Constraint-based Platform Variants Specification for Early System Verification

Andreas Burger[1], Alexander Viehl[1], Andreas Braun[1], Finn Haedicke[2,3], Daniel Große[2]
Oliver Bringmann[1,4], Wolfgang Rosenstiel[1,4]

[1]FZI Research Center for Information Technology, Haid-und-Neu-Str 10-14, 76131 Karlsruhe, Germany
[2]solvertec GmbH, Anne-Conway-Str. 1, 28359 Bremen, Germany
[3]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[4]University of Tuebingen, Sand 13, 72076 Tuebingen, Germany

[1] `[aburger,viehl,abraun]@fzi.de` [2] `[grosse,haedicke]@solvertec.de`
[4] `[bringman,rosenstiel]@informatik.uni-tuebingen.de`

To overcome the verification gap arising from significantly increased external IP integration and reuse during electronic platform design and composition, we present a model-based approach to specify platform variants. The variants specification is processed automatically by formalizing and solving the integrated constraint sets to derive valid platforms. These constraint sets enable a precise specification of the required platform variants for verification, exploration and test. Experimental results demonstrate the applicability, versatility and scalability of our novel model-based approach.

I. Introduction

The increase in complexity of distributed embedded systems and systems-on-chip (SoC) platforms over the last decade in combination with the decrease in acceptable time-to-market are issuing tremendous challenges for the platform development. Besides the growing complexity, other problems are the increase in integration of external IPs and the rising usage of reused logic in the platform design composition [1]. Due to these facts, the integration and parameterization of platform components and their configurations grow significantly which causes huge platform variant and configuration spaces.
In [2] for example, an automatic gear shifting application from Daimler Trucks is described which consists of 6.4 million valid variants. Each of them can be integrated in trucks and should be implemented, verified and tested. Another important example can be given by automotive networks like *FlexRay* [3] or *Media Oriented System Transport* [4] (MOST). A MOST network can integrate from 2 up to 64 communication devices as well as a set of different protocol parameters for each of them. This leads to a space of more than 10^{21} valid MOST network variants.
Due to these facts and the given examples, the verification of IP blocks in different platform variants and the verification of the interaction between multiple IP block instances is gaining importance. Further, the relevance of the verification of platform characteristics (e.g. network topology, component instances) and different component variants (e.g. software versions, component parameters) is growing. Therefore the focus in early verification and test is moving away from fixed virtual prototype platforms with variable test scenarios to virtual prototype platforms which are variable in the number and kind of components considering complex interdependencies, constraints and requirements.
Hence it is inevitable to devise methods which allow focusing on the model-based description and generation of feasible platform variants. This could reduce the manual effort in verification, exploration and test significantly and enables verification of IPs within different platform variants.
Therefore we demonstrate our novel approach on virtual prototyping but it can also be applied for real tests (e.g. *Hardware-in-the-loop* (HIL), *Software-in-the-loop* (SIL)). The structure of the platform variant specification is defined by UML-based [5] templates, so-called *Constrained-Structural Templates (CST)*. The hierarchically structured templates enable a high degree of structural flexibility. The usage of constraints allows to specify requirements and to realize complex interdependencies between different IPs and platform components. Equally they enable to control the huge platform variant and configuration space efficiently. This allows generating required variants for verification, exploration or test. The constraints are specified by an extended subset of the *Object Constraint Language (OCL)* [6]. They are encoded as a SAT instance on bit-vector-level to solve them by metaSMT [7]. The platform variant framework is implemented parallel and multi-threaded to overcome the overhead of generation and solving. In summary, the major contributions are:

- Precise specification of valid and feasible platform variant space
- Complete generation of required variants for verification, exploration and test
- High structural flexibility in the platform variants specification
- Fast reconfiguration capabilities of the specified platform variants space

By experimental evaluations we demonstrate our approach for verifying a MOST network using virtual prototyping. We also adopted our approach for an exploration of a FlexRay-based traffic sign recognition application. The examples demonstrate the necessity to specify the required variants for verification, exploration and test as well as the wide application field of our novel methodology.

II. Related Work

For variant and alternative modeling several approaches have been proposed (e.g. [8–12]). In [8] the *Common Variability Language* (CVL), a generic language for modeling variability in different domain models is presented. In [9] feature models are introduced for modeling software product line variants.

Feature models are also used in [10] to describe variants. They are combined with an XML-based language (XVCL) to add much more flexibility to the feature models. The approach presented in [11] allows to model variants of industrial automation systems within a product line. Another approach for variant handling is described in AUTOSAR 4.0 [12, 13]. In [14] the authors give a good overview on the current state of the art in platform-based product design and development which contain product variant management approaches.

The main weaknesses of these approaches are that they are mostly used for software or product design and have no capability to specify platforms and especially their variants. Furthermore the generation of variants is not supported and they even provide very limited methods to specify complex interdependencies between different components and variants.

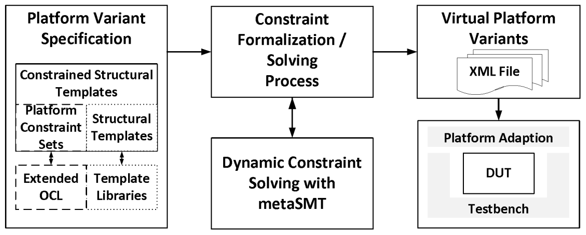
In the area of stimuli generation for test and simulation of platforms via SMT/SAT solving, several approaches have been proposed. For instance, in [15] the SystemC Verification (SCV) [16] library has been extended for stimulus generation based on SMT. The authors in [17] present a sampling algorithm combining concepts from the Metropolis-Hastings algorithm, Gibbs sampling and WalkSAT to generate solutions with an approximately uniform distribution. In both approaches the generated solutions for the constraints are stimuli parameter that can be classified on a lower system level in comparison to our approach.

In the verification and validation of large design components constraint-based random simulation remains an integral part. In [18] entropy metrics to define coverage targets of internal signals is combined with a framework which uses small randomized XOR constraints to generate stimuli for inadequately stimulated circuit regions. The authors in [19] present a methodology to analyze contradictory constraints that occur in constraint-based random simulation. Actually all of the approaches in constraint-based random simulation are used to generate test and stimuli for fixed platforms. In contradiction to our novel methodology which lifted up verification, exploration and test to variable virtual prototype platforms.

The authors in [20] describe how OCL constraints and UML models can be encoded in a SAT instance. In comparison, we support the encoding of more OCL features (e.g. if-expressions, allInstances() operation, extended OCL operators). In summary, to the best of our knowledge, no model- and constraint-based generation of platform variants has been considered so far.

III. Platform Variants Specification and Generation

This section introduces our platform variants specification approach based on constraints and structural templates. Before we describe our approach in detail we give an overview. The elaborated framework, presented in Fig. 1, is divided in three parts: the template-based platform variants specification, the automated encoding of template-attached



Platform Variant Specification
Constrained Structural Templates
Platform Constraint Sets
Structural Templates
Extended OCL
Template Libraries
Constraint Formalization / Solving Process
Dynamic Constraint Solving with metaSMT
Virtual Platform Variants
XML File
Platform Adaption
DUT
Testbench

Fig. 1. Overview of the platform variants generation

constraints including the SMT/SAT-solving process and the virtual prototype simulation. The platform variants specification contains the platform templates based on UML, see Subsection A, as well as the attached constraint sets. The UML platform templates are structured to three libraries. Within these libraries the templates are structured hierarchically by composition structure diagrams. Hence a hierarchical structure for every component of the libraries can be built. So every component can be abstracted as complex or detailed as required.

The user-defined constraint sets are attached at the different hierarchical template layers to describe the variability, the dependencies and the configuration capabilities of different platform variants. Currently we attach these constraints by using UML standard constraints [21]. The constraint sets in combination with the structural templates allow a precise specification of complex dependencies among platform templates. For example more restrictive constraint sets enable an effective reduction of the huge variant space to the relevant platforms for verification.

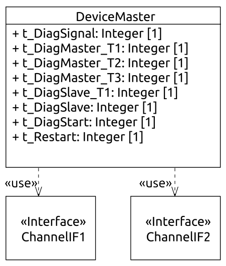
We encode the constraint sets to a SAT instance using metaSMT. The constraint solutions are derived for generating valid and feasible platform variants. These platform variants are described as XML. This enables to link the established tools for platform modeling with our platform variant approach to specify, generate and simulate platform variants. We pass the XML to our platform execution tool which registers instantiates and links the modules of the platform variant automatically [22].

A. UML Platform Templates

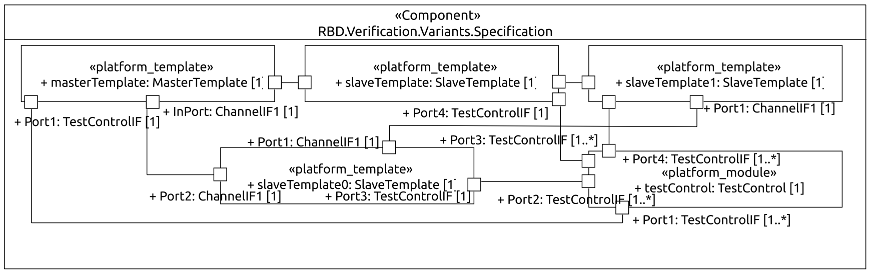
Within this section we present the UML-based structural platform templates and a small tailored UML platform profile. This profile and the platform templates form the basis for the structural specification of different variants of virtual prototype platforms (e.g. SystemC platforms). The UML platform profile consists of four relevant stereotypes:

1. <*platform_module*> is a virtual prototype module described by an instance of a class definition.
2. <*platform_template*> is a component which abstracts a part of a system for simplicity, variability or structural reasons.
3. <*platform_or_template*> is a component which contains more than one not related platform templates or modules to realize possible alternatives.
4. <*platform_constraint*> marks a template/module attached constraint.

Beside this UML profile, the UML standard profile for Primitive Types [5] is used for data types (String, Float, etc.).


DeviceMaster
+ t_DiagSignal: Integer [1]
+ t_DiagMaster_T1: Integer [1]
+ t_DiagMaster_T2: Integer [1]
+ t_DiagMaster_T3: Integer [1]
+ t_DiagSlave_T1: Integer [1]
+ t_DiagSlave: Integer [1]
+ t_DiagStart: Integer [1]
+ t_Restart: Integer [1]
«use»
«use»
«Interface» ChannelIF1
«Interface» ChannelIF2

(a) Platform module definition of a MOST DeviceMaster with timer attributes.


«Component»
RBD.Verification.Variants.Specification
«platform_template»
+ masterTemplate: MasterTemplate [1]
«platform_template»
+ slaveTemplate: SlaveTemplate [1]
«platform_template»
+ slaveTemplate1: SlaveTemplate [1]
+ InPort: ChannelIF1 [1]
+ Port1: TestControlIF [1]
+ Port4: TestControlIF [1]
+ Port1: ChannelIF1 [1]
+ Port1: ChannelIF1 [1]
+ Port3: TestControlIF [1..*]
«platform_template»
+ slaveTemplate0: SlaveTemplate [1]
+ Port2: ChannelIF1 [1]
+ Port3: TestControlIF [1]
+ Port4: TestControlIF [1..*]
«platform_module»
+ testControl: TestControl [1]
+ Port2: TestControlIF [1..*]
+ Port1: TestControlIF [1..*]

(b) Platform variant specification describing all relevant variants for the MOST Ring Break Diagnosis (RBD) process.

Fig. 2. Platform module DeviceMaster specifing a MOST master node and its timer parameters as well as the used and required interfaces. Platform variant specification defines different MOST ring variants for the verification of the Ring Break Diagnosis (RBD).

As mentioned before, the templates and modules of the platform variants specification are structured to three libraries. This well-established library approach provides clarity, supports reusability as well as reintegration of specified platform templates/modules in new platform variants specifications. The first library contains the platform modules representing virtual prototype modules (e.g. SystemC modules). Every virtual prototype module is described by a platform module definition, see Fig. 2a. Within this definition the module parameters are specified. Every parameter must have a data type and has to be marked with *public (+)*, *private (-)* or *protected (#)*. In the platform variants specification only parameters which are specified as *public (+)*, are allowed to be configured. Optionally a composition structure diagram is associated with the platform module to define a hierarchical structure.

The second element type of the first library is the platform template (<*platform_template*>). A platform template abstracts a part of a system for simplicity, variability or structural reasons. It consists of several platform modules and templates which are described in a composition structure diagram, see Fig. 2b. The last type of platform elements is the or-template (<*platform_or_template*>). They are integrated in the platform variants specification representing alternatives for specific platform components. The or-templates consist of two or more not related platform templates or modules. A valid platform variant contains always one element of them.

In the second library, interfaces are defined to guarantee a correct linking between templates and modules. Fig. 2a demonstrates the annotation of interfaces to a module definition. The linking requirements will be explained in detail in Section C. The last library contains enumerations which enable the specification of user-defined data types.

B. Platform OCL Constraints

This section introduces the constraint sets which are used to define variability, configuration capabilities and dependencies among platform elements. The constraint sets are attached to the corresponding platform modules/templates and are specified in OCL. Commonly OCL is used to define constraints at the M1 layer of the *Meta Object Facility (MOF)* Standard [23]. Hence concrete instances, respectively distinctive data (M0), of the M1 models can be verified by these constraints. Table I presents the deviation of our Platform Meta Layers to the standard MOF layers. The M3 layer is represented by the UML 2.4.1 Meta model. The elements and concepts of the UML-Meta Model are derived to build the platform templates and the platform profile. These two platform concepts build the M2 layer in our approach. The M1 layer is formed by the platform variants specification itself with the attached OCL constraints. The OCL constraints combined with the platform variants specification span the M0 layer which consists of the valid platform variant space. We deploy a subset of OCL to specify the platform constraints. We build this particular subset by adding new operators to define platform constraints more accurately. In the following the subset will be called *Platform-OCL* (P-OCL). P-OCL supports several OCL standard types and their operators like: Collection-Related Types (*Sequence, Bag*), Collection-Related Operators/Operations (*.,->, includes, size, ...*), Primitive Types (*Integer, Real, Boolean*) or Object Model Types (*Class, Attribute*).

TABLE I
MOF LAYERS

Standard MOF Layers		Platform Meta Layers
M3	Meta Meta Model	UML-Meta Model
M2	UML-Meta Model	Platform Templates/Profile
M1	User-defined UML-/Object-models	Platform Variants Specification
M0	Distinctive Data	Platform Variant Space

At first we present an OCL extension which supports probability distributions for the OCL Collection-Type *Sequence*. We implemented the *gaussian* and the *invGaussian* probability operator because these distributions are often used in validation and test. Definition 1 introduces the gaussian probability distribution for *Sequences*.

Definition 1 (P-OCL Gaussian Sequence-Operator (**gaussian()**)). Let $S = Sequence\{min..max\}$, $O = S.gaussian()\text{-}{>}\ includes(Class|Attribute)$ the OCL Expression and R is the result set of O then follows: $\forall\ X_i \in R$ and X_i are continuous random variables with $|R| = 0 \leq i < (max - min)$ because duplicates are removed from R. These random variables correspond a gaussian probability distribution with the density function $f(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$ with $\mu = \frac{max-min}{2}$ and $\sigma = \frac{\mu+min}{3}$. Thereby μ defines the mean and σ represents the standard deviation.

Additionally to the gaussian probability distribution operator an inverse gaussian probability distribution operator is added.

Definition 2 (P-OCL Inverse-Gaussian Sequence-Operator (**invGaussian()**)). Let $S = Sequence\{min..max\}$, $O = S.invGaussian()\text{-}{>}\ includes(Class|Attribute)$ the OCL Expression and R is the result set of O then follows: $\forall\ X_i \in R$ and X_i are continuous random variables with $|R| = 0 \leq i < (max - min)$ because duplicates are removed from

R. These random variables correspond an inverse-gaussian probability distribution with the density function $f(x) = \begin{cases} (\frac{\lambda}{2\pi x^3})^{\frac{1}{2}} e^{-\frac{\lambda(x-\mu)^2}{2\mu^2 x}} & x > 0 \\ 0 & x \leq 0 \end{cases}$ with $\mu = \frac{max-min}{2}$ and $\lambda > 0$. Thereby μ defines the mean and λ represents the shape parameter.

Both operators enable to specify critical boundary conditions of attribute value configurations or critical numbers of platform templates/modules. Likewise they can be used to reduce the feasible number of valid variants.
The next extension, the *active* operator, supports the constraining of or-templates.

Definition 3 (P-OCL active-Operator (**active()**)**.** Let $R_i = Class|Component$ with $i = 0, ..., n$ and $n \in \mathbb{N}$, $OT = or - template$, $R_i \in OT$ and $O = R_1.active()$ a P-OCL expression. Then follows for the resulting variants V: R_1 is integrated in all variants V.

The *active* operator defines which template or module of an or-template should be integrated in the generated platform variants. This enables to define constraint sets which integrate different platform templates/modules in valid solutions.
The following rules show the formalization of different types of P-OCL constraints in SMT/SAT expressions. Rule 1 formalizes a P-OCL *Sequence* expression. This expression optionally allows to define a step size, e.g. to reduce huge parameter ranges. Therefore the step size functionality, which is not contained in OCL Collection-Types standardly, is provided semantically by an iterator definition. This iterator definition is specified by the OCL operation *select*. In the formalization process integer values are represented as metaSMT bit-vectors. Subsequently bit-vectors are represented in rules as vectors (e.g. $\vec{b}$). In the following a natural number is converted to a bit-vector and vice versa by *conv* and $conv^{-1}$.

Rule 1. Let *rmin, rmax, stepsize* $\in \mathbb{N}$, then follows $\vec{a} \equiv conv^{-1}(rmin)$, $\vec{b} \equiv conv^{-1}(rmax)$ *and* $\vec{s} \equiv conv^{-1}(stepsize)$. The current bitwidth is defined by *bw* $\in \mathbb{N}$. The *OCL Attribute/Class Reference* is mapped to *y*. Then a formalization of a P-OCL Sequence definition is defined as follows: *Sequence{rmin..rmax}->select(e : Integer | e / stepsize = 0)->includes(Attribute/Class)*,
is mapped to: $y \geq \vec{a}$ & $y \leq \vec{b}$ & $(y$ - $\vec{a})$ % $\vec{s} == \vec{0}$

The first step of the formalization is to represent the *Attribute/Class-Reference* by a variable, here *y*. This reference-to-variable mapping has to be unique to allocate the different *Sequences* distinctly. In metaSMT, bit-vectors are represented by function *bv_uint()[bw]* with bit-width *bw*. Rule 2 demonstrates the formalization of a list definition.

Rule 2. Let be *Class/Attribute-Reference* x and n $\in \mathbb{N}$, then: *Bag{*b_0*, ...,* b_n*}->includes(Attribute/Class)* is mapped to $x < \vec{m}$, with $\vec{m} \equiv conv^{-1}(n+1)$.

A list definition in P-OCL can contain values of different Primitive Types. These lists are specified by the OCL standard type *Bags*. In order to represent a list definition in metaSMT, the list index values are formalized as bit-vectors to assign each list entry a unique index. Regarding to Rule 2 it is obvious that the variable, respectively the Class/Attribute reference, has to be smaller than the highest index number plus one. This rule is also adopted for floating point value sequences. Due to the standard definition of *Sequences* in OCL it is necessary to define floating point *Sequences* with explicit values. These results from the fact that a floating point *Sequence* interval specification is be determined infinite. One last example for the SMT/SAT formalization is given in Rule 3. It shows the formalization of an if-expression. The if-expression, subsequently described by if-condition (c), then-branch (t) and else-branch (e), is defined semantically equivalent to the Boolean formula: *(c **implies** t) **and** (**not** c **implies** e)*.

Rule 3. Let c, t and e be P-OCL constraints and the function *form* is defined by $form(\lambda) \in$ Boolean expression, with $\lambda \in$ P-OCL. Then the formalization of a P-OCL if-expression is defined by: if c then t else e is mapped to *logic_ite(form(c), form(t), form(e))*

C. Template Linking

This structural Section demonstrates the versatility in defining different system topologies, e.g. ring-, bus- or mesh-topologies, with *CST*, linking requirements and port cardinalities. The Template Linking takes place, once a solution is determined by the SMT/SAT solving process. Then the number of newly generated platform module/template instances has to be derived from the solution. Afterwards they are connected due to specific linking requirements. The linking requirements are based on templates, port cardinalities, interfaces and connectors.

1. Every newly generated platform module/template instance can be linked to their target platform modules/templates as long as target port instances are left. The number of possible port instances is defined by cardinality tags annotated to the port definitions, see Fig. 2b or 3.
2. If no target port instance is left the newly generated platform module/template instance will be connected to the previously generated instance of the same type, see the ring topology example in Fig. 3.
3. Platform modules/templates which are specified variably by constraints are only allowed to connect to non-variable platform modules/templates.
4. If two platform modules/templates are connected to each other and both are specified variably by constraints it is necessary to abstract them by a *CST*.

Fig. 3 demonstrates the linking process for two examples regarding to the above defined linking requirements. In both cases template *T1* has to be integrated three times in the final platform variant. This can be enforced by the constraint *Sequence{1..3}->includes(self.allInstances()->size())* whereby *self* refers to *T1*. *S* and *S1* are the target platform modules/templates to which *T1* is connected to. In the ring topology example has the target port instance of connector *C2* the cardinality of *[1]*. Therefore every newly generated instance of *T1* has to be connected to the next newly generated one. The bus topology is generated because the cardinality of the target port at *S* of *C1* is unlimited *[1..*]*. Hence every newly generated instance of *T1* can be connected to *S*. The given examples in this section are synthetic and should only demonstrate the definition of different system topologies by using *CSTs*, linking requirements and port cardinalities. Due

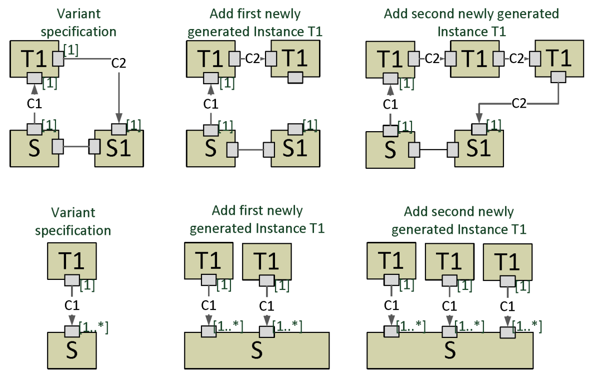

Variant specification
Add first newly generated Instance T1
Add second newly generated Instance T1
T1
S
S1
C1
C2
[1]
[1..*]

Fig. 3. Linking process for two platform variants specifications regarding to defined linking requirements.

to the limited space in this paper the example is visualized in an abstract way.

IV. Experimental Results

This section discusses our experimental results from platform variants specification for the verification of an industrial MOST application and for the exploration of a FlexRay traffic sign recognition example. All experiments have been carried out on an Intel Core 2 Quad 2.50 GHz with 4 GB of RAM using Ubuntu 12.04 64bit.

A. Ring Break Diagnosis Application (RBD)

We demonstrate the flexibility, changeability and necessity of our methodology for the verification of virtual prototypes variants by means of a MOST Ring Break Diagnosis [24] verification example. Ring Break Diagnosis is classified within ISO-OSI Data Link Layer of the MOST network interface section (NetInterface). It serves the purpose of localizing a fatal error in the network. The RBD process can be started by various triggers, which must be chosen and implemented by the System Integrator [24]. In our MOST model we implemented all different MOST layers as well as the entire functionality of RBD regarding to the state chart description of the algorithm given by the MOST specification.
In order to verify the RBD algorithm reasonably it is necessary to specify a feasible subspace of the $54*10^{21}$ valid MOST platform variants. In [22] six different verification scenarios turned out to be suggestive: *error free, ring break, excessive attenuation, multi master, all slave* and *combination.* Every scenario is specified by a different number of templates and constraints. Table II summarizes the generated variants, used templates and specified constraints for each scenario.
The number of constraints and templates remained fairly

TABLE II
Generated variants for RBD

Scenario	Variants	Templates	Constraints
error free	25133	8	38
ring break	24478	8	37
excessive attenuation	24564	8	37
multi master	25231	8	37
all slave	25117	7	29
combination	24756	8	38

constant because often one constraint or template limits the boundary to another scenario. This fast reconfiguration capability is demonstrated by changing the *multi master* to the *all slave* scenario. Therefore it is just necessary to remove the MasterTemplate from the variant specification and to relink the remaining SlaveTemplates.
In Fig. 2b the top level description of the ring break variant specification scenario is displayed. For the ring break scenario we use 8 templates and 37 constraints. Four of the 8 templates are displayed in the top level description. Each of these four templates contain one or-template and one MOST Device node. Each of the four top level templates contain a minimum of 8 constraints to specify the different timer of MOST masters or slaves as well as to trigger which channel should be used. Listing 1 shows some of these constraints. The if-constraint in line 11 to 18 take care that only one ring

```
1  -- These Sequence Constraints are attached
2  -- at MasterTemplate and slaveTemplate1.
3  Sequence{1..63}->includes(self);
4  Sequence{1..63}->includes(self);
5  slaveTemplate.allInstances()->size() +
6    slaveTemplate0.allInstances()->size() +
7    MasterTemplate.allInstances()->size() <= 63
8  -- Constraints defined within MasterTemplate
9  Sequence{2000..2500}.invGaussian()->
10   includes(self.t_Config)
11 if self.deviceChannelRB.active()
12   then SlaveTemplate.allInstances->
13     forAll(e | !e.deviceChannel.active())
14   else if slaveTemplate0.deviceChannel.active()
15   then self.allInstances
16     ->forAll(m | !m.deviceChannel.active())
17   endif
18 endif
```

Listing 1 Small subset of ring break scenario constraints

break is injected in a final platform variant. This is necessary because the RBD algorithm is designed to detect and identify the position of one ring break. For every scenario we generate approximately 25000 variant instances.
Altogether we generated for the six scenarios over 150000 variants. Thereby the solving and generation process has been done in about 2 hours. In order to ensure a suggestive verification regarding to RBD, we configured the maximum simulation time to 3500 msec. The parallelized implementation of our platform variants specification framework enables to process simulations parallel. Therefore the entire solving, generation and simulation process can be done in a few hours in case of sufficient resources. Owing to our limited resources the simulation process needs up to 5 days.

B. Traffic Sign Recognition Example

In our second example we demonstrate the versatility of our novel constraint-based platform variants specification approach. We applied our approach for an exploration use case within a heterogeneous system consisting of virtual prototype modules and modules containing hardware target code. We explored different camera modules for an automotive traffic sign recognition (TSR) platform and different hardware parallelization options. The TSR platform is implemented by a FlexRay bus system.
The top level of the variants specification of the system is defined by four structural platform templates which encapsulate the functionality of the TSR. These four templates include: the camera, the circle detection application, the traffic sign classification and the display. Each of these four templates abstracts up to 25 platform modules which provide the functionality of the camera, the circle detection or the classification. The camera exploration constraints in Listing 2 specify different camera types by display resolution,

grayscale- or colored-camera and scale factor. The second constraint set in Listing 3 specifies the different hardware parallelization options for the circle detection. The circle detection is implemented in target code for a Tilera board to explore these options on the board. The options enable to

```
Bag{320,480,576}->includes(self.roiHeight);
Bag{480,640,720}->includes(self.roiWidth);
Bag{1,3}->includes(self.unBytesPerPixel);
Bag{0.3, 0.6, 0.1}->includes(self.scale);
```

Listing 2 Camera exploration constraints

trigger a number of threads on the board which provides up to 53 cores. Thereby the circle detection module is started

```
Sequence{1..53}.includes(self.noOfCores);
Sequence{5..10}.includes(self.minRadius);
Sequence{75..80}.includes(self.maxRadius);
```

Listing 3 Exploration constraints for parallization

on a certain number of cores with a calculated sub range corresponding to a given minRadius and maxRadius. Hence different circle radii can be detecting parallel.

Due to the complete variants generation process of our approach, 71150 valid platform variants are generated. They are evaluated with regard to the frame rate and the number of recognized traffic signs. The evaluation video stream contains 6 traffic signs.

The results showed that only one camera module recognized all 6 traffic signs correctly and has no classification errors. It is defined by the parameter set: *(320, 640, 1, 0.5)* (*roiHeight, roiWidth, unBytesPerPixel, scale*). The best results for the parallelization are: *minRadius = 5*, *maxRadius = 80* and *noOfCores = 20*. Within this example we highlighted the versatile and the flexible usage of our variants specification approach for heterogenous systems.

V. Conclusion

In this paper we proposed a novel constraint-based specification approach to enable variability for virtual prototype platforms. Applying this methodology for verification, exploration and test of virtual prototype platforms enables to specify and generate precisely, plausibly and comprehensibly valid platform variants. The usability, applicability and flexibility of our approach are demonstrated for the verification of an industrial MOST application and the exploration of a traffic sign recognition application on a FlexRay bus. Likewise the MOST variant space consists of more than $54 * 10^{21}$ valid variants which points out the scalability regarding to huge variant spaces. This huge variant space can be reduced efficiently to feasible and required platform variants by applying our methodology. Next steps are a model-to-model mapping to import IP-XACT descriptions and to integrate a model-based design flow for reliability assessment of safety-relevant automotive systems [25].

VI. Acknowledgements

This work has been funded by the German Federal Ministry of Education and Research (BMBF) within project 16M3195E.

References

[1] W. R. Group. (2010) The 2010 Wilson Research Group Functional Verification Study. Wilson Research Group. [Online]. Available: http://goo.gl/rIyYu

[2] P. Montag and S. Altmeyer, "Precise WCET calculation in highly variant real-time systems," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, march 2011, pp. 1 –6.

[3] Altran GmbH. FlexRay Consortium. Altran GmbH. [Online]. Available: http://www.flexray.com/

[4] M. Cooperation, *MOST Specification*, Most Cooperation Specification, Rev. 3.0 E2, 07 2010. [Online]. Available: http://goo.gl/nRqj2

[5] OMG. (2011, August) Unified Modeling Language. Object Management Group. [Online]. Available: http://www.uml.org/#UML2.0

[6] ——. (2012, January) Object Constraint Language (OCL). ISO Release. [Online]. Available: http://www.omg.org/spec/OCL/ISO/19507/PDF

[7] F. Haedicke, S. Frehse, G. Fey, D. Große, and R. Drechsler, "metaSMT: Focus On Your Application Not On Solver Integration," in *Program Proceedings of the 1st International Workshop on Desigen and Implementation of Formal Tools and Systems (Austin, TX/USA)*, M. K. Ganai and A. Biere, Eds., 2011.

[8] OMG. (2009, March) Common Variability Language. Object Management Group. [Online]. Available: http://goo.gl/yPW7j

[9] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature Oriented Domain Analysis (FODA) Feasibility Study," nov 1990.

[10] S. Jarzabek and H. Zhang, "XML-based method and tool for handling variant requirements in domain models," in *Requirements Engineering, 2001. Proceedings. Fifth IEEE Int. Symposium on*, 2001, pp. 166 –173.

[11] C. Maga and N. Jazdi, "An approach for modeling variants of industrial automation systems," in *Automation Quality and Testing Robotics (AQTR), 2010 IEEE Int. Conference on*, vol. 1, may 2010, pp. 1 –6.

[12] AUTOSAR. (2012) About. AUTOSAR. [Online]. Available: http://goo.gl/KO4cq

[13] ——. (2011, 10) Generic Structure Template. AUTOSAR. [Online]. Available: http://goo.gl/ccByb

[14] X. F. Zha and R. D. Sriram, "Platform-based product design and development: A knowledge-intensive support approach," *Know.-Based Syst.*, vol. 19, no. 7, pp. 524–543, Nov. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.knosys.2006.04.004

[15] R. Wille, D. Große, F. Haedicke, and R. Drechsler, "SMT-based stimuli generation in the SystemC Verification library," in *Specification Design Languages, 2009. FDL 2009. Forum on*, sept. 2009, pp. 1 –6.

[16] IEEE, "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, vol. 1, pp. 1 –638, 9 2012.

[17] N. Kitchen and A. Kuehlmann, "Stimulus generation for constrained random simulation," in *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM Int. Conference on*, nov. 2007, pp. 258 –265.

[18] S. Plaza, I. Markov, and V. Bertacco, "Random Stimulus Generation using Entropy and XOR Constraints," in *Design, Automation and Test in Europe, 2008. DATE '08*, march 2008, pp. 664 –669.

[19] D. Grosse, R. Wille, R. Siegmund, and R. Drechsler, "Contradiction analysis for constraint-based random simulation," in *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, sept. 2008, pp. 130 –135.

[20] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler, "Verifying UML/OCL models using Boolean satisfiability," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, March, pp. 1341–1344.

[21] OMG, *Constraints Package*. OMG, August 2011, vol. 2.4.1, ch. 9.6, pp. 40–43. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/Infrastructure

[22] A. Braun, O. Bringmann, D. Lettnin, and W. Rosenstiel, "Simulation-based verification of the most netinterface specification revision 3.0," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, march 2010, pp. 538 –543.

[23] OMG. OMG Meta Object Facility (MOF) Core Specification. OMG. [Online]. Available: http://www.omg.org/spec/MOF/2.4.1/PDF/

[24] MOST, *MOST Media Oriented Systems Transport Multimedia and Control Networking Technology*, MOST Cooperation Std., Rev. 3.0, 05 2008.

[25] S. Reiter, M. Pressler, A. Viehl, O. Bringmann, and W. Rosenstiel, "Reliability assessment of safety-relevant automotive systems in a model-based design flow," in *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, 2013, pp. 417–422.