Towards integration of Overture into TASTE

T. Fabbri [1], M. Verhoef [2], V. Bandur [3], M. Perrotin [2], T. Tsiodras [2], P.G. Larsen [3]

[1] Dept. of Information Engineering, University of Pisa, Italy
[2] European Space Agency, ESTEC, Noordwijk, Netherlands
[3] Department of Engineering, Aarhus University, Denmark
tommaso.fabbri@for.unipi.it, marcel.verhoef@esa.int,
victor.bandur@eng.au.dk, maxime.perrotin@esa.int,
thanassis.tsiodras@esa.int, pgl@eng.au.dk

Abstract. Both TASTE and Overture have successfully demonstrated that method integration is a very promising strategy to create robust and effective tool suites. We show how the automated generation of bidirectional translation functions between VDM and ASN.1 type and value definitions allows the smooth integration of C-code generated from VDM models into TASTE. This enables rapid prototyping and early design validation directly in the target environment, while seamlessly interacting with other parts of the system specified in other notations supported by TASTE.

Keywords: Overture, TASTE, VDM, ASN.1, code generation

1 Introduction

The introduction of formal techniques in an engineering setting is notoriously difficult. Formal methods are typically focused on performing analysis on abstract models of a system, while traditional software engineering is aimed at producing source code that is usually much more detailed. Moving from the former to the latter, requires either to lower the level of detail in the formal models, or relying on translation techniques to do this (semi-) automatically, or by using a combination of both. Automatic generation of source code from formal models is certainly not new, but one of the key issues that hampers their adoption in practice is the fact that it is usually an "all or nothing" approach, which complicates the integration with other system software artifacts, i.e. legacy code, required run-time libraries, device drivers and the operating system.

In previous work [14], we explored an approach consisting in integrating a front-end formal modelling tool, called Overture [8], with a back-end software integration platform, called TASTE [6]. We show in this paper how such a tool integration can be achieved, providing us with a very powerful tool chain that allows the production of high quality executables created from a heterogeneous set of models and other software artifacts.

We first provide an overview of the TASTE environment in Section 2, followed by a short recap of the involved Overture features in Section 3, as we expect that the reader is already fairly familiar with the latter. We then discuss how the integration of the two tools is achieved in Section 4, which is demonstrated on a small case study presented in

Section 5. Finally, we draw some conclusions from our work in Section 6 and discuss how these results can be taken further.

2 The TASTE tool chain

TASTE is an acronym and stands for The ASSERT Set of Tools for Engineering, referring to the European FP6 project Automated proof based System and Software Engineering for Real-Time applications (ASSERT), which ran from September 2004 to December 2007 [5], from which the development philosophy and tool chain originate. Since then, TASTE has evolved into an active open-source initiative, dedicated to the development of high integrity distributed real-time systems [4].

TASTE addresses the problem of developing complex distributed systems, by following a strict heterogeneous model-based approach, supported by high levels of automation. This allows the user to concentrate on building the functionality for each system component in the language of choice that is most appropriate. Advanced code generation and build automation ensures consistency at each development step, removing tedious and error prone manual tasks that often are the root cause of integration and test problems. This philosophy is quite fitting with modern agile development and continuous integration approaches, however these are not at all common in this domain.

In order to achieve that goal, TASTE has adopted mature programming and modelling languages based on open standards with long-term open source and commercial support. At its core, TASTE relies on two formal modelling techniques:

AADL [3] captures the overall system architecture: the main components and their internal structure, the deployment and dynamic behaviour of these components, their interfaces (see [3], used in the TASTE Interface and Deployment Views).
ASN.1 [4] plays a pivotal role in TASTE, as it is used to provide bidirectional translations between all types used to exchange data between the constituent models (see [4], used in the TASTE Data View).

The tool chain ensures that native data types living in one model can be transfered vice versa to any other model without loss of fidelity, while enforcing all consistency requirements, such as invariants. One of the unique selling points of this approach is that the formal definition of the data type in ASN.1 and the physical encoding of the associated value are fully independent, which allows the user to change the value encoding without the need to change the interface of the application that relies on that data type. This provides both flexibility as well as the means to optimize the implementation, even in very late stages of the design. ASN.1 is both the Swiss army knife and super glue of TASTE.

For specifying system behaviour, the user has a wide range of techniques at his disposal. For example, SDL [11] can be used to specify state machines, and interfaces can be generated to directly include code generated from SCADE and Simulink. Also the inclusion of hand-written or legacy code (for example written in C and Ada) is possible. And finally, hardware/software co-design is supported by the automatic generation of both device drivers in C and for the counterpart hardware interfaces in VHDL.

[4] The main project web-site is `http://taste.tuxfamily.org`

TASTE allows to seamlessly integrate all these artifacts into a set of executables, compliant to the defined distributed system architecture in AADL. It relies on a portable and light-weight middleware called PolyORB-HI, which provides the portability abstraction across a wide range of target operating systems such as Windows, Linux, Xenomai (real-time Linux) and RTEMS. PolyORB-HI has been designed with high integrity applications in mind (i.e. ensuring low and deterministic performance overhead, low memory footprint, no dynamic memory management and so on). TASTE will produce binary application images that can be directly downloaded and executed on a wide range of embedded targets, emulators and simulators. Switching from one platform to another is merely a matter of setting a few configuration parameters and rebuilding the application. Last but not least, TASTE provides several features to assist in advanced system validation, such as the automatic generation of test suites and Python test scripts, the import and export of datasets into SQL databases, the creation of graphical user interfaces with features to record, plot and even play back data, for example using Message Sequence Charts [9].

Most if not all notations used within TASTE have a well-defined formal syntax and semantics, which not only allows to seamlessly integrate the artifacts generated from the constituent heterogeneous models, but also to perform upfront analysis. For example, schedulability of the design can already be analysed at the AADL level before even a single behavioral specification is available or the choice for the hardware platform is made. Moreover, consistent design documentation can be generated directly from the models, which is in particular useful for interface control documents. The well defined semantics also enables the integration of other tools and techniques, making TASTE extensible in a coherent and controlled way. This is the key to creating a tool chain that is aimed at improving productivity.

3 The Overture tool and `vdm2c`

Overture is a tool that enables the definition, validation and simulation of VDM models, supporting the three dialects VDM-SL, VDM++ and VDM-RT [8]. VDM-SL is the core language, with object-oriented specification facilities built on top of it as VDM++. Facilities for specifying timing and distributed architectures are in turn built on top of VDM++ as VDM-RT. Note that VDM++ is the dialect of interest in this paper, as the VDM-RT aspects are already provided by TASTE in the Deployment View.

The code generation platform of Overture [7] is being employed to create a C code generator, which is called `vdm2c`, for an executable, object-oriented subset of VDM-RT. Note that the choice for C was made deliberately over C++. At the time of writing this paper, an early prototype of this code generator was available (version 0.0.2), providing support for basic language features. Despite the tool limitations, it has proven to be sufficient to demonstrate our proof of concept.

The C code is generated under the assumption that the original VDM specification has been validated using Overture, at the very least showing that the specification is syntax and type correct. The translator follows the structure of the specification as closely as possible. It implements VDM++ classes as C structures with appropriate function pointer and name mangling mechanisms in order to implement inheritance, and it car-

ries VDM types explicitly. These VDM types are part of a generic support library which is independent of the source VDM model. The fundamental data type of the support library is the C structure `TypedValue` shown near the bottom of Listing 1.

Listing 1. Fundamental code generator data type.

```
typedef enum {
  VDM_INT, VDM_NAT, VDM_NAT1, VDM_BOOL, VDM_REAL,
  VDM_RAT, VDM_CHAR, VDM_SET, VDM_SEQ, VDM_MAP,
  VDM_PRODUCT, VDM_QUOTE, VDM_RECORD, VDM_CLASS
} vdmtype;

typedef union TypedValueType {
  void* ptr;             // VDM_SET, VDM_SEQ, VDM_CLASS,
                         // VDM_MAP, VDM_PRODUCT
  int intVal;            // VDM_INT and INT1
  bool boolVal;          // VDM_BOOL
  double doubleVal;      // VDM_REAL
  char charVal;          // VDM_CHAR
  unsigned int uintVal; // VDM_QUOTE
} TypedValueType;

struct TypedValue {
  vdmtype type;
  TypedValueType value;
};

struct Collection {
  struct TypedValue** value;
  int size;
};
```

While it is easy to understand how basic VDM types such as **nat** and **char** can be accessed in a value of type **struct** `TypedValue`, structured values such as **set** and **seq** require further explanation. The elements of a value of this type are simply stored as an array inside a value of type **struct** `Collection`. The `size` field records how many elements the collection contains. Naturally, nested types are accommodated. VDM records are treated as VDM classes with no functions or operations.

4 The Integration between Overture and TASTE

The initial step of the integration between TASTE and Overture looked at the interactive simulation of a simple heterogeneous system model [12]. The main goal of this experiment was to define reactive system behaviour as a state machine using the Specification and Description Language (SDL) and perform more complex data manipulation algorithms in VDM, as a means of achieving separation of concerns. The software architecture is presented schematically in Figure 1. The SDL modeller and simulator in

TASTE, OpenGEODE, interacts directly with the Overture interpreter through a TCP socket connection, which exposes the standard remote control API of Overture. This allows OpenGEODE to call the Overture interpreter directly in order to execute specific operations in the VDM model, whereby OpenGEODE converts all TASTE internal data types to and from their VDM counterparts. The TASTE Abstract Syntax Notation One (ASN.1) compiler, asn1scc [13], was extended to translate the ASN.1 data type definitions in OpenGEODE into their VDM counterpart in Overture. This already provided an additional level of consistency, and demonstrated that a translation from ASN.1 to VDM was feasible for all types supported in TASTE. Examples of this translation is provided later in this section.

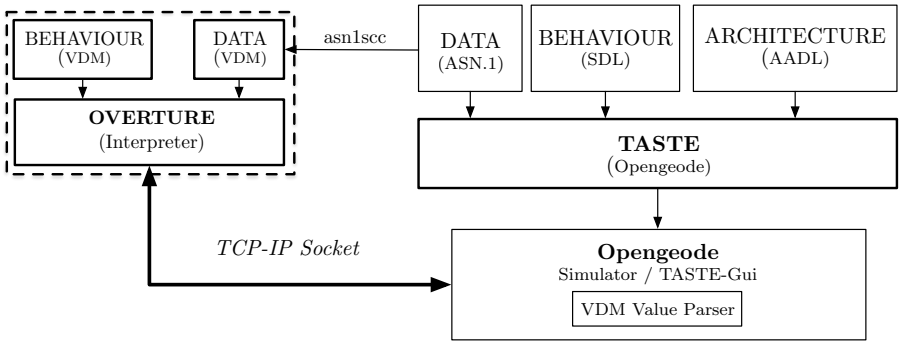

Fig. 1. Initial experiments for integrating SDL and VDM models

At this stage, the integration has involved only a few features of the TASTE tools: more precisely, a) the built-in behavioural modelling facilities (OpenGEODE) and b) the use of the ASN.1 compiler to ensure consistency of the data type definitions. This enabled the interactive simulation shown in [12] which is great for early system validation. However, the TASTE toolset is primarily aimed at the development of embedded, real-time systems: in particular TASTE is able to automatically assemble, glue together and deploy the final software system on a real target; none of these features have been exploited in the initial experiment. In order to have VDM as a new modelling language inside TASTE, a deeper integration is necessary.

As presented in Section 3, the Overture toolset currently supports a prototype code generator for the C language: vdm2c. The main idea is to allow the automatic integration of the generated C-code from Overture with other software artifacts comprising functions coded in other languages through the use of TASTE. Figure 2 presents the new integrated architecture: the compatibility of data exchanged between all software artifacts is ensured by the ASN.1 compiler. Finally, TASTE compiles and links all the pieces together and deploys the generated executable on the target platform.

To realize such integration, it is necessary to develop what makes the two generated pieces of code communicate. To have a better understanding, consider the following example where the function A (i.e. generated from Matlab-Simulink) wants to call the

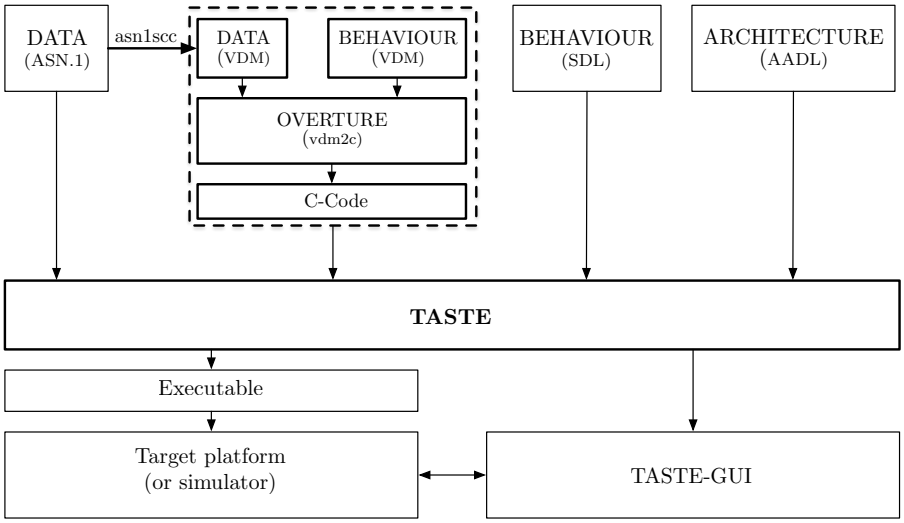

Fig. 2. TASTE Overture integrated architecture.

function `B` (generated from VDM) passing the parameter `c` of a given type. As already explained, TASTE relies on the ASN.1 language to define the exact content of the data types of the function parameters in a way that is independent from a specific implementation language. If the function `B` expects the parameter `c` of type `TypeC`, an 8-bit integer, the ASN.1 definition will include:

Listing 2. Example of a type definition using ASN.1.

```
TypeC ::= INTEGER (0..255)
```

At this stage, we would look at the `TypeC` type and find the best match in VDM language to represent it; then we would look at how `vdm2c` translates the VDM `TypeC` to C type. The same thing has to be done also in the `A` side and make sure they fit. Otherwise, we would write a function `Convert_TypeC_from_Matlab_to_VDM(...)` doing the conversion job. Despite being technically easy to perform, this is an error-prone and long process as soon as there are many functions to develop and maintain, in particular as these interfaces are likely to evolve over time. TASTE has tools that implements this whole process in an automated way, in the TASTE terminology:

`A_mapper` - starting from the ASN.1 types, the Data Modelling Toolchain [1] and/or the compiler templating facilities [13] can generate semantically-equivalent types. The `asn1scc` compiler is dedicated to safety-critical systems, and is able to generate optimized Ada and C code, but also customizable through template files for enabling support to other languages (e.g. VDM) [10].
`B_mapper` - generating the translation functions at code level that convert the data types obtained by `vdm2c` code generator from and to the `C` data types generated by

the ASN.1 compiler, as well as the piece of glue code that wraps the VDM function call with the translated parameter.

In this context, the Data Modelling Toolchain generates the VDM types starting from the ASN.1 data types definition, and currently supports all TASTE allowed data types. At the current time, the `B_mapper` supports simple data types such as `INTEGER`, `BOOLEAN`, `REAL` and `ENUMERATED` and complex types like `SEQUENCE OF`. The support of more structured data types, like `CHOICE` (the VDM union type) and `SEQUENCE` (the VDM record type) is under development [5]. Through the definition of these two components, the `A_mapper` and `B_mapper`, the VDM language is integrated into TASTE to create working prototypes. The Section 5 demonstrates these capabilities on a small case study.

5 Case Studies

First, we recall the data definition reported in Listing 2; through the invocation of the `asn1scc` compiler, the semantically equivalent VDM types are generated as reported in Listing 3, where ASN.1 constraints are translated into VDM invariants.

```
types
public TypeC = int
  inv x >= 0 and x <= 255
```

Listing 3. Generated VDM type starting from the ASN.1 definition of Listing 2.

The ASN.1 compiler `asn1scc` and `vdm2c` are both used for generating the corresponding `C` code. The ASN.1 compiler generates a `C` definition where `TypeC` is an alias of integer primitive type `asn1Sccint`, as shown in the Listing 4:

Listing 4. Native C data type, generated by the `B_mapper`.

```
  typedef asn1Sccint TypeC;
```

The `vdm2c` *maps* the defined `TypeC` into the `struct TypedValue` (see Listing 1), where the field `type` takes the value `VDM_INT` and the field `value` takes the `intVal` representation. The `B_mapper` is now used for generating functions executing the conversion between the two `C` codes obtained; recalling the example the conversion from the `asn1scc TypeC` to the VDM `TypedValue` and vice versa, the generated code is shown in Listing 5.

Listing 5. Generated convert functions by the `B_mapper`.

```
void Convert_TypeC_from_VDM_to_ASN1SCC
  (asn1SccMInt *ptrASN1SCC, TVP VDM)
{
  (*ptrASN1SCC) = (asn1SccSint)((VDM))->value.intVal;
}
```

[5] These features are currently also not supported by vdm2c.

```
void Convert_TypeC_from_ASN1SCC_to_VDM
  (TVP *ptrVDM, const asn1SccMInt *ptrASN1SCC)
{
  (*ptrVDM) = newInt((*ptrASN1SCC));
}
```

The generated functions are based on straightforward conversion as in the translation from VDM to `asn1scc` standard types, or based on the support library of `vdm2c` (see the call to `newInt`) for the definition of `TypedValue` variables. Similar functions are also automatically generated for more complex types, such as sequences, requiring iteration over the respective internal structures in order to perform the mapping. Now the basic elements are in place, we can present a small case study that demonstrates how the tools can be used in practice.

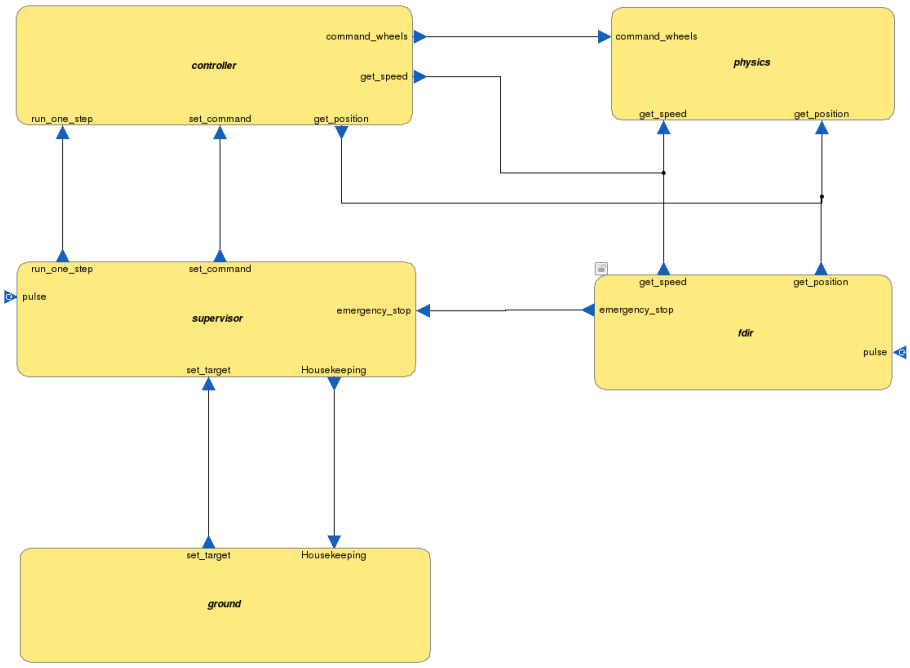

Fig. 3. Case study - TASTE Interface View

Figure 3 presents the interface view of an application consisting of five components. This top-level model, which is specified in AADL, gives an overview of all provided and required interfaces and their interconnections. A target language can be specified for each of the components. For example, the `ground` component is an automatically generated graphical user interface that allows to interact with the system. The `supervisor` is a state machine specified in SDL and the `controller` is specified

in VDM. Note that the `supervisor` has a periodic provided interface called `pulse`, for which the period is specified in the interface view, in this case 1 second. The TASTE middleware will ensure that this interface is called at this rate and that any deadlines specified are also enforced. The SDL specification of the `supervisor` is provided in Figure 4.

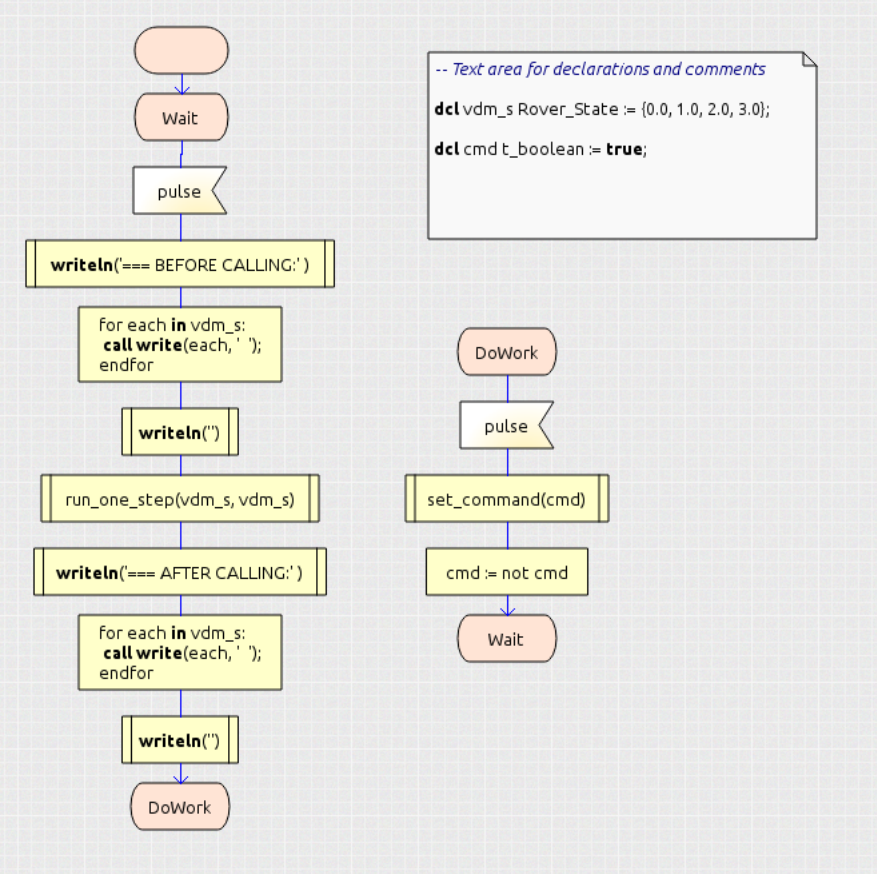

Fig. 4. Case study - TASTE supervisor SDL specification

The `supervisor` waits for the `pulse` event, then prints the `vdm_s` variable, calls the `run_one_step` operation and prints the `vdm_s` variable again. It continues in the $DoWork$ state, calls the `set_command` operation and toggles the `cmd` variable. Finally, it resumes to the $Wait$ state, which repeats the cycle at the next `pulse` event that is received. Note that `run_one_step` and `set_command` are provided interfaces of the `controller`. Also note that this model uses data types as specified in the TASTE data view, as shown in Listing 6.

Listing 6. Case study : type definitions ASN.1 (TASTE data view).

```
TASTE-Dataview DEFINITIONS ::=
BEGIN
IMPORTS T-Boolean FROM TASTE-BasicTypes;
Rover-State ::= SEQUENCE (SIZE(4)) OF REAL (0.0 .. 1000.0)
END
```

Using the `A_mapper` in `asn1scc`, TASTE will generate the following VDM model:

```
class TASTE_Dataview
types
    public Rover_State = seq of real
    inv x == len x = 4
end TASTE_Dataview

class TASTE_BasicTypes
types
    public T_Boolean = bool
end TASTE_BasicTypes
```

Listing 7. Case study: generated VDM data model.

and the following VDM specification template for the provided interfaces:

```
class controller_Interface
operations
  public Startup: () ==> ()
  Startup () is subclass responsibility;

  public PI_run_one_step: TASTE_Dataview`Rover_State ==>
    TASTE_Dataview`Rover_State
  run_one_step (-) == is subclass responsibility;

  public PI_set_command: TASTE_BasicTypes`T_Boolean ==> ()
  set_command (-) == is subclass responsibility;

end controller_Interface
```

Listing 8. Case study: generated VDM data model.

which can be conveniently modified to provide the following behavior:

```
class controller
  is subclass of controller_Interface

instance variables
  updateState : bool := false
```

```
operations
  public Startup: () ==> ()
  Startup () == updateState := true;

  public PI_run_one_step: TASTE_Dataview`Rover_State ==>
    TASTE_Dataview`Rover_State
  PI_run_one_step (vdm_state) ==
    if updateState then
      ( dcl newState : TASTE_Dataview`Rover_State :=
          [ vdm_state(1) + 1, vdm_state(2) + 2,
            vdm_state(3) + 3, vdm_state(4) + 4];
        return newState )
    else return vdm_state;

    public PI_set_command: TASTE_BasicTypes`T_Boolean ==> ()
    PI_set_command (cmd) == updateState := cmd;

end controller
```

Listing 9. Case study: user specified VDM model.

The specified behavior shown in this example is trivial of course, but it just serves the purpose of showing the process. In principle, the VDM model can be arbitrarily complex. The Overture `vdm2c` code generator can now be used to generate the C-code directly from the model shown in Listing 9. For conciseness, we do not include the generated code here, but it is available on-line, as part of the TASTE example suite [2]. The final step is the glue code and the ASN.1 conversion functions, as generated by the `B-mappers`. The ASN.1 conversion functions follow the same scheme as shown in Listing 5, but the additional glue code to link everything together, is shown below:

Listing 10. Glue code generated by the `B_mapper`.

```
#include "Vdm_ASN1_Types.h"
#include "controller.h"

static TVP controller;

void controller_startup()
{
  controller = _Z10controllerEV(NULL);
  CALL_FUNC(controller, controller, controller,
    CLASS_controller__Z7StartupEV);
}

void controller_PI_run_one_step (
  const asn1SccRover_State *IN_vdm_state,
  asn1SccRover_State *OUT_feedback )
{
  TVP ptr_vdm_state = NULL;
```

```
  Convert_Rover_State_from_ASN1SCC_to_VDM
    (&ptr_vdm_state, IN_vdm_state);
  TVP vdm_OUT_feedback;
  vdm_OUT_feedback = CALL_FUNC
    (controller, controller, controller, 1, ptr_vdm_state);
  Convert_Rover_State_from_VDM_to_ASN1SCC
    (OUT_feedback, &vdm_OUT_feedback);
}

void controller_PI_set_command(const asn1SccT_Boolean *IN_cmd)
{
    TVP ptr_cmd = NULL;
    Convert_T_Boolean_from_ASN1SCC_to_VDM(&ptr_cmd, IN_cmd);
    CALL_FUNC(controller, controller, controller, 2, ptr_cmd);
}
```

All the artifacts shown in this section are automatically collected by TASTE and built into an executable as specified in the TASTE Deployment View, which is shown in Figure 5, here indicating that all components are running on a single Intel x86 based Linux 32-bit operating system. The output of the running executable is shown in Listing 11, which demonstrates that the heterogeneous model works as specified.

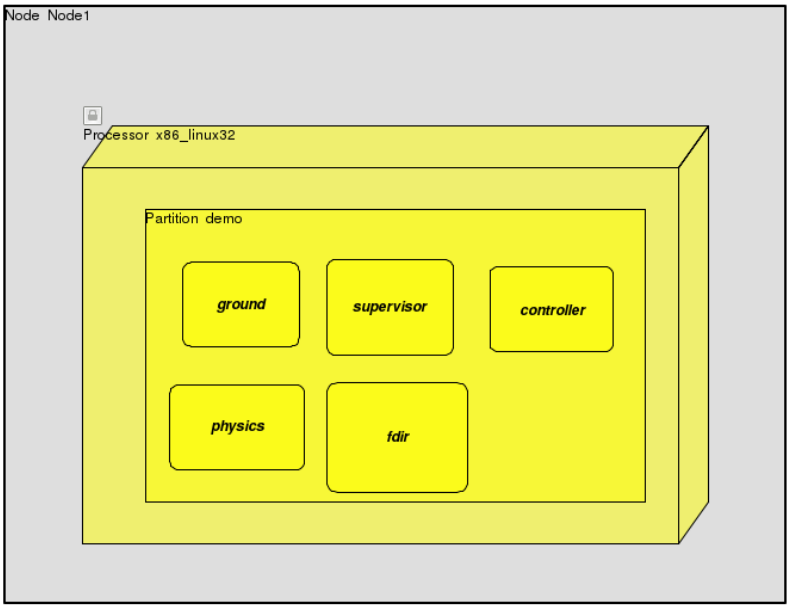

Fig. 5. Case study: simplified TASTE deployment view

Listing 11. Debug logging of the running application.

```
=== BEFORE CALLING: 0   1   2   3
=== AFTER CALLING:  1   3   5   7
=== BEFORE CALLING: 1   3   5   7
=== AFTER CALLING:  2   5   8   11
=== BEFORE CALLING: 2   5   8   11
=== AFTER CALLING:  2   5   8   11
=== BEFORE CALLING: 2   5   8   11
=== AFTER CALLING:  3   7   11   15
=== BEFORE CALLING: 3   7   11   15
=== AFTER CALLING:  3   7   11   15
=== BEFORE CALLING: 3   7   11   15
=== AFTER CALLING:  4   9   14   19
```

6 Conclusions and future work

The paper has presented the integration procedure of the VDM language as new technology for designing functionalities inside a TASTE environment. Such implementation involves the code generator `vdm2c` from the Overture side, the `asn1scc` compiler and tools from the TASTE side able to generate pieces of *glue code* to make heterogeneous technologies communicate. The case studies demonstrate how functionalities developed in the VDM language are mixed together with components developed in other languages to create a unique executable to be deployed on a target platform or tested on a simulator. At the current time only ASN.1 basic types (like `INTEGER, REAL, BOOLEAN, ENUMERATE`) and `SEQUENCE OF` basic types are completely supported.

But even in this limited setting, it has been clearly demonstrated that this kind of integration is only feasible if the steps are completely automated, and this is exactly what our solution now offers. Most of the complexity shown in the previous section is completely hidden from the user, allowing to concentrate on his main task: to specify all the system components in the most convenient formalism available in TASTE, as shown in Figure 3, Figure 4, Listing 6, Listing 9 and Figure 5. Next short term developments will be focused on the completion of the support for structured data types like `SEQUENCE` and `CHOICE`, following the on-going development of the `vdm2c` compiler, which is work in progress at this stage.

Our work has also shown that a bi-directional mapping between VDM and ASN.1 is indeed feasible. The existing Overture interpreter could therefore be extended with a feature to directly communicate to other outside tools, much akin to our initial experiments described in Section 4, but then using a communication channel based on the exchange of ASN.1 values, using a configurable encoding scheme. This would allow the coupling of Overture to other tools without the need for an external data converter, and enable the interactive use of the Overture analysis features during validation experiments.

Acknowledgments

This work has been supported by the ESA Summer Of Code In Space (SOCIS) 2016 run by the European Space Agency. The work on the VDM to C code generation from the Overture tool is supported by the INTO-CPS project (Horizon 2020, 664047)[6]. The authors would like to thank P. W. V. Tran-Jørgensen and K.G. Lausdahl for their help and participation in the development of the `vdm2c` code generator. Finally, we would like to thank the anonymous referees for valuable input on this work.

References

1. TASTE Data Modelling Tools. `https://github.com/ttsiodras/DataModellingTools/`
2. TASTE Example Suite. `https://tecsw.estec.esa.int/svn/taste/branches/stable/testSuites/WorkInProgress/Demo-TASTE-VDM/`
3. SAE AS 5506B: Architecture Analysis & Design Language (AADL) (2012), `http://standards.sae.org/as5506b/`
4. ITU X.680-X.693 : Information Technology - Abstract Syntax Notation One (ASN.1) & ASN.1 encoding rules, `http://www.itu.int/rec/T-REC-X.680-X.693-200811-I/en`
5. ASSERT: Automated proof-based System and Software Engineering for Real-Time Systems (5 2012), `http://cordis.europa.eu/project/rcn/71564_en.html`
6. Conquet, E., Perrotin, M., Dissaux, P., Tsiodras, T., Hugues, J.: The taste toolset: turning human designed heterogeneous systems into computer built homogeneous software. In: European Congress on Embedded Real-Time Software (ERTS 2010). Toulouse, France (May 2010)
7. Jørgensen, P.W.V., Larsen, M., Couto, L.D.: A Code Generation Platform for VDM. In: Battle, N., Fitzgerald, J. (eds.) Proceedings of the 12th Overture Workshop. School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446 (January 2015)
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. ITU-T Rec. Z.120 (02/2011) Message Sequence Chart (MSC), `http://www.itu.int/rec/T-REC-Z.120/en`
10. Perrotin, M., Grochowski, K., Verhoef, M., Galano, D., Mosdorf, M., Kurowski, M., Denis, F., Graas, E.: TASTE in action. In: 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016). TOULOUSE, France (Jan 2016), `https://hal.archives-ouvertes.fr/hal-01289678`
11. ITU-T Rec. Z.100 (04/2016) Specification and Description Language - Overview of SDL-2010, `http://www.itu.int/rec/T-REC-Z.100/en`
12. Taste-Team, Overture-Team: Integrating VDM and SDL. http://bit.do/sdl-vdm (2015)
13. Tsiodras, T.: ASN1SCC: An open source ASN.1 compiler for embedded systems. `https://github.com/ttsiodras/asn1scc`
14. Verhoef, M., Perrotin, M.: TASTE for Overture to keep SLIM. In: Proceedings of the 13th Overture Workshop. pp. 132–139. Center for Global Research in Advanced Software Science and Engineering, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan (June 2015), `http://taste.tuxfamily.org/wiki/images/7/7b/Taste-for-overture-verhoef-perrotin.pdf`, GRACE-TR-2015-06

[6] `http://into-cps.au.dk/`