Decoupling validation UIs using Publish-Subscribe binding of instance variables in Overture

Luis Diogo Couto[1], Kenneth Lausdahl[2], Nico Plat[3], Peter Gorm Larsen[2], and Ken Pierce[4]

[1] United Technologies Research Center, Ireland
[2] Aarhus University, Department of Engineering, Denmark
[3] West IT Solutions, The Netherlands
[4] Newcastle University, School of Computing Science, UK

Abstract. Being able to efficiently communicate the way formal models behave to stakeholders who are not experts in formal models is important for the potential commercial exploitation of such technologies. For the Overture/VDM open source initiative, this has so far been accomplished by interpreting executable models in combination with executable Java applications, typically demonstrating a conceptual Graphical User Interface (GUI). However, since that approach requires additional explicit calls to GUI functionality, this paper demonstrates a new publish/subscribe based approach enabling implicit monitoring of changes to variables during an interpretation of a formal VDM model that does not "pollute" the VDM model in the same way as existing approaches.

Keywords: Overture, VDM, user interface, Electron

1 Introduction

The Overture/VDM tool [8] supports three different dialects of VDM [4]. Overture is already capable of interpreting executable subsets of VDM [10] and it is also possible to combine such interpretation with legacy Java code [11]. However, the latter solution can be a little clumsy and "pollute" the VDM models by requiring explicit calls to operations that update, for example, a graphical user interface. The contribution of this paper is to enable an easy way to monitor instance variables as they are changed during an interpretation of a VDM model in an implicit fashion. The primary focus of our paper is on monitoring variables and displaying model information in a user interface. However, our work also supports calls from the User Interface (UI) to the model. This contribution is based on top of the web-based Electron [2] platform and it is connected to the Overture tool as a plug-in extension. Electron is a modern and popular framework for desktop application development. It was chosen because it enables use of web technologies, has a wide amount of existing libraries and modules, and has an active community supporting it.

The TEMPO[5] project investigated collaboration between different Traffic Management Systems (TMSs) by providing them with collaborative control architectures that

[5] See `http://tempoproject.eu/`.

engage with each other in automated negotiation processes [14]. TEMPO used the Overture tool to analyse both existing traffic management networks and potential collaborative designs, demonstrating the benefits of smart traffic systems. Traffic simulations produce a large amount of numerical data; it is imperative to present this in an understandable way to non-experts to make computer simulations useful. Overture is being extended with a 2D/3D visualisation library to show the negotiations between TMSs and the effect on traffic flow.

This paper focusses on extensions to the Overture tool that enable a fast, lightweight way to visualise how variables in a model change during interpretation. In the remainder of this paper, the extensions are explained in Section 2. This is followed by a short explanation of the practical application used for viewing the changes to the instance variables in Section 3. Afterwards Section 4 presents the case study in which this new technology has been used. Then Section 5 presents related work. Finally, we conclude the paper and suggest future work possibilities in Section 6.

2 The Overture Extension

2.1 Design Principles

The Overture interpreter already has features that enable it to interact with external components implemented in Java [11]. These features include the Java bridge which enables VDM models to invoke functionality implemented in Java and the `RemoteControl` interface which enables Java programs to directly control the Overture interpreter. These features have frequently been used to implement prototype user interfaces.

The extension we have developed takes into account the existing work connecting Overture to Java and seeks to address certain issues with it. As such, the extension was designed according to two key principles:

1. The extension must enable the use of modern and fast UI technologies; and
2. the UI code must not pollute the VDM model.

Principle 1 attempted to address an ongoing issue with slow UI prototyping. Pure Java interfaces are typically developed with the Swing toolkit, which is rather slow and laborious. When developing UI prototypes, it is important to be able to construct a working UI or we end up spending too much time on the UI. Swing works against this, particularly for any UI that is non-trivial. Furthermore, Swing interfaces look extremely dated and unappealing. While the aesthetics are not the primary concern of a UI prototype, it should still look presentable and adjust to the modern expectations of stakeholders, or we risk that they focus too much on the old look of the UI and thus cannot properly assess the model.

Principle 2 simply aims to improve on the existing status of the Java bridge. While there was little we could do to improve the status of Swing UIs, we have total control of the Java bridge. In the current implementation, in order to display any information in the UI, the model must explicitly call the UI to set the values. While there are alternatives, this is typically done because it is the most expedient way of displaying new data as

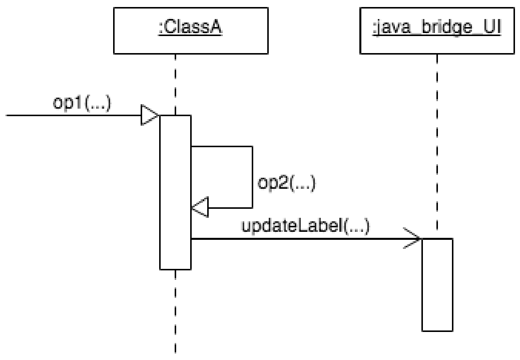


Fig. 1. Explicit call from model to UI.

soon as it is available in the model. This behaviour is shown in a sequence diagram in Figure 1.

While the explicit call approach is efficient and pragmatic, it does lead to models that are polluted with UI calls. These calls may affect the tractability of the model as well as its readability from a purely formal point of view. In addition, these explicit calls might make it impossible to execute the model without the UI present as the UI calls would fail – while this can be worked around using guards, that means polluting the model even further due to the UI.

Finally, it is worth mentioning that the existing approach typically leads to fuzzy divisions between UI, data and logic in the solution. Some UI code is in the model and the logic is often split between model and UI. Modern software development practices typically advocate for a clean separation between UI and logic, using patterns such as model-view-controller [7]. By adhering to the aforementioned design principles, the new extension will enable users to more cleanly separate their UI from their application logic.

2.2 Implementation

The new extension is based on the publish/subscribe design pattern [3]. The extension enables the UI to subscribe to statically known model variables. Whenever the value of a subscribed variable is changed, an event is broadcast with the new value. The UI can listen for this event and then react accordingly. Thus, all UI interaction is hidden from the modeller – leading to the behaviour shown in the in Figure 2, where the UI is notified of updates values in an asynchronous way without the need for explicit calls from the model.

The core of the extension is a new remote control class for the Overture Interpreter [10]. Remote control is a feature of the existing Overture interpreter Java bridge that enables a VDM model to be controlled remotely by a Java program that implements the `RemoteControl` interface [9, chapter 7]. Remote control classes are typically

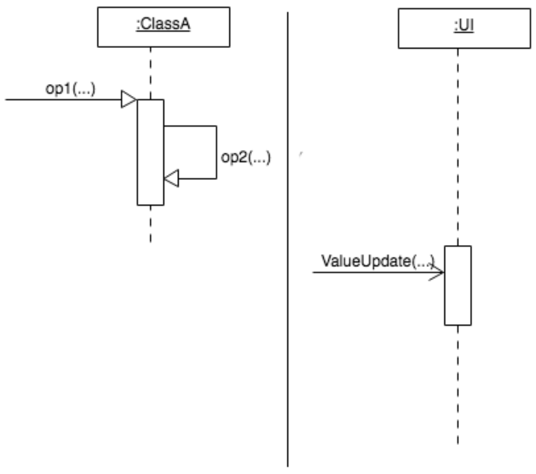


Fig. 2. No calls from model to UI.

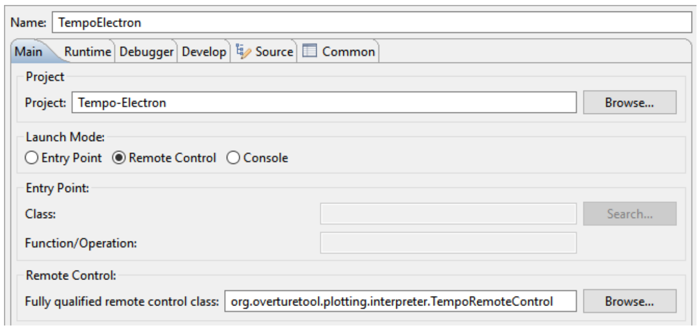
used as part of the implementation of Swing UIs in order to allow the UI to control the model. The remote control class can evaluate expressions in the context of the VDM model and invoke model operations in order to modify the model's internal state.

The remote controller of our extension also controls the interpreter in this manner but rather than directly controlling the interpreter according to a given logic, the remote control class of our extension re-exposes the control of the interpreter via a JSON based protocol over web-sockets [1]. In this way, any UI can be connected to any Overture model in a completely generic way, based on REST approaches. Most notably, this extension is backwards-compatible. Any existing VDM model can be controlled with it without a single modification being made to the model.

In addition to control over the interpreter, our protocol also defines the means by which a UI can subscribe to variables, and how the remote controller broadcasts updates to subscribed clients. The remote controller is provided as a single jar that contains everything necessary to execute it via the Overture tool, as shown in Figure 3. Because it is launched from within Overture, all the IDE features such as breakpoints and model coverage are available for models executed with a UI.

The remote control is relatively simple. It contains a Jetty-based web server [6] to expose the protocol and uses the Overture Java bridge's `RemoteControl` Java interface to control the interpreter. The main components are summarised in the class diagram shown in Figure 4.

The most notable class besides `TempoRemoteControl` is `VarListener`, that is responsible for monitoring variables in the VDM model and broadcasting messages with the updated values. This class uses the same mechanism and infrastructure already present in the interpreter to monitor invariant violations during model executions since this mechanism already hooks into every point in a model execution where a variable



Name: TempoElectron

Main | Runtime | Debugger | Develop | Source | Common

Project

Project: Tempo-Electron Browse...

Launch Mode:

Entry Point [✓ Remote Control] Console

Entry Point:

Class: Search...

Function/Operation:

Remote Control:

Fully qualified remote control class: org.overturetool.plotting.interpreter.TempoRemoteControl Browse...

Fig. 3. Overture launch configuration for remote control.

can be updated. A separate `VarListener` is attached to the `Value` objects for each variable that is to be monitored.

2.3 Control and Subscription Protocol

The protocol offered by the extension enables top level control of the Overture VDM interpreter and also defines messages to control various aspects of model execution and monitoring including: definition of model entry point, query model structure information, subscription to value changes in variables and execution of operations.

The most important messages in the protocol are illustrated in Listing 1.1 and Listing 1.2 using the VDM model shown in Listing 1.3:

RunModel tells the interpreter to start simulating the model using a previously configured entry point class and an operation provided in the message. Listing 1.1 line 2-3.

SetRootClass tells the interpreter to set the entry point to the class passed in the message. Listing 1.1 line 6-9.

GetFunctionInfo requests from the interpreter a list of all visible functions and operations that can be used as entry point from the currently configured entry point class. A response is sent with a list of available operation names. Listing 1.1 line 12-13.

GetModelInfo requests from the interpreter a list of all observable variables in a model, accessible from the defined entry point. A response is sent that recursively lists the names of all observable variables. Listing 1.1 line 16-22.

Subscribe subscribes to updates sent out when the variable passed in the message is update. Currently, only instance variables present in the class definition of the root class (both static and non-static) are supported. Listing 1.1 line 25-26.

Execute tells the interpreter to execute the model operation passed in the message. This message can only be sent when the interpreter has an active simulation and the requested operation must be visible. These messages will allow the UI to pass data into the model and control its execution. Listing 1.2 line 2-4.

ValueUpdate a message sent containing the name and updated value of a subscribed variable. This message is broadcast by the server whenever the value of a subscribed variable changes. Listing 1.2 line 7-13.

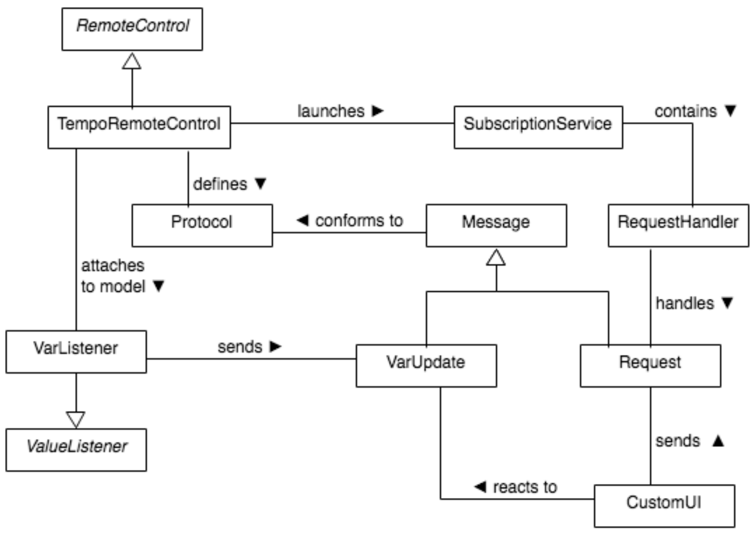


Fig. 4. The main elements of the extension.

StopServer tells the interpreter to stop the simulation. Listing 1.2 line 16.

All messages in the protocol are specified using JSON. The protocol uses a request response pattern where all messages except value change notification has a predefined request and response type. Listing 1.1 in combination with Listing 1.2 shows how the messages looks when the VDM model in Listing 1.3 is simulated by selecting the `A` class as root and observing `x` while running `op()`. The visual 2D output graph of this is shown in Figure 5.

Listing 1.1. Request and response sample messages..

```
// Obtain model classes
{"type":"REQUEST","data":{"request":"GetClassinfo"}}
{"type":"CLASSINFO","data": ["A","B"]}

// Set current root class
{"type":"REQUEST","data":
  {"request":"SetRootClass","parameter":"B"}
}
{"type":"RESPONSE","data":"OK"}

// Get available functions or operations
{"type":"REQUEST","data":{"request":"Getfunctioninfo"}}
{"type":"FUNCTIONINFO","data": ["op"]}

// Obtain state info of root class
```

```
{"type":"REQUEST","data":{"request":"GetModelinfo"}}
{"type": "MODEL", "data":
  {" rootClass": "mm", "name":"", "type":"",
    "children": [
      {"name": "x", "type":" int", "children": []}
    ]}
}

// Subscribe to a variable change from the root class
{"type":"SUBSCRIBE","data":{"variableName":"x"}}
{"type":"RESPONSE","data":"OK"}
```

In Listing 1.2 only value change messages are shown for the first two values.

Listing 1.2. Request and response sample messages..

```
// Start simulation
{"type":"REQUEST","data":
  {"request":"RunModel","parameter":"op"}
}

// Receive value updates
{"type":"VALUE","data":
  {"variableName":"x","type":"int","value":"1"}
}

{"type":"VALUE","data":
  {"variableName":"x","type":"int","value":"0"}
}

// Stop server
{"type":"REQUEST","data":{"request":"StopServer"}}
```

The VDM model in Listing 1.3 is a two class model `A` and `B` where the latter class contains a state `x` which can be observed, it alters between `1` and `0` through the execution of `op()`'s execution.

```
class A
end A

class B

instance variables
  x : int := 0;

operations

public op : () ==> ()
op() ==
for all i in set {1,...,10}
do x := x + if (i mod 2) > 0
                then 1
                else -1;

end B
```

Listing 1.3. Oscillating between +-1

The Electron application will show the following 2D graph shown in Figure 5 when the model in Listing 1.3 is executed.

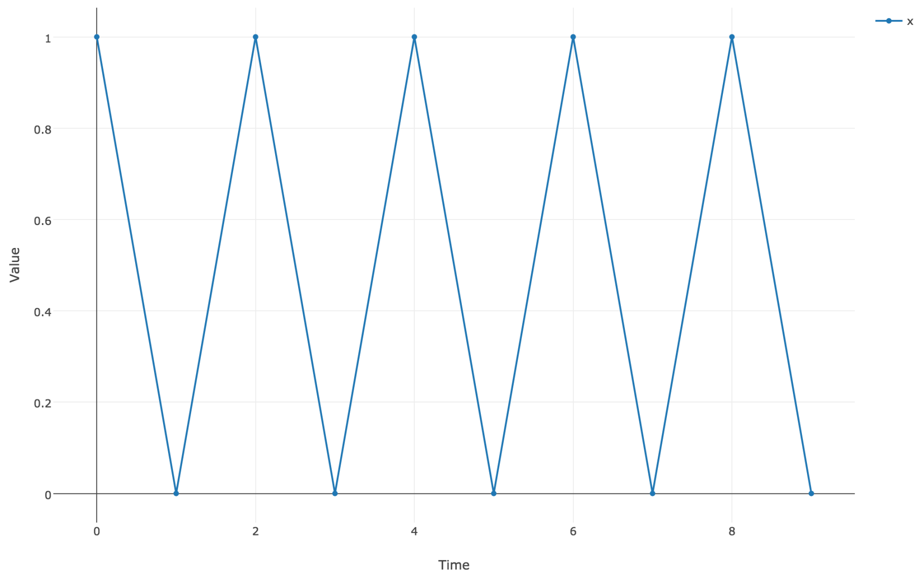
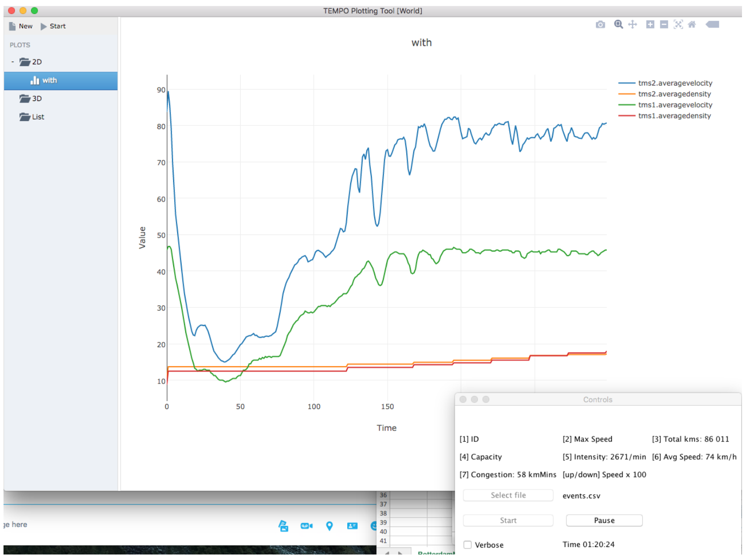


Fig. 5. 2D graph of the oscillating VDM model.

3 The Electron Application

As an example of the kinds of UIs that can be developed with our extension, we present a visualisation application based on variable plotting. The visualisation application is developed using the Electron framework that enables native applications to be developed using web technologies like JavaScript, TypeScript, HTML and CSS. Electron is compatible with Mac, Windows and Linux and therefore easily deployable to the same platforms as Java.

The plotting capability is created as an Electron application that uses web-sockets and JSON to communicate with the Overture Extension from Section 2. It enables the user to select the entry point of the model in the form of a class and operation/function. Both of these are currently limited to be used without any arguments. The user can then create plots based on variables accessible from the entry point. The variables that are offered as possibilities are all either numeric or lists (of fixed length) of numbers. Once such a configuration is completed the application can request the Overture interpreter to start simulating. The application will then plot all variable changes live during the simulation on the respective graphs. In Figure 6 an example of a 2D plot is shown.



[1] ID [2] Max Speed [3] Total kms: 86 011
[4] Capacity [5] Intensity: 2671/min [6] Avg Speed: 74 km/h
[7] Congestion: 58 kmMins [up/down] Speed x 100
events.csv
☐ Verbose Time 01:20:24

Fig. 6. 2D graph visualisation in the Electron application.

The plot configurations can also be stored and reloaded to make simulations of VDM models with more extensive plot configurations without having to redo this every time.

In addition to the conventional 2D plots it is also possible to plot using 3D as well as plot elements of list of numeric varies with fixed lengths. An example of a 3D plot can be seen in Figure 7. Note that it is possible to turn the different axes around when one is using a 3D plot. However, for an application like the one used in TEMPO the 3D effect does not give so much value so we have mostly used the 2D plots.

4 The TEMPO Case study

The TEMPO[6] case study is the first example in which the plotting extensions to Overture are used to communicate relevant parameters of a model execution to domain ex-

[6] TEMPO is an acronym for "TMS Experiment with Mobility in the Physical world using Overture". See `http://tempoproject.eu/` for more information.

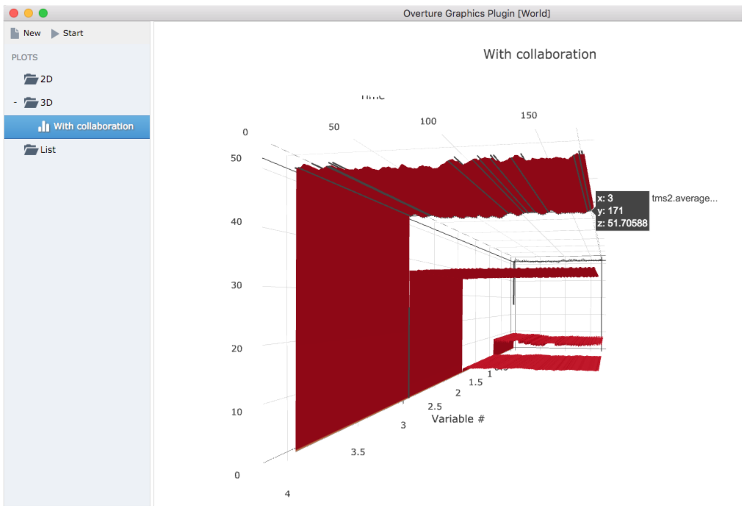


Fig. 7. 3D graph visualisation in the Electron application.

perts. It is a case study in the field of (road) traffic management, and it is based on the following idea.

Many European countries have dense road networks and significant traffic problems. The flow of traffic on Europe's roads is managed by a series of TMSs that are owned and controlled by various local and national authorities. A TMS consists of a collection of distributed systems and devices, usually installed along the roadside. These can be sensors that collect traffic data, such as cameras, radar detection systems, induction loops, or actuators that give instructions to road users via signs and signals. Current TMS architectures are usually run centrally from regional control centres. A low degree of collaboration hinders efficient management of traffic problems that straddle boundaries between authorities, because TMSs cannot communicate between regions and may have competing goals for traffic flow. While cooperation between various road authorities at a governance level has improved recently, technical barriers for collaborative TMSs are still to be removed.

The TEMPO project addresses the problem of disconnected TMSs by providing them with collaborative control architectures that engage with one another in automated negotiation processes. Negotiations are based on policies, which help reach agreement on which control measures are beneficial for the system as whole, improving the overall network performance.

Within TEMPO two VDM models have been made. One specifies a system of systems in which multiple TMSs co-exist that behave in an egocentric way: they do not communicate with one another and therefore they do no collaborate, they just apply the control measures that they have to only help themselves. The other model incorporates communication, negotiation and cooperation between various TMSs. They inform each other of their needs, and see if they can either provide help or get help to or from each other. An example of such help is setting up a diversion route. If one TMS has serious congestion problems on a part of its network then it can ask the other TMSs to provide a service "decrease output" to its network. Another TMS can then see if it has any measures that can provide this service, e.g. by setting up a diversion route that encourages travellers to avoid the congested part of the other TMS. Of course, setting up such a diversion route may cause problems for the TMS that provides the service itself, and therefore a cost (price) is associated with it. This is negotiated between the TMSs, ending up in a situation where the control measure is actually applied or not, depending on the costs.

In TEMPO, the execution of the two models (non-collaborative/egocentric versus collaborative) are compared to see if the collaborative version really provides better results. In order to do this, a simple traffic simulator has been developed which communicates a traffic situation to the model. The VDM model then executes the calculations (with or without negotiations) ending up with a series of traffic control measures to be applied. These are then communicated to the traffic simulator, which takes the measures into account, performs another round of traffic processing, provides a new traffic situation to the model, and so forth.

To determine how each model "performs" is not easy. One can think of performance parameters calculated over the execution of an entire simulation, e.g. the number of vehicle kilometres "produced" by the network, or the accumulation of congestion (product of length and duration), but these remain debatable and give little (visual) insight over what is happening during the course of the simulation itself. However, the plotting extensions are useful here. Typical indicators that a traffic engineer would look at in these situations are the average density of traffic in the network, and the average velocity (speed) of vehicles. In the model these parameters are modelled as instance variables. In the plots shown in Figure 8, generated with the extensions discussed we see density and speed evolve over time. The first plot shows the situation for a non-collaborative model and the second one for the collaborative one. The network used is a simplification of the road network in and around Rotterdam in The Netherlands. Part of this network is controlled by the municipality of Rotterdam (TMS1) and part by the national road authority Rijkswaterstaat (TMS2). It is easy to see how the plot visualisation gives a good impression of the performance of the two models.

5 Related Work

Tools for visualising VDM models have been developed in the past as well. The first ability to combine the interpretation of a VDM model with executable code was made on the VDMTools platform [5]. Here special dynamic link libraries were introduced. This means that the "pollution" of the VDM model was limited to a link to a `.dll` file and the explicit calls in the VDM model. Compared to the work presented here this is a much more cumbersome approach that requires a lot more knowledge to low-level implementation details.

For more recent related work it is in particular worth mentioning VDMPad [12]. VDMPad has an ability to explore the functionality of a VDM-SL model for example using a Read-Eval-Print Loop (REPL) capability. This lightweight approach for debugging does not require establishing new debug configurations etc. so it makes it faster

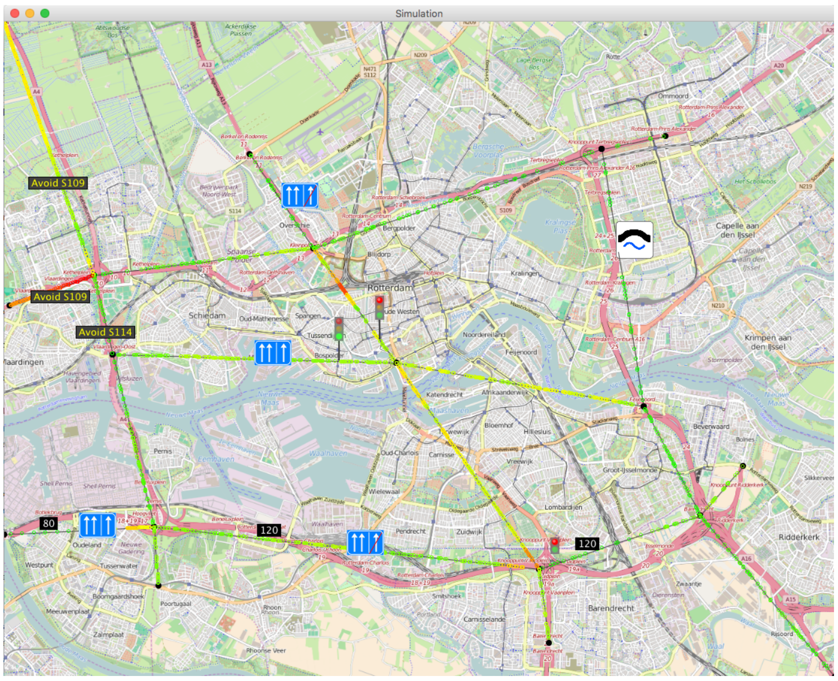


Fig. 8. Screenshot from the TEMPO simulation.

for the users to explore the behaviour. In addition it is worth mentioning that VDMPad enables a graphical view of VDM values for example instance variables which can be convenient for newcomers. The REPL functionality is also enabled in the recent WebIDE which has more IDE capabilities than VDMPad [15]. However, neither of these tools is able to combine executable code with the model and they are not able to monitor selected instance variables enabled in the work presented here. A new tool called ViennaTalk enables the combination with SmallTalk code but again it does not support the implicit monitoring presented here [13].

6 Concluding Remarks and Future Work

In this paper we have presented a new extension to the Overture tool that enables control of the Overture interpreter through a server and a protocol. The extension also features implicit monitoring of model variables, which allows UI interfaces to connect to a VDM model without needing to pollute the model with explicit UI calls. We have validated our extension by developing a UI application to monitor values in a traffic management system model.

The new extension offers greater flexibility in development of model UIs, as it allows the use of any technology that can connect to web sockets. In particular, modern web technologies such as JavaScript and HTML5 can be employed to develop better looking UIs in a faster way than the traditional Swing approach employed so far.

The protocol-based approach also enables a much cleaner decoupling between the model that represents the application and the UI. This leads to an easier separation of concerns between the model and the UI. Furthermore, both model and the UI can be evolved independently. The model does not need to be aware of the UI and the UI can rely on the protocol to act as an interface with the model.

Because both model and the UI can be developed independently, it is easier to go from a UI prototype to a production quality UI. As such, once the modelling phase is over and one begins system implementation, the UI code can be reused.

In this paper we have presented UIs focused on displaying numerical model data, as that was the primary need of the TEMPO project, which supported the development of the extension. However, the extension can enable the construction of other kinds of UIs. Any variable can be subscribed to, so it is possible to display various kinds of data. However, for complex structured data types such as records or classes, it may be necessary to write code to adequately render the data. Alternatively, the ASCII VDM representation of the data type may be used.

In the future, we would like to further validate our approach through the development of additional UIs. In particular, we would like to enrich the current collection of standard examples with user interfaces developed with this extension. Thus, the examples can showcase a new feature of Overture and the UIs themselves can serve as documentation for the extension and inspiration for users. To fully support this effort, we also hope to complete development of the protocol by supporting the `Execute` message and enabling two-way control between UI and model.

Finally, it would be worth investigating the possibility of combining our extension with the Overture code generator so that the UI code is included in the code generation process where it would be automatically modified in order to control the generated code instead of the model.

Acknowledgments Our current work is partially supported by the European Commission as a small experiment selected for funding by the CPSE Labs innovation action (grant number 644400). Thanks also go to the anonymous reviewers and Skip Balk, Pieter Buzing, John Skovsgaard and Henrik Nymann Jensen for their contributions to the different extensions.

References

1. Crockford, D.: The application/json media type for javascript object notation (JSON). RFC 4627, Internet Engineering Task Force (July 2006)
2. The Electron website. `http://electron.atom.io/` (2016)
3. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys (CSUR) 35(2), 114–131 (2003)
4. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. Wiley Encyclopedia of Computer Science and Engineering (2008), edited by Benjamin Wah, John Wiley & Sons, Inc.
5. Fröhlich, B., Larsen, P.G.: Combining VDM-SL Specifications with C++ Code. In: Gaudel, M.C., Woodcock, J. (eds.) FME'96: Industrial Benefit and Advances in Formal Methods. pp. 179–194. Springer-Verlag (March 1996)

6. The Jetty website. `http://www.eclipse.org/jetty/` (2016)
7. Krasner, G.E., Pope, S.T., et al.: A description of the model-view-controller user interface paradigm in the smalltalk-80 system. Journal of object oriented programming 1(3), 26–49 (1988)
8. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (January 2010), `http://doi.acm.org/10.1145/1668862.1668864`
9. Larsen, P.G., Lausdahl, K., Tran-Jørgensen, P.W.V., Ribeiro, A., Wolff, S., Battle, N.: Overture VDM-10 Tool Support: User Guide. Tech. Rep. TR-2010-02, The Overture Initiative, www.overturetool.org (May 2010)
10. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating A Distributed real time system using VDM. In: Qin, S., Qiu, Z. (eds.) Proceedings of the 13th international conference on Formal methods and software engineering. Lecture Notes in Computer Science, vol. 6991, pp. 179–194. Springer-Verlag, Berlin, Heidelberg (October 2011), `http://dl.acm.org/citation.cfm?id=2075089.2075107`, ISBN 978-3-642-24558-9
11. Nielsen, C.B., Lausdahl, K., Larsen, P.G.: Combining VDM with Executable Code. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) Abstract State Machines, Alloy, B, VDM, and Z. Lecture Notes in Computer Science, vol. 7316, pp. 266–279. Springer-Verlag, Berlin, Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-30885-7_19`, ISBN 978-3-642-30884-0
12. Oda, T., Araki, K., Larsen, P.G.: VDMPad: a Lightweight IDE for Exploratory VDM-SL Specification. In: Plat, N., Gnesi, S. (eds.) FormaliSE 2015. pp. 33–39. In connection with ICSE 2015, Florence (May 2015)
13. Oda, T., Araki, K., Larsen, P.G.: ViennaTalk and Assertch: Building Lightweight Formal Methods Environments on Pharo 4. In: Proceedings of the International Workshop on Smalltalk Technologies. pp. 4:1–4:7. Prague, Czech Republic (Aug 2016)
14. Plat, N., Larsen, P.G., Pierce, K.: Modelling Collaborative Systems and Automated Negotiations. In: Larsen, P.G., Plat, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 108–123. Aarhus University, Department of Engineering, Cyprus, Greece (November 2016), ECE-TR-28
15. Reimer, R.S., Saaby, K.D.: An Open-Source Web IDE for VDM-SL. Master's thesis, Department of Engineering, Aarhus University, Denmark (May 2016)