Decentralised Process Enactment

Ulf Leonhardt Anthony Finkelstein
Jeff Kramer Bashar Nuseibeh

Technical Report 95/5

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, UK

February 2, 1995

Abstract

The ViewPoints framework for distributed and concurrent software engineering provides an alternative approach to traditional centralised software development environments. We investigate the use of decentralised process models to drive consistency checking and conflict resolution in this framework. Our process models use pattern matching on local development histories to determine the particular situation (state) of the development process, and employ rules to trigger situation-dependent assistance to the user. We describe how communication between such process models facilitates the decentralised management of explicitly defined consistency constraints in the ViewPoints framework.

1 Introduction

Software engineering processes usually involve the participation of a number of people. The more people are involved, the more important becomes the collaboration and communication between the individuals. The different participants will have different views on and assumptions about the problem domain. This necessitates organised interaction including conflict detection and resolution.

Synchronisation and conflict resolution are most easily tackled by adopting the notion of central coordination, often paired with a central data repository. However, centralised control and data storage in a conceptually concurrent and distributed context are problematic when it comes to performance, reliability and flexibility. In most systems, these centralised control mechanisms are used to check and enforce consistency whenever possible. It has been recognised that such an *eager* approach does not adequately reflect the needs of concurrent and distributed software engineering processes [4].

An alternative is the decentralisation of data storage and consistency control. As a consequence, conflict detection and resolution have to be made based on interaction and local, and thus partial, knowledge about the system. The 'eager' approach to conflict detection and resolution discussed above is not viable as the complexity introduced by the distribution of control makes it too expensive. Tolerating inconsistencies is often desirable in order to avoid unnecessary restrictions on the development process [16]. Consequently, the focus is shifted from avoidance to management of inconsistencies.

Inconsistencies arise from different views and assumptions which interfere. They also indicate the need for further action by the participants in order to achieve consent on the matter in question. By addressing this issue explicitly (for example [23]), we can devise and support more

sophisticated models of cooperation and communication among the members of the development team (see also [10]). A *'lazy'* approach to consistency detection and enforcement can be taken: synchronise whenever necessary. However, inconsistency management is a complex task. Local agents have to decide what checks to invoke, when to invoke them, and how to keep track of the results. Process support therefore becomes even more crucial.

In this paper we show how fine-grained, decentralised process models can be used to drive conflict detection and resolution. These models are used to guide the developer rather than automate the development process. We describe how the process models initiate and monitor consistency checks in order to gain knowledge about the system under development. These consistency checks are the prime means of coordination for the development of such a system.

The constraint relations that are computed by consistency checks need to be derived from logically centralised notions of consistency. Therefore we also discuss how these constraints can be expressed, and how a correspondence between global and local constraints can be established. As a framework for this work we use the ViewPoints approach which has been described in earlier papers [12, 17, 21, 23].

We start our discussion with a brief account of the ViewPoints framework (section 2), followed by a "motivating example" (section 3). Then we describe our process modelling approach (section 4) and its application to decentralised inconsistency management (section 5). In a scenario walk-through at the end of this paper (section 6) we outline how our models of a concurrent software engineering process can guide the human agents involved in such a process.

2 ViewPoints

ViewPoints are the building blocks of our framework for supporting distributed software engineering. Each ViewPoint contains an artefact of the development process (for example, a partial specification) together with a thread of development activities concerning this artefact. These are locally managed, and can be characterised as a collection of loosely coupled objects that encapsulate partial knowledge.

A ViewPoint contains knowledge about the notation, tools and strategies it supports—'method knowledge'; and the results of the application of that knowledge—'specification knowledge'. A ViewPoint is structured into the following 'slots':

Style contains the representation scheme in which the partial specification contained in a ViewPoint is represented.

Work Plan contains a process model specifying what the ViewPoint user can do and how the user should do it.

Domain specifies the part of the modelled system described by the ViewPoint.

Specification contains the result of the method user's activities, i.e. the partial specification.

Work Record stores current status, history and rationale of the ViewPoint's development process. It contains a trace of all the actions in the ViewPoint's development history.

We use partially instantiated ViewPoints, ViewPoint templates, to specify method knowledge in a reusable way. Thus specific development methods can be implemented as a collection of ViewPoint templates.

2.1 Implementation

We have developed a prototype implementation of automated support for the framework called **The Viewer** [21], which has been extended following collaboration with Hewlett-Packard and Siemens [3, 13, 14]. **The Viewer** supports both method design and method use and is therefore both CASE and MetaCASE tool. Template sets supporting methods for requirements engineering and distributed systems design have been implemented [26, 18].

3 A Scenario

Our scenario is structured into five steps which, we believe, highlight some of the important issues in concurrent software engineering (see also [9]).

We use data flow diagrams as an example notation. In such a diagram, a node in a graph may be decomposed in a separate diagram. For such a diagram hierarchy we wish to ensure that decomposition diagrams exist for composite nodes (constraint 1), and that the contextual data flows are the same for nodes and their respective decomposition diagrams (constraint 2). Constraint 1 specifies syntactic completeness, constraints 2 specifies a notion of agreement.

Composite nodes are shaded grey, primitive nodes are white. We adopt the convention that the domain of each ViewPoint denotes the node of the parent diagram of which it is a decomposition. Only if the domain of the ViewPoint is labelled *top*, a parent node does not exist. In this case, the ViewPoint contains the root node of the data flow diagram hierarchy. The Work Record of the ViewPoint lists the last seven events in the development history of the ViewPoint (the full history is stored).

In this example, the following constraints on data flow diagrams form the basis of the consistency checks (for a formal specification see appendix). We will ignore in-ViewPoint constraints and checks.

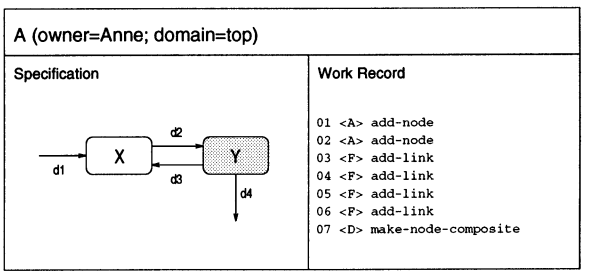

Figure 1: A ViewPoint with a simple data flow diagram in its Specification slot, and a development history listed in its Work Record. ViewPoint A contains the non-primitive node Y for which no corresponding decomposition ViewPoint exists (step 1).

Step 1 The owner of ViewPoint A, Anne, has developed a top-level data flow model of the system. She has flagged node Y for further decomposition, thus violating global constraint 1 (Figure 1). Subsequently, Anne assigns the responsibility to decompose Y to Bob.

Step 2 Bob creates a new ViewPoint B the domain of which indicates that it is a decomposition of Y. As the new ViewPoint initially contains an empty specification. Global constraint 2 is not satisfied (Figure 4).

Step 3 Bob continues developing ViewPoint B by creating a data flow diagram that decomposes Y. When he has finished both ViewPoints satisfy all local and global constraints (Figure 5).

Step 4 Now, Anne and Bob concurrently develop their ViewPoints as their understanding of the target system increases. Anne adds an output to Y (d7). Bob does the same but uses a different label (d6). He also adds another output (d9) and renames a third (d4-d8). The result again violates global constraint 2.

Step 5 While decomposing X Anne realises that its interaction with Y is much more complex than expected. Anticipating the need to restructure ViewPoint A Anne merges the decompositions

of X and Y deleting both. Consequently, constraint 1 no longer holds, because the result of the merge is neither a decomposition of X nor Y.

In each of the above steps constraints are temporarily violated. Therefore, there is a need to tolerate such constraint violations in a concurrent development process. In this context, constraints can only be checked and enforced at certain points. Hence consistency must be established by organising the application of the different checks and monitoring their result. As steps 4 and 5 show, this consistency management may be difficult, even in a such a simplified example. Consequently, guidance to users regarding the invocation of consistency checks is necessary. Such guidance must also be tailored to the development method used. In step 4, for example, more frequent checking may be required in order to avoid the accumulation of inconsistencies.

The results of previous consistency checks and all local development activities are stored in individual Work Record slots (see Figure 5, for example). Clearly, this knowledge must be taken into account when deciding when to invoke particular consistency checks.

In the following sections we describe a process modelling approach that addresses these issues. In section 6 we then apply this framework to the scenario presented above.

4 Decentralised Process Modelling

In line with the ViewPoints approach, the process modelling framework must support multiple, loosely coupled process models. At run-time there will be no explicit representation of the global process and hence no global coordination. However, it may be necessary for the method designer to "derive" the local process models from a global model, or to verify certain properties by integrating all local models into a global one.

In this section, we introduce techniques for fine-grained, local process modelling [22] in order to address some of the issues outlined above. We then discuss, how cooperation between process models and other global objectives may be achieved in this context.

4.1 Fine-grained local process models

We believe that enactable, fine-grained process models need to address the following issues:

- *Identifying* the current state of the process.
- *Deciding* what course of action is appropriate—taking into account the state of the process.
- *Enacting* the decisions made in the process.

The following sections describe our approach to solving these problems.

4.1.1 Process state

Our process models conceptually operate on the *state* of the process. Here, the term *state* denotes a view of the process which is organised according to a pre-defined schema. In our framework, the state information is structured into a set of functions and predicates which can be accessed by other components of the process model during its execution.

We have identified two different ways to implement these *observer* functions and predicates[1]: state machines, and "stateless" observation procedures. In the latter, the function or predicate is evaluated whenever it is used by the process model, it does not need to "remember" anything that has happened in the past. In contrast, if the predicate is implemented as a state machine, only events in the process trigger changes of the predicate's value.

Since each of these predicates only monitors a limited aspect of the actual process the state explosion problem can be avoided by using a number of concurrent observer predicates. These state machines may also have an infinite number of states. Here, however, we only use finite state machines in order to keep the representation of our process models clear and simple.

[1] Subsequently, we shall use the term *observer predicate* to refer to both observer functions and predicates

We use *regular expressions* as a concise and easy–to–handle notation to represent finite state machines (see [2]). Thus, we can make use of a variety of efficient and powerful tools for regular expression handling that are readily available in many programming environments.

Regular grammars define the notion of well-formed input words over some language. For each regular grammar a finite state machine can be constructed that decides whether a given sequence of input characters (word) is well-formed. Thus a regular expression defines an acceptor automaton.

This principle can be applied to ViewPoints by using the sequence of actions and events stored in the *Work Record* as input words for such acceptor automata. Essentially, this amounts to regular expression matching over the development history of a ViewPoint. This process can also be viewed as looking for known patterns of activity in the past of a ViewPoint. The value of an observer predicate associated with a grammar will therefore indicate whether or not the pattern of activity defined by the grammar has been be recognised.

4.1.2 Making decisions

The observer functions and predicates define a discrete and finite set of states for the process.

In this setting, we call the mapping of the current state into a course of action a *decision*[2]. Typically, a specific course of action will be appropriate not only for one state but for a set of similar states. We call such a set of states a *situation*. Situations can be defined by logical propositions built from the observer predicates and functions described above.

To express "decision knowledge" in our process modelling framework, we use rules of the general form

```
<situation><response>
```

The *situation* forms the pre-condition of the rule. That means, the rule fires whenever the current state matches the situation described. *Response* specifies what course of action should be taken as result of the decision made. An extension to these rules would be the addition of post-conditions in the MARVEL-style [5] to support planning activities.

4.1.3 Enaction

We distinguish three different types of such responses in decision rules:

- *Informal Guidance.* Here, we assist the user by displaying help texts, video clips, etc. Typically, such assistance would be given in complex and difficult situations.
- *Precise Recommendations.* Specific actions are recommended to the user. In this case, the user is asked to select an action from a limited number of choices. Usually, this applies to well-structured decision problems.
- *Automatic Execution of actions.* This should only occur if the correctness of the decision is reasonably certain and acceptable to the user.

4.1.4 Local architecture

Figure 2 shows how local process models for ViewPoints are structured, and how they interact with other components of the ViewPoint. We use the event trace from the work record (shown in abbreviated form as a sequence of tokens) to feed the acceptor automata defined by regular expressions. These, together with the other observer predicates, are matched against the preconditions of the decision rules. If a rule fires, the reaction is enacted on the process.

[2] The terms *decision* and *situation* are a variation of the NATURE process meta-model terminology [15], although defined in a different framework.

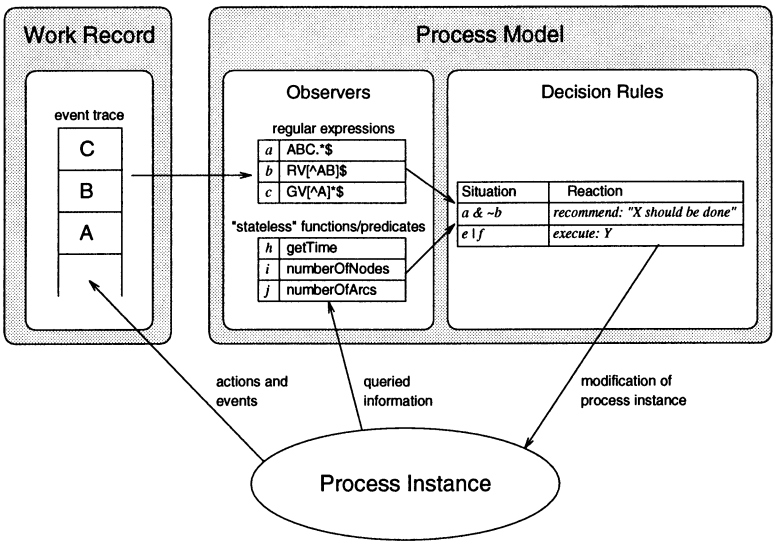

Figure 2: Local architecture

4.1.5 Notation

Our process models consist of *tests* (that is, acceptor automata defined by regular expressions), and *rules* (mapping situations into reactions).

Here, you see the definition of a simple test:

T_A:	`.*D[^R]*$`	not-successfully-checked-since-D

A test has a short and a long name which enclose a regular expression[3]. In this example, it matches if a `D`-event but no subsequent `R`-event can be found in the local Work Record of the ViewPoint. `D` and `R` are abbreviations for actions or communication events which are defined in the Work Plan.

Rules map a situation into a response to the environment.

Rule:	R_1
Situation:	$T_A \wedge \neg T_B \wedge \neg T_C$
Response:	recommend: `child-exist-check`

The pre-condition, named *situation*, is a logical proposition using tests defined in the process model. Additionally, method specific predicates may also be available in individual templates. The *response* part of the rule describes what should be done when the rule fires. The commands `display`, `recommend`, and `execute` are available to describe such responses.

4.2 Communication between process models

We believe that an implementation of the ViewPoints framework can be built on top of a communication system that supports asynchronous message-passing. Therefore we used message-passing between ViewPoints as the basic communication mechanism in the framework[4].

In this setting, ViewPoints can asynchronously send messages to and receive messages from other ViewPoints. Messages sent and received are important events in the life of a ViewPoint. Hence these events are recorded in the Work Record, making communication visible to the local

[3] We assume familiarity with the basic constructs of regular expressions as used by `Lex` [6].

[4] We assume some underlying reliable point-to-point communication medium.

process model. From the perspective of the communicating process models, a message passing transaction consists of two phases:

1. The source ViewPoint executes an action that sends a message to the destination ViewPoint. The action is appended to the Work Record of the source ViewPoint. Since such an action can be executed automatically or at least recommended to the user, a process model in this context can initiate communication transactions.

2. Upon receipt, an incoming message is automatically appended to the Work Record of the destination ViewPoint. No other processing is necessary. As a result, the local process model at the receiving end sees a message on the Work Record and can react accordingly.

We can also use this basic message-passing scheme to build other, more sophisticated communication and cooperation protocols (two-phase locking, for example). In section 5.2 we describe a protocol for two-party consistency checking using message-passing between ViewPoints.

5 Managing Consistency

In this section we describe the application of our process modelling framework to consistency management. Our intention to "automate" consistency management necessitates a formal specification of the constraints we want to impose on the system. Such constraints may apply only locally—'in-ViewPoint' constraints— or globally—'inter-ViewPoint' constraints. Inter-ViewPoint constraints form the basis of coordination between different ViewPoints.

In our framework, consistency checking is decentralised, that is, each ViewPoint checks with the ViewPoint it considers relevant.

We use local and global consistency constraint to define a desirable state of the decentralised development process. In this sense, we specify a goal for the guidance and assistance that is provided by the process models.

In the remainder of this section, we discuss how local and global consistency constraints can be managed.

5.1 Local consistency management

Achieving consistency of the local partial specification contained in some ViewPoint is a necessary subtask of global consistency management. Here, the purpose of the process model is to guide the invocation of local actions and consistency checks.

The local process model sees consistency checks like any other action performed by the ViewPoint. The result of a check is posted to the work record and therefore visible to the process model.

The consistency checks available will vary considerably from template to template. It is the method designer's task to implement specific process models together with the consistency checks required for the different templates.

The consistency checks should be 'fine-grained' because only then the process model can give fine-grained guidance (see also [22]). It is desirable to decompose more complex constraints into independent parts that can be checked separately. Once the checks have been identified, they are integrated into the process model together with the other Work Plan actions.

5.2 Global consistency management

In our framework, global consistency management initiates and monitors two-party consistency checks between ViewPoints. Therefore, we now present a process model driven enaction of two-party consistency checking.

The protocol is based on message passing between ViewPoints as introduced in section 4.2. It assumes that the set of instantiated ViewPoints is constant, that is, a fixed configuration of

ViewPoints. We then show how, by self-modification of process models, the general case of varying configurations is addressed.

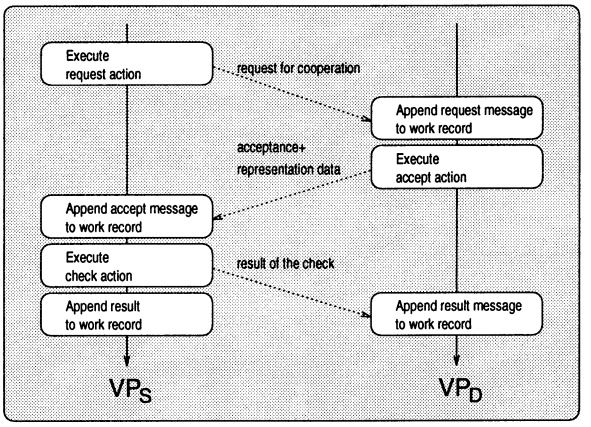

Figure 3: Protocol for two-party checks

Fixed configurations of ViewPoints Given that the set of ViewPoints is fixed and all ViewPoints know this, cooperation can be hard-coded into the local finite-state machine process models. A two-party consistency check is carried out by the protocol shown in Figure 3. The actual computation of the check involving the partial specifications contained in both ViewPoints is done by the source ViewPoint VP_S.

Variable configurations of ViewPoints We address this problem by reducing it to the fixed ViewPoints case discussed above. We do so by dynamically modifying the finite state process models as new ViewPoints are created and other ViewPoints are discarded. The results of completeness checks that look for particular ViewPoints are used to update the process model lazily. To cater for a varying set of ViewPoints , parts of the process model must be generic in order to allow for Work Record entries containing ViewPoint identifiers to be processed. The following generalisations of the process modelling architecture are necessary:

- Regular expression templates[5] with a ViewPoint identifier as a parameter have to be used to handle communication messages. Instances of such templates behave like the 'ordinary' regular expression discussed above. For example:

$T_g(v)$:	`.*q(v)[^P]*$`	child-located

  Here $q(v)$ is the generic event necessitating the abstraction of the regular grammar.

- To handle dynamically created regular expressions, rule templates are introduced. We restrict ourselves to one ViewPoint identifier as argument. For example:

Rule:	$R_4(v)$
Situation:	$T_g(v)$
Response:	recommend:
	`child-agrees-check-request-(v)`

  Such a rule would be instantiated and deleted together with the relevant regular grammars concerning a specific ViewPoint. There is an instance of this rule for each known ViewPoint with which communication can take place.

[5] These must not be confused with ViewPoint templates.

- We also want to be able to express statements like: *If all checks have succeeded do X.* Technically, this requires the expression of a universal quantification at some point in our model. Here quantification over instances of the same regular expression template plays this role. For example:

Rule:	$R_7(v)$
Situation:	$(\forall v).T_i(v)$
Response:	display: "All children known have been checked successfully"

  The pre-condition of this rule is satisfied if all instances of the template $T_i(v)$ match.

- Sometimes it is not necessary to know the identity of the other party involved when responding to a communication event. In this case, we ignore the address part of a communication message. For example:

T_f:	`.*U[^r(*)]*$`	commissioned-vp-not-sighted-yet

  Here r(*) matches any instance of r(v). Such regular expressions have one instance only, and do not require special treatment as far as the rules are concerned.

On the basis of such generic tests and rules we believe that communication of arbitrary and evolving configurations of ViewPoints can be handled.

5.3 Coordinating the checks

The framework described for two-party consistency checking necessitates cooperation among ViewPoints. Therefore the method designer developing process models has to look at the system in its entirety rather than at a specific ViewPoint template. This global perspective plays an important role by guiding and verifying the design of the local process models. The task of composing and decomposing process models, however, is non-trivial and requires tool support.

It is difficult to give general rules governing how consistency checks should be coordinated. Again, a recommended *sequence* of checks could be described. The notion of state shown has to be modified because remote actions influencing the state of a constraint will, in general, not be observable. Therefore, we have to resort to more heuristic measurements of the state. For example the *age* of a check (that is, the number of local actions and events since the last successful check) could be interpreted as reflecting the probability that the constraint still holds.

6 The Scenario Revisited

We now demonstrate our process modelling approach by applying it to the scenario described in section 3.

6.1 DFD ViewPoints

The ViewPoints used in this scenario each contain a simple data-flow diagram (DFD). Consequently, all ViewPoints are instances of the same ViewPoint template which defines a DFD technique. A simple example showing the Specification and Work Record slots of one such ViewPoint is shown in figure 1.

Checks To detect violations of the constraints listed, we use local and two-party consistency checks. They are derived from the consistency constraints outlined above.

Two-party checks We now decompose the global constraints into two-party constraints. In this example, the resulting checks are symmetric, which may not generally be the case.

- `parent-exists-check` detects whether the ViewPoint has a parent node. The check also succeeds if the domain is labelled *top*.
- `child-exist-check` takes all the composite nodes in the local diagram and tries to find the corresponding decomposition ViewPoints. As a side-effect of this check, the process model is modified by adding and/or removing generic actions, tests (that is, acceptor automata defined by regular expressions) and rules that control the interaction with decomposition ViewPoints.
- `parent-agrees-check-do-(v)` is a generic action performing a check for agreement between the local DFD and the DFD provided by `v`. The name and address of `v` is established by a previous completeness check. Agreement in this case means the contextual data flows of the child node must match those of the parent. Actually, some communication messages have to be exchanged before the check can go ahead (see section 5.2). These must also be parametrised.
- `child-agrees-check-do-(v)` performs the same operation as above, but in the opposite direction.

Work Plan All available actions are described in a ViewPoint's Work Plan. In addition to the check and communication actions described above, we also have operations on the DFD elements: Nodes {add, remove, make-composite}, Links {add, remove, rename}. All actions are assigned tokens that are used by the process model.

Process Model Also part of the work plan is the process model consisting of regular grammars T_x and rules R_i. Actions, rules and tests can have one parameter, then they are *generic*. Generic parts of the work plan are instantiated and deleted as side-effects of existence checks (for example, `parent-exists-check`).

The regular expressions T_x as presented here operate on sequences of action tokens. The action tokens for static events are fixed, and the tokens for generic events are dynamically allocated. Here are all the tokens used in the regular expression examples in the scenario:

`A`	`add-node`
`D`	`make-node-composite`
`F`	`add-link`
`G`	`remove-link`
`H`	`rename-link`
`O`	`parent-exist-check`
`Q`	`parent-exist-check-succeeded`
`R`	`child-exist-check-succeeded`
`T`	`child-exist-check-failed`
`U`	`commission-viewpoint`
`a(v)`	`parent-agrees-check-request-(v)`
`i(v)`	`parent-agrees-check-requested-(v)`
`m(v)`	`parent-agrees-check-succeeded-(v)`
`q(v)`	`child-located-me-(v)`
`r(v)`	`parent-located-me-(v)`

Except for the parametrised tokens necessitated by these dynamic allocations, the syntax and semantics of the regular expressions follow the usage by the lexical analyser generator **Lex** [6].

The rules R_i uses the regular expressions T_x as predicates in their pre-conditions. There are three alternatives that can be used in the action part of such rules:

`do:<action>` automatically executes the given *action*.

`recommend:<action>` suggests that *action* be enacted by the user.

`display:<text>` gives informal guidance by displaying the help message *text*.

6.2 Process modelling at work

We now outline how our process modelling framework applies to the scenario described in section 3. For each step, we discuss how guidance should be provided to the user. The reader may also find it useful to consult section 3 on page 3 to recall some of the details.

Step 1 Here, a DFD and thus a ViewPoint is missing to make the specification complete (Figure 1). Clearly, some local process model has to detect the violation of the constraint. Typically, ViewPoint A can discover this by executing the corresponding check.

After the check has been performed, Anne is informed of the need to create a new (or indeed, to change some existing) ViewPoint to fulfil the constraint. The process model should give Anne the choice between doing nothing, creating a new ViewPoint , or delegating responsibility (using email, for example).

Example We can use the following tests and rules to provide guidance in this situation:

T_A:	`.*D[^R]*$`	not-successfully-checked-since-D
T_B:	`.*O[^RT]*$`	exist-child-check-pending
T_C:	`.*T[^ORT]{0,10}$`	exist-child-check-just-failed

Rule:	R_1
Situation:	$T_A \wedge \neg T_B \wedge \neg T_C$
Response:	recommend: `child-exist-check`

Thus Anne would be advised to invoke a check looking for decomposition ViewPoints whenever the following conditions hold:

- The check has not been successfully carried out since the action `make-node-composite` was last performed in the local DFD (T_A).
- There are no checks of this kind pending ($\neg T_B$).
- The check has not failed less than ten actions ago ($\neg T_C$)

Step 2 The ViewPoints A and B do not agree with each other (Figure 4). Potentially, both A and B can discover the inconsistency. However, given that the Specification of B is empty, B is unlikely to initiate the check and may refuse to cooperate with A on this matter. ViewPoint A waits for some acknowledgement message from the decomposition ViewPoint because A initiated B's creation. Effectively, the checking of global check 2 is therefore suspended as long as B remains empty. The process model of B, however, will advise Bob to elaborate ViewPoint B, thus making it non-empty. The checking of global constraint 1 at ViewPoint A would detect the presence of B and update the local process model.

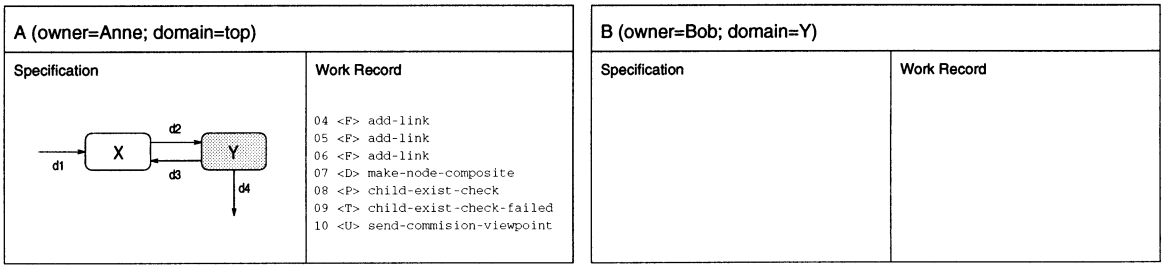

Figure 4: Now a decomposition ViewPoint for Y exists, but the inputs and outputs do not match (step 2).

Example The invocation of checks by Anne or Bob can be inhibited by the following tests and rules:

T_D:	`^.{0,10}$`	underdeveloped
T_f:	`.*U[^r(*)]*$`	commissioned-vp-not-sighted-yet

Rule:	R_2
Situation:	$\neg T_D$
Response:	display: "Viewpoint should be developed further"
Rule:	R_3
Situation:	T_f
Response:	display: "do not enact `child-exist-check` or `child-agrees-check-request` because ViewPoint creation commissioned but not acknowledged"

In this situation R_2 advises Bob to perform some additional local development before considering any non-local checks. R_3 tells Anne to wait until the ViewPoint to be created by Bob reports its existence to ViewPoint A.

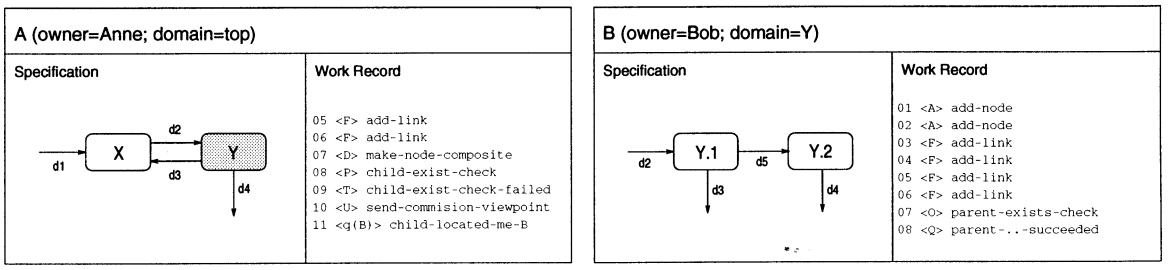

Figure 5: The specification is now consistent with respect to the formulated constraints (step 3).

Step 3 Now, all constraints are satisfied (Figure 5). All global checks are eventually initiated by A or B with their result communicated to the work record of both ViewPoints. The checks carried out separately, however, do not guarantee that both ViewPoints are consistent, even at any single point in time. To achieve this would require two-party checks to be performed as atomic transactions.

Example The initiation of the agreement check between A and B is recommended by these tests and rules:

$T_g(v)$:	`.*q(v)[^P]*$`	child-located
T_E:	`.*Q[^0]*$`	parent-probably-exists

Rule:	R_5
Situation:	T_E
Response:	recommend: **parent-agrees-check-request-(v)**
Rule:	$R_4(v)$
Situation:	$T_g(v)$
Response:	recommend: **child-agrees-check-request-(v)**

For ViewPoint A, $T_g(B)$ succeeds when ViewPoint B has been identified. Consequently, Anne is advised to check the agreement with the child whose existence is now reasonably certain. T_E tests for a recent success of the check for the existence of B's parent. If this is the case, Bob can also initiate the agreement check with ViewPoint B's parent, A.

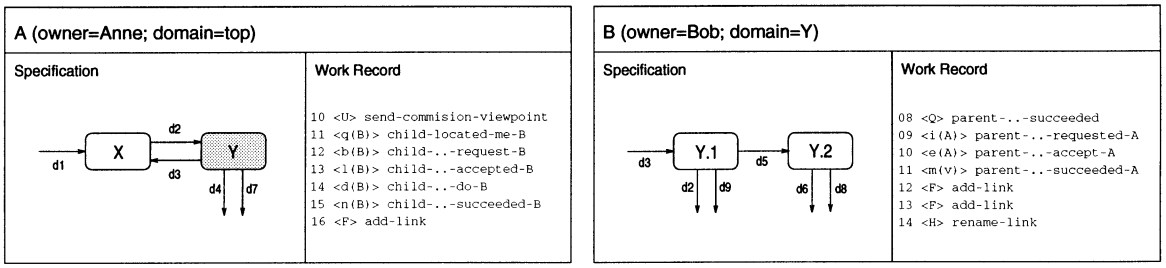

Figure 6: The outputs of Y do not match the contextual outputs of ViewPoint B (step 4).

Step 4 Again, we face disagreement between ViewPoint A and ViewPoint B (Figure 6). This step highlights a general problem of concurrent development. Clearly, both process models attempt to achieve local consistency. When local constraints are satisfied, global checking can be recommended. The 'waterfall model', for example, prescribes the sequencing of the checks with backtracking. The resolution of inconsistencies in this model is an iterative process, supported by communicating check results. However, the process model cannot 'manufacture' agreement on details like the labelling of arcs. It merely supports conflict resolution by pointing out deficiencies. The check results become part of the work record and then the process model can 'nag' the user until the constraint is satisfied (if that is what the method designer wants).

Example In this step, the development activities concerning the contextual data flows in both ViewPoints is likely to disturb any pre-existing agreement between A and B. Therefore A and B are advised to renew the check:

$T_h(v)$:	`.*m(v)[^FGHa(v)i(v)]*$`	context-links-ok

Rule:	$R_6(v)$
Situation:	$\neg T_h(v)$
Response:	recommend: `parent-agrees-check-request-(v)`

This rule applies only to Bob's ViewPoint. A symmetric rule for ViewPoint A can be analogously constructed.

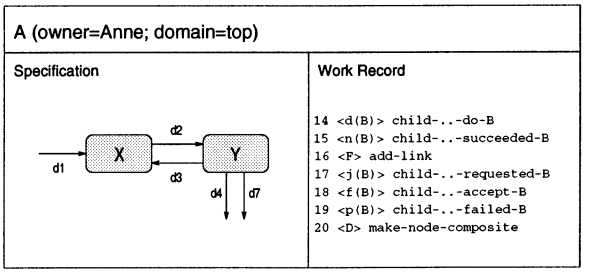

Figure 7: ViewPoint A contains the non-primitive nodes X and Y but the respective decomposition ViewPoints do not exist (step 5).

Step 5 As in step 1, the completeness constraint 1 is violated (7). Eventually Anne is advised to check the global constraints by the process model. Global check 1 fails for both X and Y. Consequently, the local process model must be updated to remove the parts no longer needed for the communication with the two ViewPoints. Otherwise, this case is similar to Step 1.

7 Related Work

Ben-Shaul and Kaiser proposed an approach towards "modelling and enaction of inter-group collaboration among independent, autonomous, and, possibly, pre-existing processes" [7]. They presented an "international alliance" metaphor to define collaboration in terms of 'treaties' which are enacted at 'summits'. Conceptually, our consistency constraints fulfil the role of such treaties, and the application of a consistency check can also be interpreted as a summit between two ViewPoints. Our approaches differ, in that we focus on intra-group collaboration whereas Ben-Shaul and Kaiser address cooperation among groups.

Jarke et al. describe decision-oriented, logically centralised process models [15]. They propose process models dedicated to guidance which use pattern matching to identify situations that can be mapped into guidance, a clear similarity to our work. They also use process models to structure process traces, a task fulfilled (in a less flexible way) by the Work Records in our framework. Jarke et al. address the issue of deriving guidance from these process traces. We have identified this problem in our framework and are currently investigating possible solutions.

Balzer [4] describes a mechanism for managing inconsistencies employing "pollution markers" to identify constraints that have been violated, which can then be either avoided or resolved. We have generalised and decentralised these concepts in our approach. In his work, Balzer also uses enforced consistency constraints, which we consider problematic to adopt in our framework.

Narayanaswamy and Goldman advocate "lazy" consistency as basis for cooperative software development [20]. They focus mainly on preventing conflicts of updates of shared artefacts, where we concentrate on conflict detection and resolution. In their setting, dependencies between products are defined by a single explicit relation. In contrast, our framework allows for many different kind of such dependencies to be defined in terms of consistency constraints. This gives us more detailed information for the resolution of inconsistencies.

8 Further Work

We have identified three major areas in need of further work. These are: guidance; support for evolution; process specification support. We see the role of process modelling as providing guidance rather than automation. In this case the way in which the guidance is framed and the means by which the guidance is actually delivered are critical. In this paper we have concentrated on process observation and decision making, however the practical utility of the techniques described will depend on advances in guidance. Some indication of our approach to this problem is given in [11]. We have indicated above some of the problems of evolution in an environment in which ViewPoints are created and (occasionally) destroyed, and we have outlined our solution to these problems. We feel that this area is in need of further work and to this end we will be exploring other grammar-based schemes, among them Activity Structures [24]. We have, as yet, little experience of how to actually arrive at process models of the form we have presented. Compositional process model design requires attention, in particular derivation of local process models from higher level cooperation policies. There are a number of improvements we would like to make to our implementation, of which the most pressing is integration with `The Viewer`; also some improvements to the user interface we provide to the process modelling capability are required.

9 Conclusions

This paper has examined the application of process modelling techniques to the problem of consistency management. The approach proposed in this paper is based on constraint checks derived from a static notion of consistency. We have outlined how consistency constraints can be decentralised in order to facilitate decentralised consistency management.

The consistency constraints are expressed in terms of agreement and completeness relations over the set of all ViewPoints. We believe that consistency constraints need to be represented explicitly to form the basis for the design and verification of mechanisms for consistency management.

We have developed an architecture for communicating local process models which is fully decentralised. In this respect, we believe, it reflects the nature of development processes in multi-perspective environments (exemplified by the ViewPoints framework). Global and local consistency checking is driven by local process models employing regular grammars and rules. This we achieve by decomposing and distributing global strategies and protocols for cooperation and communication. We have presented a communication mechanism based on message passing between ViewPoints, and consequently their local process models. We have also shown that protocols for two-party consistency checking can be built on top of this communication layer.

Our process models are 'fine-grained', and therefore capture the level of detail which we believe to be essential for adequate guidance. The granularity of process models in our framework critically depends on the granularity of consistency checks. Therefore we also favour fine-grained consistency constraints.

The application of the proposed process modelling architecture to the scenario has demonstrated the process-model driven consistency management as the innovative feature presented in this paper. We have also developed a prototype implementation of our process modelling framework which we have used to validate the ideas described.

10 Acknowledgements

We would like to gratefully acknowledge the constructive comments of Michael Goedicke. This work was partly funded by the UK Department of Trade and Industry (DTI) as part of the ESF project, by the UK EPSRC VOILA project, and by the European Union (ESPRIT BRA PROMOTER, ISI, Human Capital and Mobility).

References

[1] In A. van Lamsweerede and A. Fugetta, editors, *Proceedings of the 3rd European Software Engineering Conference (ESEC '91)*, volume 550 of *LNCS*, Milan, Italy, October 1991. Springer-Verlag.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley series in Computer Science. Addison-Wesley, Reading, Mass., 1986.

[3] L. Ballesteros. Using ViewPoints to Support the FUSION Object-Oriented Method. M.Sc. Thesis, Department of Computing, Imperial College, London, UK, September 1992.

[4] R. Balzer. Tolerating inconsistency. In *Proceedings of the 13th International Conference on Software Engineering*, Austin, Texas, May 1991. IEEE CS press.

[5] N. Barghouti and G. Kaiser. Scaling up rule-based software development environments. In van Lamsweerede and Fugetta [1], pages 380–395.

[6] Bell Telephone Laboratories, Inc., Murray Hill, New Jersey. *UNIX programmer's manual*, seventh edition, 1983. Volume 2.

[7] I. Ben-Shaul and G. Kaiser. A paradigm for decentralized process modeling and its realization in the OZ environment. In *Proceedings of the 16th International Conference on Software Engineering*, pages 179–188, Sorrento, Italy, May 1994. IEEE CS press.

[8] M. Chang and C. Woo. SANP: A Communication Level Protocol for Negotations. In *Proceedings of the 3rd European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Kaiserslautern, Germany, August 1991. North Holland.

[9] S. Easterbrook, A. Finkelstein, J. Kramer, and B. Nuseibeh. Coordinating Distributed ViewPoints: the anatomy of a consistency check. *International Journal on Concurrent Engineering: Research and Applications, Special issue on conflict management*, 2(3), 1994.

[10] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. In *IEEE Transactions on Software Engineering*, August 1994.

[11] A. Finkelstein and J. Kramer. TARA: Tool assisted requirements analysis. In *Conceptual Modelling, Databases & CASE: an integrated view of information system development.* McGraw Hill, 1991.

[12] A. Finkelstein, J. Kramer, B. Nuseibeh, M. Goedicke, and L. Finkelstein. ViewPoints: A Framework for integrating multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–58, March 1992.

[13] P. Graubmann. The HyperView Tool Standard Methods. REX technical report REX-WP3-SIE-008-V1.0, Siemens, Germany, July 1990.

[14] P. Graubmann. The Petri Net Method ViewPoints in the HyperView Tool. REX technical report REX-WP3-SIE-021-V1.0, Siemens, Germany, January 1992.

[15] M. Jarke, K. Pohl, C. Rolland, and J. Schmitt. Experience-Based Method Evaluation and Improvement: A Process Modeling Approach. NATURE Report Series 94-15, ESPRIT Project 6353, RWTH Aachen, Germany, 1994.

[16] J. Kramer. CASE Support for the Software Process: A Research Viewpoint. In van Lamsweerede and Fugetta [1].

[17] J. Kramer and A. Finkelstein. A configurable framework for method and tool integration. In *European Symposium on Software Development Environments and CASE*, volume 509 of *LNCS*, pages 233–257, Königswinter, Germany, June 1991. Springer-Verlag.

[18] Fui Kien Lai. CORE in The Viewer. M.Sc. Thesis, Department of Computing, Imperial College, London, UK, September 1993.

[19] U. Leonhardt. Process Modelling in TheViewer. Technical report, Department of Computing, Imperial College, London, UK, 1994. (in preparation).

[20] K. Narayanaswamy and N. Goldman. "Lazy" Consistency: A Basis for Cooperative Software Development. In *Proceedings of CSCW'92*, pages 257–264, Toronto, Canada, 1992. ACM press.

[21] B. Nuseibeh and A. Finkelstein. Viewpoints: A vehicle for method and tool integration. In *Proceedings of the International Workshop on Computer-Aided Software Engineering (CASE '92)*, pages 50–60, Montreal, Canada, July 1992. IEEE CS press.

[22] B. Nuseibeh, A. Finkelstein, and J. Kramer. Fine-grain process modelling. In *Proceedings of the 7th International Workshop on Software Specification and Design (IWSSD-7)*, pages 42–46, Redondo Beach, California, December 1993. IEEE CS Press.

[23] B. Nuseibeh, J. Kramer, and A. Finkelstein. Expressing the relationships between multiple views in requirements specification. In *Proceedings of the 15th International Conference on Software Engineering*, pages 187–196, Baltimore, Maryland, May 1993. IEEE CS press.

[24] W. Riddle. Activity structure definitions. *Software Design & Analysis*, March 1991. Technical Report 7-52-3.

[25] R. Smith and R. Davis. Frameworks for Cooperation in Distributed Problem Solving. In *IEEE Transactions on Systems, Man, and Cybernetics*, volume 11, January 1981.

[26] T. Thanitsukkarn. The Constructive Viewer. M.Sc. Thesis, Department of Computing, Imperial College, London, UK, September 1993.

[27] W. Turski and T. Maibaum. *The specification of computer programs.* International computer science series. Addison-Wesley, 1987.

Papers related to the ViewPoints framework can be found on `ftp://dse.doc.ic.ac.uk/dse-papers/`.

A Expressing and Checking Consistency Constraints

We use the term constraint for an explicitly defined relation over the domain of ViewPoints. A configuration of ViewPoints, that is, a set of ViewPoints, may or may not satisfy a constraint. As consistency constraints are method-specific they form a natural part of a ViewPoint template.

Constraints can specify a syntactic notion of *completeness*. Broadly speaking, by checking such *completeness* constraints we discover missing ViewPoints. Secondly, we are also looking for conflicting information with *agreement* constraints. Disagreement usually indicates the need for action on the part of the ViewPoints party to the disagreement.

The products of the development process must eventually be consistent, that is, satisfy all the defined constraints. Thus the constraints specify the goal of the development process. These constraints also provide a means of monitoring the progress of the development process. The more fine-grained the constraints are, the more information can be obtained about the process.

A.1 Formalising consistency constraints

Our intention to "automate" consistency management necessitates a formal specification of the constraints we want to impose on the system. By expressing these constraints in some formal system we implicitly build a class of states of the process that we deem to be desirable. Effectively, the constraints represent a theory, the models of which are consistent states.

We may want to distinguish between constraints that only apply locally—'in-ViewPoint' constraints—and those that are global—'inter-ViewPoint' constraints. As our goal is to distribute the checking of the constraints and perform as little global checking as possible, the constraint specification framework should reflect this distinction.

We use many-sorted first-order logic to formally express consistency constraints.

A.1.1 Local constraints

Typically, such a constraint would specify syntactic correctness for the content of the ViewPoint. Consequently, the extra-logical language introduced will be specific to the representation-style of the ViewPoint. Local constraints and local language represent a "local theory" T_{loc} for each ViewPoint. As the extra-logical language is specific to the ViewPoint template, all instances of a template can share the same set of constraints and thus have the same theory T_{loc}.

A.1.2 Global constraints

Here, we have to deal with all the different representation styles as well as with the structure of the ViewPoints framework. A specification of the latter is necessary in order to facilitate reasoning about the constraints we are interested in. Global constraints will typically include references to specific representation styles as well as to the general framework. Hence a "global theory" T_{glob} incorporating such constraints will use parts of the local languages. The distinction between agreement and completeness constraints introduced above translates into canonical forms for the representation of global constraints:

- **Completeness** constraints

  $$(G_s(x) \wedge G_a(\overline{n}))$$
  $$\Rightarrow (\exists y \in VP).(G_t(x, y, \overline{n}) \wedge R(x, y, \overline{n}))^6$$

  Here $G_s(x)$ "selects" suitable *'source'* ViewPoints for the constraint. Typically, we also want to "select" information from the representation domain with $G_a(\overline{n})$. For the so chosen domain objects x and $\overline{n}$ a *'target'* ViewPoint y which satisfies $G_t(x, y, \overline{n})$ and $R(x, y, \overline{n})$ is forced to exist, hence the interpretation as completeness constraint. We treat $G_t(x, y, \overline{n})$ as

[6] $\overline{n}$ refers to an arbitrary number of variables over the domains defined in the representation of some ViewPoint. Outermost universal quantifiers have been omitted from all formulae.

a guard for $R(x, y, \overline{n})$ in order to facilitate efficient checking procedures. An example for this kind of consistency condition would be:

> For each non-primitive node n in the representation of one ViewPoint there exists a ViewPoint in a domain with the same name as n.

Formally, this statement looks like this:

$$(n \in_{vp} x \wedge \neg prim\_node(n))$$
$$\Rightarrow (\exists y \in VP).label_n(n) =_l domain(y)$$

- **Agreement** constraints

$$(G_s(x) \wedge G_a(\overline{n}))$$
$$\Rightarrow (\forall y \in VP).(G_t(x, y, \overline{n}) \Rightarrow R(x, y, \overline{n}))$$

Here, we have a universal quantification over target ViewPoints y which is effectively restricted by the 'guarding' predicate $G_t(x, y, \overline{n})$. Hence we constrain the set of ViewPoints as determined by $G_t(x, y, \overline{n})$ to "agree" with x, the agreement relation being defined by $R(x, y, \overline{n})$. As an example we might choose:

> No two nodes in different ViewPoints have the same label.

This translates into:

$$(n_1 \in_{vp} x)$$
$$\Rightarrow (\forall y \in VP).((\neg x = y \wedge n_2 \in_{vp} y)$$
$$\Rightarrow \neg label_n(n_1) = label_n(n_2))$$

Using rules of this structure we can express and represent a global view of the system. However, in a distributed setting such a global view will not necessarily be the primary representation used to deal with inter-ViewPoint relationships. So a certain part of the knowledge about these relationships, if not all of it, will be distributed among the actual ViewPoints.

A.1.3 Distributing the constraints

Overview As there is no central global coordination at run-time, all the consistency constraints must be represented and checked in a local and decentralised fashion. Consequently, we need to create "combined local theories" $T_{l,c}$ for every ViewPoint which incorporate both the local and relevant global consistency constraints. Intuitively, if all the individual ViewPoints satisfy their respective theories $T_{l,c}$ then the set of ViewPoints as a whole also satisfies both global and local constraints, and is thus consistent. The distribution of the theory T_{glob} must be done on the template-level because instances of the same template conceptually share the same set of constraints.

Obviously, some transformation mapping T_{glob} and the different T_{loc} into template-specific $T_{l,c}$ is needed. This can be done by taking the individual T_{loc} and adding as much of the global theory T_{glob} as necessary. The converse transformation might also be of use when we want to prove certain properties over collections of ViewPoints. For the time being we shall be focusing on the top-down direction that involves two steps: modularising and localising T_{glob}.

Modularising T_{glob} We want to identify the set of constraints that need to be incorporated into the different templates. Intuitively, we can interpret the first universal quantification over ViewPoints as identifying the "*source* ViewPoint" of the constraint. In an analogous manner, we think of the second ViewPoint as "*destination* ViewPoint".

We can now 'modularise' T_{glob} by distributing the constraints to the templates of all possible source ViewPoints. If the constraint is represented in one of the canonical forms introduced above, we can unfold the universal quantification over the source ViewPoints. As a result, every template

gets a part of each global constraint. Subsequently, irrelevant constraints (that is, constraints that will always hold) can be removed from individual templates by using partial evaluation techniques.

As a result, each of the templates has at least the constraints that are relevant to its instances. These constraints represent template-specific theories $T_{g,t}$ which can be seen as modules of T_{glob}. The conjunction of the constraints in the templates is semantically equivalent to the original theory T_{glob}, that is, a given 'state' will satisfy all the partial theories $T_{g,t}$ if, and only if T_{glob} is satisfied.

Localising T_{glob} The template specific parts of T_{glob} have to be incorporated by the respective local theories T_{loc}. Technically speaking, the parts of T_{glob} have to be implemented[7] in terms of the respective T_{loc}. This is achieved by building a conservative extension $T_{l,c}$ of T_{loc} and an interpretation between theories from $T_{g,t}$ to $T_{l,c}$.

The correspondence between locally and globally represented constraints allows us to choose the degree of distribution we want to adopt. In this paper we explore the consequences of the radical approach of total distribution. But we have to bear in mind that this choice will not always be optimal when we consider performance and management concerns.

A.2 Checking constraints

For constraints concerning only the local specification, this is straightforward. If the constraint is global, that is, involves other ViewPoints as well as the local specification, a number of issues arising from the distribution of the ViewPoints have to be addressed.

1. Conceptually, constraints operate on some sort of *state*. In a distributed system, such as a collection of ViewPoints , we can only find approximations to a notion of state. Thus the results of consistency checks are also approximations at best. Alternatively, the freedom of the system can be restricted to make it behave in a more sequential way (transactions, locking, etc.). We believe, that most of the time the development process can live with approximations of consistency information.

2. Each check potentially has to consider all the ViewPoints in the system. This is neither desirable nor practical, especially for large systems. Therefore we would like to structure our ViewPoint universe in such a way that we can limit our search to a small set of ViewPoints without ever having to interact with *all* the others. One solution could be to use the domains already defined in the ViewPoints framework to restrict quantifications.

3. A ViewPoint is required by the constraints it contains to "know" about other ViewPoints. Hence we need some logically centralised service that keeps track of all the ViewPoints (that is, a name server).

4. Due to failures of the various resources involved (humans, software, hardware) checking of a constraint can be impossible. Obviously, we would like the system to be resilient against partial failures.

Location. A check can be computed by the source ViewPoint, by the target ViewPoint or by some third party (for example a "check server").

Synchronisation. We have established a check's source ViewPoint is responsible for initiating the check. Nevertheless, the check requires at least some degree of cooperation from target ViewPoints. We could by convention force the target ViewPoint to coopĕrate, but what if a particular target ViewPoint cannot respond? A sensible approach may be to require the target ViewPoint to send the relevant data to the source ViewPoint within some time limit and leave the actual application of the check to the source ViewPoint(the source ViewPoint is busy invoking and partly applying the check anyway.) However, it may be useful retain a certain degree of flexibility by

[7] We use the terminology given by Turski and Maibaum in [27]

using process models to describe the actual protocols for consistency checking. Other options include negotiation techniques derived from distributed artificial intelligence (for example [25, 8]).

Guards. If checks are not to be carried out by some third party, it seems sensible to adopt the two-party check involving just one source and one target ViewPoint as the basic transaction units of consistency checking. But it makes sense to decompose these two-party checks even further. Most of our constraints will have parts that are relatively easy to compute, and others that will be computationally expensive. The structure of the constraints given above already takes this distinction into account. $G_t(x, y, \overline{n})$ (read "guard on the target ViewPoint") is intended to designate the first, "easy" part of the check. $R(y, \overline{n})$ then refers to the expensive, "heavy-weight" component of the check. If the easy part of the check does not hold, it normally does not make sense to proceed any further. Hence the easy part can be thought of as guarding the "heavy-weight" part of the check.

Granularity. A single constraint can cover all aspects of consistency that arise between a ViewPoint and its environment. In this setting, communication and coordination seem to be much simplified. However, when such coarse-grained constraints are applied we do not get the level of detail from the result we would like in order to guide conflict resolution in a meaningful way. We can therefore identify a tradeoff between coordination cost and observational detail which governs the choice of the "optimal granularity" for consistency checks.

B The ViewPoint framework

In this chapter, the many-sorted predicate calculus shall be used to define the static notion of consistency for a configuration of ViewPoints. Firstly, the general framework will be specified. Secondly, the notion of consistency for data flow diagram ViewPoints as used in the case study is going to be defined.

SPEC 1 *This defines the basic entities of the framework (templates, viewpoints, domains, representations) and their relations. Equality is assumed to be part of our logic.*

- **sorts:**
  TEMPL, VP, DOM, REPR, REP_ST
- **relations:**
  inst_of$_t$*:* $VP \times TEMPL$
  $\_\in_d\_$: $VP \times DOM$
  contains_rep: $VP \times REPR$
  presented_in: $REPR \times REP\_ST$
  contains_style: $TEMPL \times REP\_ST$
- **functions:**
  domain: $VP \rightarrow DOM$
  tmpl: $VP \rightarrow TEMPL$
  content: $VP \rightarrow REPR$
  style: $TEMPL \rightarrow REP\_ST$
  meta_model: $REPR \rightarrow REP\_ST$
- **axioms:**
  $\vdash x = tmpl(y) \Leftrightarrow inst\_of_t(x, y)$
  $\vdash x = meta\_model(y) \Leftrightarrow presented\_in(x, y)$
  $\vdash x = content(y) \Leftrightarrow contains\_rep(x, y)$
  $\vdash x = style(y) \Leftrightarrow contains\_style(x, y)$
  $\vdash x = tmpl(y) \Rightarrow meta\_model(content(y)) = style(x)$

C Axiomatisation of the Case study

C.1 Data flow diagrams

SPEC 2 *This specification can be interpreted over data flow diagrams that are the representation style of the ViewPoints in our case study.*

- **sorts:**
  NODE, ARC, LABEL
- **relations:**
  prim_node: $NODE$
- **functions:**
  $label_n$: $NODE \rightarrow LABEL$
  $label_a$: $ARC \rightarrow LABEL$
  context: $\rightarrow NODE$
  arc_beg: $ARC \rightarrow NODE$
  arc_end: $ARC \rightarrow NODE$
- **axioms:**
  $\vdash (label_n(n_1) = label_n(n_2)) \Rightarrow n_1 = n_2$
  $\vdash (label_a(a_1) = label_a(a_2)) \Rightarrow a_1 = a_2$

This theory incorporates already the internal consistency constraints for the case study. A diagram interpretation satisfying this theory is therefore a consistent representation for ViewPoints in our case study. Note that the local constraint listed in section 6 saying that arcs must be connected to at least one node is implicit because the caculus does not allow for functions to be partial. are implicit because

C.2 Global constraints

SPEC 3 *Here we combine the framework and relevant parts of the representation. We extend SPEC1 conservatively with:*

- **sorts:**
  NODE, ARC, LABEL
- **relations:**
  prim_node: $NODE$
  $\_\in_{vp}\_$: $NODE \times VP$
  $\_\in_{vp}\_$: $ARC \times VP$
- **functions:**
  $label_n$: $NODE \rightarrow LABEL$
  $label_a$: $ARC \rightarrow LABEL$
  context: $VP \rightarrow NODE$
  arc_beg: $ARC \rightarrow NODE$
  arc_end: $ARC \rightarrow NODE$
- **axioms:**
  $\vdash (n \in_{vp} v_1 \wedge \neg prim\_node(n)) \Rightarrow$
  $(\exists v_2).(label_n(n) =_l domain(v_2))$
  $\wedge (\forall a_1 \in_{vp} v_1, a_2 \in_{vp} v_2).$
  $((arc\_beg(a_1) = context(v_2) \Leftrightarrow arc\_end(a_2) = n)$
  $\wedge (arc\_end(a_1) = context(v_2) \Leftrightarrow arc\_beg(a_2) = n))$
  $\vdash (label_n(n_1) = label_n(n_2)) \Rightarrow n_1 = n_2$

$\vdash \neg label_n(n) = "top"$
$\vdash domain(v_1) = d \wedge \neg d = "top" \Rightarrow (\exists v_2, n).(n \in_{vp} v_2 \wedge label_n(n) = d)$

The first axiom is a good example for partitioning the check into heavy and light weight parts. The name of the domain can be checked very easily, possibly by querying some domain server. The check for interface compatibility then is the more expensive part.

C.3 Distributed constraints

SPEC 4 *This is the distributed version of the global constraints (SPEC3) embedded into the normal specification of the local representation SPEC2. We arrive at this specification by extending SPEC2 with domains of remote ViewPoints and elements of the representation. Thus we have SPEC2 +*

- **sorts:**
  RVP, RNODE, RARC, RVP

- **relations:**
  $\_\in_{vp}\_ :$ $RNODE \times RVP$
  $\_\in_{vp}\_ :$ $RARC \times RVP$

- **functions:**
  $label_n$: $RNODE \rightarrow RLABEL$
  $label_a$: $RARC \rightarrow RLABEL$
  context: $RVP \rightarrow RNODE$
  arc_beg: $RARC \rightarrow RNODE$
  arc_end: $RARC \rightarrow RNODE$
  ldomain: $\rightarrow DOM$

- **axioms:**
  $\vdash \neg prim\_node(n)) \Rightarrow$
  $(\exists v_r).(label_n(n) =_l domain(v_r))$
  $\wedge ((\forall a_l)(\forall a_r \in_{vp} v_r).$
  $((arc\_beg(a_r) = context(v_r) \Leftrightarrow arc\_end(a_l) = n)$
  $\wedge (arc\_end(a_r) = context(v_r) \Leftrightarrow arc\_beg(a_l) = n)))$

Now we would have to prove that the parts of the unfolded global specification translate into this theory.

For implementation purposes it makes sense to decompose the axioms into an existence and an agreement part. The two check procedures would have to communicate the objects of interest, that is the decomposition ViewPoints.

D Case Study: Actions and Events

Here a rudimentary Work Plan for data flow diagram ViewPoints is presented. It was used in the case study in section 6 and is explained there in greater detail.

D.1 Static Part

D.1.1 Events

- **Assembly Actions**
  - <A> `add-node`
  - <B> `remove-node`
  - <C> `rename-node`
  - <D> `make-node-composite`
  - <E> `make-node-primitive`
  - <F> `add-link`
  - <G> `remove-link`
  - <H> `rename-link`
- **Internal Checks**
  - <I> `name-cons-check`
  - <J> `link-cons-check`
- **Internal Check Results**
  - <K> `name-cons-check-succeeded`
  - <L> `link-cons-check-failed`
  - <M> `name-cons-check-succeeded`
  - <N> `link-cons-check-failed`
- **External Checks**
  - <O> `parent-exists-check`
  - <P> `child-exist-check`
- **External Check Results**
  - <Q> `parent-exists-check-succeeded`
  - <R> `child-exist-check-succeeded`
  - <S> `parent-exists-check-failed`
  - <T> `child-exist-check-failed`
- **Other Events**
  - <U> `send-commission-viewpoint`[8]

D.1.2 Tests

T_A:	`.*D[^R]*$`	not-successfully-checked-since-D
T_B:	`.*O[^RT]*$`	exist-child-check-pending
T_C:	`.*T[^ORT]{0,10}$`	exist-child-check-just-failed
T_D:	`^.{0,10}$`	underdeveloped
T_E:	`.*Q[^O]*$`	parent-probably-exists

[8] Commission sent be E-mail or whatever means appropriate

D.1.3 Rules

Rule: R_1
Situation: $T_A \wedge \neg T_B \wedge \neg T_C$
Response: recommend:
`child-exist-check`

Rule: R_2
Situation: $\neg T_D$
Response: display:
"Viewpoint should be developed further"

Rule: R_3
Situation: T_f
Response: display:
"do
not enact `child-exist-check` or
`child-agrees-check-request`
because ViewPoint creation commissioned but not acknowledged"

D.2 Generic Part

D.2.1 Events

- **Generic Actions**
  <a(v)> `parent-agrees-check-request-(v)`
  <b(v)> `child-agrees-check-request-(v)`
  <c(v)> `parent-agrees-check-do-(v)`
  <d(v)> `child-agrees-check-do-(v)`
  <e(v)> `parent-agrees-check-accept-(v)`
  <f(v)> `child-agrees-check-accept-(v)`
  <g(v)> `parent-agrees-check-reject-(v)`
  <h(v)> `child-agrees-check-reject-(v)`

- **Communication Events**
  <i(v)> `parent-agrees-check-requested-(v)`
  <j(v)> `child-agrees-check-requested-(v)`
  <k(v)> `parent-agrees-check-accepted-(v)`
  <l(v)> `child-agrees-check-accepted-(v)`
  <m(v)> `parent-agrees-check-succeeded-(v)`
  <n(v)> `child-agrees-check-succeeded-(v)`
  <o(v)> `parent-agrees-check-failed-(v)`
  <p(v)> `child-agrees-check-failed-(v)`

- **Other Events**
  <q(v)> `child-located-me-(v)`
  <r(v)> `parent-located-me-(v)`

D.2.2 Tests

T_f:	`.*U[^r(*)]*$`	commissioned-vp-not-sighted-yet
$T_g(v)$:	`.*q(v)[^P]*$`	child-located
$T_h(v)$:	`.*m(v)[^FGHa(v)i(v)]*$`	context-links-ok
$T_i(v)$:	`.*n(v)[^j(v)b(v)]*$`	child-agrees

D.2.3 Rules

Rule: $R_4(v)$
Situation: $T_g(v)$
Response: recommend:
`child-agrees-check-request-(v)`

Rule: R_5
Situation: T_E
Response: recommend:
`parent-agrees-check-request-(v)`

Rule: $R_6(v)$
Situation: $\neg T_h(v)$
Response: recommend:
`parent-agrees-check-request-(v)`

Rule: $R_7(v)$
Situation: $(\forall v).T_i(v)$
Response: display:
"All children known have been checked successfully"