Transparent SystemC Model Factory for Scripting Languages

Rolf Meyer, Bastian Farkas, Syed Abbas Ali Shah, Mladen Berekovic
E.I.S., TU Braunschweig, D-38106 Braunschweig, Germany
Email: {farkas, meyer, shar, berekovic}@c3e.cs.tu-bs.de

Abstract

In this work we present a modern way to implement a model factory inside a SystemC simulation framework. We make use of a scripting language to interact with the simulation framework and dynamically load and instantiate models from external libraries. We demonstrate our approach with the SoCRocket virtual platform framework, which already comes with a generous amount of models. These models don't need to be modified in any way. With our factory and registry solution it is possible to describe a whole platform configuration in a natural scripting language syntax, while using existing models.

I. Introduction

The idea behind electronic system level (ESL) design is to further accelerate the design process. Hence, the trend for SystemC simulations moves towards runtime configurable and adaptable or even runtime intermateable models. Technologies for configuration, control and introspection gain more and more importance as they are finding their way into industry standards allowing better runtime reconfigurability. For the same reasons we have observed an increasing interest in model registries and factories. We present a solution for intermateability, which can be utilized from any scripting language, to extend their usability. Any model can be added without recompilation to a virtual platform. This is especially useful for external interface models like UART or Ethernet since the number of models needed in the platform may vary with the application. Moreover, the implementation is tested in multiple simulators: Accellera SystemC, Mentor Graphics QuestaSim, Cadence NCSim.

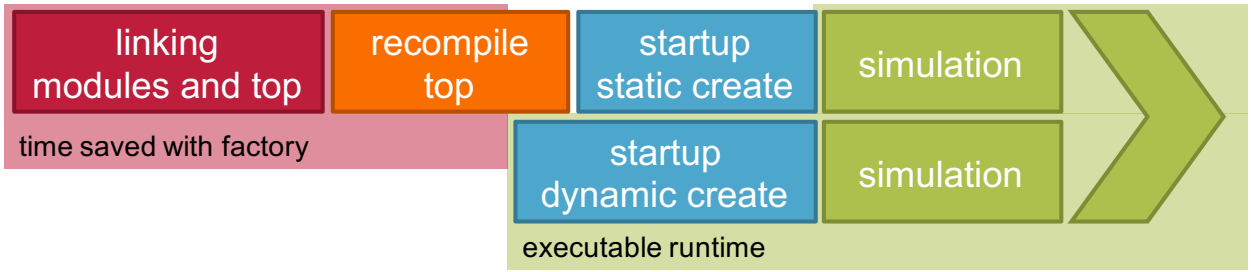

Fig. 1: Time comparison to reach the start of the simulation for static creation and dynamic creation. [1]

Other approaches restrict their solutions to be a pure registry with a factory for SystemC models and use it with their own instantiation language.[1] This approach deminishes the benefit gained from reducing recompilation time by imposing the designer to learn a new domain specific language. Our solution integrates in an existing scripting language the designer might already know. We will use Python in this example, but our approach is not limited to Python. Other implementations are possible as well. We have in fact working prototypes in Python, TCL, Perl and Ruby. For a full integration in a common scripting language not only a registry and factory is needed. We also need the ability to do type checking at runtime. This allows the scripting language to fully integrate the factory with its models to mimic a natural language user experience.

The remainder of the paper is structured in the following way: Other uses of factories will be shown in the related work section. A short introduction in the foundation of this work follows in the subsequent section. The implementation is divided in four subsections. First we explain the access of SystemC objects from a scripting language. Second we explaing in detail the implementation of the factory itself. Then we describe our natural language interface in Python. Finally we introduce an interface to load dynamic linked libraries in the last subsection. An application demonstrates the usage of our framework in the application section. Finally we conclude the paper.

II. Related Work

The topic of object factories does not come up often in research. It has been introduced in the famous "Gang of Four" book about design patterns [2]. Since then it is difficult to evaluate how many projects actually apply the factory pattern. An obvious application in the recent past have been engines and frameworks for game programming. Although it is quite difficult to get an insight since most frameworks are closed source.

The market for engines consolidates around a handful of key players and each one wants to offer the developers an easy interface to realize their visions without the need to be low level programming wizards. The natural way to do this is to provide the users (in this case game developers) with an application programming interface (API) which is easily applicable in their favorite language. Another approach is to include scripting capabilities directly in the engine. This can even be done visually instead of textually [3]. Modern games can be seen as highly complex simulation systems, much like system level design frameworks. It makes therefore a lot of sense to look into advances in that domain.

The authors of [1] describe a reconfigurable simulation framework which uses the factory pattern, but the focus there is clearly on model reconfiguration and not dynamic model loading and instantiation. The authors rely on their own description language for the platform configuration instead of a common scripting language. This fact makes the approach unattractive for developers who already have to be profiscient in several languages. Furthermore they describe a lack of lightweight, open and easy to use solutions for SystemC virtual platforms. We adress all three of these points with our factory and registry implementations.

A good overview of the related work can be found in section two of [1].

Accellera is still in the process of creating a common configuration control and introspection (CCI) standard [4]. Previous work regarding configuration mechanisms is available from Greensocs [5] which is finding its way into the configuration, control and inspection (CCI) standard.

III. Foundation

In [6] we have presented our scriptable reporting and logging framework which is the foundation for the work presented here. Our framework combines the two most powerful reporting and logging approaches for SystemC/TLM2 applications. The first approach comprises a reporting framework including a scriptable report processing back end. The default reporting tools of SystemC were extended by a comfortable scripting interface and support for smart handling of key/value pairs attached to reports. The scripting interface relies on the Python scripting language and its vast amount of available libraries. With the introduction of efficient black- and whitelisting mechanisms, the impact on simulation time can be reduced to a minimum while still benefiting from the capable scripting interface. Further speedup is expected with introduction of multi-threading support, which will be available in the near future.

The second approach combines introspection and reflection for SystemC/TLM2 applications with a comprehensive scripting library (called USI, Universal Scripting Interface). We have shown that the presented framework works well with available base-classes like `sc_object` and Cadence's `scireg` and is compatible with simulators like OSCI SystemC and MentorGraphic's Questasim. Furthermore, we have shown that our presented API greatly eases integrating third-party C++ APIs and therefore is prepared to support future standards as well as proprietary solutions with minimal efforts.

IV. Implementation

The implementation is based upon our universal scripting interface (USI), which we presented in [7] and [8]. USI enables a scripting language to interact with a SystemC simulation to fulfill sophisticated control and analysis tasks by abstracting functionality over abstract C++ interfaces. By providing a registry for third party APIs it is possible to access model internals without recompilation or preparation of the models for this purpose. In addition to fulfilling the needs of a hierarchical structure, USI is extended by a sc_module class allowing the designer to construct simple container models within the scripting language. It is possible to create a wrapper around this interface to mimic language natural style and behavior with minimal effort.

A. Universal Scripting Interface (USI)

In the simplest case, a `sc_object` is dynamically cast to an interface, for example `sc_object` or `AHBDevice` Interfaces. The cast will create a new SWIG proxy object if the interface is implemented for the specific `sc_object`, otherwise `NULL` is returned. The registration process of these simple interfaces is described in the next section.

While this technique is sufficient in most cases, some external utilities require a more complex approach. These utilities may rely on external data storages which need to be queried. To incorporate such utilities, our solution allows the integration of a custom generator function (see Listing 2, line 3). It can cope with any kind of function, creating a proxy object in the scripting language from either a `sc_object` or a hierarchical path. To demonstrate the API, the `scireg` base-class integration is explained below.

```
1 def __getattr__(self, name):
2     result = None
3     for iface in self.get_if_tuple():
4         result = getattr(iface, name, None)
5         if result: return result
6     super(InterfaceDelegate,
7       self).__getattr__(name)
8
9 def __dir__(self):
10    result = set()
11    for iface in self.get_if_tuple():
12        result.update(dir(iface))
13    return sorted(result)
```

Listing 1: Python implementation of the interface delegation

```
1 USI_OBJECT
2 USI_INIT_MODULES()
3 USI_REGISTER_OBJECT_GENERATOR(funct)
4 USI_REGISTER_OBJECT(type)
```

Listing 2: Plugin API

After construction, `USIDelegate` can be used as a normal Python object. Function calls are transparently delegated to the collected proxies as shown in Figure 2. All available SWIG or language proxies for an identified SystemC object are stored within the `USIDelegate` object. The functionality provided by the Python part of the implementation is shown in Listing 1. It simply selects the proxy object implementing the called function via the `__getattr__` method and executes it. The `__dir__` method is used to further improve the usability by enabling command completion in the interactive shell. The method `get_if_tuple` returns a tuple of valid SWIG proxy objects for the interfaces from plugins implemented on the corresponding `sc_object`. This method is implemented directly in C++, unifying the delegation and the SWIG proxy objects.

Altogether the USI enables access to SystemC models via registered utility APIs. This allows a scripting language to integrate models like native objects. But it does not introduce differentiated type for the instances or a way to construct new objects. A factory as described below is the natural extension.

B. Factory for scripting languages

USI provides us with the capability to handle SystemC object through utility APIs like native instances but how do we get these instances. In our work we created these objects in a C++ `main`-function and use the `usi.find`-function to get a reference into the scripting environment. With a model registry and factory we enable the scripting language to create these instances as well and make the C++ `main`-function obsolete. In addition to a registry for `constructor`-functions a scripting languages might need other functions. For example emulating native instance/class-behaviour requires a function to check whether a instance is implementing a class. This way an implementation can use the function to emulate the type system. Moreover a filename store the registry builds the foundation to load scripting files stored next to the C++ implementation of the model. Both in conjunction allow extending virtual classes of the models in the target scripting language with service functions. For example a structure analysis function can be added to a bus controller. Using the USI AMBA API the bus controller can iterate over all connected masters and slaves and print an online memory map. To instantiate SystemC models in a scripting language the first step is to have a simple factory creating models and returning corresponding instances. This can be easiely archived by extending the USI methology by a single function `api.create_object_by_name`. It takes group, class and instance name to create a new SystemC model. But to make this idiom more useful for the designer it is important that the creation of a model feels like the creation of a native scripting object. This way the designer uses the API as any other library of the used scripting language and does not need to learn any new details about the connecting bridge between SystemC and the scripting language. This requires more work in creating the

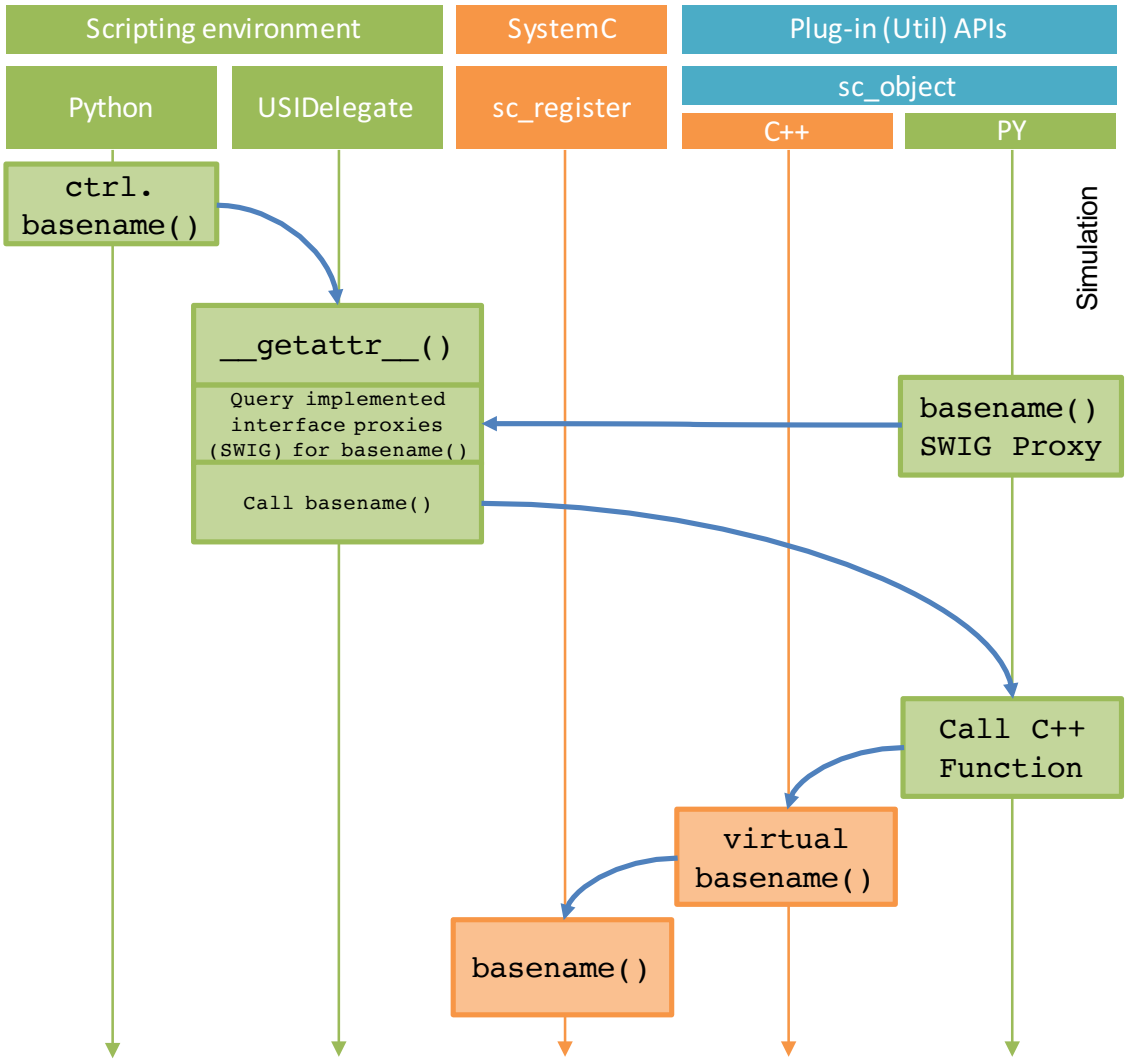

Fig. 2: The structure of an USI function lookup

bridge but simplifies the usage for the designer so that he needs to only care for the important details. Below we will explain this in further detail with a Python API as example.

C. Natural Python interface

A model factory is a natural extension to USI, but first of all the factory should feel like a normal Python Module. It must simply be loaded by `import registry`. All included modules and categories are dynamically loaded from SystemC. Which means the import statement needs to return a dynamic module. That dynamic module needs to return the modules. Secondly we need to return an abstract base class with customized instantiation function to return not an instance of that class but execute the factory function from SystemC. In Python that means it needs to return a class for a specific meta class (Listing 4) containing an overloaded `__new__`-function 3. This meta class then allows to create SystemC Object like real Python objects when instantiating a object from a class with this metaclass attached to it class. A metaclass is basically the type of the class and oversees the construction of its classes. In Python this meta class has also a function to prove if an instance is of a certain class. This function gets hooked up to the factory `is_type`-function as in Listing 4.

For a good scripting API it is important to not only implement the functionality itself but also write the introspection functions. That enables the developer to check at every point in the API what functions are available. This is enabled in Python by the `__dir__`-functions as in Listing 5

```
def delegate_new(cls, instance, *args, **kw):
    obj = api.create_object_by_name(cls.__usi_group__, cls.__usi_class__, instance)
    if hasattr(obj, 'generics'):
        generics = getattr(obj, 'generics')
        for key, val in kw.items():
            param = generics
            path = key.split("__")
            try:
                for part in path:
                    param = getattr(param, part)
                param.cci_write(str(val))
            except AttributeError as e:
                ei = sys.exc_info()
                raise AttributeError("USIDelegate␣'%s',␣'%s'␣has␣no␣generic␣'%s'" % (
                    cls.__name__, instance, key)), None, ei[2].tb_next
    return obj
```

Listing 3: Constructor function for a new SystemC object instance

```
class USIDelegateMeta(abc.ABCMeta):
    def __init__(cls, name, bases, nmspc):
        if cls.__usi_class__ != "":
            for item_name, item in cls.__dict__.items():
                if not item_name.startswith("_"):
                    usi.sc_object.attach("{}.{}".format(cls.__usi_group__, cls.
                        __usi_class__), item_name, item)

        super(USIDelegateMeta, cls).__init__(name, bases, nmspc)
        cls.__new__ = staticmethod(delegate_new)
    def __instancecheck__(cls, instance):
        return api.is_type(cls.__usi_group__, cls.__usi_class__, instance)
```

Listing 4: Python Abstract Base Meta Class

```
class USIDelegateBase(object):
     """
␣␣␣␣␣Abstract␣Base␣Class␣to␣attach␣additional␣Python␣members␣to␣SystemC␣Classes.
␣␣␣␣␣"""
     __metaclass__ = USIDelegateMeta
     __usi_group__ = ""
     __usi_class__ = ""

class Module(object):
    def __init__(self, group):
        self.group = group

    def __dir__(self):
        return list(str(name) for name in api.get_module_names(self.group))+self.
            __dict__.keys()

    def __getattr__(self, klass):
        if klass in self.__dict__:
            return self.__dict__[group]
        elif klass in list(api.get_module_names(self.group)):
            if sys.version_info >= (3,3):
                import types
                return types.new_class(klass, (), {
                    'metaclass': USIDelegateMeta,
                    '__usi_group__': self.group,
                    '__usi_class__': klass
                })
            else:
                return USIDelegateMeta(klass, (), {
                    '__usi_group__': self.group,
                    '__usi_class__': klass
```

```
31                  })
32          else:
33              return None
```

Listing 5: Module and DelegateBase class

In our factory implementation it is possible to assign constructor arguments through CCI.

D. Loading models from libraries

To further shorten compilation times the registry allows to load dynamicly linked libraries before start of elaboration. This allows to compile single models into a library and only recompile the needed part. Unfortunately Linux cannot simply load libraries from anywhere. If the libraries depend on other libraries they will only search for them in the *LD_LIBRARY_PATH*. The environment variable is read by the executable loader and cannot be modified after starting the application. This makes it hard to load libraries with models depending on each other. To overcome this shortcomming the scripting language part of the factory implements a mechanism that collects libraries. That mechanism is displayed in Figure 3. At first start the simulation will collect a list of all libraries which should be loaded until end_of_initialisation. For this to work it is only allowed to load models until this phase. Then a temp directory is created and all libraries are linked symbolicly into that temp directory. The application environemnt is copied and the LD_LIBRARY_PATH is extended by the new temp directory. In addition an environemnt variable is introduced to indicate the temp dir is added. Finally the simulation is replaced by itself with the modified environment to start from the beginning. This second time the libraries are loaded directly. Through the temporary directory each library can resolve their dependencies.

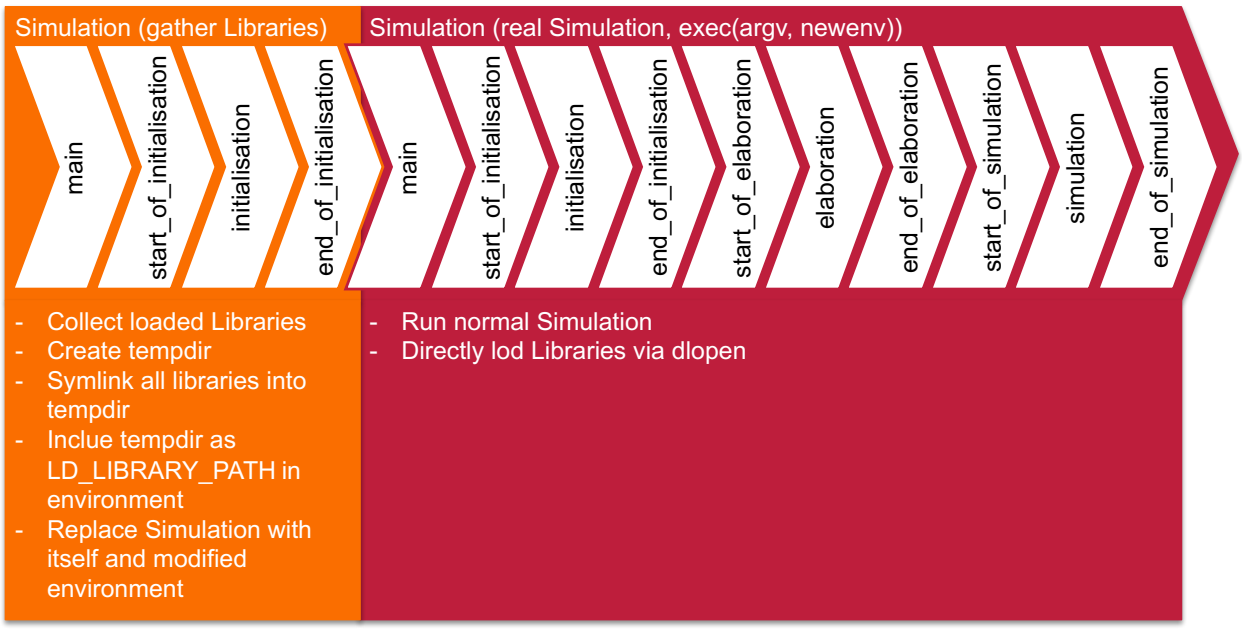

Fig. 3: The phases of the double simulation execution steps

V. Application

The presented implementation is developed as part of the SoCRocket TLM Framework and used in conjunction with the models provided by SoCRocket. The models are not build for the use with the factory and scripting language but integrate with them. All the presented components are available as open source on GitHub [9]. They are already used in the SoCRocket platform, a virtual platform (VP) framework developed for the European Space Agency as reference design. This has been tested extensively for Python with the Accellera SystemC simulator. For other languages and especially for the integration in other simulators the language interpreter has to be wrapped into a SystemC model. Fortunately this only has to be done once per language and simulator. Leaving the designer at the end with an easy-to-use workflow in which he only needs to modify a single script defining the top-level as shown in Listing 6 for Python: A working platform consists of a script importing the USI dependencies and therefore

loading SystemC and the VP models as C++ extensions to the scripting language. Followed by the definition of the structural classes (`Top`) to allow organisation in blocks. In the constructors of these classes SystemC models are instantiated. After this their members can be bound.

```
import usi
class Top(usi.Module):
    def __init__(modulename):
        super(Top, self).__init__(modulename)
        self.ahbctrl = usi.registry.AHBCtrl("ahbctrl", rrobin=True)
        self.ahbctrl.ahbOUT.bind(self.apbctrl.ahb)
        # ...
usi.registry.load('./build/models/libahbctrl.so')
top = Top(  t o p  )
usi.start()
```

Listing 6: Top-Level Python script to instantiate a AHBCtrl model inside a top class.

To register a model macros are provided which handle the registration process like the built-in SystmeC macros already known, see Listing 7. In most cases a simple macro call is enough. To register models with more parameters than the `sc_module_name` it is possible to provide a specific constructor function an registere it with the registry.

```
#include <systemc>
#include <sr_registry>
SR_HAS_MODULE(AHBCtrl);

class AHBCtrl : public sc_module {
   // ...
```

Listing 7: Registration of a simple sc_model.

The overall design flow improvements introduced by our solution results in reconfigurability without the need for recompilation and no impact to the runtime of the simulation. Due to the use of a general purpose scripting language it is relative simple to extend the system for new use cases for example to load a platform configuration directly from IP-XACT at runtime.

VI. Conclusion

We have presented a straightforward and versatile model factory implementation with configuration registry and scripting interface support. The application demonstation shows the natural usage in a pythonic environment which can also be adapted to other scripting languages if needed. Moreover the dynamic library loading support extends the factory to only recompile and link a minimal subset to further drill down compilation times. It is needed to further shrink the times of large scale design space explorations through minimization of recompilation times or completly relinguish the compilation step. Therefore it is in conjunction with other methods like [10].

The underlying mechanisms to abstract SystemC types do heavily rely on datatypes like `sc_variant` and therefore show the importance to standardize such a variant type in the future. In conjunction with USI it interfaces the upcoming CCI standard with scripting languages. By making our implementation available as open-source on Github, we encourage everyone to consider it for their frameworks and especially for the current standardization activities within the SystemC working groups.

References

[1] C. Sauer and H.-P. Loeb, "A lightweight framework for the dynamic creation and configuration of virtual platforms in systemc," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 1, pp. 5:1–5:16, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/2983626

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman, Boston, MA, 1995.

[3] Epic Games. Blueprints visual scripting. [Online]. Available: https://docs.unrealengine.com/latest/INT/Engine/Blueprints/index.html

[4] Accellera, "Accellera working group for Configuration, Control and Inspection," *Website http://www.accellera.org/activities/committees/systemc-cci/*, 2015.

[5] C. Schröder, W. Klingauf, R. Günzel, M. Burton, and E. Roesler, "Configuration and control of systemc models using tlm middleware," in *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '09. New York, NY, USA: ACM, 2009, pp. 81–88. [Online]. Available: http://doi.acm.org/10.1145/1629435.1629447

[6] R. Meyer, J. Wagner, B. Farkas, S. Horsinka, P. Siegl, R. Buchty, and M. Berekovic, "A scriptable standard-compliant reporting and logging framework for systemc," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 1, Oct 2016. [Online]. Available: http://doi.acm.org/10.1145/2983623

[7] R. Meyer, J. Wagner, R. Buchty, and M. Berekovic, "Universal scripting interface for systemc," in *DVCon Europe Conference Proceedings 2015*, Nov 2015.

[8] J. Wagner, R. Meyer, R. Buchty, and M. Berekovic, "A scriptable, standards-compliant reporting and logging extension for systemc," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, July 2015, pp. 366–371.

[9] SoCRocket sources. [Online]. Available: https://github.com/socrocket

[10] S. A. A. Shah, B. Farkas, R. Meyer, and M. Berekovic, "Accelerating mpsoc design space exploration within system-level frameworks," in *The IEEE Nordic Circuits and Systems Conference (NORCAS), 1-2 November 2016 Copenhagen, Denamrk*, Nov 2016.