Finding inconsistencies between views using relation partition algebra

Finding Inconsistencies between Views using Relation Partition Algebra

J. Muskens, M.R.V. Chaudron and R.J. Bril

Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven, P.O. Box 513,
5600 MB Eindhoven, The Netherlands
J.Muskens@tue.nl, M.R.V.Chaudron@tue.nl, R.J.Bril@tue.nl

Abstract. The inconsistency allowed by architecture and design languages is a source for software engineering problems. Inconsistencies arise due to the use of multiple views. In this paper we present an approach that aids architects and designers in finding inconsistencies between different views. This approach supports intra phase consistency checking and inter phase consistency checking. Therefore the approach is suitable for detecting consistency problems between for example multiple diagrams in a UML design as well as between a design and the implementation.
The approach is based on verification of constraints and obligations that are imposed on views using relation partition algebra. The constraints and obligations are driven by the development process and therefore differ between projects. The challenge is to enable consistency checking without imposing constraints on the development process. To this end we developed a unifying approach for verification of constraints and obligations. Constraints and obligations can be imposed between views within a development phase as well as between views of different phases, independent of which view is considered to be leading, and independent on the view in which you want to see the violations. This enables consistency checking in an arbitrary process.

1 Introduction

1.1 Background

Our research was carried out in the context of the Robocop and Space4U projects[1]. The goal of these projects is the definition of a component based software architecture for the middleware layer of high volume embedded appliances. Development of robust, reliable and manageble systems is a critical issue in these projects. Our contribution to these projects focusses on terminal management and development support activities. In this paper, we consider the latter topic.

In the context of Robocop and Space4U development support consists of a number of activities. For example: automatic generation of template code for rapid development of components and systems, analysis support for extra-functional properties at design-time and consistency checking. In this paper we will discuss techniques for consistency checking at the architecture and design phases of software development.

1.2 Motivation

The purpose of our research is to investigate how we can aid architects and designers in finding inconsistencies between views. These views can be artefacts of a single development phase as well as artefacts of different phases. For example we aim to support consistency checking between different diagrams of a design and consistency checking between architecture and design (or design and implementation).

[1] These projects are funded in part by the European ITEA program and they are joint projects of various European companies, together with private and public research institutes.

UML [3] is becoming the defacto standard for software engineering projects. UML and UML case tools offer a lot of freedom. They offer the possibility to describe a system using different views [9]. For a specific view it is also possible to use different diagrams. These different diagrams have overlapping information. For example, the dependencies in a class diagram and the messages in a message sequence diagram are related. As a result these different diagrams can give rise to inconsistencies.

Architecture description, detailed design and implementation are different views on a system that also contain overlapping information. Therefore this can also give rise to inconsistencies.

Inconsistencies increase the chance of errors and complicates the management of software development [6]. There is a need for verification of consistency in and between architecture, design, and implementation. The consistency requirements depend on the development process and development practices used within each particular project. The techniques discussed in this paper can be used to check the consistency constraints for a particular project.

1.3 Overview

The remainder of this paper is structured as follows. Section 2 discusses a number of examples of inconsistency problems and how these problems can be detected using relation partition algebra. Section 3 generalizes these examples and presents an approach for detecting consistency problems based on restrictions and obligations that one diagram imposes on another. Section 4 shows how to extend the approach to enable verification of cardinalities of relations. In Section 5 we discuss the applicability of the approach followed by some conluding remarks in section 6.

2 Examples of Consistency Checks

In this section we will present a number of examples that illustrate how Relation Partition Algebra (RPA) can be used to find consistency problems. In this paper we use the following rules from RPA [2]:

$$\begin{aligned}
I &= \{<x,x> \mid \forall x\} \\
A^{-1} &= \{<y,x> \mid <x,y> \in A\} \\
A - B &= \{<x,y> \mid <x,y> \in A \wedge <x,y> \notin B\} \\
A \cup B &= \{<x,y> \mid <x,y> \in A \vee <x,y> \in B\} \\
A \cap B &= \{<x,y> \mid <x,y> \in A \wedge <x,y> \in B\} \\
A; B &= \{<x,z> \mid (\exists y : <x,y> \in A \wedge <y,z> \in B)\} \\
A^{+} &= \bigcup_{n=1}^{\infty} A^n, \text{where } A^n = A; A^{n-1} \text{ for } n \geq 2 \\
A^{*} &= A^{+} \cup I \\
A \uparrow B &\equiv B^{-1}; A; B \\
A \downarrow B &\equiv B; A; B^{-1} \\
\lceil A \rceil &= \{<x,y,1> \mid <x,y> \in A\} \\
M_2; M_1 &= \{<x,z,n> \mid n = \textstyle\sum_{<x,y,n_1> \in M_1 \wedge <y,z,n_2> \in M_2} n_1 \times n_2\}
\end{aligned}$$

The examples are based on a number of (design and implementation) views on a software game called Tic Tac Toe. They illustrate that one view imposes constraints or obligations on a different view and that there is a notion of a leading view and following view. Furthermore these examples show that there is a choice in whether you are interested in the voilations of constraints / obligations (in the following view) or the constraints / obligations that are violated (in the leading view).

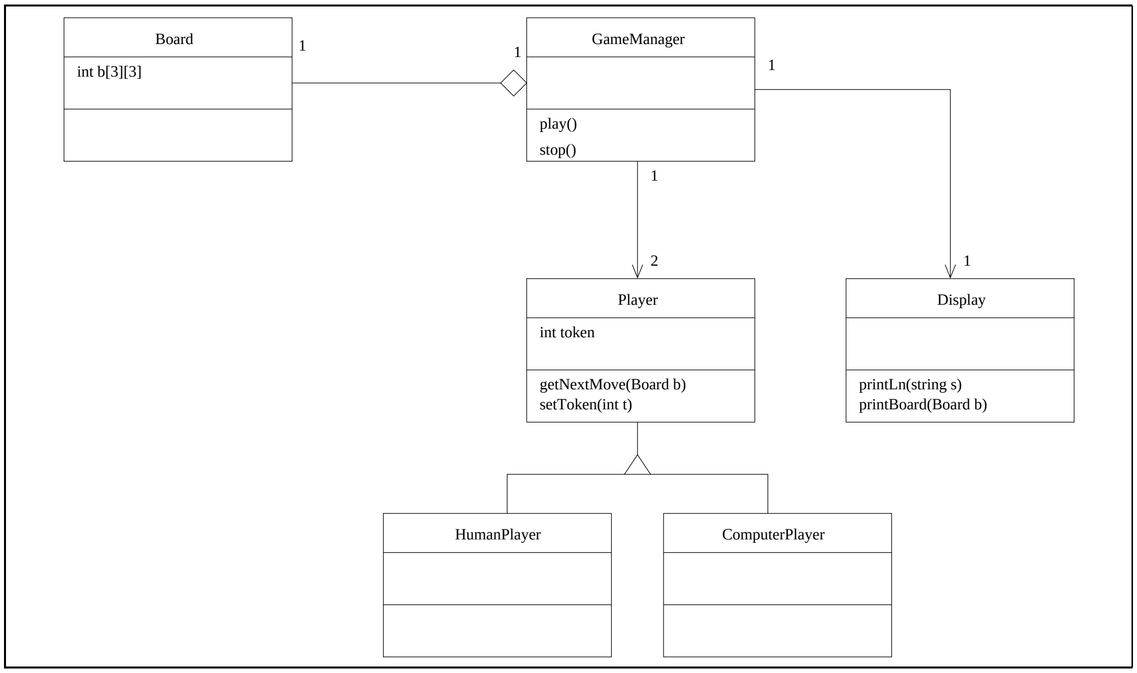

Fig. 1. Tic Tac Toe class diagram

2.1 Class Diagram and Message Sequence Diagrams

In this subsection we will illustrate how consistency between message sequence diagrams and class diagrams can be verified. For this purpose we show how to detect:

– Missing dependencies in the class diagram based on information in the message sequence diagrams (MSCs): This is an example of obligations imposed on the class diagram by the MSCs.
– Method invocations in the MSCs that are not allowed based on the class diagram: This is an example of constraints imposed on the MSCs by the class diagram.

Which of the two is appropriate depends on which view one considers to be leading. After describing the different views, we will start with the first case. In this case we assume that when the MSCs show a method invocation of an object of class x on an object of class y, then the class diagram should show a dependency between class x and class y.

Consider the following example. We are designing a game called Tic Tac Toe. The structure of the application is showed in Figure 1. The game is implemented using a GameManager that controls the input of two Players and uses a Display for the output. There are two type of players HumanPlayers and ComputerPlayers. The class Board is used to store the current game situation. The structure information of the Tic Tac Toe class diagram is represented by the following sets and relations.

$$\begin{aligned}
CLASS = \{ & \text{GameManager, Board, Player, HumanPlayer,} \\
& \text{ComputerPlayer, Display}\} \\
METHOD = \{ & \text{GameManager.play, GameManager.stop,} \\
& \text{Player.getNextMove, Player.setToken,} \\
& \text{HumanPlayer.getNextMove, HumanPlayer.setToken,} \\
& \text{ComputerPlayer.getNextMove, ComputerPlayer.setToken,}
\end{aligned}$$

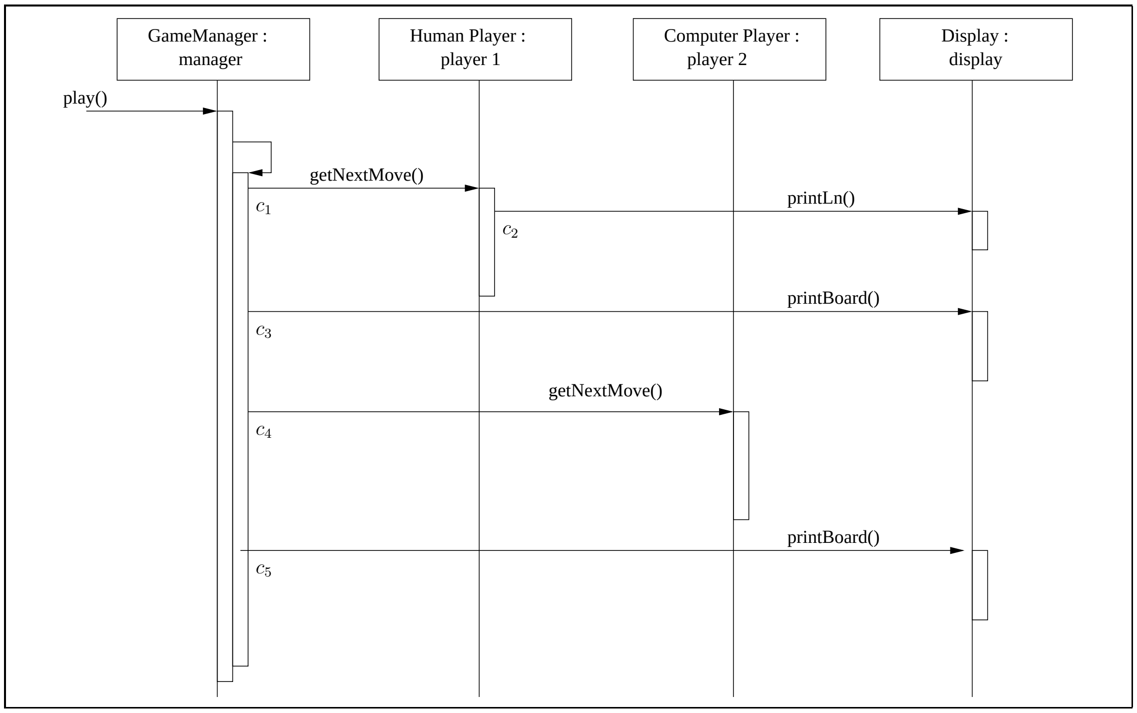

Fig. 2. Example message sequence diagram

$$\begin{aligned}
&\text{Display.printLn, Display.printBoard}\}\\
IMPLEMENTS = &\{\text{<GameManager.play,GameManager>},\\
&\text{<GameManager.stop,GameManager>},\\
&\text{<Player.getNextMove,Player>, <Player.setToken,Player>},\\
&\text{<HumanPlayer.getNextMove,HumanPlayer>},\\
&\text{<HumanPlayer.setToken,HumanPlayer>},\\
&\text{<ComputerPlayer.getNextMove,ComputerPlayer>},\\
&\text{<ComputerPlayer.setToken,ComputerPlayer>},\\
&\text{<Display.printLn,Display>},\\
&\text{<Display.printBoard,Display>}\}\\
INHERITANCE = &\{\text{<HumanPlayer,Player>,<ComputerPlayer,Player>}\}\\
DEPENDENCY = &\{\text{<GameManager,Player>,<GameManager,Display>}\}\\
AGGREGATION = &\{\text{<GameManager,Board>}\}
\end{aligned}$$

The dynamics of our game are described using a message sequence diagram (see Figure 2). There is a manager that fetches the next move of the 2 players. After each move the output is updated using the printBoard method of the display. The HumanPlayer (Player1) writes some comments to the output using the printLn method of the display, during his move. The information of the message sequence diagram is represented by the following sets and relations.

$$\begin{aligned}
OBJECT = &\{\text{manager,player1,player2,display}\}\\
TYPE = &\{\text{<manager,GameManager>,<player1,HumanPlayer>},\\
&\text{<player2,ComputerPlayer>,<display,Display>}\}\\
CALL = &\{c_1, c_2, c_3, c_4, c_5\}
\end{aligned}$$

$$\begin{aligned}
NEXT &= \{< c_1, c_2 >, < c_2, c_3 >, < c_3, c_4 >, < c_4, c_5 >\} \\
CALLER &= \{<\text{manager},c_1 >,<\text{player1},c_2 >,<\text{manager},c_3 >,<\text{manager},c_4 >,<\text{manager},c_5 >\} \\
CALLEE &= \{<\text{player1},c_1 >,<\text{display},c_2 >,<\text{display},c_3 >,<\text{player2},c_4 >,<\text{display},c_5 >\} \\
MESSAGE &= \{<\text{HumanPlayer.getNextMove},c_1 >,<\text{Display.printLn},c_2 >, \\
&\quad <\text{Display.printBoard},c_3 >,<\text{ComputerPlayer.getNextMove},c_4 >, \\
&\quad <\text{Display.printBoard},c_5 >\}
\end{aligned}$$

Next we will discuss how we can detect missing dependencies in the Tic Tac Toe class diagram based on the calls in the message sequence diagram. We assume that a dependency from (a superclass of) class x and on (a superclass of) class y must exist if an object of class x invokes a method on an object of class y. We will construct a lower bound on for the dependencies in the class diagram based on messages in the MSC. This lower bound contains the *obligations* from the MSC on the class diagram.

We construct the relation between the objects calling each other ($CALLER;CALLEE^{-1}$). This relation contains the dependencies between objects. The relation need to be 'lifted' to dependencies between classes, in order to be able to compare with the dependencies in the class diagram. Once we have identified the dependencies between classes based on the message sequence diagram we verify that all these dependencies are also in the class diagram. This last check is not completely straight forward, since we have to take inheritance into account.

$$\begin{aligned}
\textit{Rule} : ((CALLER;CALLEE^{-1}) \uparrow TYPE) \subseteq \\
(DEPENDENCY \downarrow INHERITANCE^{*})
\end{aligned} \tag{1}$$

Missing dependencies are given by $((CALLER;CALLEE^{-1}) \uparrow TYPE) - (DEPENDENCY \downarrow INHERITANCE^{*})$. In our example we can deduce that one dependency is missing in the class diagram. This is the dependency between HumanPlayer and Display. This is motivated by the following hints:

$$\begin{aligned}
< player1, display > &\in CALLER;CALLEE^{-1} \\
< HumanPlayer, Display > &\in ((CALLER;CALLEE^{-1}) \uparrow TYPE) \\
< HumanPlayer, Display > &\notin (DEPENDENCY \downarrow INHERITANCE^{*})
\end{aligned}$$

We can also investigate the second possibility. This is the case in which the class diagram imposes *constraints* on the MSCs. We assume that a method invocation of an object of class x on an object of class y is only allowed when the class diagram shows a dependency between class x and class y. The allowed dependencies between objects $((DEPENDENCY \downarrow INHERITANCE^{*}) \downarrow TYPE)$ provide an upper bound for the actual dependencies between objects as indicated in the MSCs ($CALLER \oslash CALLEE$).

$$\begin{aligned}
\textit{Rule} : (CALLER;CALLEE^{-1}) \subseteq \\
((DEPENDENCY \downarrow INHERITANCE^{*}) \downarrow TYPE)
\end{aligned} \tag{2}$$

In our example we can show that there is a dependency between player1 and display that is not allowed. This is an inconsistency that can arise because class diagrams and MSCs contain overlapping information. Eventhough MSCs are used to describe the dynamics it also implies some structural information (dependencies). Information that does not cannot cause inconsistency problems. For example, the order in which methods are invoked according to a MSC cannot cause consistency problems in the class diagram.

2.2 Class Diagram and Profile

In this subsection we discuss how we can verify a class diagram against a profile. A profile can be used to describe certain design rules. In this example the profile (design rules) is the leading

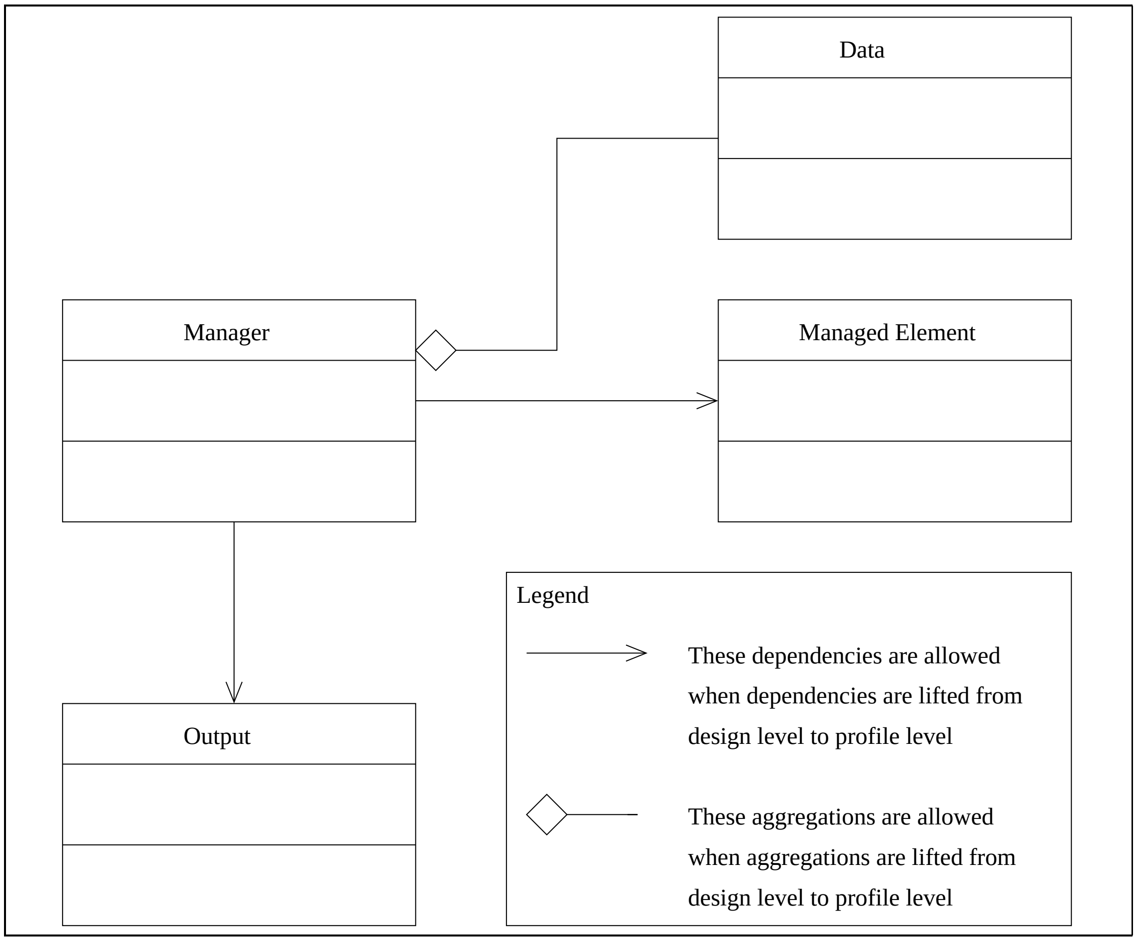
Data
Manager
Managed Element
Output
Legend
These dependencies are allowed when dependencies are lifted from design level to profile level
These aggregations are allowed when aggregations are lifted from design level to profile level

Fig. 3. Profile describing allowed dependencies between classes

diagram which imposes *constraints* on the class diagram. The following procedure can be used to verify whether a class diagram adheres to these design rules.

The procedure is illustrated based on a small example. We present a profile (see Figure 3) and verify whether the design rules expressed in the profile are obeyed in the class diagram of the Tic Tac Toe game (see Figure 1). In this case the profile describes which dependencies and aggregations are allowed to exist between classes of specific categories in the design. The information in the profile is represented by the following sets:

$$\begin{aligned} METACLASS &= \{\text{Manager, Managed Element, Data, Output}\} \\ METADEPENDENCY &= \{<\text{Manager,Output}>,<\text{Manager,Managed Element}>\} \\ METAAGGREGATION &= \{<\text{Manager,Data}>\} \end{aligned}$$

There also exists a relation between the classes in the profile and the classes in the design. This relation expresses that a class in the design is of a certain category. This information is represented by the following relation:

$$\begin{aligned} CATEGORY = \{&<\text{Board,Data}>,<\text{GameManager,Manager}>, \\ &<\text{Player,ManagedElement}>,<\text{HumanPlayer,ManagedElement}>, \\ &<\text{ComputPlayer,ManagedElement}>,<\text{Display,Output}>\} \end{aligned}$$

Next we discuss how we can detect dependencies and aggregations in the class diagram that are not allowed according to the profile. Dependencies in the class diagram that are not allowed are found by 'lowering' the allowed dependencies from the profile to the class diagram. The allowed dependencies (*constraints*) provide an upper bound for the actual dependencies that are in the class diagram.

$$Rule : DEPENDENCY \subseteq (METADEPENDENCY \downarrow CATEGORY) \tag{3}$$

The rule above identifies whether there are violations of the design rules in the Class diagram. We can find the violations as by $DEPENDENCY - (METADEPENDENCY \downarrow CATEGORY)$. The following rule is more useful if we are interested in which design rules are violated.

$$Rule : (DEPENDENCY \uparrow CATEGORY) \subseteq METADEPENDENCY \tag{4}$$

The violated design rules are given by $(DEPENDENCY \uparrow CATEGORY) - METADEPENDENCY$.

The rules (*constraints*) showed above are not violated by the class diagram of the Tic Tac Toe application. In the previous subsection we found that a dependency between HumanPlayer and Display was missing based on information in the message sequence diagram. However, adding this dependency results in violation of a design rule.

Verification that the class diagram does not contain any aggregations that violate the design rules is done in a similar fashion. The rule is shown below.

$$Rule : AGGREGATION \subseteq (METAAGGREGATION \downarrow CATEGORY) \tag{5}$$

If we are interested in which design rules are violated the following rule is more useful.

$$Rule : (AGGREGATION \uparrow CATEGORY) \subseteq METAAGGREGATION \tag{6}$$

2.3 Package Diagram and Class Diagram

In this subsection we will illustrate how the consistency between class diagrams and package diagrams can be verified. For this purpose we show how to detect missing dependencies in the package diagram based on the information in the class diagrams. In this example a class diagram imposes *obligations* on the package diagrams, the other possibility would be that the package diagram imposes *constraints* on the class diagram. The latter is not discussed in this subsection. We assume that when a class of package x has a dependency on a class of package y, then the package diagram should show a dependency between package x and y.

Consider the package structure illustrated in Figure 4. There is a Management package that contains the GameManager and Board classes. There is a Graphics package that contains the Display class. There is a Players package that contains the Player, HumanPlayer and the ComputerPlayer. Finally the package diagram shows that dependencies are allowed between classes of the Management package and classes of the Graphics package. The information above is represented by the following sets and relations.

$$\begin{aligned} PACKAGE &= \{\text{Management,Graphics,Players}\} \\ CONTAINS &= \{\text{<GameManager,Management>,<Board,Management>,} \\ &\quad \text{<Player,Players>,<HumanPlayer,Players>,} \\ &\quad \text{<ComputerPlayer,Players>,<Display,Graphics>,} \end{aligned}$$

Next we discuss how we can detect missing dependencies in the package diagram based on the information in the class diagram. Missing dependencies in the package diagram are found by lifting the dependencies between classes to the package diagram. The lifted dependencies (*obligations*) provide a lower bound for the actual dependencies in the package diagram.

$$\begin{aligned} Rule : (DEPENDENCY \downarrow INHERITANCE^{*}) \uparrow CONTAINS) \subseteq \\ \{\text{<Management,Graphics>}\} \end{aligned} \tag{7}$$

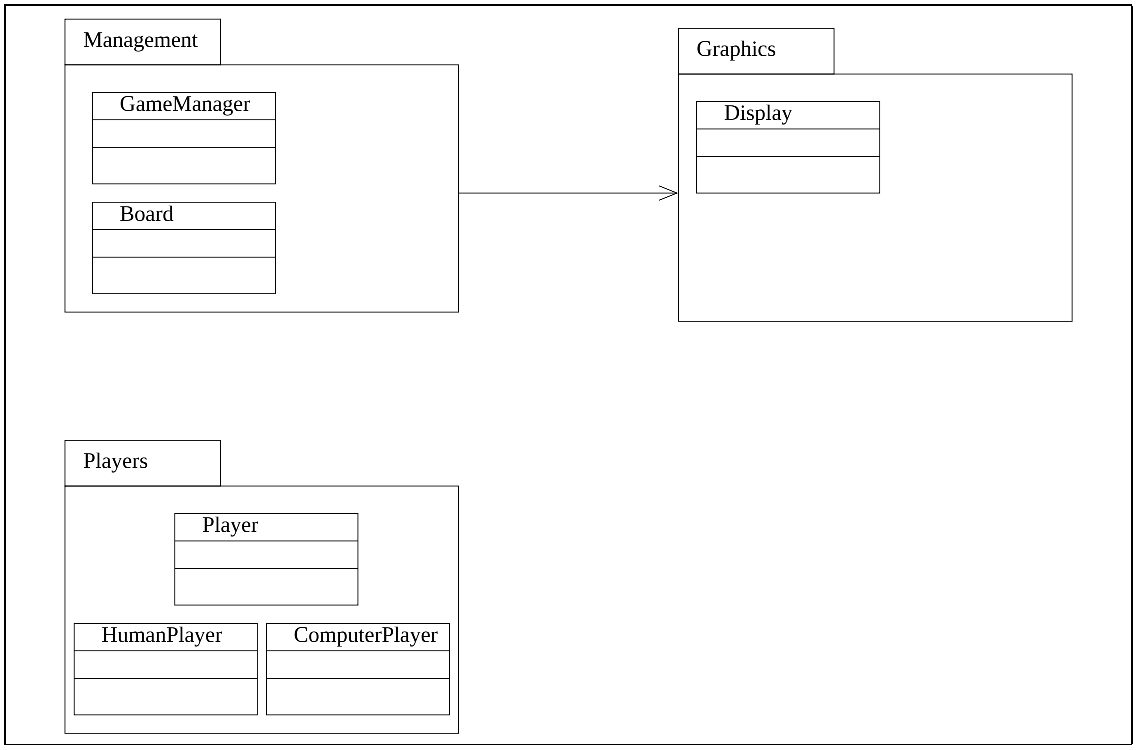

Fig. 4. Tic Tac Toe package diagram

2.4 Layering

Verification of layering in systems using relation partition algebra is already discussed in [1][10]. In this section we will briefly argue that the same reasoning can also be applied on structures of classes and packages. High level package diagrams can be used to specify the layering rules.

Consider the Tic Tac Toe example again. The layering rules are described in a high level package diagram (see Figure 5). The Graphics package will be put in the UI Layer package and the Management and Players packages will be put in the Logic Layer. The dependencies between the packages remain.

We can use the techniques described in [7] to verify whether the dependencies of sub-packages and classes obey the layering rules, as long as we don't allow inheritance to cross package boundaries.

2.5 Design and Implementation

In this subsection we will illustrate how consistency between implementation and design can be verified. This is an example of inter-phase consistency checking. For this purpose we show how to detect includes in C++ files that are not allowed according to the class diagram. This means the class diagram is leading and imposes *constraints* on the implementation.

Consider the implementation illustrated in Figure 6. The boxes represent implementation files in C++. The arrows represent includes of a header file by another file. For each class in the class diagram (see Figure 1) there are two implementation files except for the Player class. This information is represented by the following set and relations.

$$\begin{aligned} FILES = \{&\text{Board.c, Board.h, GameManager.c, GameManager.h,}\\ &\text{Display.c, Display.h, Player.h, HumanPlayer.c} \end{aligned}$$

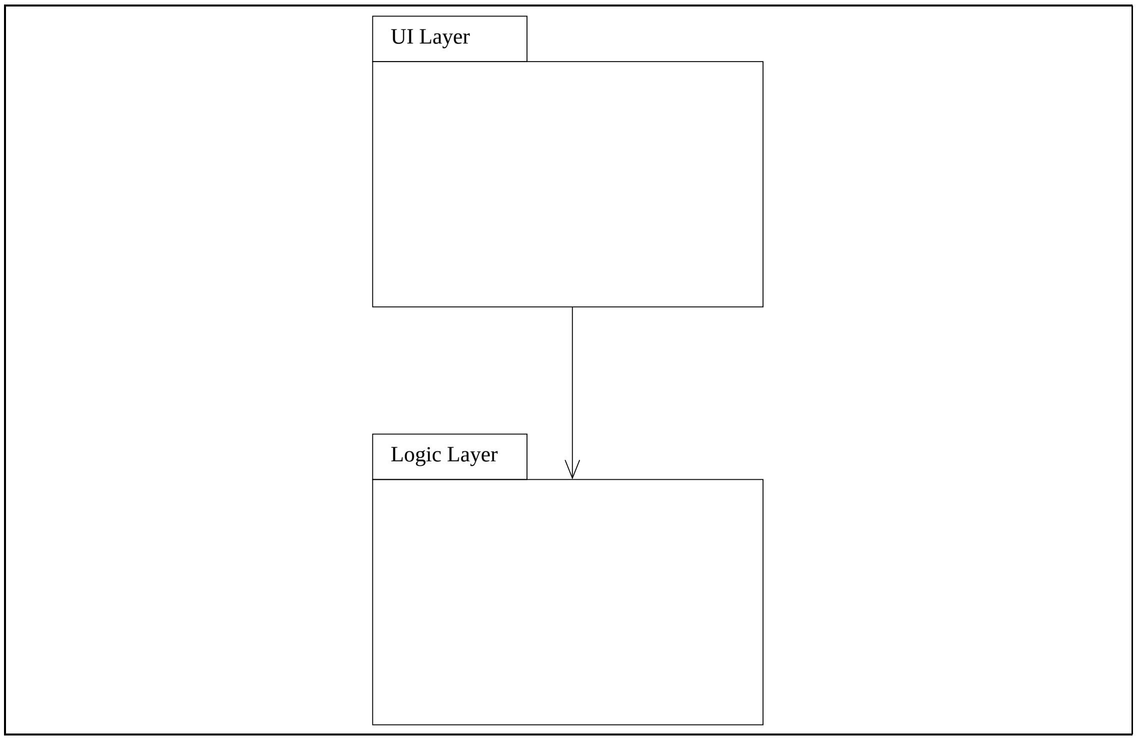

Fig. 5. Package profile

$$\begin{aligned}
& \text{HumanPlayer.h, ComputerPlayer.c, ComputerPlayer.h}\} \\
INCLUDES = & \{\text{<Board.c,Board.h>, <GameManager.c,GameManager.h>,} \\
& \text{<Display.c,Display.h>, <HumanPlayer.c,HumanPlayer.h>,} \\
& \text{<ComputerPlayer.c,ComputerPlayer.h>, <GameManager.h,Board.h>,} \\
& \text{<GameManager.h, Display.h>, <GameManager.h,Player.h>,} \\
& \text{<HumanPlayer.h, Player.h>, <ComputerPlayer.h,Player.h>}\} \\
IMPLEMENTS = & \{\text{<Board.c,Board>, <Board.h,Board>,} \\
& \text{<GameManager.c,GameManager>, <GameManager.h,GameManager>,} \\
& \text{<Display.c,Display>, <Display.h,Display>,} \\
& \text{<HumanPlayer.c,HumanPlayer>,<HumanPlayer.h,HumanPlayer>,} \\
& \text{<ComputerPlayer.c,ComputerPlayer>, <ComputerPlayer.h,ComputerPlayer>,} \\
& \text{<Player.h,Player>}\}
\end{aligned}$$

Next we discuss how we can detect includes between implementation files that are not allowed due to the constraints expressed in the design. Files that are part of the implementation of class c_1 are allowed to include:

- files that implement c_2, if c_1 depends on c_2;
- files that implement c_2, if c_1 aggregates c_2;
- files that also implement c_1 or one of the parent classes of c_1.

We construct an upper bound for the $INCLUDES$ relation. This upper bound is constructed by lowering the union of the dependencies, aggregations and the identity relation lifted with $INHERITANCE^*$.

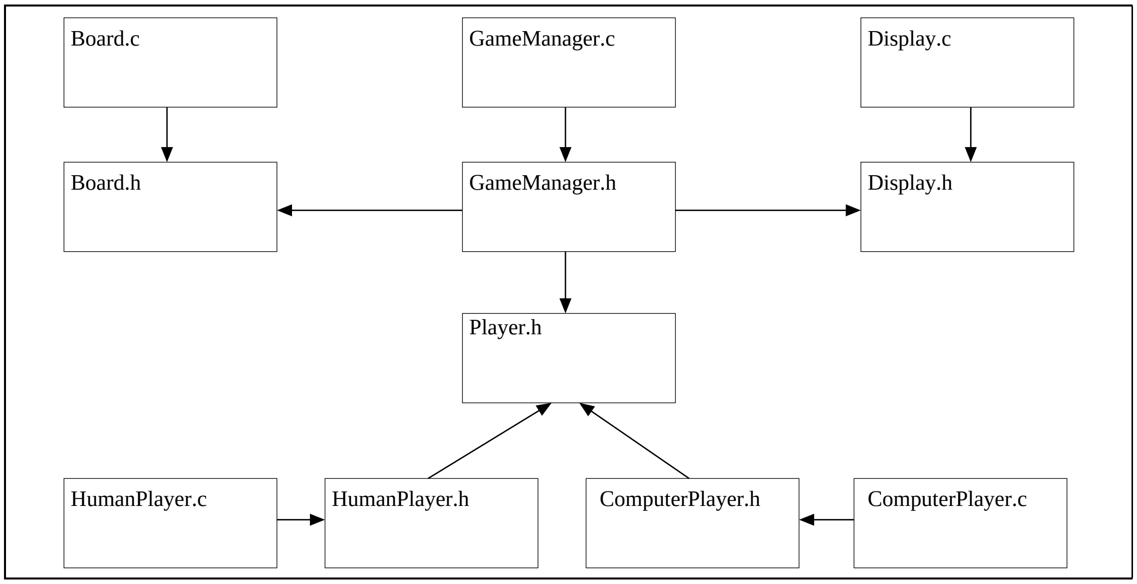

Fig. 6. Implementation of Tic Tac Toe game

$$\begin{aligned} &Rule : INCLUDES \subseteq \\ &\quad (DEPENDENCY \cup AGGREGATION \cup (I \uparrow INHERITANCE^{*})) \downarrow IMPLEMENTS \end{aligned} \tag{8}$$

The rule above is very convenient to find include relations in the implementation that violate the constraints induced by the design. In the example there are no violations. To show these violations in the design (class diagram) the following rule is convenient.

$$\begin{aligned} &Rule : INCLUDES \uparrow IMPLEMENTS \subseteq \\ &\quad DEPENDENCY \cup AGGREGATION \cup (I \uparrow INHERITANCE^{*}) \end{aligned} \tag{9}$$

3 A General Approach to Consistency Checking

In the previous section we showed a number of examples that illustrated how relation partition algebra can be used to verify consistency between different views. All these examples followed the same pattern. In this section we show that this pattern can be generalized to an approach that can be applied on a large variety of consistency problems.

Consistency problems arise due to the usage of multiple views when describing a software design. Relations between elements in one view impose constraints or obligations on elements and their relations in other views.

One problem when looking at consistency is to determine which view is leading. For example, a message sequence diagram indicates there is a dependency between two classes and this dependency is not shown in the class diagram. Which diagram is wrong? In UML there is no notion of a leading diagram. Different design processes give rise to different sequences in which UML diagrams are created, and also which diagram is considered to be the most important. We feel that the selection of the leading view should be determined by the design process instead of being prescribed by the design support tools.

Once we have determined which view is leading we need to identify the rules for the constraints and/or obligations in the other view. The different approaches used for constraints and obligations are discussed in the following subsections.

3.1 Conventions

Let X and Y be two views of software development artefacts. View X contains a set of elements X, which are related. This relation is represented by R_X. View Y contains a set of elements Y, which are related. This relation is represented by R_Y. We assume the views X and Y are related due to the fact that a mapping M_{YX} exists between Y and X. M_{YX} is a part of relation meaning that it is *functional* and *acyclic*.

$$functional : (< x, y >\in M_{YX} \wedge \ < x, z >\in M_{YX}) \Rightarrow y = z$$
$$acyclic : M^{+}_{YX} \cap I = \emptyset$$

3.2 Rules for constraints

There are two possibilites. View X is leading or view Y is leading. First we assume view X is leading. We want to express that the relation R_Y is constrained by relation R_X (we will use R^{con}_X to express that this relation contains the constraints). For example: $< y_1, y_2 >$ is only allowed to be an element of R_Y if and only if $< x_1, x_2 >\in R^{con}_X$, $< y_1, x_1 >\in M_{YX}$, and $< y_2, x_2 >\in M_{YX}$. This rule can be expressed as follows:

$$Rule : R_Y \subseteq R^{con}_X \downarrow M_{YX} \tag{10}$$

The rule above is very convenient to find the violations in Y. The violations in Y are given by $R_Y - R^{con}_X \downarrow M_{YX}$. The next rule is very convenient to find the constraints in X that are not met.

$$Rule : (R_Y \uparrow M_{YX}) \subseteq R^{con}_X \tag{11}$$

The following table shows how to relate these general rules to the examples of section 2. The instantiations of the first row can be used to get rule (2), and the second row to get rule (3) and (4).

X	R^{con}_X	Y	R_Y	M_{YX}
$CLASS$	$DEPENDENCY$	$OBJECT$	$(CALLER; CALLEE^{-1})$	$TYPE$
$METACLASS$	$METADEP.$	$CLASS$	$DEPENDENCY$	$CATEGORY$
$LAYERS$	$ALLOWED$	$PACKAGE \cup CLASS$	$DEPENDENCY$	$CONTAINS$

Table 1. Example of applicability of constraints where view X is leading

Next we discuss the second possibilty. We assume view Y is leading. We want to express that relation R_X is constrained by R_Y (we use R^{con}_Y to express that this relation contains the constraints). For example: $< x_1, x_2 >$ is only allowed to be an element of R_X if there exists an $< y_1, y_2 >\in R_Y$, $< y_1, x_1 >\in M_{YX}$, and $< y_2, x_2 >\in M_{YX}$. This rule can be expressed as follows:

$$Rule : R_X \subseteq R^{con}_Y \uparrow M_{YX} \tag{12}$$

The rule is very convenient to find the violations of the constraints in X. These violations are given by $R_X - R^{con}_Y \uparrow M_{YX}$. Finding the constraints in Y that are not met is not possible. The constraints expressed in Y imply that for all elements $< x_1, x_2 >$ of R_X there should be at least one element in R_Y that 'allows' $< x_1, x_2 >$:

$$\forall_{<x_1,x_2>\in R_X} : (\exists_{<y_1,y_2>\in R_Y} :< y_1, x_1 >\in M_{YX} \wedge < y_2, x_2 >\in M_{YX})$$

The problem is that R_Y^{con} cannot be used as an upper bound for $R_X \downarrow M_{YX}$ because this would be equivalent to the following expression:

$$\forall_{<x_1,x_2>\in R_X} : (\forall_{<y_1,y_2>\in R_Y} :< y_1, x_1 >\in M_{YX} \wedge < y_2, x_2 >\in M_{YX})$$

3.3 Rules for obligations

There are two possibilities. View X is leading or view Y. First we assume view X is leading. We want to express that the relation R_Y has some obligations due to relation R_X (we will use R_X^{obl} to express that this relation contains obligations). For example: $< y_1, y_2 >$ needs to be an element of R_Y if $< x_1, x_2 >\in R_X^{obl}$, $< y_1, x_1 >\in M_{YX}$, and $< y_2, x_2 >\in M_{YX}$. This rule can be expressed as follows:

$$Rule : R_X^{obl} \subseteq (R_Y \uparrow M_{YX}) \tag{13}$$

The rule is very convenient to find the obligations of view X that are not met. These obligations are given by $R_X^{obl} - (R_Y \uparrow M_{YX})$. Finding the violations in Y is not possible for the same reason that it was not possible to find the voilations of constraints on X expressed in Y (see previous subsection).

Next we discuss the second possibility. We assume view Y is leading. We want to express that relation R_X has some obligations due to relation R_Y (we will use R_Y^{obl} to express that this relation contains obligations). For example: $< x_1, x_2 >$ needs to be an element of R_X if $< y_1, y_2 >\in R_Y^{obl}$,$< y_1, x_1 >\in M_{YX}$, and $< y_2, x_2 >\in M_{YX}$. This rule can be expressed as follows:

$$Rule : R_Y^{obl} \subseteq (R_X \downarrow M_{YX}) \tag{14}$$

The rule above is very convenient to find the obligations expressed in view Y that are not met. The following rule is convenient to find the missing relations in X.

$$Rule : (R_Y^{obl} \uparrow M_{YX}) \subseteq R_X \tag{15}$$

The following table shows how to relate these general rules to the examples of section 2. The instantiation of the first row can be used to get rule (1), instantiation of the second row can be used to get rule (7).

X	R_X	Y	R_Y^{obl}	M_{YX}
$CLASS$	$DEPENDENCY$	$OBJECT$	$(CALLER; CALLEE^{-1})$	$TYPE$
$PACKAGE$	<Management,Graphics>	$CLASS$	$DEPENDENCY$	$CONTAINS$

Table 2. Example of applicability of obligations where view Y is leading

3.4 Exceptions

In this subsection we discuss how to deal with exceptions on rules. Exceptions can be made for several reasons. Exceptions can be made to get more fine grained constraints and obligations. For example you want to have a strict layering except for one specific component that is allowed to by-pass some layers. These exceptions can be permanent, for example due to performance reasons. Exceptions can also be made temporarily. This often happens due to time pressure, because there is no time to solve the violations.

Exceptions can be made for constraints and obligations. Consider we have the following rule for constraints on view Y imposed by view X.

$$Rule : R_Y \subseteq R_X^{con} \downarrow M_{YX} \tag{16}$$

We want to make exceptions for some relations in view Y. The exceptions on the constraints are elements of E_Y. The constraints of view X are lowered to view Y. The relations in view Y minus the exeptions should be a subset of the lowered constraints. This is expressed by the following rule.

$$Rule : R_Y - E_Y^{con} \subseteq R_X^{con} \downarrow M_{YX} \tag{17}$$

We express execeptions in view Y. Expressing exceptions in view X results in the following rule.

$$Rule : R_Y \subseteq (R_X^{con} \cup E_X^{con}) \downarrow M_{YX} \tag{18}$$

The rule above is sub-optimal since $E_Y^{con} \subseteq E_X^{con} \downarrow M_{YX}$. Table 3 shows in which view to specifiy the exceptions based on the leading view and whether we deal with constraints or obligations.

	constraints	obligations
X is leading	Y	X
Y is leading	X	Y

Table 3. Where to define the exceptions

3.5 Overview

In this section we discussed how to express rules that can be used to verify consistency between views. We identified that the following aspects influence the rule that needs to be used:

– Do we want to express *constraints* or *obligations*?
– Which view is leading?
– In which view do we want to see the violations?

For each of the three questions above there are two possibilities. As a result there are 2^3 possible rules, in two cases this rule cannot be found. The resulting 6 rules are shown in Figure 7. Some of these rules are the same. We observe that the rule for finding violations on constraints from view X on view Y is the same as finding violations for obligations from view Y on view X and vice versa.

Essentially consistency checking comes down to verification of constraints and obligations imposed from one view on another. Constraints provide an upper bound for the relations in a view and obligations a lower bound.

$$obligations \subseteq relations \subseteq constraints$$

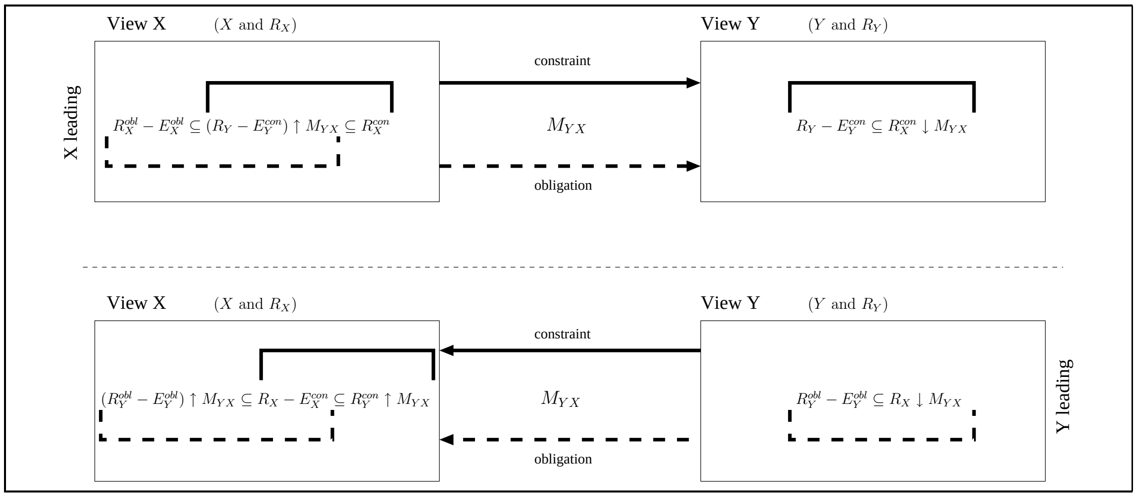
View X
$(X \text{ and } R_X)$
View Y
$(Y \text{ and } R_Y)$
X leading
constraint
$R_X^{obl} - E_X^{obl} \subseteq (R_Y - E_Y^{con}) \uparrow M_{YX} \subseteq R_X^{con}$
M_{YX}
$R_Y - E_Y^{con} \subseteq R_X^{con} \downarrow M_{YX}$
obligation
View X
$(X \text{ and } R_X)$
View Y
$(Y \text{ and } R_Y)$
constraint
$(R_Y^{obl} - E_Y^{obl}) \uparrow M_{YX} \subseteq R_X - E_X^{con} \subseteq R_Y^{con} \uparrow M_{YX}$
M_{YX}
$R_Y^{obl} - E_Y^{obl} \subseteq R_X \downarrow M_{YX}$
Y leading
obligation

Fig. 7. Overview of rules for verification of obligations and constraints

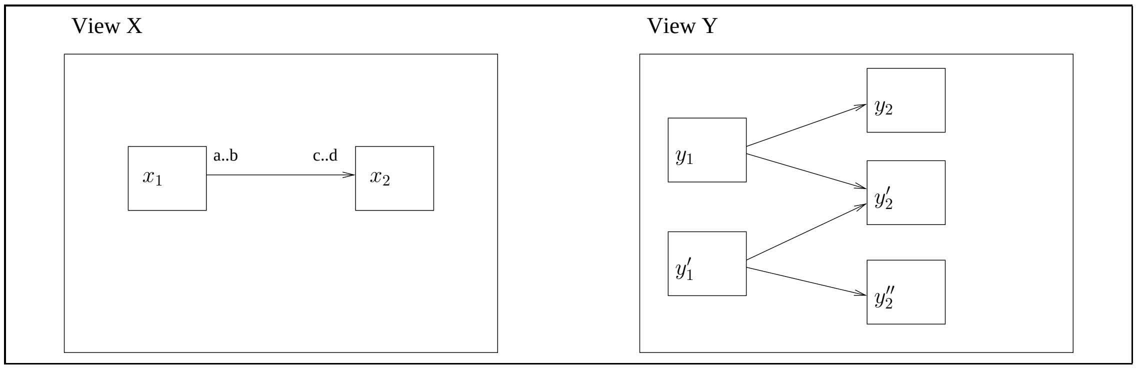

Fig. 8. Example of constraints and obligations with cardinalities

4 Cardinality Based Consistency Checking

In this section we briefly discuss how to extend our approach for consistency checking with cardinalities. The approach presented in section 3 provides a way to express whether relations between certain elements must exist or are not allowed to exist. Next we will illustrate how to indicate that the number of relations between certain elements is in a specific range.

We assume there are 2 views, view X and view Y (see Figure 8). View X contains a set of elements X. These elements are related, this is represented by relation R_X. View Y contains a set of elements Y. These elements are related, this is represented by relation R_Y. The views X and Y are related due to the fact that a relation M_{YX} exists between Y and X.

$$\begin{aligned} X &= \{x_1, x_2\} \\ R_X &= \{< x_1, x_2 >\} \\ Y &= \{< y_1, y_1', y_2, y_2', y_2'' >\} \\ R_Y &= \{< y_1, y_2 >, < y_1, y_2' >, < y_1', y_2' >, < y_1', y_2'' >\} \end{aligned}$$

View X is leading and we want to express that there are some constraints and obligations on view Y. These constraints and obligations are based on the cardinalities of the relations in R_X. Each relation $< x_1, x_2 >$ in the leading view X imposes constraints as well as obligations on the relations in view Y. The following rule shows the constraints and obligations.

$$\begin{aligned} Rule : \forall < y_1, x_2, n > \in \lceil R_Y \rceil ; \lceil M_{YX} \rceil : n \in [c..d] \\ \forall < x_1, y_2, m > \in \lceil M_{YX}^{-1} \rceil ; \lceil R_Y \rceil : m \in [a..b] \end{aligned} \tag{19}$$

5 Discussion

The techniques presented in this paper can be applied in different development processes and for different consistency problems. It can be applied in a top-down approach where a high level diagram is refined in more detailed ones and the high-level diagram imposes constraints on the more detailed ones and also in a bottom-up approach where detailed diagrams impose obligations on the high-level diagrams. We notice that one of the strenghts of UML and its tools is flexibility. Little consistency is enforced by UML case tools and this is one of the reasons why they are able to support the large number of different development processes. We also notice that there is a need for consistency checking and that the required checks highly depend on the development process and practices.

This paper describes how, probably project specific, consistency checks can be specified and verified using relation partition algebra. We offer a unifying approach for verification of constraints and obligations imposed on diagrams. The result is a way to verify consistency without imposing constraints on the development process. Consistency can be verified within a development phase (intra development phase consistency), for example by checking consistency between two views in a design. Consistency can also be verified beteen different development phases (inter development phase consistency), for example checking consistency between architecture and design or between design and implementation.

6 Concluding Remarks

6.1 Related Work

Relation Algebras are used for software manipulations and analysis. Since 1994, when Relation Partition Algebra (RPA) was defined at Philips [2] it has been applied in various areas of software architecture analysis. RPA has been used to express software-related metrics as well as for dedicated analyses, e.g. detecting cyclic dependencies [1], reverse architecting [7] [8], and verification of module architectures in component based systems [10]. Also Holt [4] [5] suggests to use a Relation Algebra (Tarski Algebra) as a theoretical basis for software manipulations.

The idea of imposing constraints on a design is also used in the work on UML profiles [11]. In our approach we consider the architecture profile to be a leading view that imposes constraints on a other views.

6.2 Contributions

Relation partition algebra has been used to verify constraints on relations between entities based on their containing entities. Typical examples are constrainst from sub-systems toward components and from components towards files. Usually these techniques are used to verify constraints from early phases in the development process on the later phases.

Our contributions consist of a unifying approach for verification of constraints as well as *obligations*. The techniques that have been used to verify inter development phase consistency (architecture vs. design) can also be used for intra development phase consistency (different views in a design).

In our approach the constraints and obligations are not imposed based on the development support tools and / or languages. Constraints and obligations are imposed by the development process and yield "leading" and "following" views. This results in a situation where the containing view is not necessarily leading.

In order to be applicable in industrial development projects, the approach supports making exceptions on constraints and obligations. Specific violations can be allowed for example due to performance (structural) issues or time pressure (temporal). In the latter case it is interesting to check whether the number of violations has increased since the previous consistency check. This is supported by the approach, making exceptions for all the old violations during the consistency checking will result in only the new violations.

6.3 Conclusions

Consistency can be verified during the development process using relation partition algebra. In this paper we have presented a unifying approach for verification of consistency (constraints and obligations) between views. We noticed that constraints and obligations are driven by the development process more than the development support tools and languages. The strength of currently used architecture and design description languages (like UML) is that they are very free and flexible, therefore it does not constrain the development process. This freedom enables the introduction of inconsistencies in a design. Our approach enables verification of consistency using constraints and obligations that are suitable within a specific development process. The latter is possible since only constraints and obligations that are relevant are used and we select the leading views in alignment with that process.

References

1. L. Feijs, R. Krikhaar, and R. van Ommering. A relational approach to support software architecture analysis. In *Software - Practice and Experience, 28(4)*, pages 371–400, Apr. 1998.
2. L. Feijs and R. van Ommering. Theory of relations and its applications to software structuring. In *Philips internal report*, 1995.
3. M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition.* Addison-Wesley, 2003.
4. R. Holt. Binary relation algebra applied to software architecture. Technical Report CSRI Technical Report 345, Computer Systems Research Institute, 1996.
5. R. Holt. Structural manipulations of software architecture using tarski relational algebra. In *Proceedings of Fifth Working Conference on Reverse Engineering, IEEE Computer Society*, pages 210–219, 1998.
6. Z. Huzar, L. Kuzniarz, G. Reggio, J. Sourrouille, and M. Staron (editors). *Proceedings of Workshop on Consistency Problems in UML-based Software Development II.* Oct. 2003.
7. R. Krikhaar. Reverse architecting approach for complex systems. In *International Conference on Sofware Maintenance (ICSM'97)*, pages 4–10, Oct. 1997.
8. R. Krikhaar. *Software Architecture Reconstruction, Ph.D. Thesis.* University of Amsterdam (UvA), 1999.
9. P. Kruchten. The 4+1 view model of architecture. In *IEEE Software, Vol.12 No. 6*, pages 42–50, Nov. 1995.
10. A. Postma. A method for module architecture verification and its application on large component-based systems. In *Information and Software Technology, Vol. 45 (4)*, pages 171–194, Mar. 2003.
11. P. Selonen and J. Xu. Validating UML models against architectural profiles. In *Proceedings of the 9th European Software Engineering Conference*, pages 58–67. ACM Press, 2003.