Automated Exploration of Alternative System Architectures with VDM-RT

Kenneth Lausdahl and Augusto Ribeiro

Aarhus School of Engineering, Dalgas Avenue 2, DK-8000 Aarhus C, Denmark

Abstract. Choosing the optimal deployment of a distributed embedded application onto alternative hardware configurations is often difficult and time consuming. When developing a new product, a company must chose a hardware architecture that ensures both that the system behaves correctly according to its functional and timing specifications but also keeps its production cost at a minimum. The investigation to find this tradeoff between cost and performance can be very expensive if carried out at implementation time. A company can save money and development time if there is a possibility to quickly explore the design alternatives before the start of the implementation. In this paper we describe a method and associated tool support to assist in finding the best system design solution.

1 Introduction

As distributed real-time embedded systems become more and more prevalent around us, new techniques must to be used to ensure lower costs of development while keeping the service quality high. The quality of these kind of systems is normally not only measured by functional correctness but also by timing behaviour correctness. Because timing correctness is so essential when developing such systems, the implementation cost can increase considerably if it is discovered at later stages of the development that the selected hardware architecture cannot fulfil the timing parameters. Typical design questions that cross an architect's mind are [13]:

1. Does the proposed architecture meet the performance requirements of all applications?
2. How robust is the chosen architecture with respect to changes in the application or architecture parameters?
3. Is it possible to replace components by cheaper, less powerful equivalents to save cost while maintaining the required performance targets?

Models of software/hardware can be used to assist the system architect in answering these questions. They have previously been used to explore and validate different deployment architectures even before the implementation cycle starts [14,12]. By doing so, it is possible to gain knowledge, at an early stage, of the product that is being developed, even before any deployment decisions have been made. Usually, companies that wish to develop a distributed embedded system have certain target Printed Circuit Boards (PCBs) in mind, which support a small limited range of CPUs. These PCBs establish the architecture of the hardware and by using models and simulation techniques,

one can gain insight into which PCB should be selected in order to fulfil the project requirements. Furthermore, one can identify which kinds of CPUs and buses are needed to respect the speed/capacity requirements of the application. Having a method and tool support to test out all the interesting PCB/CPU/bus combinations and identify which ones satisfy the system timing invariants, would give the system architect an advantage when making such initial design choices.

The modelling language VDM-RT, enables a system architect to do these kinds of different simulations, but the process of changing the system architecture and application deployment is very cumbersome and lacks both flexibility and tool support. Everything concerning deployment is tightly connected and mixed with the application construction in the *system*[1] class. This makes it difficult and impractical to explore different architectures from a modeller's point of view. When one wants to make such changes, one must either overwrite the **`system`** class, and by doing so losing the previous system, or create a new project and copy all the files, except the system class, and then create a new system class.

In this paper we show how deployment of VDM-RT models can be modified to support exploration of different hardware configurations without changing the model and how this can help a system architect to find the good designs. This can be seen as a part of a larger effort in order to explore the different design alternatives of an embedded distributed system in the style used in the DESTECS project [2]. We consider "the best design" to be any solution that solves the proposed problem. It is then up to the architect to decide which is the best one based on system invariants and additional cost analysis.

The remainder of this paper is set out as follows. In section 2 an introduction to VDM and the VDM-RT dialect is presented. Section 3 illustrates how we separate the model from its deployment without loosing expressiveness. In section 4, the typical design questions are addressed and solutions are presented to show how this work assists the answering of these questions. Section 5 illustrates how this work can be used to explore an in-car-navigation system. Lastly, section 6 concludes this work with remarks and suggestions for future improvements.

2 The VDM Real-Time Dialect

The Vienna Development Method (VDM) [1,8,4] was originally developed at the IBM laboratories in Vienna in the 1970s and as such it is one of the longest established formal methods. The VDM Specification Language is a language with a formally defined syntax, static and dynamic semantics. Models in VDM are based on data type definitions built from simple abstract types such as **`bool`**, **`nat`** and **`char`** and type constructors that allow user-defined product and union types and collection types such as (finite) sets, sequences and mappings. Type membership may be restricted by predicate invariants. Persistent state is defined by means of typed variables, again restricted by invariants. Operations that may modify the state can be defined implicitly, using pre- and post-condition predicates, or explicitly, using imperative statements. Such operations denote relations between inputs and pre-states and outputs and post-states, allowing for non-

[1] The **`system`** class describes the system architecture and its deployment.

determinism. Functions are defined in a similar way to operations, but may not refer to state variables.

Three different dialects exists for VDM: The ISO standard VDM Specification Language (VDM-SL) [5], the object oriented extension VDM++ [6] and a further extension of that called VDM Real Time (VDM-RT) [15,7]. All three dialects are supported by the open source tool called Overture [9].

VDM++ and VDM-RT allow concurrent *threads* to be defined. In VDM-RT, the concurrency modelling can be enhanced by deploying objects on different CPUs with buses connecting them. Operations called between CPUs can be asynchronous, so that the caller does not wait for the call to complete.

VDM-RT has a special **`system`** class where the modeller can specify the hardware architecture, including the CPUs and their bus communication topology; the dialect provides two predefined classes for the purpose, `CPU` and `BUS`. CPUs are instantiated with a clock speed (Hz) and a *scheduling policy*, either *First-come, first-served (FCFS)* or *Fixed priority (FP)*. Only one **`system`** is allowed to be declared at a time for a single model.

The initial objects (artifacts) defined in the model can then be deployed to the declared CPUs using the CPU's `deploy` operations. Buses are defined with a transmission speed (bytes/s) and a set of CPUs which they connect. Object instances that are not deployed to a specific CPU (and not created by an object that is deployed), are automatically deployed onto a *virtual CPU*. The virtual CPU is connected to all real CPUs through a *virtual bus*. Virtual components are used to simulate the external environment for the model of the system being developed.

In figure 1 a graphical representation of an in-car navigation radio system is shown, which illustrates deployment with three CPUs connected by a single bus.

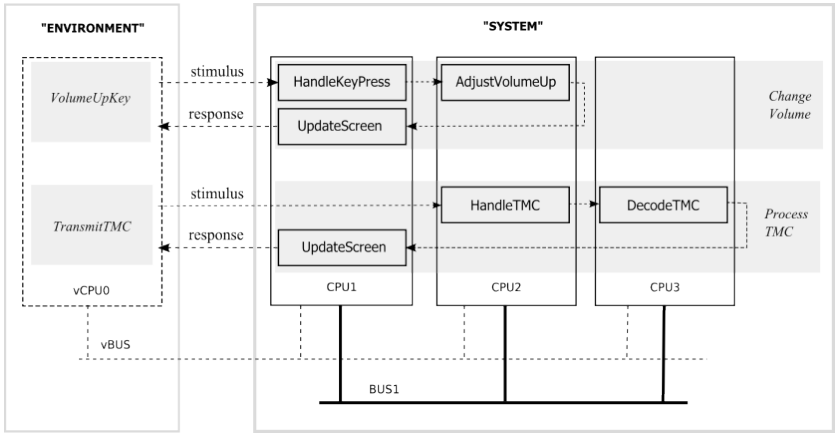

Fig. 1. Overview of the In-car Radio Navigation System

The in-car-navigation system shown in figure 1 is represented as a **`system`** class in listing 1.1. Firstly, the application artifacts are declared as instance variables (`mmi`,

radio and navigation). The definition of the hardware appears after: in this case three CPUs are declared (CPU1, CPU2 and CPU3) with a bus (BUS1) which connects them. Finally, the system architecture and deployment of the artifacts to the specific CPUs appear in the last section.

```
system RadNavSys
instance variables
  -- create artifacts
  static public mmi : MMI := new MMI();
  static public radio : Radio := new Radio();
  static public navigation : Navigation := new Navigation();

  -- create CPUs (policy, capacity)
  CPU1 : CPU := new CPU (<FP>, 22E6);
  CPU2 : CPU := new CPU (<FP>, 11E6);
  CPU3 : CPU := new CPU (<FP>, 113E6);

  -- create bus (policy, capacity, topology)
  BUS1 : BUS := new BUS (<FCFS>, 72E3, {CPU1, CPU2, CPU3})

operations
  public RadNavSys: () ==> RadNavSys
  RadNavSys () ==
    ( navigation.setMmi(mmi);
      radio.setMmi(mmi);
      radio.setNavigation(navigation);
      mmi.setRadio(radio);

      CPU1.deploy(mmi,"MMI");
      CPU2.deploy(radio,"Radio");
      CPU3.deploy(navigation,"Nav");
      ...
    );
end RadNavSys
```

Listing 1.1. A typical **system** class

Special system invariants based on timing constraints can be validated through post analysis of log files [3]. This enables the modeller to express time constraints on operations and instance variables; e.g. when volumeUp is called then no later than three time units later the volume must be incremented. Log files used for post analysis can be directly created by the VDM interpreter [10] enabling automated checking of such time constraints, allowing a systematic rejection of models which do not meet the time constraints either because the models are wrongly specified or the architecture used is not powerful enough. At the this point in time only post analysis is possible but a version to check system invariants at run-time is being investigated [11].

3 Ensuring Separation Between Software And Hardware

To enable automated exploration of hardware architectures for VDM-RT models, changes must be made to the way deployment is expressed. Currently, the modeller must create new projects with a custom **system** class for each architecture to be explored. This method is inefficient and difficult to automate. The basic problem with the current system definition is the close coupling between system architecture and system deployment. Ensuring a clear separation between architecture and deployment allows a system to be configured and tested against any number of hardware architectures without the hassle of creating new test projects or changing the system architecture.

This section will present a different approach to express deployment from the current VDM-RT **system** class explained in section 2 while preserving the same run-time properties.

This section will present a different approach to express deployment from the current VDM-RT **system** class shown in section 2 while preserving the same run-time properties. Instead of a single **system** class defining the deployment, our approach uses a four level structure to define deployment, keeping a clear separation between the model and the actual deployment. This allows tool automated exploration at all levels:

Abstract Software Architecture: Defines artifacts and how they depend on each other;
Abstract Hardware Architecture: Defines the abstract hardware architecture in terms of nodes and communication channels, i.e. without speeds/capacities or policies;
Configuration: Defines deployment of artifacts presented in Abstract Software Architecture to nodes from the hardware present in the Abstract Hardware Architecture;
Deployment: Defines a concrete deployment using the Configuration; similar to the constructor in the current **system** class.

For each of the levels above, a concrete definition and the relation between them will be presented as VDM-SL types and functions in the following sections.

3.1 Abstract Software Architecture

The Abstract Software Architecture (ASA) is used to describe which application artifacts exist in the system and where inter-artifact calls occur. It represents the software system at its most abstract point where only artifacts of applications are referred e.g. `mmi`, `radio` and `navigation` from section 2. The ASA contains dependencies between the different artifacts representing the inter-artifact calls in the system. This dependency description is used both (1) to check that a hardware architecture contains the required communication channels and (2) for automatic exploration of hardware architectures fitting the software model. Listing 1.2 shows the VDM types used to represent the ASA of a system.

```
types
Artifact : seq of char

ASA ::
```

```
  artifacts    : set of Artifact
  dependencies : map Artifact to set of Artifact
inv mk_ASA(artifacts,dependencies) ==
  dom dependencies subset artifacts
  and
  dunion rng dependencies subset artifacts
  and
  forall key in set dom dependencies &
        key not in set dependencies(key);
```

Listing 1.2. Abstract Software Architecture types.

The `Artifact` type denotes a named system instance variables (e.g. `mmi`); the `artifacts` set denotes the set of artifacts which can be deployed; the `dependencies` map denotes the dependencies between the artifacts.

3.2 Abstract Hardware Architecture

From an abstract point of view, a computing system is no more than a set of processing nodes which communicate via channels. We name this representation: Abstract Hardware Architecture (AHA). Listing 1.3 presents VDM types capable of representing an abstract hardware architecture.

```
types
Node ::
  id : nat1;

ComChannel ::
  nodes : set of Node;

AHA ::
  nodes    : set of Node
  channels : set of ComChannels
inv forall c in set channels & c.nodes subset of nodes;
```

Listing 1.3. Abstract Hardware Architecture as a VDM type

A processing node is represented by `Node` which has an identifier and a communication channel is represented by `ComChannel`, which contains the set of nodes it connects. `AHA` defines a hardware architecture containing several nodes and channels connecting them. AHAs can either be automatically generated based on the maximum number of artifacts in the system or manually specified which is often the desired solution for an industry where existing PCBs are available from previous projects.

3.3 Configuration

A configuration describes how a system is deployed to an abstract architecture. This allows a system to be deployed onto a hardware configuration without explicitly spec-

ifying the limitations of the hardware like CPU speed and bus capacity. A configuration defines a relation between artifacts from an ASA and the computing nodes from an AHA. The dependencies stated by an ASA must be reflected in the communication channels of the AHA for the configuration to be valid. This check is done by the function `checkDependencies`. Listing 1.4 defines a configuration of an ASA to an AHA. A configuration can be created either by automatic permutation of artifacts onto the nodes of an AHA or by manually specifying the relations. The latter is the normal case for an industry where specialized nodes such as processors with integrated GPS[2] modules are used, which will require a GPS artifact to be explicitly deployed to a specific node.

```
types
NodeArtifactRelation : map Node to set of Artifact;

Configuration ::
  asa : ASA
  aha : AHA
  relation : NodeArtifactRelation
inv mk_Configuration(asa,aha,relation) ==
  checkDependencies(asa, aha, relation);
```

Listing 1.4. Deployment Configuration

3.4 Deployment

The deployment of a system is the process of restricting the computational power of the nodes and the communication channels. A node must be limited to the computational power of a specific CPU with a maximum number of instructions it can perform per second. The same applies to buses where the transfer rate is limited. Listing 1.5 shows the `Deployment` type which represents a mapping between `Nodes` and `ComChannels` to concrete CPUs and buses.

```
Deployment ::
  config : Configuration
  buses  : map ComChannel to BUS
  cpus   : map Node to CPU
inv mk_Deployment(config,buses,cpus) ==
  (forall channel in set config.aha.channels &
        channel in set dom buses)
  and
  (forall node in set config.aha.nodes & node in set dom cpus)
  and card config.aha.channels = card dom buses
  and card config.aha.nodes = card dom cpus;
```

Listing 1.5. Specifies the type of each computational node and communication channel

[2] Global Positioning System

Computational nodes and communication channels are abstractions of the actual physical implementation where a circuit board is manufactured, which among other things consists of the main components CPUs and buses which VDM-RT can reason about. In listing 1.6 two VDM types are listed. `CPU` represents a computational `Node` where the node is limited from being infinitely fast to a specific frequency slowing down the execution of instructions. The same applies to the `BUS`, which is a limited version of the `ComChannel`, where a transmission speed limits the number of bytes which can be transmitted per second.

```
CPU ::
  id                  : nat1
  speed               : nat1
  brand               : seq of char
  scheduling          : <FP> | <FCFS>;

BUS ::
  id              : nat1
  speed           : nat1
  type            : <FILO>;
```

Listing 1.6. Hardware types

3.5 New Deployment Work-flow

To use the four layered separation described above some changes must be made to the deployment work flow. However, not all of the above levels require the modeller's direct attention, since most of the changes are conceptual separations of system elements. It is important to understand that the output of the separation proposed above can be mapped to the current VDM-RT system class without losing details. The difference is that this clear separation between the different levels, enables the modeller to do exploration at all levels. It also enables tools to be developed to assist this process.

The work-flow in ordinary VDM-RT can be described with the following steps:

1. Defining the VDM-RT model.
2. Identifying the static artifacts of the model.
3. Defining the hardware nodes: `CPU` and `BUS` and instantiating the artifacts.
4. Deploying the artifacts to the CPUs.

This is currently all done in a single class called **`system`** with no clear indication of what is artifacts and what is hardware and deployment.

The work-flow with the new sub divided structure:

Model development: The first step is to develop the actual VDM model as in the current VDM-RT workflow.

System configuration: The modeller configures the artifacts of the system as usual in a VDM-RT system class.

Extract artifacts and dependencies: If all artifact relations are expressed as either artifact constructor arguments or parsed as arguments to operations on artifacts, then this step can be automated. Artifacts will be extracted from the **`system`** class and their dependencies from the system constructor, enabling an ASA to be created[3]:

Composing a new AHA: The ASA defines the artifacts and their required dependencies while the AHA define an abstract hardware architecture which respects the dependencies from the ASA extracted from the artifacts dependencies. Such an AHA can either be automatically generated based on the ASA or it can be manually specified by the user.

Configuration: The configuration defines how each artifact is linked to a node of the given AHA. This can be specified manually by the user or a range of configurations can be generated from the pair (ASA, AHA).

Deployment: The final deployment is the limitation of an AHA. This can again be specified by the user to a single fixed deployment or the user can enter a set of possible CPUs which could be used per node allowing a range of deployments to be generated to explore these different CPU limitations.

Evaluation: Finally, the model can be executed with a single specific deployment and its system invariants can be checked either through post-analysis or at run-time both leading to an accept / reject verdict of the tested deployment. This indicates to the modeller if this configuration is acceptable to the system leaving the decision of which to choose to the modeller.

The steps described above can be expressed through the formula 1.

$$\left(ASA + AHA\right) \rightarrow^{*} Configuration \rightarrow^{*} Deployment \equiv \texttt{system} \qquad (1)$$

The arrow $\rightarrow^{*}$ denotes that many elements can be generated with respect to the left side of the arrow. In the first case one or more *configurations* can exist which configures a particular pair of ASA and AHA. Each configuration defines how the ASA is mapped onto the AHA but does not restrict the hardware in any way. Similar to the *configuration*, one or more *deployments* can exist which restricts a particular *configuration* by limiting each computational node to a specific frequency and each communication channel to a specific speed. Finally it can be seen from the **`system`** in section 2 that the left side of the formula below is equivalent to the information in the system class in VDM-RT.

4 The Exploration of Alternative System Architectures

Exploring alternative system architectures is supported by VDM-RT and in section 3 it has been described how the process of deployment can be split up into levels which can be explored for alternatives. The goal is to provide the means to answer the questions

[3] In this paper we do not deal with references which can be passed between artifacts at run-time which also leads to new dependencies.

stated in the introduction. However because these questions are seen from the modellers point of view, we will try to relate them to the levels of the formula 1 to make it easier to describe how this work provides (partial) answers to these questions.

The requirements extracted from the questions are as follows:

- Exploring alternative artifact distribution on a fixed hardware configuration.
- Exploring alternative hardware configurations for an ASA.
- Exploring alternative deployment parameters for a fixed configuration.

The questions require the exploration to support different distribution of artifacts on a fixed distributed hardware platform; the ability to explore parameters for a specific hardware such as CPU capacity; and finally a way to validate such a system architecture. Futhermore we can add the ability to generate hardware architectures, but this may be mainly of academic value. The requirements stated above are covered in the following subsections. In addition the validation is addressed in section 4.4.

4.1 Exploring Alternative Artifact Distribution On A Fixed Hardware Configuration

To explore alternative artifact distribution, an ASA is required to obtain the artifacts and their dependencies. Since the hardware configuration is fixed, an AHA is also provided by the modeller. This gives the pair (ASA, AHA) as input to the exploration of alternative artifact distribution. Formula 2 illustrates where this takes place in the overall work flow where the underlined part denotes what is produced. The result of the generation of alternative distributions is a set of `Configuration`s all for the same system.

$$\left(ASA + AHA\right) \rightarrow^* \underline{Configuration} \rightarrow^* Deployment \equiv \textbf{system} \tag{2}$$

Listing 1.7 shows the signature of a VDM function which produces the desired set of configurations:

```
createAltDisbs : ASA * AHA -> set of Configuration
createAltDisbs(asa, aha) == is not yet specified;
```

Listing 1.7. Signature of a function for generation of Configurations from an ASA.

4.2 Exploring Alternative Hardware Configurations For An ASA

When the goal is to find the optimal hardware configuration for a given system it can often be difficult and time consuming to create all possible combinations of nodes and communication channels. It is however important to understand that this is possibly only of academic value since industrial companies often have of-the-shelf hardware platforms which they want to explore. AHAs can be generated from the number of

unique artifacts from an ASA. The formula 3 shows where in the overall work flow this exploration contributes again using the underlined part as the produced aspects.

$$\left(ASA + \underline{AHA}\right) \rightarrow^* Configuration \rightarrow^* Deployment \equiv \textbf{system} \quad (3)$$

A signature of the VDM function to create the AHAs is shown in listing 1.8. It takes an ASA as input and returns a set of AHAs.

```
createAHAs : ASA -> set of AHA
createAHAs(asa) ==
  let maxNodes = card asa.artifacts
  in
    ...
```

Listing 1.8. Signature of a VDM function for the automatic AHA generation.

4.3 Exploring Alternative Deployment Parameters For A Fixed Configuration

Exploring alternative deployment parameters for an otherwise fixed system is one of the most important requirements because this relates directly to the costs of the final product. If a cheaper CPU can be used in mass production, money can be saved by the manufacturer of such a system. Exploring alternative deployment parameters means that one can come up with all possible limitations of the hardware, reducing either a CPUs computational capacity or limiting the bandwidth of a bus. An unlimited range of such deployments can be generated however this is not useful in practice since only a small number of CPUs and buses can be used in a specific hardware topology. In most cases, a PCB design already exists which supports a fixed number of different CPUs from a specific family. Thus the exploration is based on knowing that a small list of possible CPUs or buses are available to be used as nodes. The exploration generates a set of deployments and takes an otherwise fixed system as input together with a set of available CPUs per node of the AHA and a set of available buses for each communication channel. Formula 4 shows where in the overall work flow this takes place.

$$\left(ASA + AHA\right) \rightarrow^* Configuration \rightarrow^* \underline{Deployment} \equiv \textbf{system} \quad (4)$$

The signature of a VDM function is shown in listing 1.9 which takes a fixed configuration of a system plus two maps where the available CPUs and BUSs are given for the resources in the AHA.

```
exploreDeployParams : Configuration *
                      map Node to set of CPU *
```

```
                        map ComChannel to set of BUS
                        -> set of Deployment
exploreDeployParams(config, nCm, cBm) == is not yet specified;
```

Listing 1.9. Signature of a VDM function for alternative deployment parameter exploration.

4.4 Evaluation Of The Architectures

The ability to automatically determine if a specific deployment is good enough is very important now that we have presented the functionality to automatically generate alternatives as early as the AHA in the work flow. The potentially results is a very large number of deployments, since a split in the flow at an early stage doubles the output of all later steps. Currently, the only way to determine if a deployment is "good enough" is by manually inspecting the execution log through the graphical viewer named Real-Time Log Viewer. This viewer is able to illustrate how the scheduler creates threads, shifts them in and out in relation to time etc. To overcome the challenge of manual inspection work is being done in [11] to enable run-time checking of system invariants. Such invariants can then express time constraints in the system, which is exactly what is needed when deployments have to be validated. If the modeller provides system invariants expressing the critical time constraints of the model then the run-time checking of these invariants will be able to tell us if a given deployment has not violated any invariants and thus be accepted.

5 Case Study: In-car Radio Navigation

This case study is based on an already known case explored in both [3] and [13]. How this new structure can be used to do deployment exploration will be presented. The new way to express a system configuration is shown in listing 1.10, it can be seen that no deployment is included within the system class. This is very similar to the system from section 2.

```
system RadNavSys
instance variables
  -- create artifacts
  static public mmi : MMI := new MMI();
  static public radio : Radio := new Radio();
  static public navigation : Navigation := new Navigation();

operations
  public RadNavSys: () ==> RadNavSys
  RadNavSys () ==
    ( navigation.setMmi(mmi);
      radio.setMmi(mmi);
      radio.setNavigation(navigation);
      mmi.setRadio(radio);
```

```
    );
end RadNavSys
```

Listing 1.10. In-Car-Navigation system.

A new grammar for deployment in VDM-RT is proposed in listing 1.11, allowing the deployment elements: AHA, configuration and deployment to be specified. All the elements can be generated through exploration as explained in section 4. The listing 1.11 illustrates how the deployment of the in-car-navigation system can be done with this new syntax. The deployment is specified with all elements, but without the ASA, since it can automatically be extracted from the system class in listing 1.10. Any of the blocks **aha**, **configuration** and **deployment** can be left empty in the grammar, indicating that they should be automatically generated. However by explicitly specifying all blocks only a single deployment will exist as in the original system definition from section 2.

```
aha

Channel1 := {node1, node2, node3}

configuration

node1 := {mmi};
node2 := {radio}
node3 := {navigation}

deployment

node1 := CPU(200MHz, <FP>)
node2 := CPU(100MHz, <FP>)
node3 := CPU(1000MHz, <FP>)
Channel1 := BUS(72E3, <CSMACD>)
```

Listing 1.11. New deployment specification for the In-Car-Navigation system

What if the deployment specified in listing 1.11 is an acceptable deployment but the modeller likes to do future investigation through the third question: *Is it possible to replace components by cheaper, less powerful, equivalents to save cost while maintaining the required performance targets?* One option is to try out deployments where one of the nodes is limited to one of three different CPUs as shown in listing 1.12. It can be seen that the grammar allows nodes to be defined with a set of `CPU`s instead of a single `CPU` this allows the exploration to use permutations of CPUs for each node.

```
deployment

node1 := {CPU(200MHz, <FP>),
          CPU(100MHz, <FP>),
```

```
          CPU(50MHz, <FP>)}
node2 := CPU(100MHz, <FP>)
node3 := CPU(1000MHz, <FP>)
Channel1 := BUS(72E3, <FCFS>)
```

Listing 1.12. Alternative deployment block for exploration of deployment parameters.

When the exploration is done for the **`deployment`** block as shown in listing 1.12, three alternatives will be generated, one with each type of CPU. All these alternatives can then automatically be validated against the same tests to see if all of them fulfils the system invariant[4]. If so the modeller can freely decide which option is the best choice.

6 Concluding Remarks

Choosing the optimal architecture for a system is challenging, not only can it be difficult to determine but VDM-RT currently lacks the ability to allow exploration of alternatives in an efficient way without the need of duplicating the model. This work has proposed a way to enable exploration through separation of model and deployment, where exploration is possible at all levels of the deployment process. The common questions a modeller might ask when choosing a optimal architecture have been addressed and exploration functions proposed. We think that this work will help the system architect to determine an optimal architecture for a given system by enabling easy automated exploration. Such an exploration will be able to create all alternatives of AHA, Configuration and deployments and evaluate them against system invariants. VDM-RT priority settings for functions and operations has not been addressed in this work but will be future investigated in the near future.

The plan is to implement the features described in this paper in the Overture platform such that it can be exploited for automatic co-model analysis in the DESTECS project as well. We expect that this will be completed before the end of 2011.

Acknowledgements

This work was partly supported by the EU FP7 DESTECS Project. We appreciate the input we have had from the different partners on this work. In addition we would like to thank Nick Battle for valuable input on this paper.

References

1. Bjørner, D., Jones, C. (eds.): The Vienna Development Method: The Meta-Language, Lecture Notes in Computer Science, vol. 61. Springer-Verlag (1978)
2. Broenink, J.F., Larsen, P.G., Verhoef, M., Kleijn, C., Jovanovic, D., Pierce, K., F., W.: Design support and tooling for dependable embedded control software. In: Proceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems. ACM (April 2010)

[4] The system invariants are not included in this paper but can be found in [3].

3. Fitzgerald, J.S., Larsen, P.G., Tjell, S., Verhoef, M.: Validation Support for Real-Time Embedded Systems in VDM++. In: Cukic, B., Dong, J. (eds.) Proc. HASE 2007: 10th IEEE High Assurance Systems Engineering Symposium. pp. 331–340. IEEE (November 2007)
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. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edn. (2009), ISBN 0-521-62348-0
6. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005), `http://www.vdmbook.com`
7. Hooman, J., Verhoef, M.: Formal semantics of a VDM extension for distributed embedded systems. In: Dams, D., Hannemann, U., Steffen, M. (eds.) Concurrency, Compositionality, and Correctness, Essays in Honor of Willem-Paul de Roever. Lecture notes in Computer Science, vol. 5930, pp. 142–161. Springer-Verlag (2010)
8. Jones, C.B.: Systematic Software Development Using VDM. Prentice-Hall International, Englewood Cliffs, New Jersey, second edn. (1990), iSBN 0-13-880733-7
9. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. ACM Software Engineering Notes 35(1) (January 2010)
10. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating a Distributed Real Time System using VDM. Submitted for publication (2011)
11. Ribeiro, A., Lausdahl, K., Larsen, P.G.: Run-Time Validation of Timing Constraints for VDM-RT Models. Submitted for publication (2011)
12. Verhoef, M.: On the use of VDM++ for Specifying Real-Time Systems. In: Fitzgerald, J.S., Larsen, P.G., Plat, N. (eds.) Towards Next Generation Tools for VDM: Contributions to the First International Overture Workshop, Newcastle, July 2005. pp. 26–43. School of Computing Science, Newcastle University, Technical Report CS-TR-969 (June 2006)
13. Verhoef, M.: Modeling and Validating Distributed Embedded Real-Time Control Systems. Ph.D. thesis, Radboud University Nijmegen (2008), ISBN 978-90-9023705-3
14. Verhoef, M., Larsen, P.G.: Interpreting Distributed System Architectures Using VDM++ – A Case Study. In: Sauser, B., Muller, G. (eds.) 5th Annual Conference on Systems Engineering Research (March 2007), Available at http://www.stevens.edu/engineering/cser/
15. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. pp. 147–162. Lecture Notes in Computer Science 4085 (2006)