraints. In our case study, for example, we would like to impose that a process of the distributed functional view and its corresponding task in the distributed control view are mapped onto the same computer in the physical view. Another requirement could be that each process must be on the same site as any data to which it has read or write access.

To address this need, we propose a small language of constraints and define a complete checking algorithm for the generic graphs introduced in Section 2. We illustrate the interest of this language with the case study introduced in Section 3.

4.1 A simple constraint language

The syntax of our constraint language is the following:

$$\begin{array}{rcl} C & ::= & \forall x_1 : \tau_1, \ldots,\ x_n : \tau_n.\ P \\ P & ::= & P_1 \wedge P_2 \mid P_1 \vee P_2 \mid \mathit{edge}(x, \alpha, y) \mid \mathit{path}_{E_t}(x, y) \mid \neg P \\ E_t & \subseteq & \mathcal{T}_e \cup \overline{\mathcal{T}_e} \end{array}$$

We also use $\Rightarrow$ in the following but we do not introduce it as a basic connector since it can be defined using $\vee$ and $\neg$. The semantics of the constraint language is presented in Figure 10. A generic graph Gg satisfies a constraint C if all its instances Gi satisfy the constraint. Constraints $\mathit{path}_{E_t}(x, y)$ are defined with respect to a set of edge types E_t. $\overline{\mathcal{T}_e} = \{\overline{\alpha} \mid \alpha \in \mathcal{T}_e\}$ is a set of annotated types used to accept inverse edges in paths. For example, $\mathit{path}_{\{\alpha,\overline{\alpha}\}}(a, b)$ is true if there exists an undirected path, that is a path made of undirected α-edges, between nodes a and b. Note that we consider only simple paths of the instance graphs. *Simple paths* correspond to non-cyclic paths of the generic graph (condition $\mathit{Class}(a_i) \neq \mathit{Class}(a_j)$ at the bottom of Figure 10).

Examples of the use of the language to express both inter-view and intra-view compatibility constraints are provided in Section 4.3. We now turn our attention to the design of a verification algorithm for this language of constraints.

4.2 A constraint checking algorithm

The semantics of the language of constraints presented in Figure 10 is not directly suggestive of a checking algorithm because it is defined with respect to the

$$
\begin{array}{l}
Gg \models C \;\Leftrightarrow\; \forall Gi \in \mathit{Sem}(Gg),\; Gi \vdash C \\
\textbf{where} \quad Gi = \langle Ni, Ei \rangle \quad \text{and} \quad Gg \overset{\mathit{Class}}{\rightleftharpoons} Gi \\
\\
\begin{array}{lcll}
Gi \vdash & \forall x_1:\tau_1,\ldots,x_n:\tau_n.\ P & \Leftrightarrow & \forall x_1:\tau_1,\ldots,x_n:\tau_n \in Ni.\ Gi \vdash P \\
Gi \vdash & P_1 \wedge P_2 & \Leftrightarrow & Gi \vdash P_1 \;\wedge\; Gi \vdash P_2 \\
Gi \vdash & P_1 \vee P_2 & \Leftrightarrow & Gi \vdash P_1 \;\vee\; Gi \vdash P_2 \\
Gi \vdash & \mathit{edge}(x,\alpha,y) & \Leftrightarrow & (x,\alpha,y) \in Ei \\
Gi \vdash & \mathit{path}_{E_t}(x,y) & \Leftrightarrow & \exists a_1,\ldots,a_{k+1} \in Ni.\ \exists \alpha_1,\ldots,\alpha_k \in E_t. \\
 & & & \quad a_1 = x \wedge a_{k+1} = y \\
 & & & \wedge\ \forall i \in [1,k].\quad ((a_i,\alpha_i,a_{i+1}) \in Ei \wedge \alpha_i \in \mathcal{T}_e) \\
 & & & \qquad\qquad\qquad \vee\ ((a_{i+1},\alpha_i,a_i) \in Ei \wedge \overline{\alpha_i} \in \overline{\mathcal{T}_e}) \\
 & & & \wedge\ \forall i,j \in [1,k].\ i \neq j \Rightarrow \mathit{Class}(a_i) \neq \mathit{Class}(a_j) \\
Gi \vdash & \neg P & \Leftrightarrow & Gi \nvdash P
\end{array}
\end{array}
$$

Fig. 10. Semantics of the language of constraints

(potentially infinite) set of all the instances Gi of a generic graph Gg. In this subsection, we sketch a checking algorithm, Check, and provide some intuition about its correctness and completeness, stated as follows:

Property 3 $\mathit{Check}(C, Gg) \Leftrightarrow Gg \models C$

Space considerations prevent us from presenting the algorithm thoroughly; the interested reader can find a detailed account of the algorithm and proofs in [8].

The Check procedure outlined in Figure 11 takes two arguments: a property $\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ P$ in canonical form and a generic graph Gg. Properties in canonical form are properties in conjunctive normal form with negations bearing only on relations $\mathit{edge}(x, \alpha, y)$ and $\mathit{path}_{E_t}(x, y)$. The transformation of properties into their canonical form is straightforward.

The first step of the algorithm consists in considering all the possible classes X_i (denoting nodes of the generic graph) consistent with the types of the variables x_i. The interesting case in the definition of Verif is the disjunction which is based on a proof by contradiction. The function Contra returns true if its argument contains a basic property and its negation. The function Gen is called with three arguments: the context *Class*, which records the class of each variable, the generic graph Gg and the set of basic properties[3] $\{\neg B_1, \ldots, \neg B_n\}$. Gen is the core of the algorithm: it generates all the basic properties that can be derived from this initial set. Gen is defined as a composition of intermediate functions:

– GenPe expands $\mathit{path}(x, y)$ properties into sequences of edges by introducing fresh nodes variables. All the simple paths between $\mathit{Class}(x)$ and $\mathit{Class}(y)$ in the generic graph Gg are considered. As a consequence, GenPe returns a set of tuples $\langle \mathit{Class}, Gg, S \rangle$ which corresponds to a logical disjunction. This justifies the use of the $f^\star$ notation to ensure that the subsequent functions

[3] Basic properties are properties of the form $\mathit{edge}(x, \alpha, y), \mathit{path}_{E_t}(x, y)$ or their negation.

$$\text{Check}(\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ P,\ Gg\ ) = \bigwedge_{X_1:\tau_1,\ldots,X_n:\tau_n \in Ng} \text{Verif}([x_i \mapsto X_i],\ Gg,\ P)$$

$$\textbf{where } Gg = \langle Ng, Eg, m_s, m_d, s\rangle$$

$$\begin{aligned}
\text{Verif}(\mathit{Class}, Gg,\ P_1 \wedge \ldots \wedge P_n) &= \text{Verif}(\mathit{Class}, Gg,\ P_1) \wedge \ldots \wedge \text{Verif}(\mathit{Class}, Gg,\ P_n)\\
\text{Verif}(\mathit{Class}, Gg,\ B_1 \vee \ldots \vee B_n) &= \text{Contra}(\text{Gen}(\langle \mathit{Class}, Gg, \{\neg B_1, \ldots, \neg B_n\}\rangle))
\end{aligned}$$

$$\textbf{where } \text{Contra}(S) = \exists B \in S.\ \neg B \in S$$

$$\begin{aligned}
\text{Gen}(\langle \mathit{Class}, Gg, S\rangle) = \ & \text{Lub} \circ \text{Gen}_{\text{Ni}}^{\star} \circ \text{Gen}_{\text{Pi}}^{\star}\\
& \circ\ \mathit{iter}\ (\text{Gen}_{\text{Sub}}^{\star} \circ \text{Gen}_{\text{Eq}}^{\star} \circ \text{Gen}_{\text{Ed}}^{\star} \circ \text{Gen}_{\text{So}}^{\star})\\
& \circ\ \text{Gen}_{\text{Neg}}^{\star} \circ \text{Gen}_{\text{Pe}}(\langle \mathit{Class}, Gg, S\rangle)
\end{aligned}$$

$$\textbf{where} \quad f^{\star}(s) = \{f(x) \mid x \in s\}$$

$$\mathit{iter}\ f\ x = \begin{cases} x & \text{if } f(x) = x\\ \mathit{iter}\ f\ f(x) & \text{otherwise}\end{cases}$$

$$\text{Lub}(\{\langle \mathit{Class}_1, Gg, S_1\rangle, \ldots, \langle \mathit{Class}_n, Gg, S_n\rangle\}) = \bigcap_{i\in[1,n]} S_i \text{ s.t. } \neg\text{Contra}(S_i)$$

Fig. 11. Constraint checking algorithm

are applied to all the tuple elements of this set. The Lub function is then used to compute the intersection of all the resulting (non contradictory) sets; it corresponds to disjunction elimination in an inference system.

– Gen_{Neg} exploits the generic graph and condition (4) of the semantics of generic graphs (see Figure 3) to produce all valid $\neg \mathit{edge}$ and $\neg \mathit{path}$ relations.
– Gen_{So} computes the relation *solid* as defined in Section 2.4 and Gen_{Ed} applies condition (5) to derive all possible new edges.
– Gen_{Eq} exploits the multiplicities in the generic graph to derive all possible equalities between nodes. For example, if the generic graph is such that

$$m_d(\mathit{Class}(x), \alpha, \mathit{Class}(y)) = 1$$

then Gen_{Eq} derives $y_1 = y_2$ from $\mathit{edge}(x, \alpha, y_1)$ and $\mathit{edge}(x, \alpha, y_2)$.
– Gen_{Sub} uses the equalities produced by Gen_{Eq} to derive new properties (for example $\mathit{edge}(x_2, \alpha, y)$ can be derived from $x_1 = x_2$ and $\mathit{edge}(x_1, \alpha, y)$). The application of Gen_{Sub} can lead to the derivation of new *solid* relations by Gen_{So}, hence the iteration. The termination of *iter* is ensured by the fact that no new variable is introduced in the iteration steps (so only a finite number of basic properties can be generated by Gen).
– Gen_{Pi} generates the path properties implied by the edge properties.
– Gen_{Ni} generates negations that follow from deduction rules such as:

$$\mathit{edge}(x, \alpha, y) \wedge \neg \mathit{path}_{E_t \cup \{\alpha\}}(x, z) \Rightarrow \neg \mathit{path}_{E_t \cup \{\alpha\}}(y, z).$$

It can be applied as a final step since negations are not used by the preceeding functions.

The correctness and completeness proofs described in [8] are based on an intermediate inference system (which is itself complete and correct with respect to the semantics of generic graphs and the language of constraints). Correctness of the algorithm is straightforward. Completeness and termination rely on the application ordering of the intermediate functions of Gen. The proof uses the fact that an intermediate function F cannot generate a property which could be used (to derive new properties) by a function F' that is not applied after F.

The Check procedure outlined here is a naïve algorithm derived from the inference system. We did not strive to apply any optimisation here. In an effective implementation, the intermediate functions of Gen would be re-ordered so as to detect contradictions as early as possible (and the algorithm would stop as soon as two contradictory properties are generated).

4.3 Automatic verification of compatibility constraints

We return to the case study introduced in Section 3 and we show how inter-view and intra-view compatibility relations can be expressed within our constraint language. We consider three constraints involving the distributed functional view, the control view and the physical view. In this subsection, we let Gg stand for the generic graph $\langle Ng, Eg, m_s, m_d, s\rangle$ made of all the (related) views defined in Section 3.

1. **Process-Task consistency constraint**: "A process of the distributed functional view and its corresponding task in the distributed control view must be placed on the same computer in the physical view." The mapping between DFV and PhV associates data with the processes that use them, while the mapping between DCV and PhV is driven by concurrency concerns. The compatibility constraint imposes that a tradeoff must be found that agrees on the placement of processes on the available computers. It is expressed as follows in our language:

   $\forall p : \textsc{process},\ t : \textsc{task},\ c : \textsc{computer}.$
   $\quad \mathit{edge}(p, \textsc{map}, t) \wedge \mathit{edge}(p, \textsc{map}, c) \Rightarrow \mathit{edge}(t, \textsc{map}, c)$

   The canonical form of this constraint is:

   $\forall p : \textsc{process},\ t : \textsc{task},\ c : \textsc{computer}.$
   $\quad \mathit{edge}(t, \textsc{map}, c) \vee \neg \mathit{edge}(p, \textsc{map}, t) \vee \neg \mathit{edge}(p, \textsc{map}, c)$

   The application of the Check procedure to this constraint and Gg results in a call to:

   $$\text{Contra}(\text{Gen}(\langle \mathit{Class}, Gg, \{\neg \mathit{edge}(t, \textsc{map}, c), \mathit{edge}(p, \textsc{map}, t), \mathit{edge}(p, \textsc{map}, c)\}\rangle))$$

   Then, whatever the mapping of the nodes variables p, c, t on class nodes of the complete diagram, Gen adds the relation $\mathit{edge}(t, \textsc{map}, c)$ to the initial set. It results in a contradiction with $\neg \mathit{edge}(t, \textsc{map}, c)$ and Check returns true. The relation $\mathit{edge}(t, \textsc{map}, c)$ is generated by the GenSo and GenEd functions from the relations $\mathit{edge}(p, \textsc{map}, t)$ and $\mathit{edge}(p, \textsc{map}, c)$ using the fact that the

MAP edges are solid edges.

2. **Site consistency constraint**: "A process should be on the same site as any data to which it has read or write access. A site is taken as a set of computers linked by a local network. " This second constraint is expressed as follows in our language:

$$\begin{array}{l} \forall p : \textsc{process},\ d : \textsc{data},\ m_1, m_2 : \textsc{computer},\ n : \textsc{network}. \\ \quad \big(\mathit{edge}(p, \textsc{r}, d) \vee \mathit{edge}(p, \textsc{w}, d)\big) \\ \quad \wedge\ \mathit{edge}(p, \textsc{map}, m_1) \wedge \mathit{edge}(d, \textsc{map}, m_2) \wedge \mathit{edge}(m_2, \textsc{c}, n) \\ \quad \Rightarrow\ \mathit{edge}(m_1, \textsc{c}, n) \end{array}$$

The application of the Check procedure shows that this constraint does not hold. After translation of the contraint into its canonical form, Verif has to examine a conjunction of two disjunctions. One of these disjunctions leads to a call to Gen with the initial set of relations:

$$\{\mathit{edge}(p, \textsc{r}, d), \mathit{edge}(p, \textsc{map}, m_1), \mathit{edge}(d, \textsc{map}, m_2), \mathit{edge}(m_2, \textsc{c}, n), \neg\mathit{edge}(m_1, \textsc{c}, n)\}$$

Together with the mapping: [$p \mapsto$ `process speed correction`, $d \mapsto$ `railway topology`, $m_1 \mapsto$ `train computer`, $m_2 \mapsto$ `computer 2`, $n \mapsto$ `local network`], the Gen function does not generate the expected contradiction. The reason lies in the R link between the nodes `railway topology` and `process speed correction` in the distributed functional view (Figure 6) and the fact that these two nodes are mapped onto two computers (`computer 2` and `train computer`) belonging to two different sites (Figure 9): `computer 2` is linked to `local network` and `train computer` is linked to `local train network` (Figure 8). This example shows that the Check procedure can easily be extended to return counterexamples.

3. **Communication consistency constraint:** "Two processes communicating by message passing are not allowed to share data." This constraint imposes that message passing is used only for communication between distant processes. The constraint is expressed as follows in our language:

$$\begin{array}{l} \forall p_1, p_1', p_2, p_2' : \textsc{process},\ d : \textsc{data}. \\ \quad \wedge \begin{array}{l} (\ \mathit{edge}(p_1, \textsc{r}, d) \vee \mathit{edge}(p_1, \textsc{w}, d)\ ) \\ (\ \mathit{edge}(p_2, \textsc{r}, d) \vee \mathit{edge}(p_2, \textsc{w}, d)\ ) \end{array} \wedge\ \mathit{edge}(p_1', \textsc{m}, p_2') \\ \Rightarrow \neg(\ \mathit{path}_{\{\textsc{r},\textsc{w},\overline{\textsc{r}},\overline{\textsc{w}}\}}(p_1, p_1') \wedge \mathit{path}_{\{\textsc{r},\textsc{w},\overline{\textsc{r}},\overline{\textsc{w}}\}}(p_2, p_2')\ ) \end{array}$$

This constraint, which is an example of an intra-view consistency relation, is not satisfied by the diagram of Figure 6. This is because process `prediction` and `process speed correction` share the data `railway topology` and still communicate (indirectly) by message passing (following a $\{\textsc{r}, \textsc{w}, \overline{\textsc{r}}, \overline{\textsc{w}}, \textsc{m}\}$ path through the nodes `send state` and `update state`).

These constraints illustrate that inconsistencies can naturally arise in the specification of multiple views. Counterexamples produced by the extended Check procedure provide useful information to construct a correct architecture. A simple solution to satisfy the last two constraints consists in providing each train with a local copy of the `railway topology` data and adding an update mechanism as was done for the "state" variable (`global state` and `train state` in Figure 6). The two copies could then be mapped onto two different computers.

5 Conclusion

We have presented a framework for the definition of multiple view architectures and techniques for the automatic verification of their consistency. It should be noted that we have defined views as collections of uninterpreted graphs. The intended meaning of a diagram (or a collection of diagrams) is conveyed indirectly through the constraints. This uninterpreted nature of graphs makes it possible to specify a great variety of views. In Section 3 we just sketched three views used in the train control system case study. Following the approach presented in this paper, we have been able to address some of the requirements (both functional and non functional) listed in [5]. For example, distribution and fault-tolerance are expressed directly as views. Fault-tolerance views are refinements[4] of the distribution views and they give rise to constraints like "Data A and Data B must not be on the same memory device" or "Process A and Process B must be mapped onto two different processors".

An interesting byproduct of the work described here is that we can apply the algorithm of Section 2 to check the consistency of UML diagrams. UML also includes a language called OCL [14] for defining additional constraints on diagrams. OCL constraints can express navigations through the diagrams and accumulations of set constraints on the values of the node attributes. Even if our language of constraints and OCL are close in spirit, they are not directly comparable because OCL does not include recursive walks through the graph similar to our $path_{E_t}$ constraint; on the other hand, we did not consider constraints on node attributes. Previous attempts at formalizing certain aspects of UML and OCL (based on translations into Z) are reported in [6, 9]. In contrast with the framework presented here, they do not lead to automatic verification methods. Recent studies have also been conducted on the suitability of UML to define software architectures. In particular, [10] and [13] present detailed assessments of the advantages and limitations of UML in this context. In comparison with these approaches, we do not really use UML here, considering only graphical and multiplicity notations as a basis for describing the structure of the architecture. As mentioned in [10], a graphical notation may sometimes be too cumbersome to express correspondences between views. Since our diagrams are translated into constraints, we can easily allow a mixed notation (including diagrams and constraints) allowing architects to use the most appropriate means to define their views.

More generally, we put forward a two-layer definition of software architecture views: the basic or structural layer is defined using a graphical (or mixed)

[4] Refinement is defined formally here as the expansion of nodes by subgraphs.

notation and the specific or semantical layer is defined on the top of the first one, through node and edge attributes. We have only considered simple type attributes here because they are sufficient both to express interesting properties and to raise non-trivial consistency issues. We are currently working on the integration of more sophisticated attribute domains to describe other aspects of software architectures (such as performances or interaction protocoles).

Finkelstein et al. have proposed a framework supporting the definition and use of multiple viewpoints in system development [7]. They consider views expressed in different formalisms (Petri nets, action tables, object structure diagrams, ...). Correspondances between views are defined as relations between objects in a logic close to our language of constraints. The logical formulae are used as rules to help the designers to relate the views. An important departure of the work presented here with respect to [7] is that our consistency checking is performed on a family of architectures (which is potentially infinite); their verification is simpler since it applies to fixed viewpoint specifications (which correspond to a particular instance graph in our framework). To check consistency, they first require the designers to provide a translation of the views into first order logic. Consistency is then checked within this common formalism and meta-rules are used to report inconsistencies at the view level. Their philosophy is that "it is not possible (or even desirable) in general to enforce consistency between all the views at all times because it can unnecessarily constrain the development process". So they put stress on inconsistency management rather than consistency checking itself.

Consistency checking has also been carried out in the context of specification languages like Z, Lotos or Larch [16, 3]. The traditional approach, which can be called "implementation consistency", is summarized as follows [3]: "n specifications are consistent if and only if there exists a physical implementation which is a realization of all the specifications, ie. all the specifications can be implemented in a single system". In contrast, our approach decouples the issues of consistency and conformance for a better separation of concerns and an increased tractability. In this paper, we have been concerned exclusively with consistency. Our approach to conformance consists in seeing the software architecture views as collections of constraints that can be exploited to guide the development process. For example, the functional view specifies constraints on the possible flows of data between variables; the control view adds constraints on method or procedure calls and their sequencing. We are currently designing a generic development environment which takes architectural constraints as parameters and guarantees that only programs that conform to these constraints can be constructed. In addition to the views already mentioned in this paper, this environment should make use of a "development view", which represents the hierarchical organization of programs and data into development units (classes in Java).

References

1. R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80. IEEE Computer Society Press, May 1994.
2. M. Bourgois, D. Franklin, and P. Robinson. Applying RM-ODP to the air traffic management domain. EATCHiP Technical Document, Eurocontrol, Brussels, March 1998.
3. H. Bowman, E. Boiten, J. Derrick, and M. Steen. Viewpoint consistency in ODP, a general interpretation. In E. Najm and J.-B. Stefani, editors, *Proceedings of the 1st IFIP International Workshop on Formal Methods for Open Object-Based Distributed Systems*, pages 189–204. Chapman & Hall, March 1996.
4. S. C. Cheung and J. Kramer. Checking subsystem safety properties in compositional reachability analysis. In *Proceedings of the 18th International Conference on Software Engineering*, pages 144–154, Berlin - Heidelberg - New York, March 1996. Springer.
5. E. de Jong. Software architecture for large control systems: a case study. In D. Garlan and D. Le Métayer, editors, *Proceedings of Coordination'97*, volume 1282 of *LNCS*, pages 46–63. Springer-Verlag, September 1997.
6. Andy Evans. Reasoning with the unified modeling language. In *Proceedings of Workshop on Industrial-Strength Formal Specification Techniques (WIFT'98)*, 1998.
7. A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. *IEEE Transactions on Software Engineering*, 20(8):509–578, August 1994.
8. P. Fradet, D. Le Métayer, and M. Périn. Consistency checking for multiple view software architectures: application to the design of a train control system. Technical Report 1249, IRISA/INRIA, Rennes, France, 1999.
9. A. Hamie, J. Howse, and S. Kent. Interpreting the object constraint language. In *Proceedings of Asia Pacific Conference in Software Engineering.* IEEE Press, 1998.
10. C. Hofmeister, R. L. Nord, and D. Soni. Describing software architecture with UML. In P. Donohoe, editor, *Proceedings of Working IFIP Conference on Software Architecture*, pages 145–160. Kluwer Academic Publishers, February 1999.
11. P. Inverardi, A. Wolf, and D. Yankelevich. Checking assumptions in components dynamics at the architectural level. In D. Garlan and D. Le Métayer, editors, *Proceedings of Coordination'97*, volume 1282 of *LNCS*, pages 46–63. Springer-Verlag, September 1997.
12. Philippe B. Kruchten. The 4 + 1 view model of architecture. *IEEE Software*, 12(6):42–50, November 1995.
13. Nenad Medvidovic and David S. Rosenblum. Assessing the suitability of a standard design method for modeling software architecture. In P. Donohoe, editor, *Proceedings of Working IFIP Conference on Software Architecture*, pages 161–182. Kluwer Academic Publishers, February 1999.
14. UML. Object Constraint Language, version 1.1, September 1997. Available at `http://www.rational.com/uml`.
15. UML. Unified Modelling Language notations guide, version 1.1, September 1997. Available at `http://www.rational.com/uml`.
16. Pamela Zave and Michael Jackson. Conjunction as composition. *ACM Transactions of Software Engineering and Methodology*, 2(4):379–411, October 1993.