SPECIFIC TARGETED RESEARCH PROJECT
INFORMATION SOCIETY TECHNOLOGIES

FP6-IST-2005-033606

VIsualize all moDel drivEn programming
VIDE

WP 11	Deliverable number D11.2 Model Simulation Industrial use cases

1.1 Detailed description of the Use Case

Models are often perceived and used as a vehicle for communicating general ideas of a software system inside a team and as an aid when thinking about the problem. In this sense, the abstraction associated with the notion of model means treating a model informally and skipping some essential details (like e.g. exact data model and behaviour details). The situation is different for executable models, where the abstraction implies only suspending the decision of the final platform realisation, though the semantics is precisely defined and behaviour can be completely specified. Hence, in this case, practically all the analysis and validation normally available only after software is fully implemented, can be now performed already at the stage of a completing a platform-independent model. Having defined a meta-model for our modelling language and a precise semantics behind its constructs makes it possible to assure model consistency (in a similar or broader scope than e.g. in type checking during traditional programming language code compilation) and to enforce quality standards for its structure and behaviour. However, some issues can be disclosed only through a more dynamic checking of the model, e.g. by its execution. Such a feature can be useful for several related purposes:

- Early (PIM level) debugging – this assumes employing a model execution engine capable of stepwise execution of code and having its constructs traceable to respective platform-independent model elements. This way the model could be run and debugged in a way analogous to typical programming language environments.
- Early (PIM level) testing – executing a model directly in terms of its particular functionality elements: either by ad-hoc statement calls from the model editor (as well as queries, to check their effect on sample data) or from external components (e.g.

workflow engine, other applications, GUI (Graphical User Interface) components etc.) – thorough published web service operations.

- Early (PIM level) validation – this case is similar to the above, but has a different focus. It is intended to run a model as a prototype of the prospective software in order to confirm it matches the expectations of non-IT stakeholders (e.g. the future users). In this case it is especially important to provide means for prototyping GUI and making it execute and interact with the executable model.

In other words, the use case can be described as aimed at achieving an analogous insight into a model as in case of target platform implementation (to allow running and debugging it), but making it much more direct and straightforward by providing that functionality directly from a UML tool and interpreting the results of code execution in terms of UML elements.

1.2 Current development approach

Current use of modelling languages (at least in the business software field) seems to be dominated by informal modelling, where a model serves for outlining an overall architecture or requirements. Apart from that, more precise kinds of models are used, but are usually limited to structure modelling. In that case they can serve as an input for class skeletons integration in the target platform programming language, or for generating SQL schema definitions for database. Behavioural models are used less commonly and, if so, they play a more conceptual role, being not involved in code generation.

This means, that in order to validate system behaviour or test the model functionality, a whole path towards the platform specific code needs to be traversed. Namely, the following steps need to be performed (see Figure 1):

1. Incomplete code generation – in order to get target platform language's source code containing class skeletons.
2. Coding the system behaviour in the target platform language, using respective IDE.
3. Validating the behaviour using the target IDE.
4. Coding sample data input (platform-specific) to populate data sources and run test cases.
5. Preparing platform-specific invocations of the functionality to be tested.
6. Optionally – preparing Web service interfaces to the functionality being tested.
7. Running the tests and interpreting the responses (platform specific).
8. Identifying necessary corrections and applying them to the code, or – if they involve structural changes – updating the model and repeating the cycle.

© Copyright by VIDE Consortium

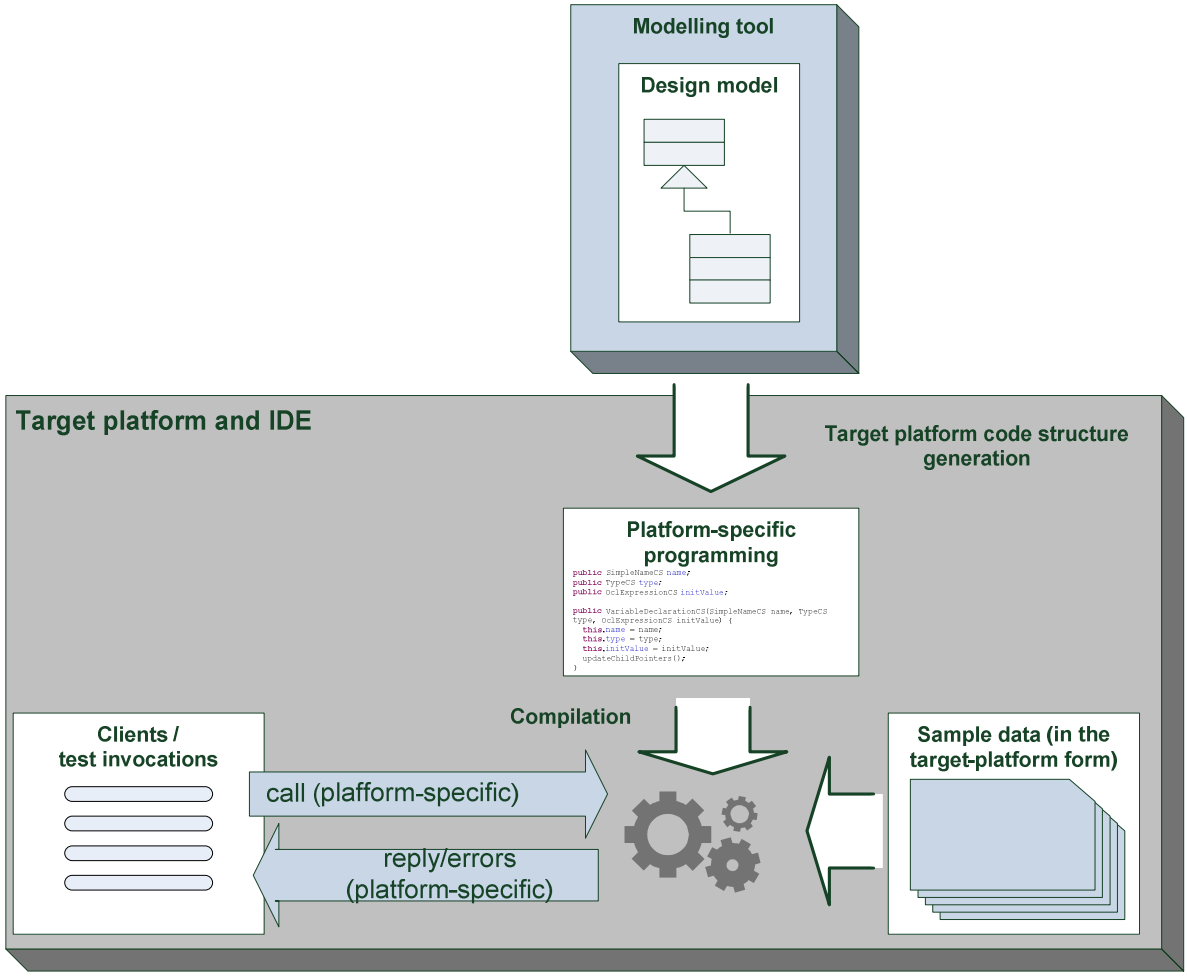

Figure 1: Code dynamic validation with absence of model execution capability

As can be seen, most of the development effort is located on the platform-specific side. Most of the tests need to be performed at that phase, which means that their results are not reusable as multiple target platforms are considered. Moreover, if the transformation between a platform-independent class model and the target language structures generated from it is not trivial (consider for example differences in handling inheritance, objects removal or parameter passing), applying necessary changes to the model based on the code execution brings additional complication.

1.3 VIDE contributions

Shifting the means of precise and complete behaviour specification to the PIM level, allows performing the validation and testing of the model once, even in case of multiple target platforms (assuming the presence of respective model compilers, reliable in terms of preserving the semantics when transforming to a target platform). Moreover, even if one target platform is considered, a more directly available testing environment is an advantage. Sample data, and test invocations can be encoded in a platform independent manner, and the execution results are expressed in terms of the PIM rather than its platform-specific counterparts. The overall process of testing the system behaviour is simpler, and offers many opportunities for making the steps underlying model execution even more transparent. Hence, the goal can be achieved by traversing a shorter path (as shown in Figure 2). This path consists of the following steps need to be performed:

1. Activating the Model Execution Engine component.

© Copyright by VIDE Consortium

2. Feeding a model into it.
3. Coding sample data creation in a platform-independent way, to populate data source in order to run test cases.
4. Running the tests and interpreting the responses (provided in terms of the PIM).
5. Identifying necessary corrections and applying them directly to the model.

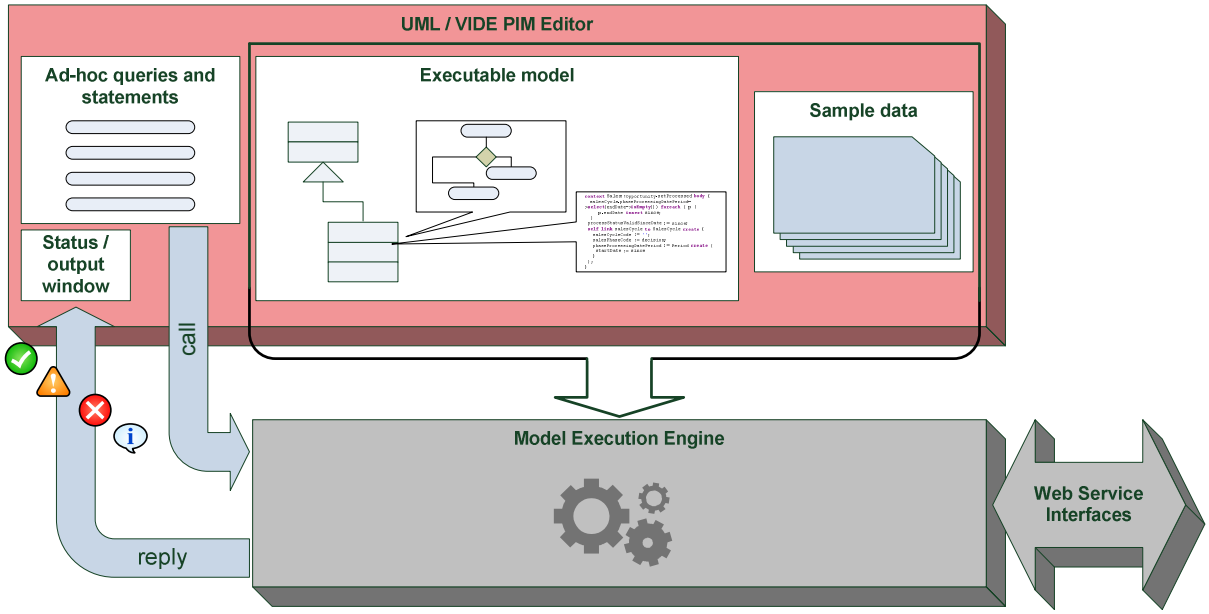

Figure 2: Model execution capability available in VIDE

With VIDE PIM support for Web service interfaces, selected functionality can be nominated to provide Web service operations that become available right after running the model. This allows using various external clients to invoke modelled behaviour. Particularly, those interfaces can be used for the presentation layer (GUI) prototyping, which may significantly increase the expressiveness of the software under development when presenting it to stakeholders.

1.4 Example based on VIDE toolset

The concept outlined above have been realised in the VIDE prototype in its basic form. The developer can connect to the Model Execution Engine and make model execute on it (according to a semantics compliant with the one assumed for respective UML 2.1 constructs). Web service interfaces – for publishing as well as for consuming Web service operations – are also supported by the model execution engine. Although the Model Execution Engine uses its internal language that is different from the UML, this language remains very similar and on the same level of abstraction. In consequence, the messages returned by the Model Execution Engine are quite informative to VIDE developers.

The diagram below presents a minimalistic sample of a model that can be tested using the Model Execution Engine in its current shape. It consists of two domain class ***Product*** and **Order**, as well as a «module»-stereotyped class ***ProductList*** that provides an environment, a persistent storage and behaviour to manipulate the instances of the former class. Note also the following constructs in the model:

- Stereotypes «PublishedService» and «PublishedOperation» in class ProductList, making the respective UML behaviour invokable as Web service.

© Copyright by VIDE Consortium

- Multi-valued **prod : Product** property – for a global storage of Product class instances.
- WRITE_SAMPLE_DATA() – a disposable method introduced for the time of testing. It contains statements creating sample data instances. A similar method could be created in order to ease batch invocation of different queries or statements – instead of having them executed directly from the ad-hoc statements invocation window of the VIDE editor.
- «Id» – a stereotype indicating that the Model Execution Engine has to take care of generating a unique value for this attribute in newly created attributes (code generated by model compilers is supposed to assure the same on their target platforms).
- Unidirectional association between Product and Order classes.

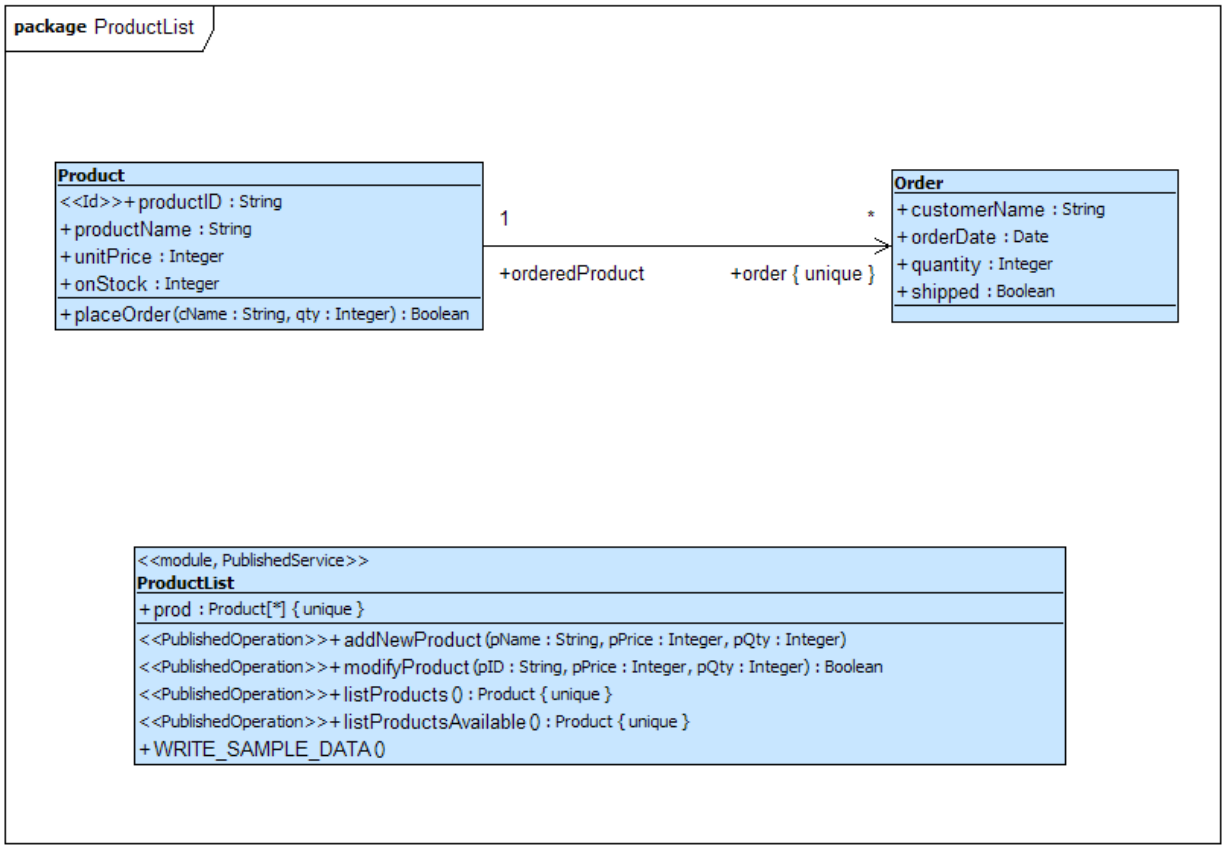

Figure 3: A minimalistic sample model illustrating the model execution capability

Related models: The folder \Products as a whole constitutes a VIDE project that can be imported to Eclipse (using Import -> Existing Projects into Workspace), setting \Products as the root directory of the project imported.
Any other VIDE project available in the library that has behaviour fully developed using Textual VIDE Language can be also tested against this use case to observe the behaviour of a more complex model.

In order to run the example, the following steps need to be performed:

1. Import the example project provided as a .zip archive.
2. Ensure the VIDE perspective is selected.
3. Open the model file (Products.uml file) in the Repository Browser.

© Copyright by VIDE Consortium

4. Start ODRA Model Execution Engine using the script creating a new database file and establish a connection from the editor ("Connect to an existing server" toolbar command) as described in the documentation.
5. Now, you can open the existing ProductList.odra file, or generate a new one from the model by using the Repository Browser's **Generate classes and contexts** command button.
6. Having the .odra file opened in the editor, press the **Execute** toolbar button to send the model into the Model Execution Engine. You shout observe the following message in the ODRA Results View:
```
Module created: ProductList.
Current module: ProductList.
Endpoint ProductListEndPoint installed.
Current module: .
```
7. Make sure the ProductList class is selected in the Repository Browser in order to set the context for statement execution against a model instance.
8. Create or open the statements.vide textual file. Type WRITE_SAMPLE_DATA() operation invocation. Press the "Execute" toolbar button to invoke the statement.

Now, the example is running and sample data are available in it. Next, tthe following steps can be performed:

- Invoking other statements or queries (by repeating steps 5-7 from the above list) and observing the engine responses in the "ODRA Results View" window.

- Creating and executing ad-hoc OQBE queries. A sample query "toBeDelivered" can be invoked using the following steps (see the screenshot in Figure 4).

  1. Open the OQBE diagram file (toBeDelivered.oqbe_diagram)
  2. Press the "Execute OQBE" toolbar command.
  3. Observe the results of query invocation shown in the "ODRA Results View" window (see the figure below).

- You may also invoke the sample model's published behaviour from external tools using the Web Service interfaces. When the engine is running and is initiated with sample data, respective service interface will be available at: http://localhost:8888/ProductsSample/ProductListService?WSDL (see Figure 5).

© Copyright by VIDE Consortium

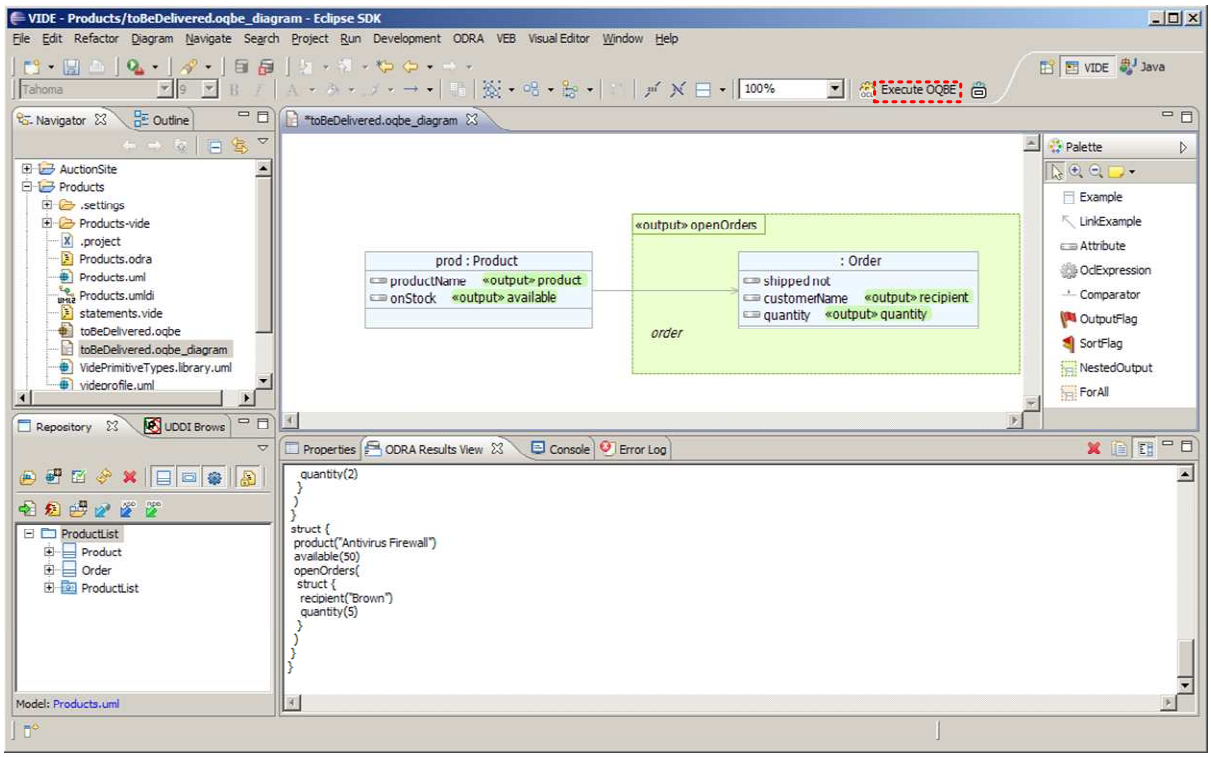

Figure 4: Using the Visual Expression Builder for creating and invoking ad-hoc queries

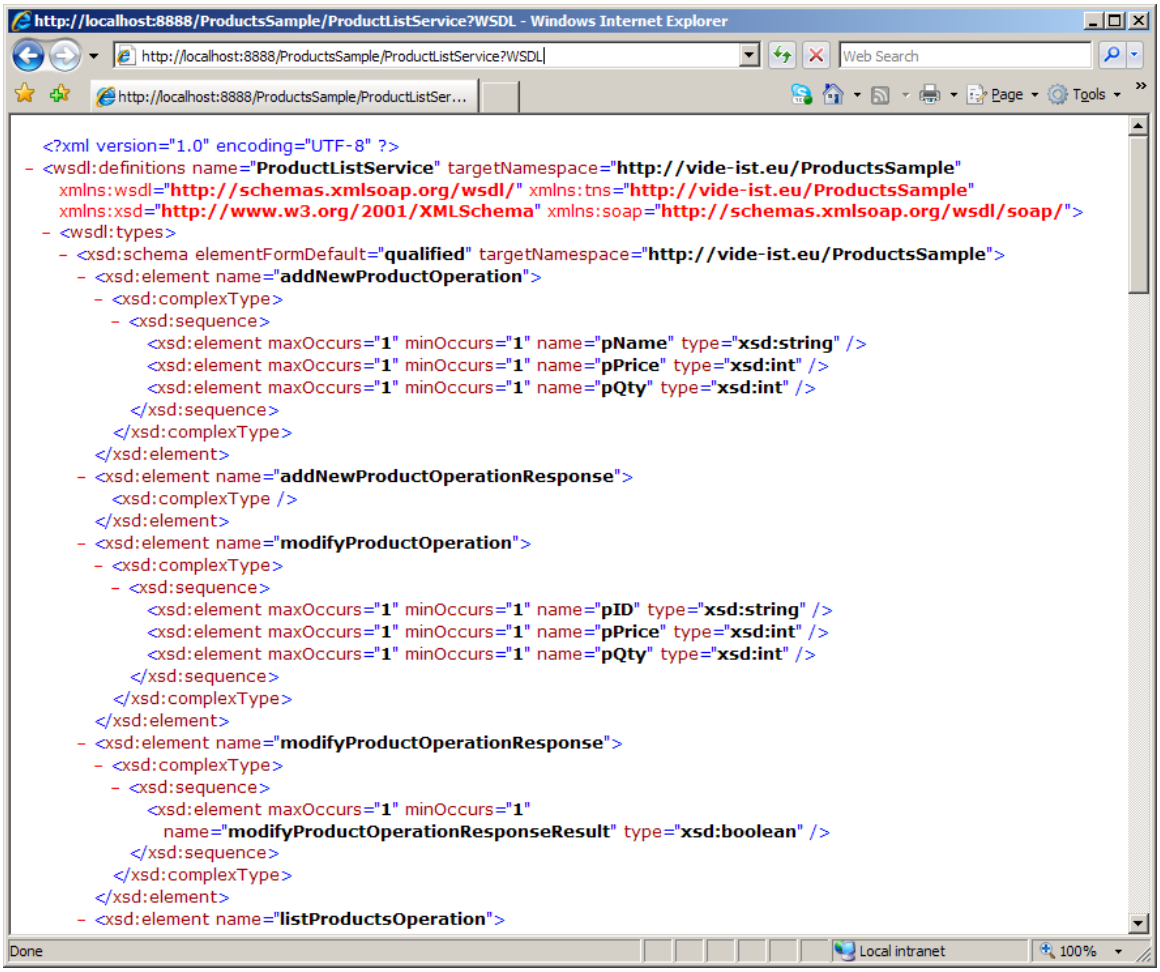

Figure 5: Definition of web service operations made available by running VIDE model

© Copyright by VIDE Consortium

Normally, based on the results observed during the model execution, you may wish to update the model. Please see the VIDE Cookbook document for a more detailed description of the editor functionality and the language constructs.

Compared to the current prototype, the following future improvements would be desirable to fully realize the concept outlined of UML model simulation:

- Making the steps underlying model execution more transparent (currently the Model Execution Engine code generation and passing its result to the engine are two separate commands that have to be invoked).
- Annotating the Model Execution Engine internal language syntax tree with references to UML model, so that all the runtime messages can be presented in terms of the model.
- Providing the Model Execution Engine with a means of stepwise execution of code, and provide respective controls for debugging to the VIDE editor.
- Extending the VIDE editor with a utility easing the creation of sample date to be fed into the model.

© Copyright by VIDE Consortium