TOWARDS AN INTEGRATION OF DIFFERENT SPECIFICATION METHODS BY USING THE VIEWPOINT FRAMEWORK

Bettina E. Enders[1] Michael Goedicke[1] Torsten Heverhagen[2]
Rudolf Tracht[2] Peter Tröpfner[1]

[1] Specification of Software Systems, Dept. of Mathematics and Computer Science
[2] Automation Technology, Dept. of Mechanical Engineering
University of Essen, Germany

[1] {enders|goedicke|troepfner}@informatik.uni-essen.de
[2] {torsten.heverhagen|tracht}@automat.uni-essen.de

ABSTRACT

In industrial interdisciplinary projects where engineers of different sorts have to work together, a framework is necessary in order to support such an interdisciplinary development process. For example, the design information created by the production automation engineer has to be set into relation to the software architecture the software designer has to produce which will control the automation plant. Such requirements result in the need to correctly and consistently integrate various specification methods and development processes applied during the development.

This contribution focuses on an approach to a correct and consistent integration of different design notations playing an important role in the development process of production automation systems. The integration of heterogenous specification methods concerning the various heterogenous components of such production automation systems can be achieved by using the ViewPoint framework. This framework provides an intuitive understanding of different specifications involved in the development process, on the one side. On the other side the ViewPoint framework allows for a consistent and correct integration of these specification methods due to its existing formalization by distributed graph transformations.

1. INTRODUCTION AND RELATED WORK

Developing software systems in industrial interdisciplinary projects as in the area of production automation systems many different aspects have to be considered. In many cases these aspects are formulated using different specification notations. Among these descriptions are, for example, specifications of various system components including descriptions of their desired behaviour. In addition specifications have to be considered concerning the communication between these system components in order to perform specific collaborative tasks. A third important aspect is the fact that all these specifications of various kinds belong to many different development stages. This means that different specification notations have to be considered and integrated across different specification stages. Thus, a correct and consistent integration strategy covering all these aspects is needed.

There is a considerable body of results available which uses a view or viewpoint based approach to integrate different representation schemes and software development methods. See, for example, [7] for a review. However, in contrast to these approaches which consider mostly pure information systems development the approach presented here focuses specifically at the integration of development methods from different engineering disciplines: production automation engineering and software engineering. To the best of our knowledge this has been considered only in [5]. This contribution, however, does not consider a precise formal framework, which is the aim of the contribution presented here.

This contribution focuses on an approach to a correct and consistent integration using the ViewPoint framework. This framework allows to integrate specifications using different notations and belonging to different development stages.

After the introduction of a sample production automation system used as running example throughout this paper in chapter 2 the ViewPoint framework is briefly introduced and, subsequently, the formalism of distributed graph transformation is presented in chapter 3. The last section of this chapter shows how the technique of distributed graph transformation is used as formal underlying semantics for the ViewPoint framework. Chapter 4 contains preparations for the integration: the two specification methods state charts and sequential function charts are introduced and are used further for specifying the example system *stoppage of a pallet* as part of the whole production automation system. In chapter 5 a sketch of consistency checks and consistency check actions using ViewPoints is given first. This includes a brief introduction to the notion of correspondence relation and the integration itself. The main point is to establish a correspondence relation allowing to apply suitable checks

and check actions between different ViewPoints, thus leading to the integration of different specifications expressed by respective different ViewPoints. The last chapter closes with a conclusion and remarks on future research.

2. THE SAMPLE PRODUCTION AUTOMATION SYSTEM

Figure 20 shows a simplified technology schema of our sample production system. It consists of three assembly stations and a transportation system. The task of an assembly station is to assemble electro-mechanical devices like switches for surface mounting. The transportation system uses pallets to transport the needed component parts to the assembly stations. Pallets are stopped at certain points in front of an assembly station using a stopper.

The stopper shown in Figure 1 is a pneumatic cylinder. Normally a pallet is moved continuously by the conveyor belt. If the pneumatic cylinder is in extended position the pallet is blocked and stopped. This stopping mechanism is used in this paper as the example scenario.

It is assumed that two engineering teams (an assembly station team and a transportation system team) describe the same aspect of the system -- the *stoppage of a pallet.* Both teams believe that they have to control the stoppage of a pallet. It is also assumed that the teams use different design languages. The transportation system team uses a Programmable Logic Controller (PLC) to control the transportation system. PLCs are programmed in IEC 1131-3 languages. Here, Sequential Function Charts (SFCs) are chosen. The assembly station team uses an industrial PC to control the assembly station specified using State Charts (SCs). Thus there is an obvious need to integrate different approaches to system design.

3. THE VIEWPOINT FRAMEWORK AND ITS FORMALIZATION BY DISTRIBUTED GRAPH TRANSFORMATION

3.1 The ViewPoint Framework

ViewPoints have been successfully used in a wide variety of domains to express different views and plans of participants in a development process. A *ViewPoint* can be thought of as a combination of the idea of an ‘actor‘, ‘knowledge source‘, ‘role‘ or ‘agent‘ in the development process and the idea of a ‘view‘ or ‘perspective‘ which an actor maintains [3]. In software engineering terms a *ViewPoint* is defined to be a locally manageable object or agent encapsulating partial knowledge about the system, its domain, and the design process [6]. The knowledge is specified in a particular suitable representation scheme. An entire system is described by a set of related, distributable, and loosely coupled ViewPoints.

A single ViewPoint (cf. Figure 2) consists of five slots. The *style slot* contains a description of the scheme and notation used to describe the knowledge of the ViewPoint. The *domain slot* defines the area of concern addressed by the ViewPoint. The *specification slot* contains the actual specification of a particular part of the system which is described in the notation defined in the style slot. The fourth slot is called *work plan* and encapsulates the set of actions by which the specification can be built as well as a process model to guide application of these actions. Two classes of work plan actions are especially important*: In-ViewPoint check actions* and *Inter-ViewPoint check actions* are used for checking consistency within a single ViewPoint and between multiple ViewPoints respectively. The last slot of a ViewPoint called *work record* contains the development history in terms of the actions given in the work plan slot.

A *ViewPoint template* (cf. Figure 2) is a kind of ViewPoint type and is described as a ViewPoint in which only the style and the work plan slots are specified while all the others are empty. When creating a new ViewPoint the developer has the opportunity to use an existing ViewPoint template instead of designing the entire ViewPoint from scratch.

The ViewPoint framework is independent from any particular development method and actively encourages multiple representations. Software development methods, techniques and specification methods are generally defined as sets of ViewPoint templates encapsulating the notations provided as well as the rules how they are used. While these rules may refer to a single isolated ViewPoint template integration of methods is realised by rules referring to multiple ViewPoint templates. Using this latter kind of rules the integration of different specification methods is achieved (see the next chapter).

A detailed description of the ViewPoint framework is given in [6], [3]. In the next section the technique used as formal underlying semantics for the ViewPoint framework is sketched: *distributed graph transformation.*

3.2 Distributed Graph Transformation

Graph transformation means the rule-based manipulation of graphs: *if* a certain structure exists *then* it may be transformed into a new one. In system modelling graphs are used to describe states while rules describe state transitions. Among several existing approaches to graph transformation in this contribution we follow the *algebraic double-pushout approach* as presented in [1] which contains an introduction and an overview of the important theoretical results achieved so far. In this approach each node and edge of a graph may be typed and attributed by further data types. A rule consists of three graphs, the *left* and the *righ- hand sides* as well as a gluing graph. The gluing graph is part of the other two graphs and contains all those nodes and edges to be preserved during a transformation. All additional graph objects at the

left-hand side are to be deleted while the ones at the right-hand side not contained in the gluing graph are to be newly created. The gluing graph is necessary to specify how to insert the newly created objects. An application of a rule to a graph is performed by first finding a match of the left-hand side into the graph and then applying the rule to the graph at this match. A start graph describing the initial state together with a set of rules specifying permitted actions form a *graph transformation system.*

Distributed graph transformation as presented in [12] distinguishes two levels of abstraction: the *network* and the *object* level. The network level contains the description of a system's network topology by a network graph and its dynamic reconfiguration during runtime by network rule applications. Each network node may contain the description of a local system again described by a graph transformation system. At this abstraction level graphs describe object structures and rules model object interactions.

The combination of the network graph structure and the local object structures is specified by distributed graphs in order to describe distributed object structures. A *distributed graph* consists of one network graph where each network node is refined by a local graph describing local object structures. Each network edge is refined by a total graph morphism defining relations between local systems. Local systems present object structures accessible for other local systems in export interfaces and those required from remote export interfaces in import interfaces. Communication between local systems only takes place via export and import interfaces.

A *distributed graph rewrite rule* consists of a *network rule* and *object rules* for all network nodes preserved by the network rule. Distributed graph rewrite rules are useful for describing dynamic network reconfiguration by a network rule as well as dynamic local data structures in each component by a set of local rules. Furthermore, remote interactions can be described by distributed rules. In this case all network nodes where parts of the interaction should take place are incorporated into the network graphs of the rule.

A *distributed graph transformation* may change both the network structure as well as object structures in local systems and their interfaces. For performing these actions first all distributed matches have to be found. If all matches are fixed there always exists a unique result for a distributed graph transformation [12].

In the next paragraph it is discussed how distributed graph transformation is used as formal underlying semantics for the ViewPoint framework.

3.3 Formalisation of the ViewPoint Framework by Distributed Graph Transformation

While application-internal states and behaviour of a single ViewPoint can be described by distributed graph transformation at the local level distributed graph transformation at the network level is well suited for coordinating a distributed ViewPoint configuration.

The five slots of a single ViewPoint are formalised by the following aspects of distributed graph transformation:

- The style slot is represented by a local graph transformation system (a start graph and a set of rules).
- The domain slot is described by a local graph (mostly, a single node attached with a domain label suffices).
- The specification slot is represented by a local graph. It should be noted that formalising rules for transferring the specification in the original notation to its graph-based formalisation as well as for transferring the graph-based formalisation back to the original notation have to be specified by the developer of the ViewPoint template. However, these rules are in most cases easy to develop since most requirements engineering methods are based on diagrammatic notations.
  For common linear languages there exists already an approach in [8] based on syntax trees. Suitable techniques for more complex cases concerning the transformation from and to general notations will be investigated in future research.
- The actions of the work plan slot are formalised by distributed graph transformation rules for accessing multiple ViewPoints. Network graph rewrite rules for coordinating a ViewPoint configuration and local graph rewrite rules for accessing a single isolated ViewPoint.
- The work record slot is represented by a local tree the edges of which are labelled with applied development actions (i.e. applied graph rewrite rules defined in the work plan). Then links have to be established from the tree nodes to nodes of the specification graph for tracking the effects of development actions.

A formalisation of the specification slot as well as of parts of the work plan slot is given in [9] which focuses on ViewPoint-oriented software development while a detailed formalisation of assembly, In-ViewPoint check and especially Inter-ViewPoint check actions allowing for integration of specification methods is presented in this contribution.

4. THE NEED FOR INTEGRATING STATE CHARTS AND SEQUENTIAL FUNCTION CHARTS

In this chapter essential preliminaries for the integration of two particular specification methods using the ViewPoint framework will be presented. These are *State Charts* (SC) and *Sequential Function Charts* (SFC).

The important point here is that in industrial interdisciplinary projects generally engineers of different disciplines have to work together. In this contribution specification methods concerning production automation

systems will be considered. Such systems consist of several system components collaborating with each other in order to perform specific production requests. The important goal of designing a flexible and modular software architecture for production automation systems of this kind requires to investigate the communication and collaboration between the involved system components. This results in the need of integrating heterogenous descriptions of the respective system components. Among the notations of such descriptions one can find typical engineering notations, like IEC 1131-3, as well as those mainly used by computer scientists like Realtime-UML. Thus, it is an important task to integrate correctly and consistently quite different specification methods during the entire development process.

In this paper the two specification methods SFC as an engineering method and SC as a method used mainly by software designers are discussed. The *stoppage of a pallet* as part of the transportation system of the production automation system serves as an example (cf. explanations of chapter 5). This small part of the dynamics of the whole complex system will be described by a SC as well as by a SFC in order to be able to relate these two different specifications to each other. For the same part of the entire system will be presented as a SC as well as described by a SFC it will become more obvious whether the two specifications contain consistent parts and where mismatches exist between them.

First, a brief comparison of the SC and SFC methods is presented in order to describe their relationship. A SC ViewPoint for the example as well as a SFC ViewPoint are described in order to sketch how to use ViewPoints. Then, In-ViewPoint checks and In-ViewPoint check actions are presented to show how inconsistencies within an isolated single ViewPoint can be detected and handled respectively. Finally, correspondence relations are introduced which are needed to integrate different specification methods by applying Inter-ViewPoint checks and Inter-ViewPoint check actions between multiple distributed ViewPoints.

4.1 Comparison of State Charts and Sequential Function Charts

State Charts and Sequential Function Charts do both have a very rich syntax. A complete comparison of these languages is beyond the scope of this contribution. We use the example specifications for the *stoppage of a pallet* for comparison (Figure 21). For SCs the notation of UML-Realtime [2] and for SFCs the notation presented in [13] is used. Both diagrams express the same aspect: The pneumatic cylinder can be at two positions - extended and rested. In extended position the pallet is blocked. In rest position the pallet is released or unblocked. State Charts consist of states and transitions. Sequential Function Charts consist of steps and transitions. In SCs the actual situation of the system is determined by the active state. In SFCs active steps determine the system situation. In both languages changes of system states are expressed by transitions.

Transitions in SCs must have a condition. In UML-Realtime this condition is normally a signal. In most cases an action is connected to a transition. This action is executed if the transition is fired. In the example with transition `block` an action is associated forcing the pneumatic cylinder to go into extended position, and a signal `timeout_sign` sent by a timer. With transition `unblock` an action is associated forcing the rest position of the cylinder, and a signal `ext_sign` which is sent by a control device. Transitions `Initial` and `Final` are for initialisation and finalisation of the system.

Transitions in SFCs must also have a condition but may not have an associated action. Conditions are Boolean expressions which can contain input variables that are set by external signals. With $\mathtt{Trans}_1$ a Boolean input variable `Unblock` is associated which is set by the signal of a control device. With $\mathtt{Trans}_2$ an expression is associated which becomes `true` if the step `Unblocked` is longer than 32 ms (milliseconds) active. $\mathtt{Trans}_0$ is always active.

States in SCs can have an entry and exit action. Furthermore, they can consist of an inside state chart which describes the behaviour of the system in this state. State `unblocked` contains an entry action which causes a timeout signal from a timer after 32 ms.

Steps in SFCs can have associated actions. In the example with step `Blocked` an action is associated which forces the pneumatic cylinder to go into extended position. With step `Unblocked` an action is associated which forces the rest position of the cylinder.

4.2 State Chart and Sequential Function Chart ViewPoints for the Example

In order to build a SC specification for the example *stoppage of a pallet* graph rewrite rules specifying assembly actions located in the work plan slot of the SC ViewPoint are needed. Figure 3 shows the first *rule add isolated state (String ViewPoint; State state; StringList actions, variables)* specifying the addition of an isolated state within a SC. By this parameterised rule the left-hand side is replaced by the right hand side in case of the existence of a match within the graph to be modified. In addition to the match, the negative application condition `NAC` has to be satisfied for an application. Thus, if a state with name *state* does not yet exist within the specification slot of the ViewPoint *ViewPoint* then exactly this *state* will be added together with the corresponding *actions* and *variables* a state in a SC may have. The gluing graph necessary due to the algebraic double-pushout approach used here is not shown at this point and in the following. Since it has to be a subgraph of both hand sides of the rule it is empty in this case. Finally, the set `NV` at the bottom of the rule specifies all those variables attached to nodes which by all means have to be actualized before a match

takes place. For dealing with `NV` and `EV`, an analogous set of variables for edges, cf. [8].

In a specification process it is most natural to sometimes delete specification parts if they become redundant or inconsistent. Therefore, also rules like the one for deleting a state in a SC as presented in Figure 4 are important. Further, Figure 22, Figure 23, and Figure 5 to Figure 8 show assembly rules for adding and deleting a transition between two states, the start state, and, finally, a transition between a state and the start state. Analogously to the assembly rules concerning the start state, there also exist rules concerning the final state*: add stop (String ViewPoint), delete stop (String ViewPoint), connect stop (String ViewPoint, state), and disconnect stop (String ViewPoint, state).*

The assembly rules presented above allow to build a SC specification of a ViewPoint specifying *the stoppage of a pallet*. First, an empty specification slot for the ViewPoint VP_{SC} which shall contain the SC specification for the example will be created by performing the trigger rule (located in the work plan) *create* VP_{Spec} (VP_{SC});

Suppose, now, the following series af actions will be performed (recorded in the work record):
add start (VP_{SC});
add isolated state (VP_{SC}, blocked, „“, „“);
connect start (VP_{SC}, blocked);
add isolated state (VP_{SC}, unblocked, set_timeout, 32);
add transition (VP_{SC}, blocked, unblocked, unblock, unblock, ext_sign);
add transition (VP_{SC}, unblocked, blocked, block, block, timeout-sign);
add stop (VP_{SC});
connect stop (VP_{SC}, blocked);

Then the specification slot of the ViewPoint VP_{SC} will be equipped with the SC as presented in Figure 9. It should be noted at this point that the rule for adding a transition between two states (cf. Figure 22) specifies a name, an operation and a signal for the transition to be added. However, in the example stated here the name and the operation of a transition always are equal which is the reason for denoting the transitions only once by their name (cf. transitions `block` and `unblock` in Figure 9).

Analogously, an SFC specification can be built by applying corresponding assembly rules. The specification slot of the ViewPoint VP_{SFC} will be equipped with the SFC as presented in Figure 10 by applying the following series of rules (see also above):

create VP_{Spec} (VP_{SFC});
add isolated step (VP_{SFC}, Blocked);
add isolated transition (VP_{SFC}, $Trans_1$, Unblock);
connect step to transition (VP_{SFC}, Blocked, $Trans_1$);
add isolated step (VP_{SFC}, Unblocked);
connect transition to step (VP_{SFC}, $Trans_1$, Unblocked);
add isolated transition (VP_{SFC}, $Trans_2$, Unblocked.T_gt_32ms);
connect step to transition (VP_{SFC}, Unblocked, $Trans_2$);
add start stop cond (VP_{SFC}, $Trans_2$, Blocked);

Concerning these SC and SFC specifications a simple example for checking and handling an inconsistency within a single isolated ViewPoint is presented in the next section.

4.3 In-ViewPoint Check and In-ViewPoint Check Action

Considering the SC specification in Figure 9 it could be useful to check whether a specific state in the VP_{SC} specification has a single transition. The assembly rules given in the previous section for building a SC specification allow specifying such a single edge provided that the algebraic double-pushout approach permits non-injective matches (cf. [1]). In this contribution the definition of inconsistency according to [4] is used: an inconsistency is regarded as any situation in which two parts of a specification do not obey some relationship that should hold between them. Thus, if a single edge is found then this shall be a detected inconsistency to be handled.

The detection can be realised by the *In-ViewPoint check* specified by the *In-ViewPoint check rule* given in Figure 11. If this rule matches in a graph to be modified then a single unwanted edge at a specific state is found. One kind of handling such a detected inconsistency (cf. the process model for handling inconsistencies proposed in [4]) can be the deletion of this edge. This can be realised by the *In-ViewPoint check action* specified by the *In-ViewPoint check action rule* given in Figure 12.

We now have introduced assembly rules for creating ViewPoint specifications, In-ViewPoint check and In-ViewPoint check action rules for detecting and handling inconsistencies within a single isolated ViewPoint, respectively. Below correspondence relations are presented allowing the integration of different specification methods by applying suitable Inter-ViewPoint check and Inter-ViewPoint check action rules (cf. next chapter).

4.4 Correspondence Relations

In contrast to In-ViewPoint checks and In-ViewPoint check actions which handle inconsistencies within a single isolated ViewPoint Inter-ViewPoint checks and Inter-ViewPoint check actions take place between multiple distributed ViewPoints. As pointed out above this concept realizes the integration of different specification methods.

Suppose, a specific ViewPoint – containing a SC specification – initiates an inconsistency check wrt a particular feature within another ViewPoint which contains a SFC specification (cf. Figure 13). First a relationship between these two ViewPoints has to be established in order to establish a correspondence between relevant elements concerning the inconsistency check

required. Then, the particular inconsistency can be checked and potentially be handled as well.

The distributed graph transformation approach serves nicely this purpose. Multiple ViewPoints are given as a distributed graph. In Figure 14 it is shown how a so-called *correspondence relation* may be established between a `state` in the ViewPoint VP_{SC} and a `step` in the ViewPoint VPS_{FC}. This is realised according to distributed graph transformation (cf. [12] for a more detailed explanation) by

- first, specifying a unidirectional edge leading from node VP_{SC} to node VP_{SFC} (one arrow between the two interface nodes at the network level).
- second, making the `state` an import element (an import interface graph consisting of the single node `state`) and the `step` an export element (an export interface graph consisting of the single node `step`) such that these two interface graphs may be connected by a total graph morphism (at the local level).

Here and in the following an import interface node is presented by a light grey background and black characters with `i`-extension. An export interface node is presented by a dark grey background and white characters with `e`-extension.

Applying the graph rewrite rules *import state* (VP_{SC}, blocked), *export step* (VP_{SFC}, Blocked), *and connect state and step* (VP_{SC}, VP_{SFC}, blocked, Blocked) would realise the steps given above of establishing a correspondence relation between a `state` of the ViewPoint VP_{SC} and a `step` of the ViewPoint VP_{SFC}. Both specify the *stoppage of a pallet*. This application results in the distributed graph in Figure 15 allowing, later on, to apply Inter-ViewPoint checks and Inter-ViewPoint check actions, respectively. In this case (and in the following cases) the third rule given above realises connections at the network as well as at the local level.

The interesting point here is that the edge at the network level and the total graph morphism at the local level **belong to the specification** of the distributed graph. This morphism provides the basis for applying inconsistency checks and inconsistency check actions as Inter-ViewPoint check and Inter-ViewPoint check action rules respectively (cf. next chapter).

It should be noted here that the *notion correspondence relation* has already been introduced in [10] where it is defined as a bi-directional relation. In this approach, however, unidirectional relationships between multiple ViewPoints are important. This justifies the slightly different definition of a correspondence relation given here.

In the next chapter the integration of the two specification methods SC and SFC will be presented by applying Inter-ViewPoint checks and Inter-ViewPoint check actions to the corresponding ViewPoints specified by one distributed graph through a suitable correspondence relation.

5. INTEGRATION: ESTABLISHING A CORRESPONDING FEATURE BETWEEN SPECIFICATION PARTS

A useful requirement during the development process concerning a flexible and modular software architecture for the production automation system could concern investigating and comparing corresponding parts of the dynamics between the SC and the SFC, respectively.

For example, in the SC it is ensured to get into the state `blocked` from the state `unblocked` by the transition `block`. This can easily be seen considering the correct syntax of the SC in the specification slot of the ViewPoint VP_{SC} (cf. Figure 9). Because this is an important feature to be investigated in the following it is called *SC-feature*.

Suppose now, the SC ViewPoint VP_{SC} would have the intention during the specification process to proof or disprove the existence of a suitable feature corresponding to the SC-feature within the SFC specification. Suppose further, the specification of the SFC ViewPoint VP_{SFC} would lack the transition $\texttt{Trans}_2$ including its connections to the rest of the graph (cf. Figure 10 and Figure 16).

The goal of checking wrt and establishing a (possibly missing) suitable feature corresponding to the SC-feature described above within the SFC specification means an integration of the two ViewPoints VP_{SC} and VP_{SFC} wrt this feature. An integration can be realized by

- establishing a suitable correspondence relation, then (cf. last section of previous chapter)
- applying a suitable Inter-ViewPoint check, and, finally
- applying a suitable Inter-ViewPoint check action.

5.1 Establishing a suitable correspondence relation

A suitable correspondence relation is achieved by identifying exactly which parts of the specifications of the two ViewPoints have to do with the requirement of checking wrt. and, if necessary, of establishing a feature within VP_{SFC} corresponding to the SC-feature initiated by VP_{SC}. In the SC specification of ViewPoint VP_{SC} the SC-feature means as described above the following dynamics part that the state `unblocked` may be left through the transition `block` reaching the state `blocked`. In the SFC specification of ViewPoint VP_{SFC} a corresponding feature is specified if there exists a transition by which the step `Blocked` can be reached from the step `Unblocked`. The requirement to check and establish a feature within VP_{SFC} corresponding to the SC-feature initiated by VP_{SC} implies the following policy:

- first, it has to be found out that the feature corresponding to the SC-feature is missing within VP_{SFC} by a suitable Inter-ViewPoint check initiated by VP_{SC}
- second, the missing feature has to be added within VP_{SFC} by a suitable Inter-ViewPoint check action initiated by VP_{SC}.

This knowledge allows to specify a correspondence in the following manner: the state `blocked` corresponds to the step `Blocked`, and analogously, the state `unblocked` to the step `Unblocked`. There is no other item within the SC corresponding to any element of the SFC **and** being relevant also for the required feature. Thus, the relevant interface graphs within both ViewPoints and their correct connection, i.e. a total graph morphism (cf. [12]), can be specified by the rules

import state (VP_{SC}, blocked);
import state (VP_{SC}, unblocked);

specifying the import interface graph in ViewPoint VP_{SC}

export step (VP_{SFC}, Blocked);
export step (VP_{SFC}, Unblocked);

specifying the export interface graph in ViewPoint VP_{SFC},

connect state and step (VP_{SC}, VP_{SFC}, blocked, Blocked);
connect state and step (VP_{SC}, VP_{SFC}, unblocked, Unblocked);

specifying the connection between corresponding elements at the network as well as at the local level.

Obviously, the connection between the two ViewPoints VP_{SC} and VP_{SFC} at the local level is specified by a total graph morphism. The resulting distributed graph is depicted in Figure 17.

The correspondence relation realised as described above is a necessary prerequisite to check this specification wrt inconsistencies and for potentially handling detected inconsistencies in an appropriate way. The inconsistency check as well as the handling of detected inconsistencies is performed by Inter-VierwPoint check and Inter-ViewPoint check action rules, respectively.

In the next section we show how an Inter-ViewPoint check rule initiated by the SC ViewPoint VP_{SC} can detect that a feature corresponding to the SC-feature is missed within the SFC ViewPoint VP_{SFC}.

5.2 Applying a suitable Inter-ViewPoint check rule

A distributed graph rewrite rule being able to detect an inconsistency does **not change** anything within a distributed graph to be checked. Rather, its left and right hand sides are specified by the same graph and its successful match within the distributed graph reveals an inconsistency.

The *rule check transition not exists (String ViewPoint1, ViewPoint2; State state1, state2; Step step1, step2; Transition1 transition1; String operation; Signal signal)* if actualised by correct parameters for application, e.g., *check transition not exists* (VP_{SC}, VP_{SFC}, blocked, unblocked, Blocked, Unblocked, block, block, timeout_sign), can be seen in Figure 24. Due to readability it is given here in a concrete form rather than in an abstract one. The rule checks whether a transition concerning a feature corresponding to the SC-feature **does not exist** within the SFC specification of ViewPoint VP_{SFC}. This is exactly specified by *the negative application condition* `NAC` specifying that the rule matches only if such a transition does not exist. The left and right-hand sides L and R of the rule specify the relevant sub-graphs of the two ViewPoint specifications. These are exactly the two interface graphs described in the previous section together with a body element in the SC, namely the edge attached by the name/operation `block` and by the signal `timeout_sign`. A body element is not part of an interface graph, i.e. is hidden (cf. [12]). It is presented, though, within this rule because it will become the correspondent part of an edge still missed within the SFC specification.

For the first moment the rule *check transition not exists* seems to be too complicated for the task it is designed for because it does not really change anything. In order to ensure a mathematically correct and a consistent integration, however, it is important to use a **uniform** transformation mechanism. Additionally, in case of a complex implementation later on simplifying techniques will be investigated for realizing rules of such a kind.

The Inter-ViewPoint check rule described above detects the inconsistency of a missing transition concerning the required feature within VP_{SFC}. This inconsistency is handled by a suitable Inter-ViewPoint check action in the next section.

5.3 Applying a suitable Inter-ViewPoint check action rule

Having detected the inconsistency of a missing transition concerning the required feature within VP_{SFC} – as described in the previous section – its handling will be presented in this section. This is performed by an Inter-ViewPoint check action specified by a distributed graph rewrite rule. There are several possibilities of handling inconsistencies (cf. the penultimate section above). In the case discussed here the missing transition concerning the required feature in the SFC specification of ViewPoint VP_{SFC} will be added. More in detail, the missing transition with its connections to the rest of the SFC graph will be inserted by a suitable distributed graph rewrite rule.

The rule *add transition (String ViewPoint1, ViewPoint2; State state1, state2; Step step1, step2; Transition1 transition1; String operation; Signal signal; Transition2 transition2)* if actualised with the correct parameters for application, namely, *add transition* (VP_{SC}, VP_{SFC}, blocked, unblocked, Blocked, Unblocked, block,

block, timeout_sign, $Trans_2$), can be seen in Figure 25. In contrast to the rule in the previous section this rule changes something. It inserts a transition `Trans2` for the detected missing one together with its connections with the rest graph within ViewPoint VP_{SFC}. In detail, these connections are represented by two edges leading from step `Unblocked` to transition `Trans2` and from transition `Trans2` to step `Blocked`, respectively. The important point here is that the condition belonging to a SFC transition can not be specified exactly by such a rule. Only certain knowledge *about* this condition may be specified. The condition belonging to the transition `Trans2` has to depend on the result of the operation `set-timeout` belonging to the state `unblocked`, i.e., more precisely, it depends on the signal `timeout_sign` belonging to the transition `block` in the SC. As can be seen, this dependence is indicated in the specification through the right hand side of the rule in Figure 25. A more concrete specification of the condition attached to the transition `Trans2` can finally be done by the requirements engineer dealing with the ViewPoint VP_{SFC}.

However, the rule also does something more (cf. Figure 9). The original body element edge `block` in the SC becomes an import interface element leading from VP_{SC} to VP_{SFC}. Thus, while actualised in an appropriate way a total graph morphism is maintained as required by the distributed graph transformation approach (cf. [12]).

The application of the Inter-ViewPoint check action rule described above to the distributed graph depicted in Figure 17 results in the distributed graph in Figure 18. This graph now contains the originally missed transition within the SFC specification of the ViewPoint VP_{SFC}. More concretely, this transition means the required feature corresponding to the original SC-feature within the SC specification of the ViewPoint VP_{SC}, specifying the dynamics of getting from state `unblocked` to state `blocked` by the transition `block`.

At this point the requirement initiated by the ViewPoint VP_{SC} is achieved. The correspondence relation is no longer necessary. It can be removed. The following distributed graph rewrite rules perform this task:

disconnect state from step (VP_{SC}, VP_{SFC}, blocked, Blocked);
disconnect state from step (VP_{SC}, VP_{SFC}, unblocked, Unblocked);
disconnect edges (VP_{SC}, VP_{SFC}, block, edge (Trans2, Blocked));

specifying the disconnections between the two ViewPoints VP_{SC} can VP_{SFC},

hide connection step to transition (VP_{SFC}, Unblocked, Trans2);
hide connection transition to step (VP_{SFC}, Trans2, Blocked);
hide transition (VP_{SFC}, Trans2);
hide step (VP_{SFC}, Unblocked);
hide step (VP_{SFC}, Blocked);

making the export interface graph as subgraph of the SFC a body graph again,

hide state (VP_{SC}, unblocked);
hide state (VP_{SC}, blocked);
hide transition (VP_{SC}, block, block, timeout_sign);

making the import interface graph as subgraph of the SC a body graph again. The result of transforming the distributed graph depicted in Figure 18 into two single isolated graphs again is shown in Figure 19.

Integration of different single isolated ViewPoints during a specification process is possible by specifying suitable correspondence relations, checking suspected inconsistencies and, finally, handling possibly detected ones. All these tasks are realisable correctly and consistently by distributed graph transformation rules.

6. CONCLUSION AND FUTURE RESEARCH

In this contribution we presented an approach how different specifications of important component features in the domain of production automation systems can be integrated. In order to achieve a correct and consistent integration of different specification methods the ViewPoint framework was used. As an underlying formal semantics the technique of distributed graph transformation is applied. We investigated as example system/component the *stoppage of a pallet* as part of the entire production automation system. It was specified by a state chart ViewPoint as well as by a sequential function chart ViewPoint. These two specifications could then be integrated by establishing a suitable correspondence relation and, further, by applying suitable Inter-ViewPoint check rules as well as Inter-ViewPoint check action rules.

For future research it is challenging to investigate integration concerning other and more complex aspects during development processes. Interesting considerations for the integration are e.g.

- specifications of different system components and using the same specification method,
- specifications of different system components and using also different specification methods,
- specifications of either system components and using either specification methods, but being in different specification stages, thus, pushing forward the development process,

Also the joint work of [11] will be used to accomplish the integration task.

Results of these and similar research aspects promise to provide correct and consistent development processes of software systems for production automation systems by a suitable integration of arbitrary relevant specifications.

ACKNOWLEDGEMENTS

The authors would like to thank Torsten Meyer for his helpful inspirations within the world of ViewPoints and of distributed graph transformation.

REFERENCES

[1] Corradini, A., Montanari, U., Rossi, F., Ehrig H., Heckel, R., and Löwe, M., *Algebraic Approaches to Graph Transformation -- Part I: Basic Concepts and Double Pushout Approach*, in Rozenberg, G. (ed.), Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1, World Scientific, 1997.

[2] Douglass, B. P., *Doing Hard Time -- Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*; Addison-Wesley, 1999.

[3] Easterbrook, S., Finkelstein, A., Kramer, J., and Nuseibeh, B., *Coordinating Distributed ViewPoints: the anatomy of a consistency check*, Concurrent Engineering: Research & Applications, vol. 2, CERA Institute, USA, 1994.

[4] Easterbrook, S., and Nuseibeh, B., *Using ViewPoints for Inconsistency Management*, BCS/IEE Software Engineering Journal, pp. 31-43, 1996.

[5] Finkelstein, A., Nuseibeh,B., Finkelstein, L., Huang, J., *Technology Transfer: Software Engineering and Engineering Design*, IEEE Computing & Control Engineering Journal, vol. 3(6), pp. 259-265, November 1992.

[6] Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., and Goedicke, M., *ViewPoints: A Framework for Integrating Multiple Perspectives in System Development*, Int. Journal of Software Engineering & Knowledge Engineering, vol. 2(1), 1992.

[7] Finkelstein, A., Sommerville, I., *The ViewPoints FAQ*, Software Engineering Journal, vol. 11(1), pp. 2-4, 1996.

[8] Goedicke, M., *On the Structure of Software Description Languages: A Component Oriented View*, Habilitation Thesis, Research Report No. 473/1993, University of Dortmund, Germany, 1993.

[9] Goedicke, M., Enders, B., Meyer, T., and Taentzer, G., *Tool Support for ViewPoint-oriented Software Development: Towards Integrating Multiple Perspectives by Distributed Graph Transformation*, Proceedings International Workshop and Symposium AGTIVE - Applications of Graph Transformations with Industrial Relevance, Monastery Rolduc, Kerkrade, NL, September 1-3, 1999.

[10] Goedicke, M., Meyer, T., and Taentzer, G., *ViewPoint-oriented Software Development by Distributed Graph Transformation: Towards a Basis for Living with Inconsistencies*, Proceedings of the 4th IEEE International Symposium on Requirements Engineering, Limerick, Ireland, 1999.

[11] Große-Rohde, M., Heverhagen, T., Enders, B., Goedicke, M., *Integration of UML-Realtime and the IEC1131-3 language*, forthcoming.

[12] Taentzer, G., Fischer, I., Koch, M., and Volle, V., *Distributed Graph Transformation with Application to Visual Design of Distributed Systems*, to appear in Rozenberg, G. (ed.), Graph Grammar Handbook 3: Concurrency and Distribution, World Scientific, 1999.

[13] Programmable Controllers - Part 3: Programming languages (IEC 1131-3:1993), 1993.

FIGURES

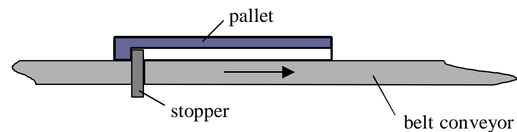


Figure 1 The stopping point

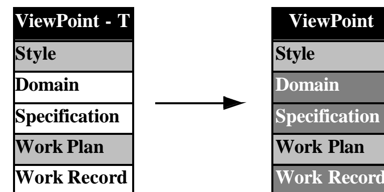


Figure 2 The structure of *aViewPoint* instantiated from a corresponding suitable *ViewPoint template*

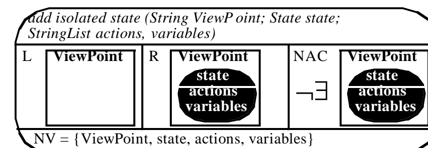


Figure 3 Assembly rule for adding an isolated state into a state chart specification provided that it does not yet exist

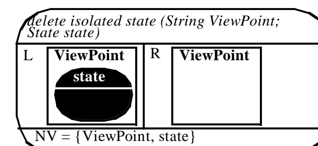


Figure 4 Assembly rule for deleting an isolated state in a state chart specification

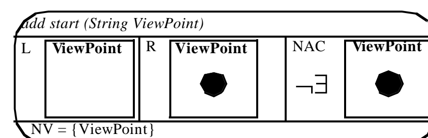


Figure 5 Assembly rule for adding the start state in a state chart specification provided that it does not yet exist

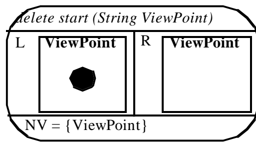


Figure 6 Assembly rule for deleting the start state in a state chart specification

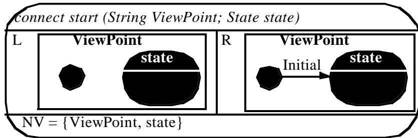


Figure 7 Assembly rule for adding the transition between the start state and an isolated state in a state chart specification

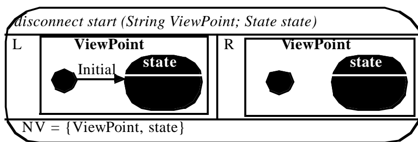


Figure 8 Assembly rule for deleting the transition between the start state and an isolated state in a state chart specification

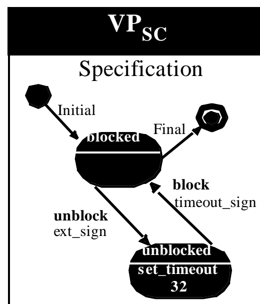


Figure 9 Specification slot of the ViewPoint VP_{SC}: SC specification built by the assembly rules given in the work plan

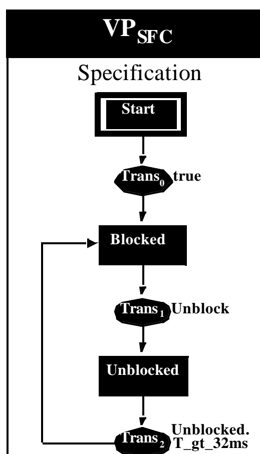


Figure 10 Specification slot of the ViewPoint VP_{SFC}: SFC specification

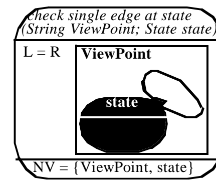


Figure 11 Inconsistency check rule for checking whether a state in a state chart specification has a single transition

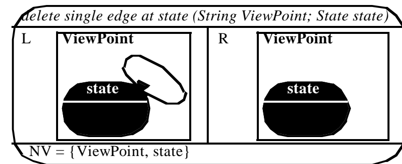


Figure 12 Inconsistency check action rule for deleting a detected single transition at a state in a state chart specification

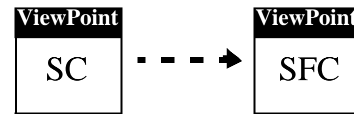


Figure 13 Requirement of an inconsistency check within a SFC ViewPoint initiated by an SC ViewPoint

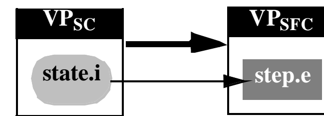


Figure 14 A correspondence relation leading from the SC ViewPoint VP_{SC} to the SFC ViewPoint VP_{SFC}: a uni-directional edge (at the network level) and a total graph morphism (at the local level)

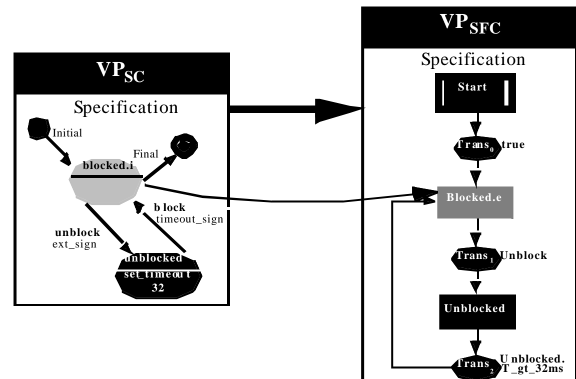


Figure 15 Distributed graph as a result of applying graph rewrite rules for establishing a correspondence relation between a state of the ViewPoint VP_{SC} and a step of the ViewPoint VP_{SFC}

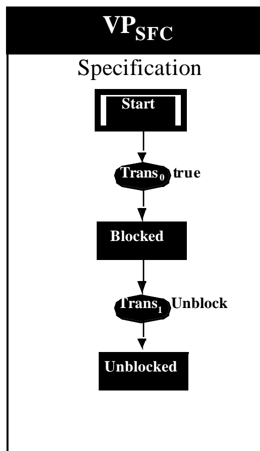


Figure 16 Specification slot of ViewPoint VP_{SFC}: SFC specification without transition between `Unblocked` and `Blocked`

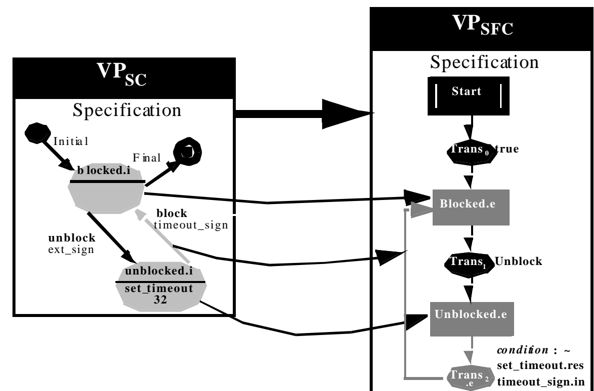


Figure 18 Result after application of an Inter-ViewPoint check action rule to the distributed graph depicted in Figure 17

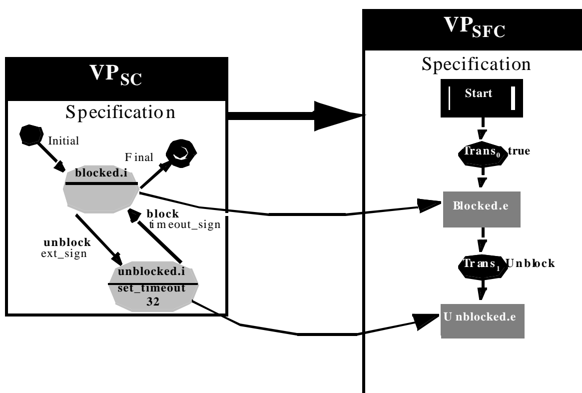


Figure 17 Established correspondence relation between the two ViewPoints VP_{SC} and VP_{SFC}

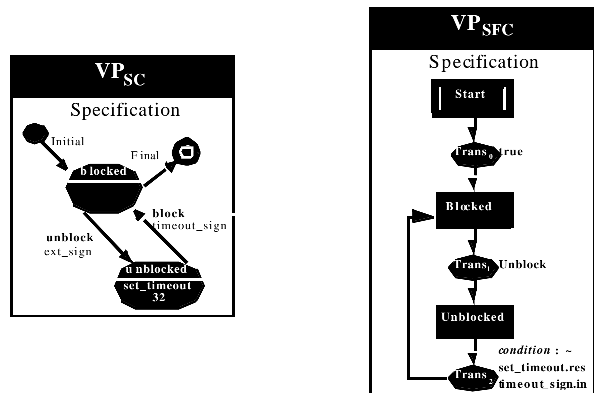


Figure 19 Result after removing the correspondence relation in the distributed graph depicted in Figure 18

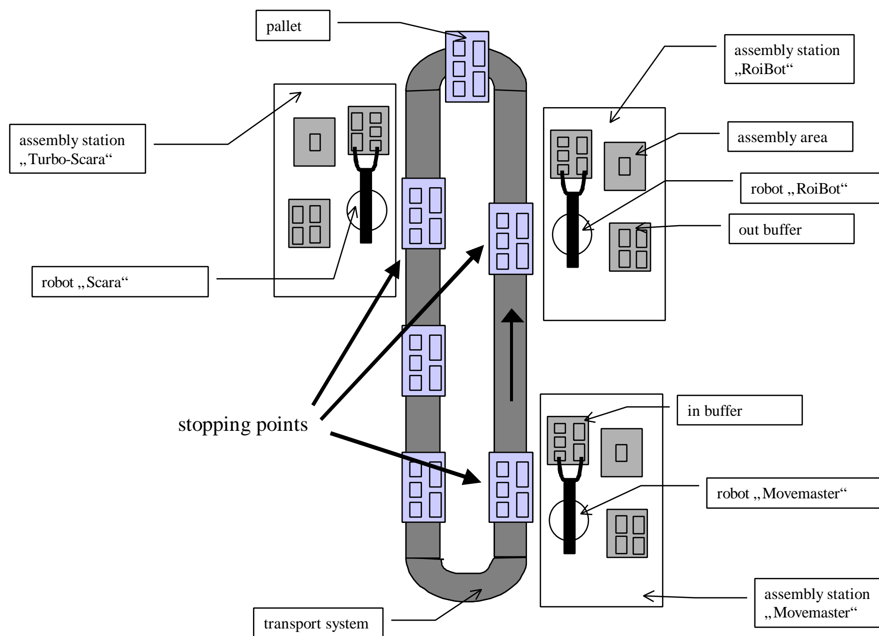
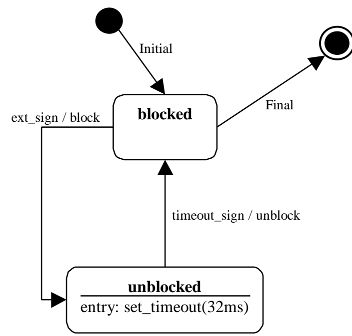
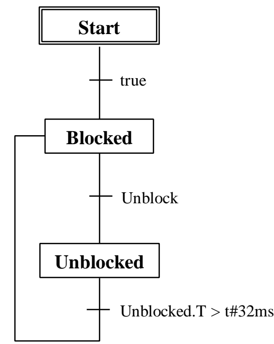


Figure 20 Technology schema of the production automation system



State Chart



Sequential Function Chart

Figure 21 State Chart and Sequential Function Chart for the stopper

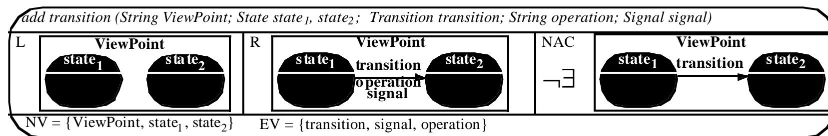


Figure 22 Assembly rule for adding a transition between two states in a state chart specification provided that it does not yet exist

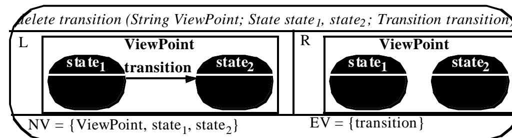


Figure 23 Assembly rule for deleting a transition between two states in a state chart specification

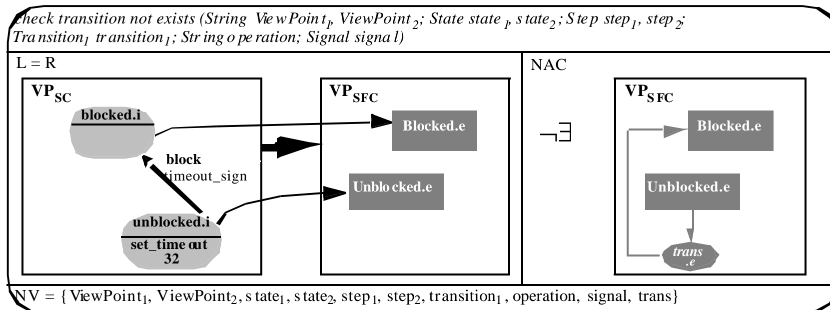


Figure 24 Inter-ViewPoint check rule for checking the inconsistency of a missing transition within the ViewPoint VP_{SFC} initiated by the ViewPoint VP_{SC}

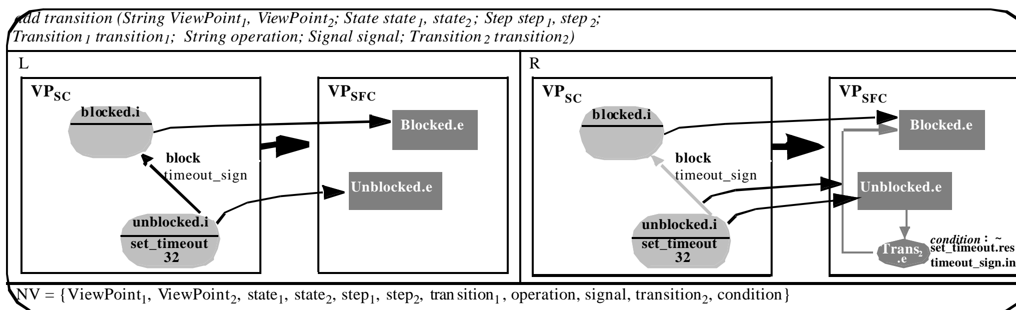


Figure 25 Inter-ViewPoint check action rule initiated by the ViewPoint VP_{SC} for handling the inconsistency of a missing transition within the ViewPoint VP_{SFC} detected by the Inter-ViewPoint check rule depicted in Figure 24