A Multi-View Component Modelling Language for Systems Design: Checking Consistency and Timing Constrains

Juan de Lara
Escuela Politécnica Superior
Universidad Autónoma
Madrid, Spain
jdelara@uam.es

Esther Guerra
Ingeniería Informática
Universidad Carlos III
Madrid, Spain
eguerra@inf.uc3m.es

Hans Vangheluwe
School of Computer Science
McGill University
Montreal, Canada
hv@cs.mcgill.ca

Abstract

In this paper, we present our component modelling language called MiCo and show how different kinds of consistency – syntactic, static semantics and dynamic (timing) constrains – can be checked. MiCo has several views for structural and behaviour specification. The latter includes a view for specifying protocols for the component ports. For this view, we introduce a higraph-based notation to describe timed event regular languages.

The language has been defined using a meta-modelling approach using the AToM^3 tool. The different diagrams of the language are defined by selecting the necessary elements from the global meta-model. In the generated environment, syntactic consistency is automatically achieved by building a unique model, gluing the different view models that the users have built. The gluing is performed by executing some automatically generated triple graph grammars. In addition, the designer can specify additional triple rules for static semantics consistency checking. As an example of dynamic semantics checking, we show how, when connecting two ports, event language inclusion can be checked, taking into account the timing constraints.

Keywords: Meta-Modelling, Visual Languages, Multiple Views, Consistency, Timing Constraints, Component Models, Graph Transformation.

1. Introduction

The complexity of man-made systems is growing at a phenomenal rate. In the most general case, these systems are composed of hardware and software elements. For both kinds of element, one has to gather their functional requirements, evolve them into a design, build the system, test and so forth. Diagramatic notations are pervasive in most of these phases. Different diagrams are needed to describe different perspectives of the system to be built. For example the UML [21] proposes several diagrams to describe structure, behaviour and element interaction, deployment, etc. SysML [20] enriches UML and broadens its applicability to mixed hardware, software and mechanical domains.

Component-Based Modelling (CBM) seeks increasing productivity, quality and reusability through the use of a high abstraction level concept: the component. This is a *"modular unit with well-defined interfaces that is replaceable within its environment"* [21]. The internals of a component are hidden and interaction occurs via *provided* and *required* interfaces, usually exposed via ports. Thus, CBM promotes a shift from monolithic specifications of systems to composing systems by plugging components together through their interfaces. Rich specification techniques [3] become necessary in order to express component characteristics from different viewpoints. This includes not only functional requirements, but also non-functional, such as timing and reliability constraints.

Thus, in multi-view and component-based VLs, mechanisms to ensure consistency between the different view models become essential. There are different levels of consistency. The simpler one is *syntactic consistency*, which ensures that the combination of all the view models results in a syntactically correct system description. *Semantic consistency* can be divided in static and dynamic consistency. The former extends syntactic consistency with additional language-specific restrictions (well-formedness rules). The latter requires restrictions on the way the different parts of the system behave.

The aim and contributions of this paper are, on the one hand, to present our approach to formally handle the different kinds of consistency. The approach is based on graph transformation to build a unique model with the different view models, where all consistency checks and further analysis can be performed. On the other hand, we present a new notation for the specification of port protocols that includes timing constraints. The notation is included in our modelling language MiCo (which stands for <u>Mi</u>nimal

Components). It permits port conformance analysis when connecting components, promoting correct reuse. In addition, these concepts have been newly incorporated in our meta-modelling tool AToM^3 [4].

This paper is organized as follows. Section 2 introduces some graph transformation concepts that will be needed later. Section 3 shows our approach to consistency checking. Section 4 presents our systems modelling language, through an example of the modelling of a wireless home video system. In subsections 4.1 and 4.2, we present our notation for protocol behaviour specification. Section 5 discusses tool support for the previous concepts. Section 6 compares with related research and finally section 7 ends with the conclusions and prospects for further research.

2. Graph Transformation

Graph transformation [8] is a declarative and visual means to manipulate graphs based on rules. A graph grammar is made of a set of rules and a starting graph. Rules are chosen non-deterministically and are tried to be applied to the graph. A rule is applicable if an occurence of the left hand side (LHS) is found in the graph. If the rule is applied, the identified part in the graph can be substituted by the right hand side (RHS). Rules can be equipped with a set of additional application conditions, the simpler form of them are *negative application conditions* (NACs). In this way, in order for the rule to be applicable, no occurrence of the NAC has to be found in the graph. Note that it is possible to have non-injective occurrences of the LHS and NAC (i.e. two or more edges or nodes in the LHS can be identified into a single one in the graph).

One of its formalizations (the one we use in this paper) is called Double Pushout (DPO) and is based on category theory [8]. The categorical approach has the advantage that the results are valid for any adhesive category and not just for graphs. In particular, in multi-view environments, we are interested in rewriting triple graphs [19]. A triple graph is made of three graphs and two span morphisms from one of them to the other two: $G_1 \xleftarrow{v_1} LINK \xrightarrow{v_2} G_2$. Graph triples can be manipulated by means of triple graph grammars [19] (TGGs). In [10] we defined TGGs using the DPO approach, extended them with application conditions and used them in combination with meta-modelling to describe concrete-to-abstract syntax transformations. Here we use them as a means to glue the different view models into a unique repository. Thus, we use triple graphs, where one of the graphs is a view model, another is the repository, and the graph in between is used to relate the elements in the view and the repository.

3. Multiple Views and Consistency

A VL may be composed of a number of diagrams. The definitions of all of them can be based on a unique meta-model, which relates their abstract syntax concepts. This is for example the case of UML2.0. The different diagram definitions may have some overlapping parts in the meta-model. If we identify the overlapping parts for each pair of diagrams, then we obtain the structure shown in Figure 1. In the figure, we have defined a meta-model with three diagrams. Two of them overlap with the third, but not between them. This structure can be described in categorical terms as a co-limit construction where in addition all the squares formed by two diagrams, their overlapping parts and the complete meta-model are pullbacks (thus, the squares commute $f_1 \circ o_{11} = f_2 \circ o_{12}$, $f_2 \circ o_{22} = f_3 \circ o_{23}$ and $f_1 \circ o_{31} = f_3 \circ o_{33}$). In addition, at the model level this structure also holds. In the actual implementation of this approach in AToM^3 (see section 5), the morphisms between each diagram meta-model and the global meta-model are inclusions, but in general they can be any function.

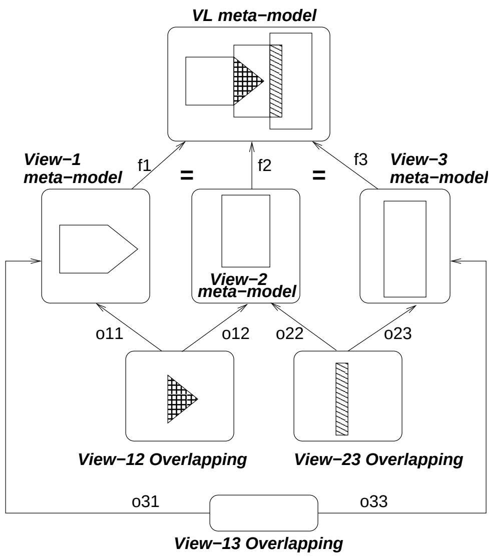


Figure 1. Multi-View Meta-Model as co-limit.

At the model level, in order to guarantee syntactic consistency, we build a unique model (called *repository*) by gluing all the view models of the user. This gluing operation is performed by automatically generated triple graph grammar rules (derived from the meta-model information). Updating the repository (as a result of a view model modification) may leave some other view models in an inconsistent state. At this point, other automatically generated triple rules update the necessary view models. In this way, we have grammars that propagate changes in the two directions: from the view models to the repository and the other way around. This can be seen in Figure 2, where the mechanism is similar to the model-view-controller (MVC) framework. In the figure, each model has in its upper-right corner the meta-model it is compliant with. Step 1 is usually performed by the user, steps 2, 3 and 4 are automatically performed by the tool.

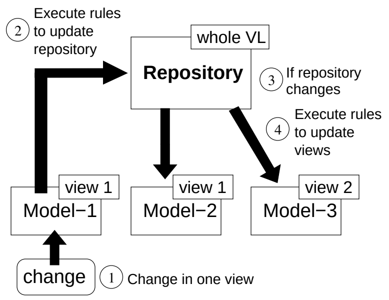


Figure 2. Change Propagation.

For static semantics consistency, the VL designer may provide additional triple rules. In this way, both syntax and static semantics can be checked in a uniform way. For dynamic semantics consistency, the VL designer must provide the necessary mechanisms. Usually, these will perform checkings at the repository level, probably performing a model transformation into another domain for analysis, but other specific checkings can be provided as well. Note how, transformation into another domain can also be expressed as graph transformation [6].

4. The MiCo Language

We have developed a component-based VL for modelling and simulation called MiCo (Minimal Components). The language was presented in [5], but we have extended it with hierarchy and the possibility to assign specifications to ports. Moreover, we have separated the diagrams into different views, which are now supported by AToM^3, with different kinds of consistency checkings. Our aim was to make the language as small as possible but still usable, and make their semantics precise. Here we give an overview of the language and focus on the specification of port behaviour by means of protocols, in which timing constraints can be included. As a running example, we present a model of a system for wireless video broadcast.

There are two kinds of diagrams in MiCo: structural and behavioural. In the first kind, we find *Specification Diagrams*, in which components are defined. These can be either simple components or be composed of other components. In this diagram one declares the ports, their types (the events they can send and receive) and the inner structure of the composite diagrams. In the *Connections Diagram*, users can specify constraints (regarding multiplicity) in the way the components can be connected through ports. This is an optional diagram, as one can just use the information of the port types. In the *Events Diagram* it is possible to define the event types and their attributes, together with the event hierarchy. Abstract and concrete events can be defined in this diagram. In the *Run Time Diagram*, it is possible to specify a run time configuration, made of component instances (that we call *processes*), which execute the behaviour defined in its component type. In the *Behaviour Diagram* one can describe the behaviour of each simple component type. The behaviour of a composite component is described in terms of its inner components. By now, simple component behaviour can be specified using *events graphs* [5], a visual discrete-event modelling notation. Finally, in the *Protocol Diagram*, it is possible to specify the order in which events should be received (in case of an input port) or sent (in case of an output port), together with timing constraints. This kind of diagrams is specified by a notation that we call *timed event machines* (TEMs), and is described in section 4.1.

As an example, we describe a system for video broadcast. The run-time configuration is shown in Figure 3, depicting a system made of a video player, connected to a wireless transmitter (able to transmit video and audio) with a projector, speakers and a mobile device. The projector and the speakers receive video and audio data respectively, while the mobile device receives both. In addition, the latter component may send control data to the video player, which is transmitted via the wireless transmitter.

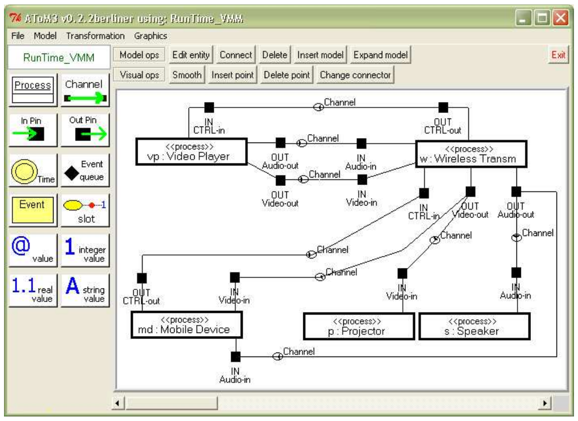


Figure 3. Run-time Diagram for the Example

Figure 4 shows an event diagram with the types of the events the system handles. They are either data or control (the subclassification is made using the usual UML notation of hollow arrows). A specification diagram is shown in the top-most window in Figure 8. Here, the *Video Player* component is being defined, together with its ports and static type. Finally, Figure 5 shows the behavioural type of the *CTRL-in* port of the *Video Player* component, in the form of a TEM. This diagram will be explained in next section.

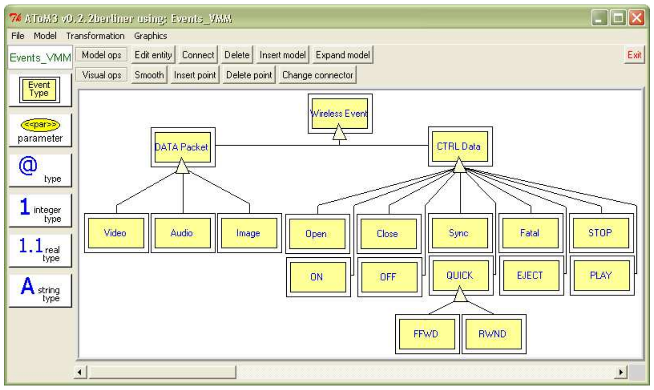

AToM3 v0.2.2berliner using: Events_VMM
File Model Transformation Graphics
Events_VMM
Model ops Edit entity Connect Delete Insert model Expand model Exit
Visual ops Smooth Insert point Delete point Change connector
Event Type
<<par>> parameter
@ type
1 integer type
1.1 real type
A string type
Wireless Event
DATA Packet
CTRL Data
Video
Audio
Image
Open
Close
Sync
Fatal
STOP
ON
OFF
QUICK
EJECT
PLAY
FFWD
RWND

Figure 4. Events Diagram for the Example

4.1. Specifying Timed Port Protocols: Timed Event Machines

We specify port protocols by means of a notation that we call *timed event machines* (TEMs). We use this notation to specify timed event languages, borrowing concepts from higraphs [12] (graphs with hierarchy). We find more natural to let events be the main concepts in the diagram. In this way, events are the nodes of the graph. TEMs allow specifying protocols with timing constraints, which serve as a behavioural type for ports. Our purpose is to be able to check whether connecting two ports is possible. As a TEM defines a regular language (plus timing information), when two ports are connected, we can check if the first one is a sublanguage of the other. In addition, the timing constraints should hold. Being able to specify such functional and timing constraints facilitates correct component reuse and replaceability.

Figure 5 shows a protocol description for the $CTRL-In$ port of the *Video Player* component. The diagram is made of nodes, which are the events that the port may receive. A transition from an event to another one expresses the order in which events occur. The same event may appear several times in the diagram. Transitions are labelled with two real numbers, which define a closed interval (open to the right if the upper bound is ∞). In a time belonging to this interval the target event should occur. In the more general case, the interval can be $[0, \infty)$. In case of an event having two or more outgoing transitions, this means that one of the target events should occur in the given interval.

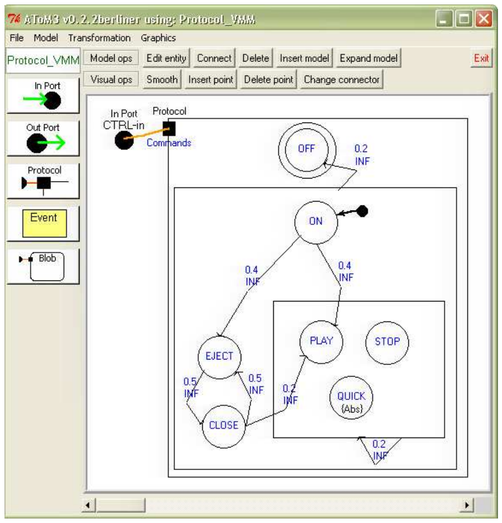
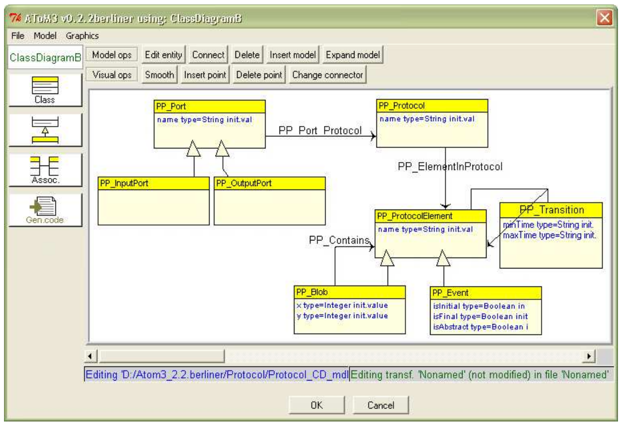

AToM3 v0.2.2berliner using: Protocol_VMM
File Model Transformation Graphics
Protocol_VMM
Model ops Edit entity Connect Delete Insert model Expand model Exit
Visual ops Smooth Insert point Delete point Change connector
In Port
Out Port
Protocol
Event
Blob
In Port CTRL-in
Protocol
Commands
OFF
0.2 INF
ON
0.4 INF
0.4 INF
EJECT
PLAY
STOP
0.5 INF
0.5 INF
0.2 INF
CLOSE
QUICK (Abs)
0.2 INF

Figure 5. The Protocol Description for the CTRL-In Port of the Video Component.

It is necessary to identify at least one starting event, which is marked with a small arrow. In the figure, it is the ON event. In addition, one or more final events are needed, and are marked with a double circle (OFF event in the figure). A TEM may contain abstract events as we have an event diagram where event hierarchy is defined. The semantics of a TEM with abstract events is equivalent to a *flattened* one, substituting the abstract events by all the concrete events in their hierarchy tree, and connecting all the input and output transitions of the abstract event to each concrete event. In the example, the $QUICK$ event has two children: $FFWD$ and $RWND$, thus we can find an equivalent graph by replacing $QUICK$ by these two events.

Similar to higraphs, events can be aggregated inside *blobs*. These are represented as rectangles that may receive or emit transitions. Blobs can be nested. If a blob receives a transition, then it is equivalent to all events inside the blob receiving such transition. If a blob is the origin of a transition, then it is equivalent to start the transition in each event inside the blob. In the example, the inner blob (and its self-loop transition) can be replaced by 9 arrows (with the same interval as label), starting and ending from each one of the three events inside (thus, forming a complete clique). The outer blob can be substituted by 6 transitions, starting from each one of its inner events, and ending in the OFF event. The meta-model for this notation, which is the protocol view of MiCo, is shown in Figure 6.

Each timed state machine defines a *timed regular expression* [2]. This is like an untimed regular expression but a time interval can be present between two consecutive symbols. Their semantics are given in terms of time-event sequences over a certain alphabet of actions Σ. Such



Editing 'D:/Atom3_2.2.berliner/Protocol/Protocol_CD_mdl Editing transf. 'Nonamed' (not modified) in file 'Nonamed'

Figure 6. Meta-Model for Timed Event Machines (Protocol View).

sequences have the form $t_0 a_1 t_1 a_2 ... a_n t_n$, where $t_i \in \mathbb{R}^+$ represents time lapses between the $a_i \in \Sigma$. A regular expression like $a_{[l,u]}$ denotes $\{r\ a | r \in [l,u]\}$.

The diagram in Figure 5 defines the following timed regular expression:

$$\begin{aligned} tre = \; & on \cdot eject_{[0.4,\infty)} \cdot (close_{[0.5,\infty)} \cdot eject_{[0.5,\infty)})^* \cdot \\ & (\lambda + close_{[0.5,\infty)}) \cdot off_{[0.2,\infty)} + \\ & on \cdot (eject_{[0.4,\infty)} \cdot (close_{[0.5,\infty)} \cdot eject_{[0.5,\infty)})^* \cdot \\ & close_{[0.5,\infty)} \cdot play_{[0.2,\infty)} + play_{[0.4,\infty)}) \cdot \\ & (play_{[0.2,\infty)} + stop_{[0.2,\infty)} + ffwd_{[0.2,\infty)} + \\ & rwnd_{[0.2,\infty)})^* \cdot off_{[0.2,\infty)} + on \cdot off_{[0.2,\infty)} \end{aligned}$$

The regular expression can be obtained with standard techniques in automata theory (see for example [14]). Basically we have to describe all the possible paths from all the initial events to all the final events, and then the timing constraints can be added. This latter step is much easier than for a timed automaton (see for example [2]), because in our case, the arrival of an event resets the clock. In this way, the next event does not depend on when the last event occurred, but only of the interval depicted in the transition. Thus, we do not need the concept of extended timed regular expression, as defined in [2]. Note how, TEMs are indeed a special case of *Time Petri nets* [15] (a so called state machine [17]). However, abstract events and blobs make TEMs more compact than the corresponding Petri net.

4.2. Using Timed Event Machines

TEMs define a behavioural type for the protocols. We use them mainly for two purposes. The first one is protocol conformance testing when connecting two ports. Several connections may appear here: input-to-output (*binding*), input-to-input of an inner component (*subsuming*), and output-to-output of an outer component (*delegation*). In all cases, what we need to check is that the source protocol defines a sublanguage of the target protocol. This can be done using standard techniques of automata theory, but timing constraints have to be taken into account. Thus, we have to check that the target TEM can simulate the source TEM, and that for each transition step of the source TEM, its interval is a subinterval of the target protocol transition. In this paper we restrict ourselves to checking systems in which an input port does not receive more than one connection.

As an example of protocol conformance, Figure 7 shows two protocols belonging to two ports that are to be connected. The one to the left shows the specification for the output port $Video - out$ of the $VideoPlayer$ component. The one to the right shows the specification for the $Video - in$ input port of the $WirelessTransmitter$ component. The first protocol defines the following expression:

$$\begin{aligned} t_1 = \; & open \cdot (sync_{co} \cdot (\lambda + sync_{cs} \cdot (\lambda + sync_{cs}))) \cdot \\ & [video_{sf} \cdot (video_{cf})^* \cdot \\ & (sync_{cf} \cdot (\lambda + sync_{cs} \cdot (\lambda + sync_{cs})))]^* \cdot \\ & (video_{sf} \cdot (video_{cf})^* \cdot (close_{cf} + \\ & sync_{cf} \cdot sync_{cs} \cdot sync_{cs} \cdot fatal_{cs})) + \\ & open \cdot sync_{co} \cdot sync_{cs} \cdot sync_{cs} \cdot fatal_{cs} \end{aligned}$$

where $co = [0.5, 2]$, $cs = [0.09, 1.12]$, $cf = [0.06, 0.08]$ and $sf = [0.06, 0.09]$.

The second protocol generates the following expression:

$$\begin{aligned} t_2 = \; & (open + sync + close + fatal + ON + OFF + \\ & STOP + FFWD + RWND + EJECT + \\ & PLAY) \cdot (open_i + sync_i + \\ & close_i + fatal_i + ON_i + OFF_i + STOP_i + \\ & FFWD_i + RWND_i + EJECT_i + PLAY_i + \\ & video_i + audio_i + image_i)^* + \lambda \end{aligned}$$

where $i = [0.04, \infty)$. Recall from Figure 4 that events $open$, $sync$, $close$ and $fatal$ inherit from $CTRLdata$, while $video$, inherits from $DATAPacket$. As $L(\lfloor t_1 \rfloor) \subseteq L(\lfloor t_2 \rfloor)$[1] and $co, cs, cf, sf \subseteq i$, the language of the first event machine is a sublanguage of the second. Thus, the first component can be connected to the second.

The second use of TEMs is for component replaceability testing. In our framework, a component can be substituted by another one in a certain environment if:

- The new component has at least the same number of input and output ports.
- Each input port of the replaced component accepts a subset of the events that the corresponding input port

1 We use the operator $\lfloor \cdot \rfloor$ on timed regular expressions to refer to the underlying untimed regular expression.

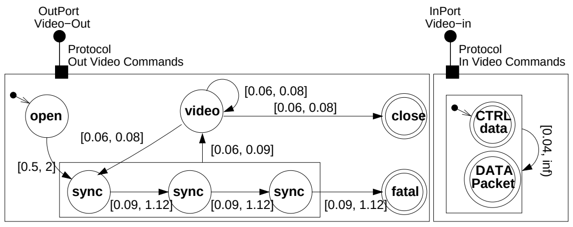


Figure 7. Protocols for the Example

of the new component, and the other way around for output ports.

- Each input port of the replaced component (inp_i^{OLD}) defines a sublanguage of the corresponding input port of the new (inp_i^{NEW}) and the other way around for output ports. That is $L(inp_i^{OLD}) \subseteq L(inp_i^{NEW}) \forall i$ and $L(outp_j^{NEW}) \subseteq L(outp_j^{OLD}) \forall j$.

5. Tool Support

AToM^3 [4] is a meta-modelling tool that allows specifying the syntax of a VL by means of a meta-model, and then a tool for editing diagrams for the VL is automatically generated. Model manipulation (change propagation, simulation, optimisation, translation into another formalisms and code generation) can be expressed by means of graph rewriting or Python code.

The tool supports the generation of environments for VLs with (possibly overlapping) multiple views. The different view meta-models are restricted to be subsets of the classes and relationships of the whole VL meta-model. Moreover, each one can use only a part of the attributes and constraints specified in the complete meta-model. Figure 8 shows the generated environment for the MiCo language. Six views, corresponding to the 6 different diagrams proposed by the language, have been defined. As it is shown on the left of the background window, AToM^3 adds one button for each view. The top-most window shows a specification diagram being edited.

Syntactic consistency between view models, as well as change propagation, is achieved by means of triple graph grammars. They are automatically derived by the tool from the view meta-models information. Thus, a triple graph grammar relating each view and the *repository* is generated. Its purpose is to build the unique *repository* model made of the gluing of all the view models. In this way, its triple rules handle the creation, deletion and edition of elements in the view models, by performing the same actions in the *repository* model. These rules are executed when the user has edited one diagram (step 2 in Figure 2). Figure 9 shows one of the syntactic consistency rules generated for the class *OutputPort*. This rule relates a view (upper side

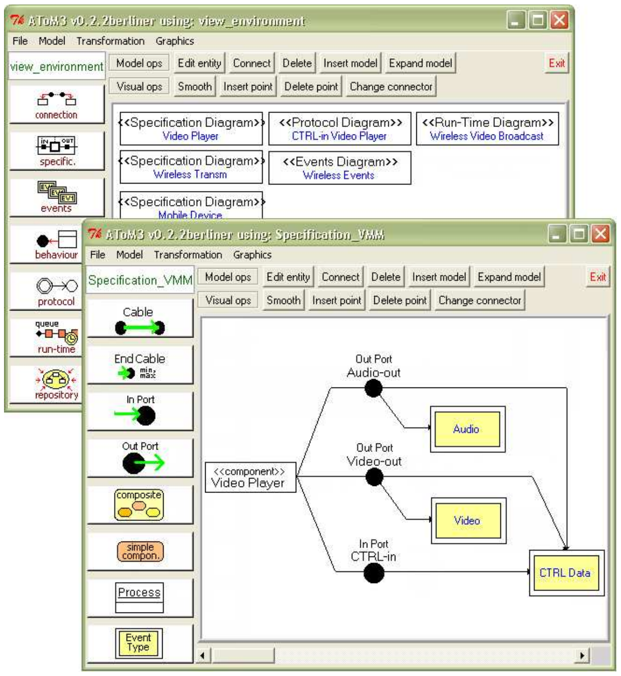

AToM3 v0.2.2berliner using: view_environment
File Model Transformation Graphics
view_environment
Model ops
Edit entity
Connect
Delete
Insert model
Expand model
Exit
Visual ops
Smooth
Insert point
Delete point
Change connector
connection
specific.
events
behaviour
protocol
run-time
repository
<<Specification Diagram>> Video Player
<<Protocol Diagram>> CTRL-in Video Player
<<Run-Time Diagram>> Wireless Video Broadcast
<<Specification Diagram>> Wireless Transm
<<Events Diagram>> Wireless Events
<<Specification Diagram>> Mobile Device
AToM3 v0.2.2berliner using: Specification_VMM
Specification_VMM
Cable
End Cable
In Port
Out Port
composite
simple compon.
Process
Event Type
Out Port Audio-out
Out Port Video-out
In Port CTRL-in
<<component>> Video Player
Audio
Video
CTRL Data

Figure 8. Generated Environment for MiCo.

of the rule) and the *repository* (lower side). If an *OutputPort* object has been created in the view, but it is not in the *repository*, the rule adds it to the *repository* and creates a correspondence relation between both. Afterwards, if the same object is added to a different view, another rule relates the new instance of the object with the existing one in the *repository*. Note that objects in the *repository* have a counter with the total number of its instances along the view models. When an object is newly created in the *repository*, its counter is set to 1; each time the object is referenced, the counter is incremented by 1. Another two rules manage the deletion of objects from the views. The first one decrements the counter of instances of an object if it is removed from a view. The second one deletes an object from the *repository* if its counter reaches zero, which means that there is no instance of the object left in any view. Finally, a last rule copies the values of attributes from the objects in the views to the related object in the *repository* if they are different. See [11] for a detailed explanation of the generated rules for a different language.

Change propagation is performed by another (automatically generated) triple graph grammar relating the *repository* and each view. It is tried when the *repository* is modified by the execution of a previous grammar (steps 3 and 4 in Figure 2). It contains a rule for each element in the view meta-model, which copies the value of its attributes from the *repository* to the views if they are different. Note that, for each view, only the rules for the overlapping elements with the initially modified view should be tried. This infor-

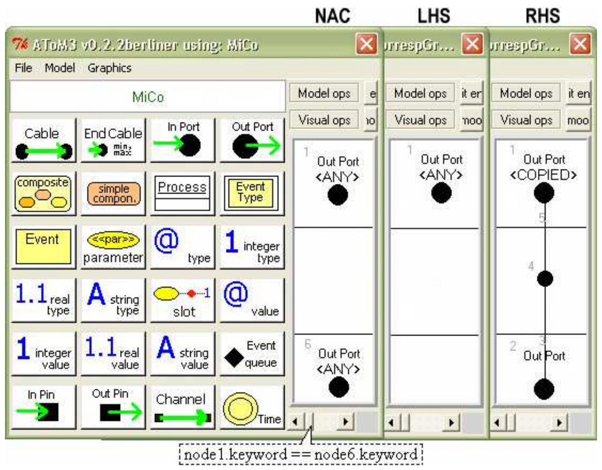


Figure 9. One of the Automatically Generated Syntactic Consistency Rules for MiCo.

mation is given by the pullback objects (see Figure 1).

Static semantics consistency can also be specified by means of triple graph grammars. Whereas the syntactic consistency rules are automatically generated by the tool, the semantic ones have to be defined by the VL designer. Figure 10 shows one of the defined rules between the *Connections Diagrams* view (upper side) and the *repository* (lower side). It checks the static type of two ports before connecting them. An output port can be connected to an input port only if the events generated by the first are also accepted by the second. In other case, an error message is shown and the connection in the view is deleted.

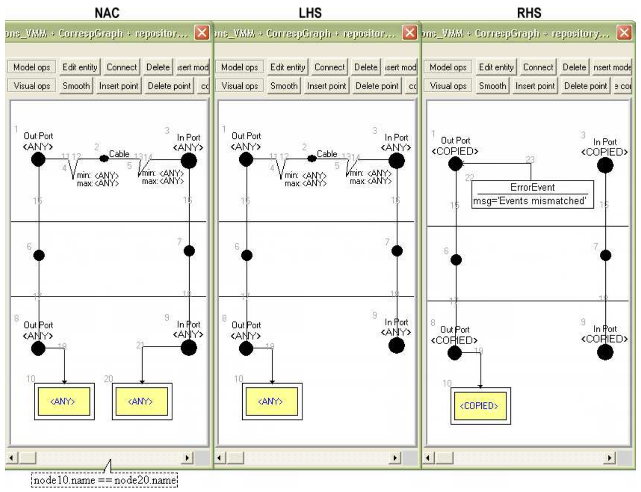


Figure 10. A Static Semantics Consistency Rule for MiCo.

Static semantics consistency can also be implemented as regular graph grammars applicable over the *repository* model. In this case, errors are shown to the user, who should correct them by hand in the corresponding view in order to obtain a consistent specification. This would be an iterative process until no consistency error is left. This second approach results in less restrictive environments, but more work is delegated to the user.

Finally, dynamic consistency can be provided in several ways. In the general case, some regular graph grammar can be used to translate the *repository* into some semantic domain for further analysis. As we explained in section 4.2, in our MiCo language we want to know if the protocols defined in output and input ports allow their connection. With this purpose, first we use a graph grammar to flatten both TEMs. Then, a Python program checks that the language defined by the TEM in the output port is included in the language defined by the TEM in the input port.

6. Related Work

A common approach for consistency checking and (possibly) change propagation in languages with multiple views is defining a common repository where the different view models are somehow related. The way in which these relations are built can be specified by using either a textual or a visual notation. Examples of the former include DOME [7], MetaEdit+ [16] or Pounamu [22]. Examples of the latter include JComposer [9] as well as AToM^3 [4]. In JComposer, *change propagation and response graphs* (CPRGs) allow to visually attach events to components and relations; the reaction to the events can implement semantic constraints on the attached components. In AToM^3 both syntactic and static semantics consistency can be specified by means of triple graph grammars. The formers are automatically generated by the tool. We think the advantages of using TGGs are its graphical nature, but in addition its theoretical background makes transformations subject to analysis.

There are other approaches to the specification of protocols for component frameworks. We distinguish between textual and graphical notations. In the first category, SOFA components [18] allow specifying protocols in a textual notation similar to regular expressions. Protocols are assigned to components (and not to ports) and do not contain timing information. Other textual approaches, such as Wright [1], are based on CSP [13]. All these approaches do not consider timing information.

In the visual approaches, we can find UML 2.0 protocol state machines. These are a kind of high-level Statecharts that show the usage of an interface. Method calls (our events) are shown in the arrows of the Statecharts and may have pre- and post- conditions. However, UML does not provide a formal basis for protocol conformance testing. Other visual notation that allows setting timing constraints is Time Petri nets [15]. However, our possibility to aggregate events inside blobs makes our TEMs more compact. Moreover, our aim with MiCo and TEMs is to design a visual, formal, simple, yet usable notation. Using such a simple notation as TEMs has the advantage that it maximally constraints the user for the task to be done. If we had used Petri nets, users would have had more freedom, but also higher risks of building wrong models. The simplicity of MiCo makes it easy to design a formal semantics for it. In this paper, we only showed the formal approach to consistency, but the operational semantics of MiCo are based on graph transformation (the semantics of a simpler version of the language was shown in [5]). Common visual approaches such as UML and SysML offer much richer design elements, but the complexity of their definition makes formal analysis much harder.

7. Conclusions and Future Work

In this paper we have shown our approach for static and dynamic consistency checking in VL with multiple views. The approach is based on building a repository model, by gluing all the view models. Graph transformation rules perform static consistency checking by updating the repository and the view models. Syntactic consistency can be automatically obtained from the meta-model. Static semantic consistency can be achieved by specifying additional triple rules. Once the repository is built, we can perform dynamic consistency checking. This may be done by transforming the models to be checked into another domain, and involves domain specific analysis techniques and/or simulation.

We have shown our VL for component-based system modelling called MiCo. The language defines several structural and behavioural diagrams, including the possibility to specify functional and timing requirements for the component ports. We have introduced a visual notation called *timed event machines* for this purpose. This allows conformance checking, and makes reutilization and replaceability of components easier. We have shown the implementation of these concepts in the meta-modelling tool AToM^3. The tool is able to generate customized modelling environments with automatic support for syntactic consistency by means of triple graph grammars.

In the future, we will work in supporting conformance checking in the case of several output ports connected to an input port. In an asynchronous case, this would involve calculating the parallel composition of the timed event machines of the output ports.

Acknowledgements: This work has been supported by the Spanish Ministry of Science and Technology with project TIC2002-01948.

References

[1] Allen, R. J., Garlan, D. 1997. *A Formal Basis for Architectural Connection*. ACM Transactions Software Engineering and Methodology. 6(3), pp: 213 - 249.

[2] E. Asarin, P. Caspi, O. Maler. 2002. *Timed Regular Expressions* Journal of the ACM 49, No.2, pp.: 172-206.

[3] Damm, W. 2005. *Controlling Speculative Design Processes Using Rich Component Models*. Invited talk at ACSD (Application of Concurrency to System Design), pp.: 118-119.

[4] de Lara, J., Vangheluwe, H. 2002. *AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling.* Proc. ETAPS/FASE'02, LNCS 2306, pp.: 174 - 188. Springer.

[5] de Lara, J. 2004. *Distributed Event Graphs: Formalizing Component-based Modelling and Simulation*. Proc VLFM, ENTCS (Elsevier), 127(4), pp.: 145-162.

[6] de Lara, J., Taentzer, G. 2004. *Automated Model Transformation and its Validation with AToM^3 and AGG*. LNAI 2980. Springer, pp.: 182-198.

[7] DOME Home page at *http://www.htc.honeywell.com/dome/download.htm*

[8] Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. 1999. *Handbook of Graph Grammars and Computing by Graph Transformation.* (1). World Scientific.

[9] Grundy, J. C., Mugridge, W.B., Hosking, J.G. 1998. *Visual Specifi cation of MultiView Visual Environments*. IEEE Symposium on Visual Languages, pp.: 236-243.

[10] Guerra, E., de Lara, J. 2004. *Event-Driven Grammars: Towards the Integration of Meta-Modelling and Graph Transformation*. LNCS 3256, pp.: 54-69. Springer.

[11] Guerra, E., Díaz, P., de Lara, J. *Supporting the Automatic Generation of Advanced Modelling Environments with Graph Transformation Techniques*. To appear in Proc. JISBD'05, Granada, Spain.

[12] Harel D. 1988. *On Visual Formalisms*. Communications of the ACM. Vol 31, No. 5. Pp.: 514-530.

[13] Hoare, C. A. R. 1985. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, N.J.

[14] Hopcroft, J., Motwani, R., Ullman, J. 2001. *Introduction to Automata Theory, Languages, and Computation. 2nd ed.* Addison-Wesley.

[15] Merlin, P. M., Farber, D. J. 1976. *Recoverability of communication protocols – implications of a theoretical study*. IEEE Transactions on Communications, 24(9):1036-1043.

[16] MetaEdit+ Home page at: *http://www.metacase.com/*

[17] Murata, T. 1989. *Petri Nets: Properties, Analysis and Applications*. Proceedings of the IEE, Vol. 77(4). Pp.: 541-580.

[18] Plasil, F., Visnovsky, S. 2002. *Behavior Protocols for Software Components*. IEEE Transactions on Software Engineering, 28(11) pp.: 1056-1076.

[19] Schürr, A. 1994. *Specifi cation of Graph Translators with Triple Graph Grammars.* LNCS 903, pp.: 151-163. Springer.

[20] SysML home page at *http://www.sysml.org/*

[21] UML2.0 specification: *http://www.omg.org/*

[22] Zhu, N., Grundy, J.C. and Hosking, J.G., 2004. *Pounamu: a meta-tool for multi-view visual language environment construction* Proc. 2004 VL/HCC, IEEE CS Press, pp. 254-256.