META II: LINGUA FRANCA DESIGN AND INTEGRATION LANGUAGE

Michael Masin, Alberto Sangiovanni-Vincentelli, Alberto Ferrari, Leonardo Mangeruca, Henry Broodney, Lev Greenberg, Michael Sambur, Dolev Dotan, Sergey Zolotnizky and Sasha Zadorozhniy

IBM Research – Haifa Research Laboratory

August 2011
Final Report

Approved for public release; distribution unlimited.

The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

See additional restrictions described on inside pages

TABLE OF CONTENTS

Section **Page**

LIST OF FIGURES ..... **iv**

1 SUMMARY **1**

1.1 General *1*
1.2 Team *1*

2 INTRODUCTION **2**

2.1 Lingua Franca Ecosystem *2*
2.2 Common Semantics *3*
2.2.1 Common Semantics Comprehension 4
2.3 Existing Work *5*
2.4 IBM Commercial Tools *5*
2.5 Language Elements *6*
2.5.1 Semantic Foundations 6
2.5.2 Models of Computation 6
2.5.3 Integration Language 6
2.5.4 Contracts 6
2.5.5 Variability Modeling 7

3 Methods and Assumptions **9**

3.1 Challenge problems *9*
3.1.1 Electric Power System 9
3.1.2 Gas Turbine Engine 10
3.2 Models of Computation *11*
3.3 Variability *12*
3.4 Contracts *12*

4 Results and Discussions **13**

4.1 MoC Integration Framework *13*
4.1.1 Early Example of MoC Integration 13
4.1.2 SysML Extensions 14
4.1.3 Latest example of MoC integration 16
4.1.4 Latest Evolution of SysML Extensions 18
4.1.5 META II Tool Flow for Hybrid Simulation 22
4.2 Black Box Integration *23*
4.2.1 Overview 23

Section **Page**

4.2.2 Modeling 23
4.2.3 Integration Process 24
4.2.4 Pros and Cons 24
4.2.5 EPS Usecase 24
4.3 Contracts 25
4.3.1 Specification Layer Overview 25
4.3.2 Contracts Meta-model 27
4.3.3 Semantics of Dynamic Asserts 29
4.3.4 Contract Algebra 31
4.3.5 Applications 34
4.4 Concise modeling 41
4.4.1 General 41
4.4.2 Planes / Layers 41
4.4.3 SysML Extensions Profile 43
4.4.4 Concise Profile 43
4.4.5 Constraints Modeling Stereotypes 45
4.4.6 Variability Profile 46
4.4.7 Objectives and Algebras 49
4.4.8 Exemplary Design Process With Concise Modeling 51
4.4.9 Concise Plug-in 53

5 Conclusions 56

5.1 Dealing With Scale 56
5.2 MoC Integration 57
5.3 Contract Based Design 57

6 References 58

LIST OF ACRONYMS, ABBREVIATIONS, AND SYMBOLS 59

LIST OF FIGURES

Figure 1 - Lingua Franca Ecosystem 3
Figure 2 - Common Semantics Comprehension 5
Figure 3 - EPS Use Case Diagram 9
Figure 4 - Secondary EPS Geometrical Layout 10
Figure 5 - Gas Turbine Engine Model 11
Figure 6 - Diagram of March PI Meeting Demo 13
Figure 7 - MoCC Package Block Diagram 15
Figure 8 - Specific MoCC Package 16
Figure 9 – Model Structure for the MoC Integration Demo 17
Figure 10 –SLD Component of the MoC Integration Demo 17
Figure 11: SysML Extensions for META II Semantic Integration 18
Figure 12: DESYRE Adaptor Library 20
Figure 13: The Electrical Model of Computation 21
Figure 14: System's Electrical View 21
Figure 15: Automated Tool Flow for Hybrid Simulation 23
Figure 16 - Concise Structural Assert 25
Figure 17 - Complex Monitor 27
Figure 18 - Contracts Meta-model 27
Figure 19 – Contract Block Example 28
Figure 20 - Contract Usage Example 28
Figure 21 - Example of LF Automata 33
Figure 22 - Properties of Continuous Signals 33
Figure 23 -Requirements Transformation Flow 35
Figure 24 - Visual Expression (all_buses_powered) 36
Figure 25 - EPS Relay Contract 36
Figure 26 - Generator Control Unit Contract 37
Figure 27 - Case 1 (closed,closed) [Ebeg, Eend] 39
Figure 28 - Case 2 (closed,open) [Ebeg, Eend) 39
Figure 29 - Case 3 (open, closed) (Ebeg, Eend] 40
Figure 30 - Case 4 (open,open) (Ebeg, Eend) 40
Figure 31 – Primary EPS Functional view 42
Figure 32 – Secondary EPS Functional view 42

Figure 33 – Secondary EPS Technical View 43
Figure 34 - Index (geometry) View 43
Figure 35 - Constraint Diagram 46
Figure 36 - Variability Realization Package 47
Figure 37 - Variability Abstraction Package 48
Figure 38 - Choice tree 48
Figure 39 - Textual Approach in Algebras Definition 50
Figure 40 - Parametric Diagram as Algebra Definition 51
Figure 41 - IBM Design Optimization Process 52
Figure 42 - Design Optimization Process in Detail 53
Figure 43 - Gas Turbine Engine Concise Model 55
Figure 44 - Abstraction as the Key to Scale 57

Approved for public release; distribution unlimited

1 SUMMARY

1.1 General

IBM Research is leading a crack team of researchers with a common goal of defining a common language for cyber-physical systems modeling. The main vision that is driving the effort is to give the Systems Engineering (SE) community the ability to formally express design models and to reason about their interaction on a common semantic basis, eventually allowing automated composition and full virtual verification of complete systems.

1.2 Team

IBM team's inspiration comes from Alberto Sangiovanni Vincentelly, a pioneer of Electronic Design Automation (EDA) and a universal expert in hybrid systems research. Alberto's input is instrumental in every aspect of our activity and IBM is very fortunate for have him on the team.

A veteran of European hybrid systems research Advanced Laboratory for Embedded Systems (ALES) from Rome (Italy) brings invaluable experience and a broad basis in tool and language integration for heterogeneous systems. Represented by Alberto Ferrari and Leonardo Mangeruca, ALES team handles the difficult part of defining the mathematical framework for common semantics and of integrating the various models in a single tool.

We are taking our cues on design processes and methodologies from United Technologies Research Center (UTRC) team, who also provides us with use cases for our language and tools validation and demonstration. UTRC team is represented by Brian Murray and Alessandro Pinto.

Dr. Michael Masin, from IBM Haifa Research Laboratory (HRL) is the Principle Investigator for this activity. Lev Greenberg is leading the contracts language and semantics effort. Henry Broodney is in charge of the variability and concise modeling activity.

2 INTRODUCTION

Increased systems complexity is causing schedule and costs overruns in the vast majority of defense and aerospace development projects. Despite the wide adoption of domain specific software tools, which are able to model, simulate and verify the most intricate corners of mechanics, flow dynamics, thermal interactions, electric power and logic and many more, the interaction between those tools, within and across the various domains, remains a thing of the future. Modeling, which is ubiquitous in the various domains, focuses on the interaction of object within the domain thus not having semantic meaning outside the domain. Also many modeling languages do not carry precisely defined semantics, but rather employ execution semantics of the specific tools they are used in. Thus integration between two languages and often between the same language in two separate tools is impossible. To address the above IBM team, basing its effort on the experience of prof. Sangiovanni and of ALES, is working on the definition of an integration language with rigorous semantics that would be able to bind the other languages and tools together. The work is based on the Tagged Signal Model (TSM) [1] developed by prof. Sangiovanni and his colleague Edward Lee. The specific behavior and the semantics of various tools and languages are called Models of Computation and Communication (MoCC).

Contemporary engineers lack ability to formally express the requirements and components, so that composition of the latter would be possible. Furthermore a formal description of both will facilitate computer based reasoning about the components' composition and about the satisfaction of the requirements by a specific components' architecture. IBM proposes to adopt an already circulating approach of contract based design (or assume-guarantee reasoning) and augment it by a specific contract language definition with a semantics defined using the results of the activity outlined in the previous paragraph.

In addition, for a true computer aided design of complex systems, we would like the software tools to help us come up with the best architectures for our systems. In order to do that the new language must be able to model systems, their composition rules, constraints, parameter relations and more, so that a computer is able to generate architectures from the above inputs, thus significantly decreasing system design times, reducing the number of unsuccessful design iterations and improving the overall quality of the systems.

We call our proposed language Lingua Franca. SysML, as the predominant global standard for systems modeling, serves as the base for the language.

2.1 Lingua Franca Ecosystem

In order for the reader to have perspective about the place of the new language in the Systems Engineering ecosystem we bring forth Figure 1 which was the centerpiece of our presentation at the 5th META PI meeting (July 2011, Arlington, VA).

The Tagged Signal Model (TSM) is the core of the language semantics foundation in the following onion representation of the ecosystem. The TSM facilitates integration between domains and serves as the basis for the rest of the language elements.

The three sides of the language coin are the MoCC integration language, variability modeling package and the contracts language.

Next are the tools that use the language. The forefathers of those tools are the various prototypes that IBM team has demonstrated in the course of the PI meetings. The tools use the language and

their prototypes serve as the validators to the language, a sort of assurance that the language can express what the design process, via the tools, needs to have expressed.

And the most important layer is the participants of the design process. There are three levels of designers in the ecosystem, ones that do the actual designs (System Engineers), ones that develop the tools and lastly the ones that develop the methods and deal with the Tagged Signal Model. This situation is very similar to what is happening in electronic design.

In addition to designers there are several other, no less important groups, that affect the process and are going to use and benefit greatly from the enhanced formality of the language. These are, for example, Marketing people, Customers and other non-engineering stakeholders.

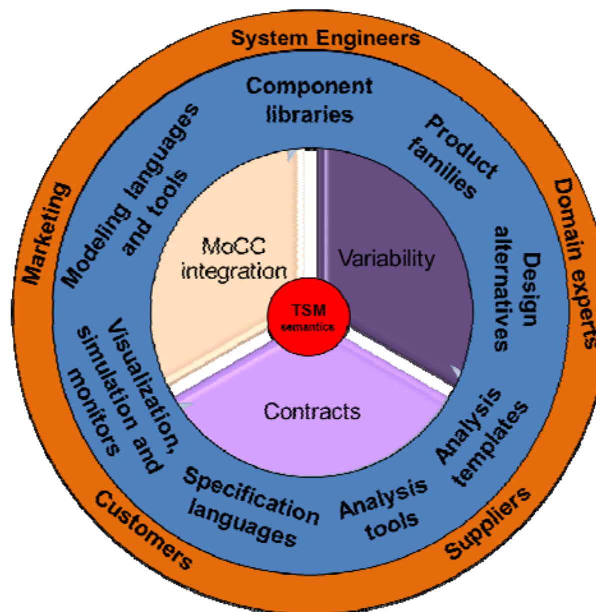


Figure 1 - Lingua Franca Ecosystem

2.2 Common Semantics

Models of cyber-physical systems, or parts thereof, whether geometrical, behavioral, thermal, mechanical, etc., are formalized with respect to a precise mathematical framework that we call the **model's semantics**. The focus is on the **unification** of the relevant semantics (e.g., 3D geometry, spatiotemporal ordinary and partial differential equations, stochastic processes, discrete systems, hybrid systems) for cyber-physical modeling, while retaining their analytical power.

The unification of semantics has the purpose of enabling integrated cross-domain formalization, analysis and synthesis (Sinnig 2007) (Q. D.-V. Zhu 2006) (Q. Zhu 2007) (Madl 2006). While different unifying semantics are used for different analytical purposes, an overarching semantic unification is still required to provide the general meaning of integrating different domain specific models, so that all analytical results, possibly obtained over different semantics, *concur to a consistent interpretation and understanding of the overall system*. To take advantage of the analytical power of the different semantics, the approach described in the present document also

advocates the specification of different semantics as libraries within the META language to enable both domain specific and integrated cross-domain analytical flows.

The semantics concept has two facets – Denotational and Operational. Denotational semantics attempts to formalize the meaning of a language in mathematical terms, which provide meaning to the language constructs. Operation semantics is the definition of how execution results are obtained from the model. Having defined Denotational semantics allows formal verification and optimization, whereas Operational only allows joint execution of models.

In the approach described in the present document, and detailed in Appendix B the common semantics is fine-grained so that that all domain specific semantics relevant for the META II project can be represented. The common semantics may evolve over time, if needed to account for new semantics.

The current status of the proposed common semantics can represent physical modeling over space and time (ordinary and partial differential equations, hybrid systems) as well as for timed and untimed digital systems and their integration with physical systems. The common semantics can and will be extended to probabilities and stochastic processes over space and time.

The structural representation of the cyber-physical design is captured by the approach presented in the present document in the *META integration language*. This language is designed to specify the integration of models defined using different domain specific languages and semantics. The *language only captures integration aspects*, which include also the original domain specific semantics in which the models are defined. Semantic annotations enable the user to be aware of semantic integration aspects and explicitly introduce appropriate semantic adaptation components. The effects of the integration can be verified using the analysis tool chain, that is also aware of the semantic integration made *explicit* in the integration language. The integration language supports both *denotational and operational semantics*, so that the integration of analysis tools is *systematic and semantically robust*. The analysis tools chain is not necessarily targeting the common foundational semantics, whose purpose is to provide rigorous mathematical definition of the meaning of the overall system model. On the contrary, the analysis tool chain exploits the analytical power of the different semantic domains supported by the integration language.

The main differentiation of the proposal outlined here with respect to existing approaches to the modeling of heterogeneous systems is the focus on the integration of existing domain specific languages and semantics and corresponding analysis and synthesis capabilities. The proposed language is used to define a semantically robust integration of IPs and analytical methods and results provided using several domain specific languages and tools.

2.2.1 Common Semantics Comprehension

As stated previously, most tools have no formally (mathematically) defined semantics and the only semantics they have is defined, often incidentally, by the actual functionality of the simulators and solvers. These semantics are generally undocumented and are difficult to discover. Hence only black box interactions are possible, when the integrating tool is not aware of the nature or the behavior of the black box component, but rather only of the component's boundary. An integration tool with understanding of the internal semantics may achieve different results.

Approved for public release; distribution unlimited

Another aspect of semantics is its importance in the eyes of the very people who need it. The word means different things to different people and we've seen such differences even in the PI meeting workshops. The regular model transformation today is purely syntactic and thus non-generic and sometimes flawed, since the same model will yield different results in a different tool. In our experience also through the course of interaction with other META performers, we see that once the idea settles in one's mind the need becomes clear.

The vision of common semantics is, however, not easily attainable one. Figure 2 is an excerpt from one of our PI meeting insight slides. The left hand box represents the state of the art today with the center box representing the vision of common semantics. The task of defining formal semantics for all tools and languages is gargantuan in nature, both due to its own complexity and due to the fact that a formal definition must be devised for the actual source modeling environments. Thus the plausible reality, the sprouts of which we have attempted to show in our demonstrations, is to have an integration layer between the integrative analysis tool and the source models. At the very least a black box simulation or computation should always be possible. As time progresses and tools' semantics are formalized more analyses and automatic design activities can take place.

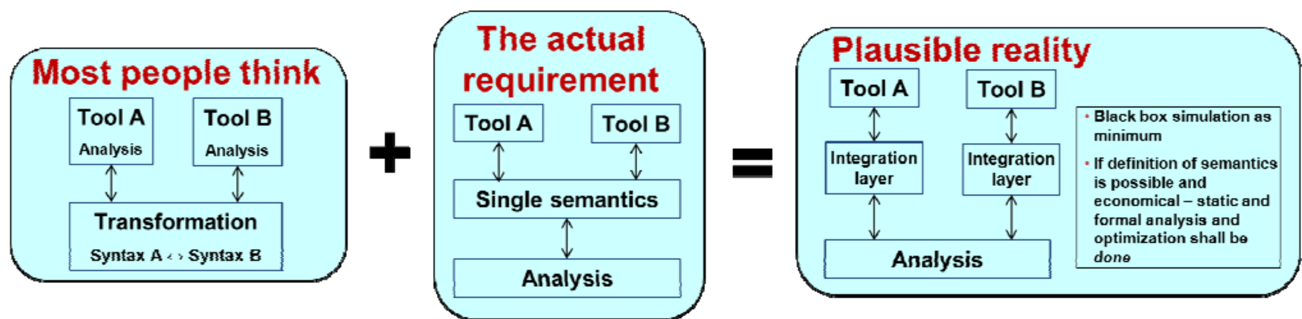


Figure 2 - Common Semantics Comprehension

2.3 Existing Work

IBM effort is, in part, fed by the work done by Alberto Sangiovanni, his colleagues and his students, in the field of common semantics, heterogeneous tools and contract based design [REFS]. In addition our ALES partners bring experience from the European Union (EU) SPEEDS project, where a contract framework and a Hosted Simulation protocol were developed. In addition we make use of the DESYRE simulator which is a TSM based commercial tool developed by ALES.

IBM Research has developed a Product Lines Engineering approach for software modeling. This approach is partially integrated into the new language variability package.

2.4 IBM Commercial Tools

IBM Corporation is a leading tool provider for the SE industry. Our work is mostly based on the Rational Rhapsody modeling tool and iLog CPlex Studio optimization environment. Our prototypes are mostly plug-ins for Rhapsody.

2.5 Language Elements

2.5.1 Semantic Foundations

The META II language has been founded over rigorous mathematical foundations by defining a reference common semantic domain based on the Tagged Signal Model (TSM) mathematical framework. The definition of the reference common semantic domain has been detailed in a document along with examples on how it can be used to define Models of Computation (MoC) and to provide semantics both to domain specific languages and models expressed with such languages.

The reference common semantic domain is able to capture different domain semantics, including those required to specify spatiotemporal ordinary and partial differential equations, timed and untimed digital systems, discrete event models, 3D geometrical and kinematic models, thermal models. The TSM mathematical framework has also been extended with the capability of representing probabilistic and stochastic semantics, including probabilistic uncertainty models, Markov Chains, Stochastic Hybrid Systems.

The reference common semantic domain is also able to represent operational semantics, for the definition of computation protocols, such as the Functional Mockup Interface, the SPEEDS Hosted Simulation protocol, the Ptolemy actor semantics.

The reference common semantic domain has been demonstrated with some simple examples, among which the PID controller example has been discussed in the Semantics Workshop at the PI meeting in May.

2.5.2 Models of Computation

Models of computation (MoCC) provide semantics to the models referred to through the Lingua Franca language constructs. Constructs are provided within the Lingua Franca language to

- Associate modeling elements such as blocks and ports with semantic elements (models of computation);
- Express requirements that MoCCs impose on modeling elements they are associated with; for example, the discrete time model of computation requires that each modeling element it is associated with specify the sampling period and initial time offset as attributes.

The language described in the present document enables to specify MoCCs and their modeling requirements as SysML profile extensions and associations between modeling elements and semantic elements as SysML stereotypes in the Lingua Franca model.

2.5.3 Integration Language

The META II integration language has been conceptualized as an extension of the SysML standard language. The language includes means to relate blocks and block ports to semantic domains. Semantic domains specify attributes that blocks and/or block ports must assign value to.

2.5.4 Contracts

Lingua Franca contracts framework based on Assumption/Guarantee approach was developed. Assumption/Guarantee asserts are represented as processes in framework of TSM. Semantics and syntax of the contracts are partially defined, while common semantics of contracts enable

analysis of contracts specified in different Domain Specific Languages (DSL) such as PSL, CSL, PRIMATIC, etc. We have shown flow from requirements to formal contracts and monitors generation for UTRC use cases. Usage of contracts was demonstrated as part of design space exploration flow.

Our language enables to specify contracts as white/gray/black boxes in stereotyped SysML model.

The contract based design approach facilitates a central concept of the META program – Platform Based Design (PBD). PBD is a formalization of what Systems Engineers have been mostly doing for quite a while. In contemporary competitive environment virtually no system design is started from scratch, but from an assortment of existing engineering assets (components), which the Systems Engineer uses, to the maximum extent possible, to create an architecture that satisfies the requirements. From time to time there will be no component that for some function and then the engineer will create a placeholder for that component and produce a requirement specification. PBD approach, recognizing the above, attempts to formalize that process analyzing its philosophy in order to create tools and methods that will aid the Systems Engineer improving designs and reducing costs and schedules.

2.5.5 Variability Modeling

2.5.5.1 Background

System modeling in SysML is a relatively new practice. The architecture is usually represented by a collection of blocks interconnected by flows. The flows represent data or physical matter or energy. The main objectives of system modeling normally are: conveying information to peers, simulating the system's behavior to verify its correctness or generating software code for system control.

However, before any analysis can be done, the Systems Engineer (SE) must model the architecture or a set of architectures are the best candidates for the system in question.

There are two main problems that the SE is facing. The first is that it may be impractical to model the entire system in SysML, which is a predominantly graphic language, due to the system's sheer size and complexity. For example, modeling a network of one hundred Ethernet switches is time consuming, error prone and the resulting model would most certainly be unreadable.

The second, even more acute problem is that the optimal architecture is unknown at the time of modeling. The SE needs to create the various alternatives and reason about their merits. The alternatives creation process is often manual, resulting in missed variants that may have been the best choice.

2.5.5.2 Objective

The purpose of variability modeling is:

- Define complete composition rules for a system without explicit modeling of a system
- Define variability points in a system:
    - Components variability – selection of a component out of an existing library
    - Topological variability – selection of which components are connected to which
    - Geometrical variability – selection of specific location for each component

  - Relational variability – selection of specific components based on the existence or non-existence of others
- Define constraints for variability choices (above)
- Provide information for automatic formulation of an optimization problem in order to find an optimal system architecture
- Allow automatic expansion of the concise model into a concrete system architecture:
  - Based on a set of expansion parameters (solution)
  - Conserving composition rules to allow subsequent simulation and verification of the expanded system

3 METHODS AND ASSUMPTIONS

3.1 Challenge problems

Every engineering method is best explained by applying it to a real world use case. IBM has contracted a fellow META performer, United Technologies Research Center (UTRC), to provide use cases on which the newly developed language and methodologies will be tested.

3.1.1 Electric Power System

The lead use case provided by our partners is an Electric Power System (EPS) for an experimental Unmanned Aerial Vehicle (UAV). Figure 3 depicts the general structure of the EPS. The EPS is split into two major parts – the primary EPS and the secondary EPS.

The primary EPS includes the power generation infrastructure and ends with the Alternating Current (AC) and Direct Current (DC) buses. The main goal of this part of the system is to make sure that all buses are powered in as many scenarios as possible. It consists of four power sources (right and left generators, an Auxiliary Power Unit (APU) and an inverter). A relay network connects the sources to the two AC buses, which in turn are powering the DC buses through a Transformer Rectifier Units (TRU). The external power input and the Ram Air Turbine (RAT), which are both normally present in all aircraft, were dropped to reduce the use case complexity.

The secondary EPS starts with the buses and includes the power distribution infrastructure to the loads of the system. The core of the secondary EPS is a collection of Power Distribution Boxes (PDB), which house electric protection and power management devices. Each PDB drives several loads.

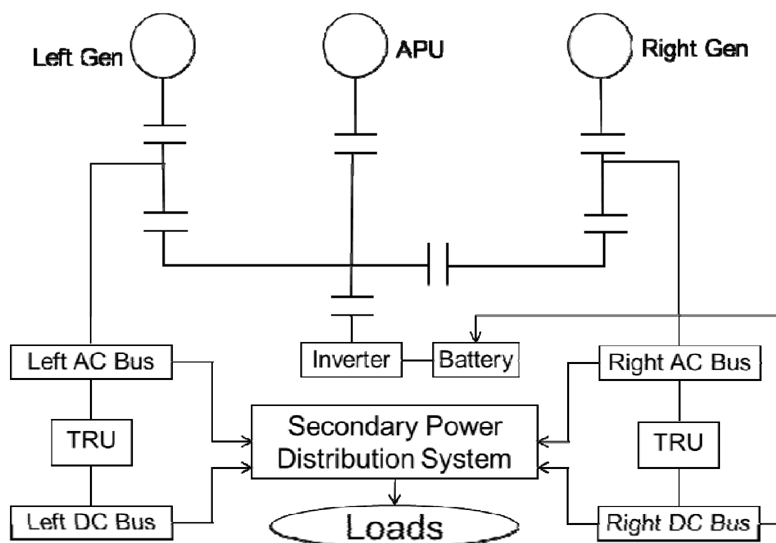


Figure 3 - EPS Use Case Diagram

Since we wanted to address the geometrical aspect of the architecture, UTRC provided geometrical data for the possible locations of the PDB network and the loads. Figure 4 depicts the secondary EPS geometry with the dark rectangles being the optional locations of PDB and the white rectangles and the red circles are the concrete locations of the exemplary loads in the system. The green lines depict the right and left power buses.

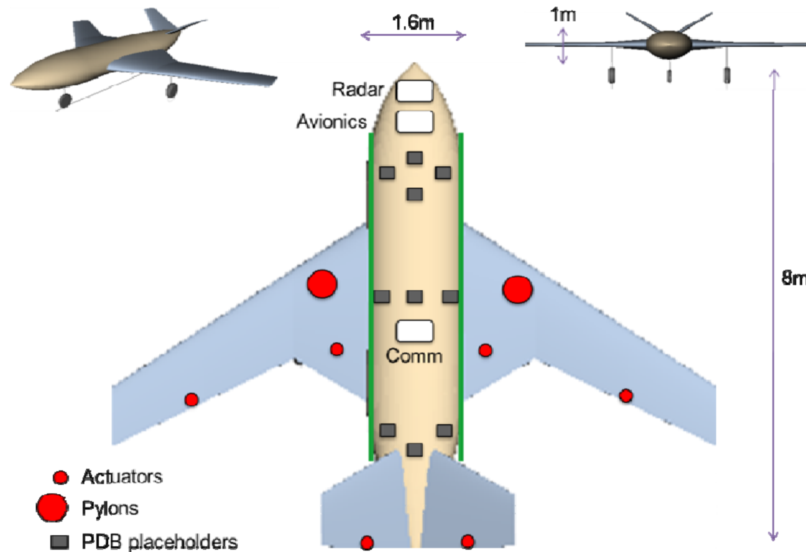


Figure 4 - Secondary EPS Geometrical Layout

In addition we have received and used:

- A list of requirements in the form of contracts. Following are some exemplary requirements:
    - Power sources can never be paralleled
    - All buses must be powered in case of no more than 1 failure beyond the Minimum Equipment List (MEL – list of failure combinations that do not affect the mission worthiness of the aircraft) in steady state (breaks of up to 2ms allowed)
    - TRUs can never be paralleled in steady state (transients allowed)
    - Critical loads must be powered for at least 30 minutes in case of loss of all mechanical power sources (Generators, APU)
- A Simulink analysis model – capable of accepting an architecture and component data and generating current/voltage/power on the nodes.
- Simulink models of components – generators, relays, TRU.

3.1.2 Gas Turbine Engine

Another use case we have used for validation of the concise modeling approach is a Gas Turbine Engine model provided by Pratt & Whitney, who are a part of the UTRC team.

The use case calls for generating architectures for the engine, under a set of limitations. The engine consist of standard elements – compressors, combustor, turbines, nozzles and gears. Figure 5 is a high level SysML description of the engine.

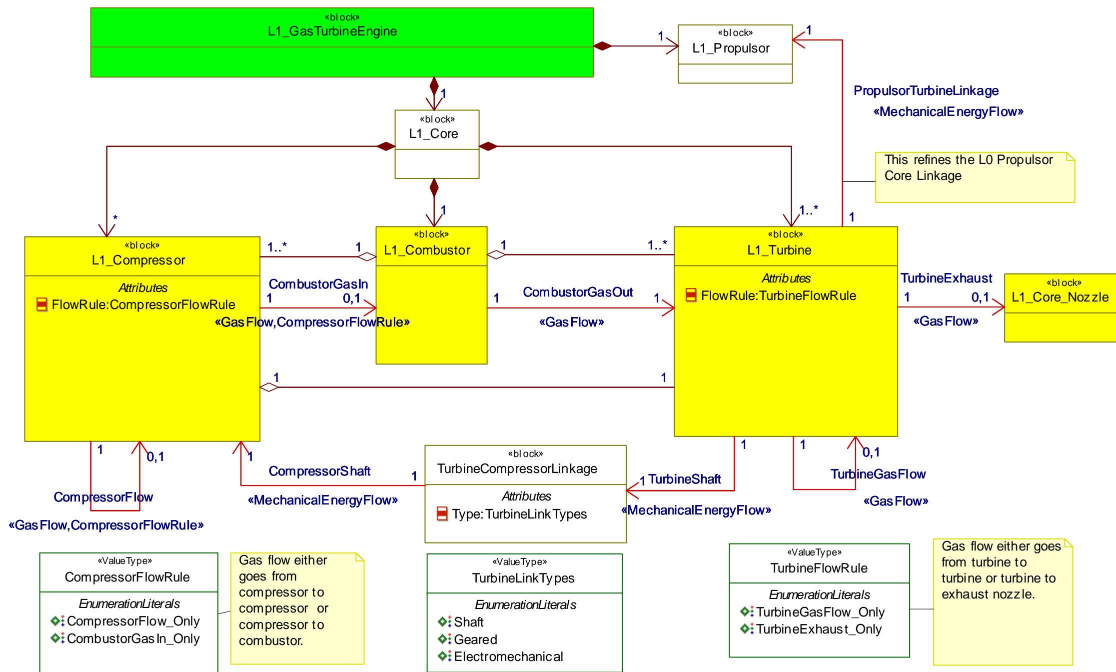


Figure 5 - Gas Turbine Engine Model

3.2 Models of Computation

The approach to dealing with multiple Models of Computation has two major aspects – specify the MoC in the design model and be able to integrate it with MoCs.

Since Lingua Franca integration language is based on SysML, the logical choice for the MoC specification is the SysML extension mechanism through stereotyping.

Integration aspect is more complex. There are several levels on which models can be integrated.

We identify three levels of integration: black box, white box and clear box. These three levels encompass the three cases that we have identified.

Black box model internals are inaccessible for the integration framework and the interaction occurs on the interface/boundary protocol level. Black boxes are required to have precise operational semantics that ensures a rigorous integration with the rest of the system. The language will be able to integrate the black box component, if it already supports its corresponding operational semantics. The black box approach is envisioned to be primarily used for integration of legacy components and can be used for types check/simulation based analysis.

A white box model ensures the integration of components from different specification and analysis tools. White box components allow the integration of components defined in different tools and can be used for formal analysis, Design Space Exploration (DSE) and optimization. Such integration is possible if the corresponding models of computation are supported by the language.

Clear box components are components natively specified within the integration language. Integration of white box and clear box components is defined both at the denotational and at the operational level, ensuring the applicability of a potentially wider range of analyses.

3.3 Variability

In variability modeling we have taken a pragmatic approach to language development. Using the EPS use case we have searched for variation points in the design and for most generic form that can still convey the general construction of a system.

With the help of our UTRC partners we have explored the information available to Systems Engineers during the architectural design phase.

We have also tried to reuse, as much as possible, the Product Lines Engineering (PLE) asset the IBM already has in place.

3.4 Contracts

We have examined previous work done in contract based design, mainly the EU SPEEDS project result. From the beginning our objective was to find a convenient way to apply contracts in design flows. The concept of contract libraries needed to be explored and expanded and groundwork needed to be laid to tie formal contract to their TSM definitions.

4 RESULTS AND DISCUSSIONS

4.1 MoC Integration Framework

Appendix B contains the theoretical background for all discussions on Models of Computation, including tool integration and heterogeneous contracts. This paper was generated in the course of work on META.

4.1.1 Early Example of MoC Integration

The earliest (in META) example of MoC integration was displayed at the 3rd PI meeting (March 2011).

Figure 6 shows the diagram of the third scenario of the heterogeneous simulation (based on the EPS use case). The model consist of the following groups of components: electrical elements (generators, contactors, TRUs, AC and DC busses), sensors, a controller and two system contract monitors. The electrical elements have been modeled by continuous time differential algebraic equations, with discrete control inputs for contactors and generators and discrete outputs for the DC busses that are read by the four sensors. The sensors are modeled as a discrete time component, while the controller and the contract monitors are modeled as discrete event components, so that the simulation spans three different models of computation, each with its own solver/scheduler.

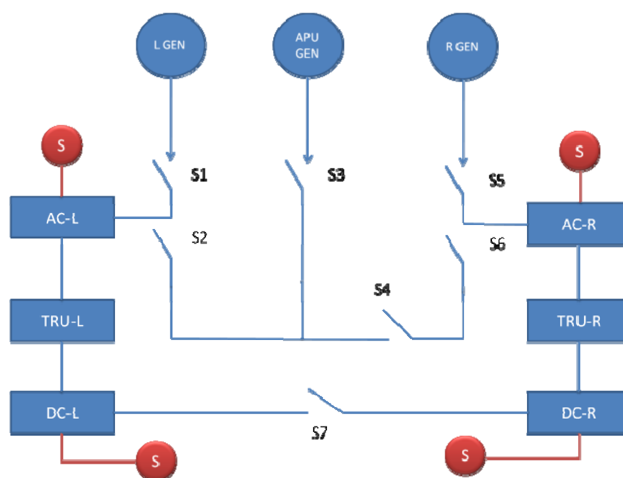


Figure 6 - Diagram of March PI Meeting Demo

The demo is organized in three scenarios:

In the first scenario, the connection between the DC-L and the DC-R busses is missing. Then, it is shown that in certain failure conditions it not possible to power all busses when a single generator is on. This is shown because the scenario leads to a contract violation detected by the corresponding monitor.

In the second scenario, the connection between the DC-L and the DC-R busses is added without the S7 contactor. Then, it is shown that, in the same failure conditions as the first scenario, the controller parallels two AC sources, which violates a different contract requirement. The contract violation is detected in the simulation scenario by the corresponding contract monitor.

In the third scenario, the contactor S7 is added and it is shown that both contracts are now satisfied in the given failure conditions.

4.1.2 SysML Extensions

A prototype of a SysML profile has been developed, where semantic domains are defined in a SysML profile and relations between blocks and/or block ports to the semantic domains can be provided through appropriate stereotypes. Semantic adaptation between blocks is specified through specific connectors, called thick connectors, which are stereotyped with the appropriate semantic adaptation scheme. For example a sampler can be used for adapting the continuous time semantic domain with the discrete time semantic domain, while a zero-hold can be used to do the reverse adaptation.

In subsequent version of the SysML profile for the META II integration language, it has been decided to avoid associating semantic adaptation functionality to block connectors, to be compliant with the SysML semantics for connectors. The adaptation mechanism is instead specified by the used through an explicit block.

There are two basic types MoCCs stereotypes: denotational and operational. Denotational stereotypes define to which MoCC (denotational semantics) particular component belongs to, including parameters of the MoCCs (for example a discrete time MoCCs must have period and phase specified) which implemented using stereotypes tags are.

Operational stereotypes define operational semantics which is used to analyze (for example by simulation) a particular component. There is an «implements» relation between a corresponding denotational and operational MoCCs.

In addition separate stereotypes are defined for blocks and ports, so to define heterogeneous components one can mark a port of the block as a different MoCCs.

A basic MoCCs package in Figure 7 defines the most general MoCCs, which will be used be define more specialized MoCCs using generalization relation with a «specialize» stereotype.

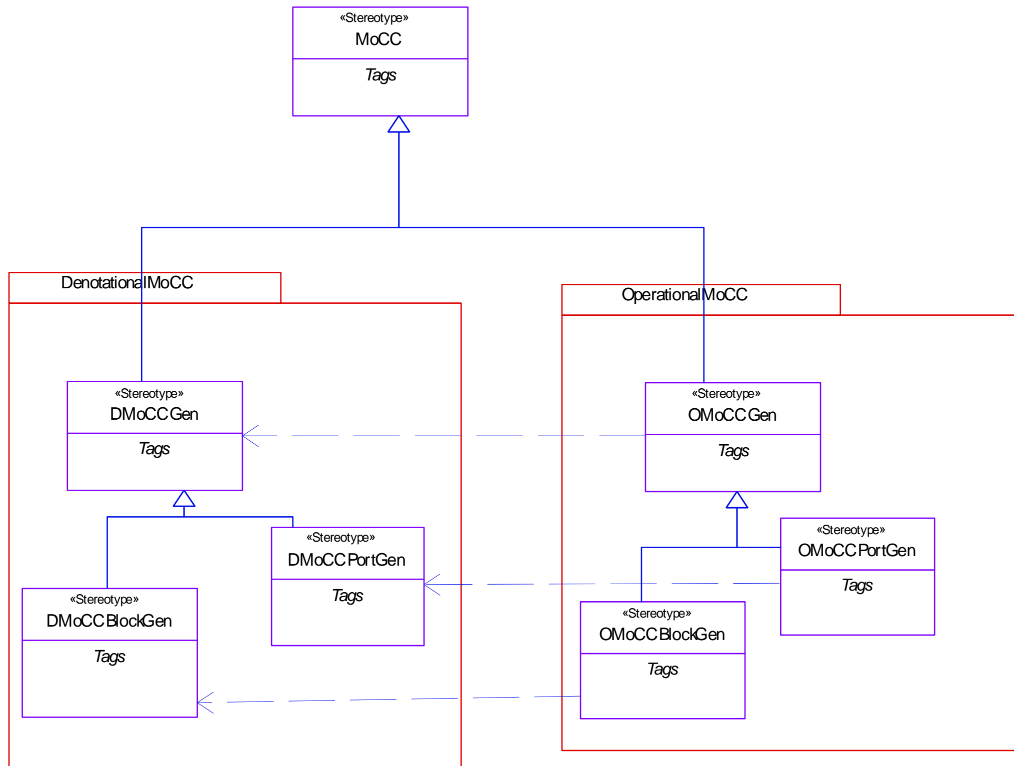


Figure 7 - MoCC Package Block Diagram

In Figure 8 we show a specific MoCC package, which defines Differential Algebraic Equation (DAE), Discrete Event (DE), Timed Data Flow (TDF) and Electrical Network denotational MoCCs. In addition ActorOriented, Simulink, Rhapsody operational MoCCs are defined. For each Operational MoCCs an interface should be provided which enable integration of all Operational MoCCs. In Figure 8 such an interface is displayed for ActorOriented Operational MoCC.

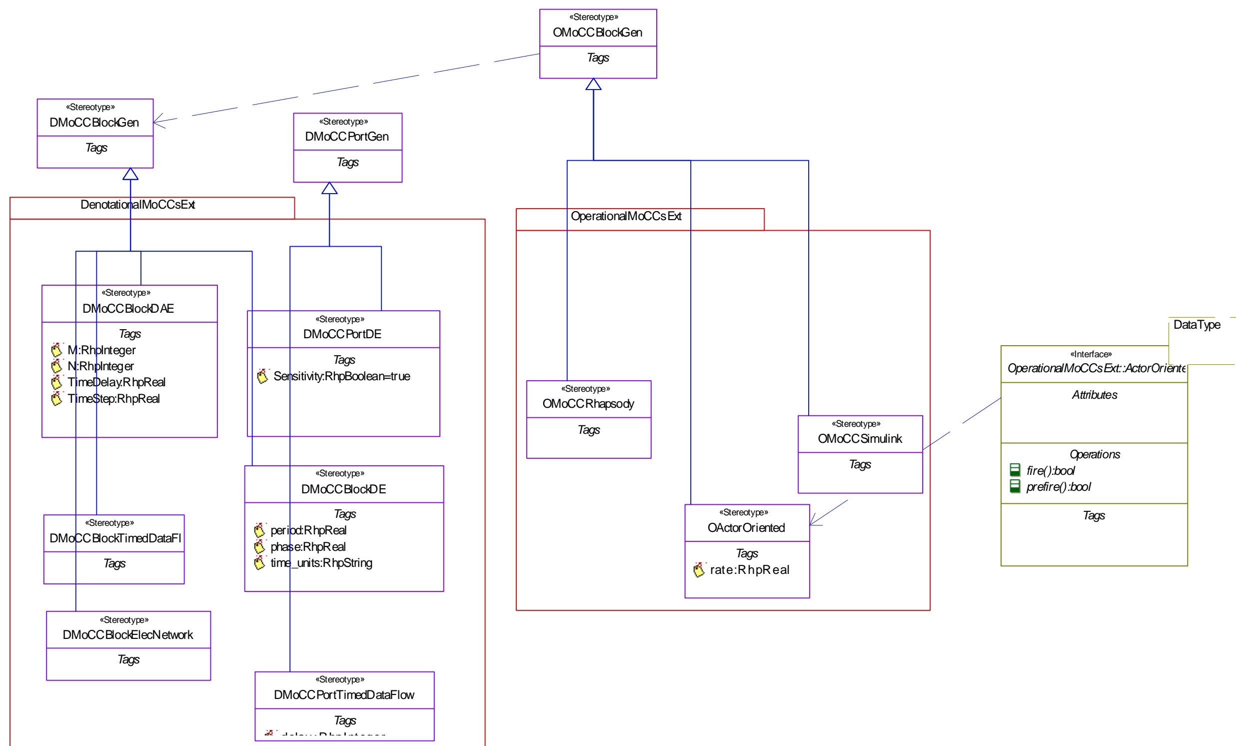


Figure 8 - Specific MoCC Package

This MoCC scheme was used in the hybrid simulation demo at the 5th PI meeting (July 2011).

4.1.3 Latest example of MoC integration

At the 5th and 6th PI meeting we have shown the evolution of the MoC integration framework.

Figure 9 and Figure 10 show the structure of the model. The demo now includes multiple domain views and integrates components generated from different tools. The electrical view, represented by the System Level Design (SLD) component, has been modeled by continuous time differential algebraic equations directly in the DESYRE simulation environment, which provides the computation engine for the heterogeneous simulation. The thermal view, represented by the ThermalAlg component, is a model of the propagation of heat from the generators to the loads and vice-versa and is modeled by an ordinary differential equations defined in the Matlab/Simulink tool and automatically imported in the DESYRE simulation framework, through the Real-Time Workshop code generation tool provided with the Matlab/Simulink distribution. The controller is modeled as a mixed discrete time/discrete event Rhapsody Statechart finite state machine, whose executable code is automatically generated by the IBM Rhapsody modeling environment, wrapped and compiled into an executable Dynamic Link Library (DLL) that is imported into DESYRE for the heterogeneous simulation. The contract monitor and the fault handler are currently natively modeled within DESYRE. With this demo we have shown that we can perform multi-domain heterogeneous simulation spanning several different models of computation, by coordinating and synchronizing the different solvers within the same simulation framework.

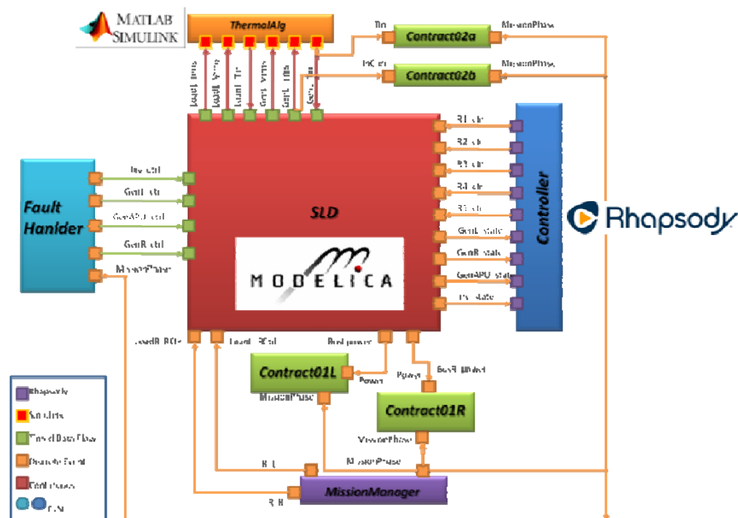


Figure 9 – Model Structure for the MoC Integration Demo

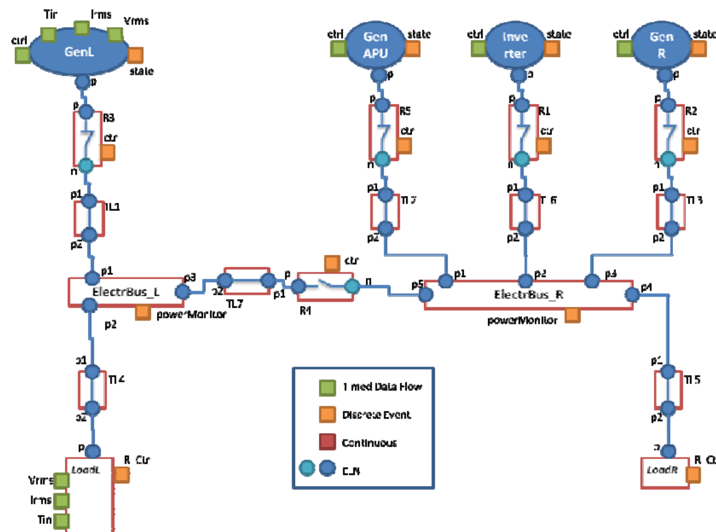


Figure 10 –SLD Component of the MoC Integration Demo

We have shown the META II language's capabilities of integration of different domain-specific tools, working with different models of computation. The structure of the integrated model specified in the META II integration language is shown in Figure 9. The behavior of the different components are defined in domain-specific tools: the electrical network (SLD component) is specified using electrical modeling offered by the Modelica language; the thermal coupling of the electrical components is defined using the MATLAB/Simulink language; the control logic is defined using the Rhapsody Statechart language; the remaining components are defined directly within the DESYRE simulation framework. Each domain is defined using different models of computation. The electrical network is defined using electrical modeling components (based on algebraic differential equations), the thermal effects are modeled using Simulink continuous time components, the Rhapsody controller is modeled using Rhapsody's discrete event, and the components defined within DESYRE are modeled using SystemC's discrete event and timed dataflow processes.

4.1.4 Latest Evolution of SysML Extensions

Figure 11 shows the definition of models of computation in the META II integration language. The example shows how models of computation are specified together with their attributes and their generalization relations. The specification of models of computation is organized in profiles. Any model of computation must be a specialization of the MoCC stereotype contained in the General package of the predefined MoCCProfile profile. This ensures that any model of computations has two string attributes by default: wxURi and bxURI. The wxURI attribute specifies a reference to the whitebox definition of the component associated with the given model of computation. For example, “modelica://MyPackage/MySubpackage/MyModel” states that the whitebox associated with the component is a modelica model called MyModel and defined in the modelica package called MyPackage.MySubpackage. Similarly, the bxURI attribute specifies a reference to the blackbox definition of the component associated with the given model of computation. For example, “desyre://Modelica/MyModelFMU” states that the blackbox for hybrid simulation associated with the component is a DESYRE component called MyModelFMU contained in the Modelica library of DESYRE.

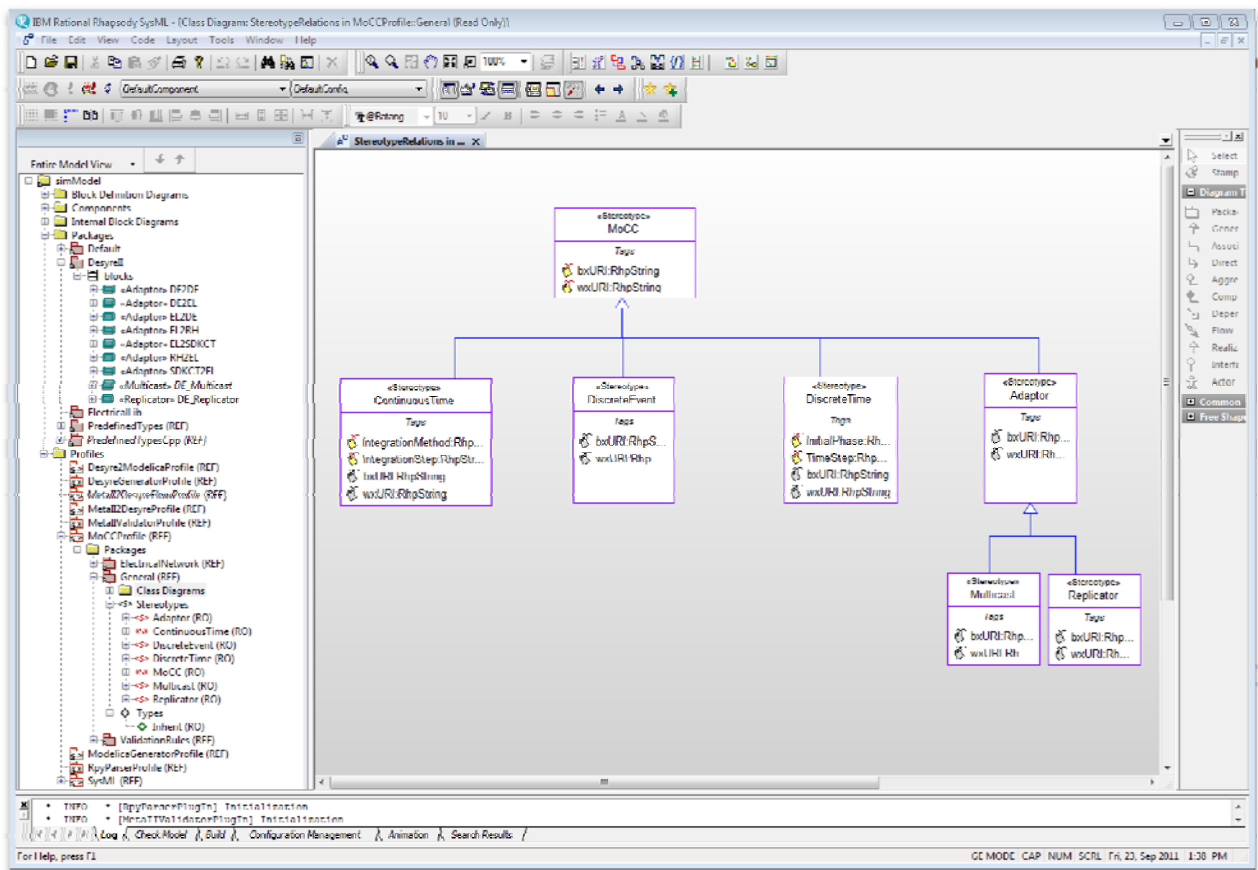


Figure 11: SysML Extensions for META II Semantic Integration

The class diagram in Figure 11 also shows the generalization relations between the MoCC stereotypes. An MoCC M1 is more general than and MoCC M2 if it can be regarded as an abstraction of it, i.e., informally, there is a determinate mapping from the set of all finite and infinite traces of M2 to the set of all finite and infinite traces of M1. The generalization relation allows checking the compatibility of the models of computation associated with the ports connected to the same connector. In particular, the MoCC associated with the input end of the

connector must be equal to or a generalization of the MoCC associated with the output end of the connector.

In the general package the stereotypes for some basic models of computation are specified, as shown in Figure 11, namely ContinuousTime, DiscreteEvent, and DiscreteTime. It can be seen that a specific model of computation may require additional attributes. For example, the ContinuousTime MoCC requires the IntegrationMethod and IntegrationStep attributes, while the DiscreteTime MoCC requires the InitialPhase and TimeStep attributes. The package general defines three additional stereotypes: Adaptor, Multicast and Replicator. The Multicast and Replicator stereotypes are specializations of the adaptor stereotype. These stereotypes are used for components that are only required to adapt different models of computation at the interface, for connecting components defined over incompatible models of computation. This approach enables the target analysis tool, such as DESYRE, to implement the appropriate MoCC adaptation scheme in the most flexible and efficient way. To this purpose, the target analysis tool may provides the META II user with a library package of adaptors, as the DesyreII package shown in Figure 11. In order to avoid creating adaptors for each supported data type, the General package defines an Inherit data type that can be used on the adaptor's port definition. The data type of the adaptor's ports are inherited through the connectors attached to it. The Multicast is used when a single output port is connected to multiple input ports associated with different models of computation. The Replicator is used when a single output port is connected to multiple input ports associated with the same model of computation.

Figure 12 depicts the DESYRE adaptor library, showing in particular the ports of the Electrical to DiscreteEvent adaptor and of the DiscreteEvent Multicast. The Electrical model of computation is specified in the ElectricalNetwork package of the MoCCProfile profile, as shown in Figure 13. Note that the ports of the adaptor are associated with different MoCC stereotypes that are the stereotypes of the MoCCs involved in the adaptation process. Note furthermore that the DiscreteEvent Multicast has an input port associated with the DiscreteEvent stereotype and several output ports associated with different MoCC stereotypes. The Internal Block Diagram in the figure shows how adaptors are instantiated to connect the Controller and the SLD component. The diagram does not specify the connection of several ports of the SLD component, which are specified in distinct Internal Block Diagrams representing different views of the system.

Approved for public release; distribution unlimited

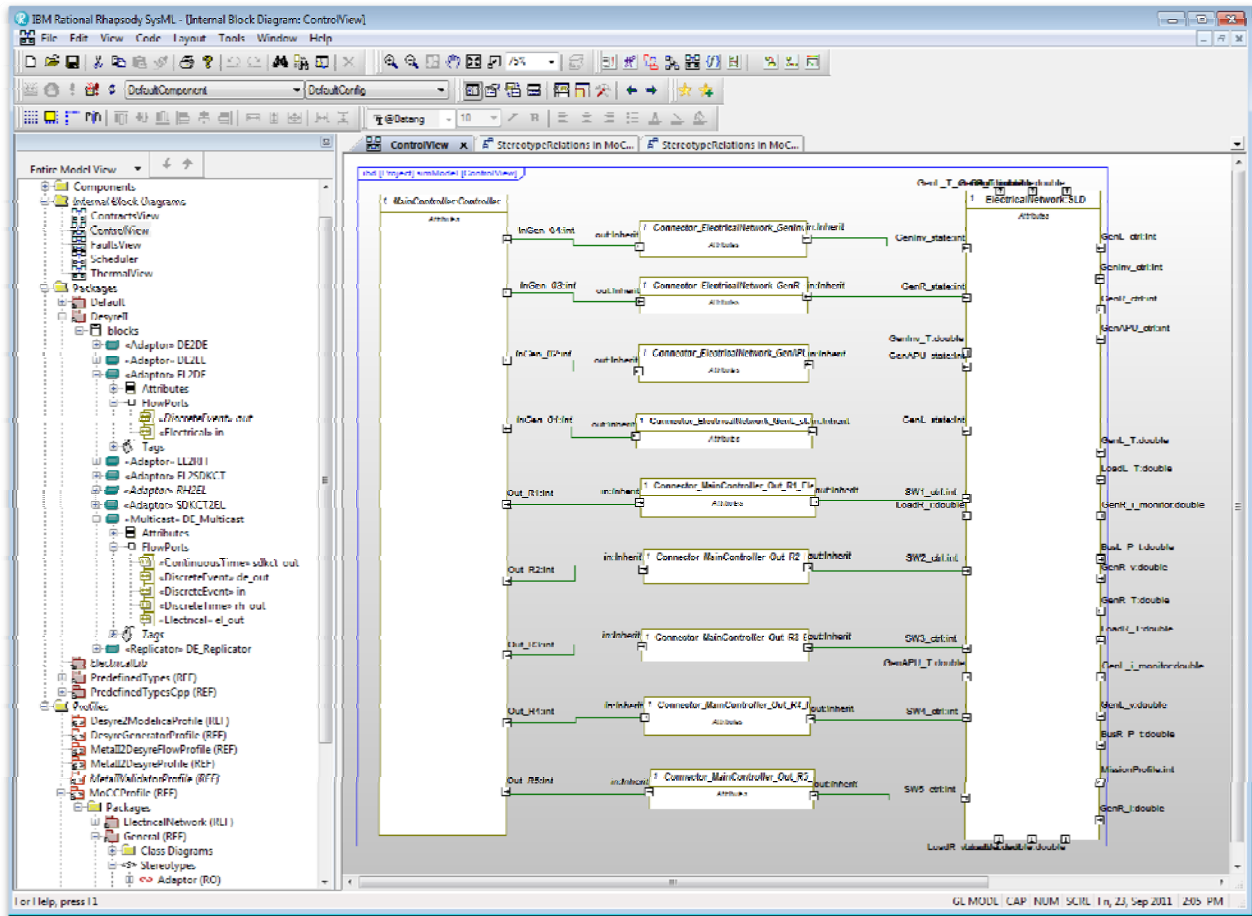


Figure 12: DESYRE Adaptor Library

Figure 13 shows the Electrical model of computation stereotype. This stereotype defines the basic components that can be represented in an electrical network with references to corresponding whitebox definitions of them, which precisely define their semantics. These stereotypes can be used to define an electrical network within the Rhapsody META II integrated model, as shown in Figure 14.

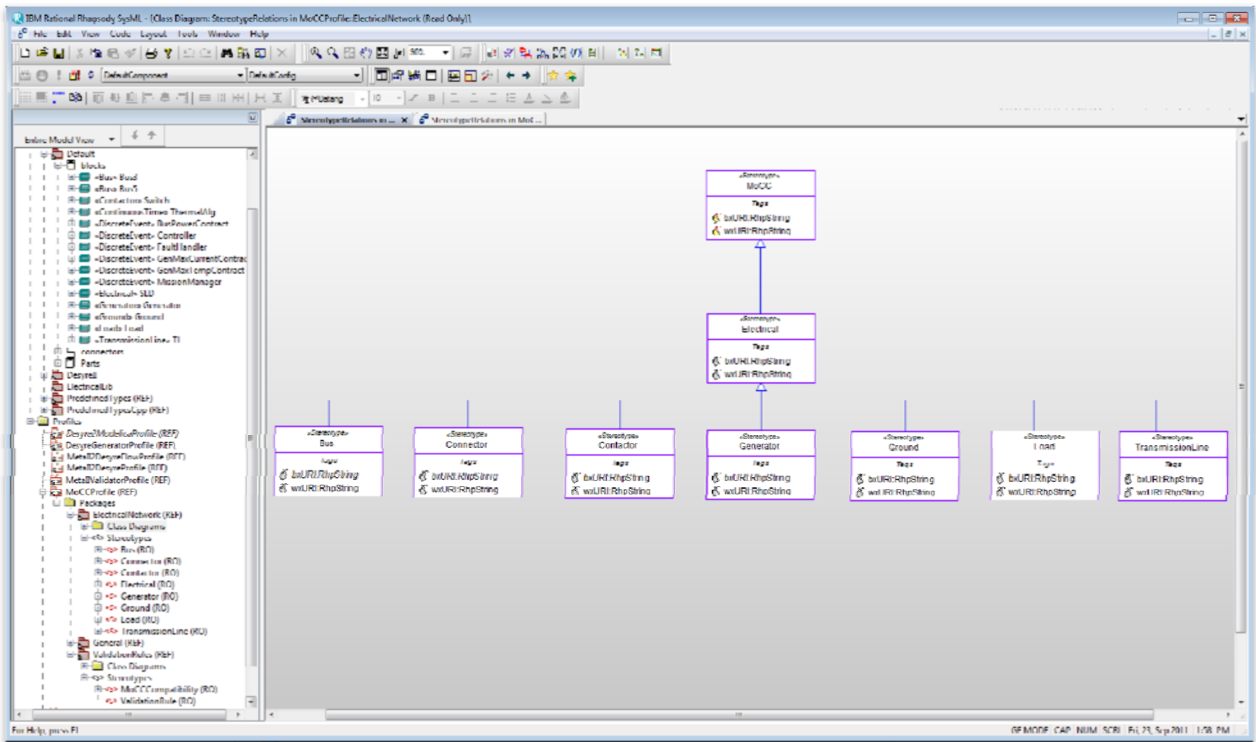


Figure 13: The Electrical Model of Computation

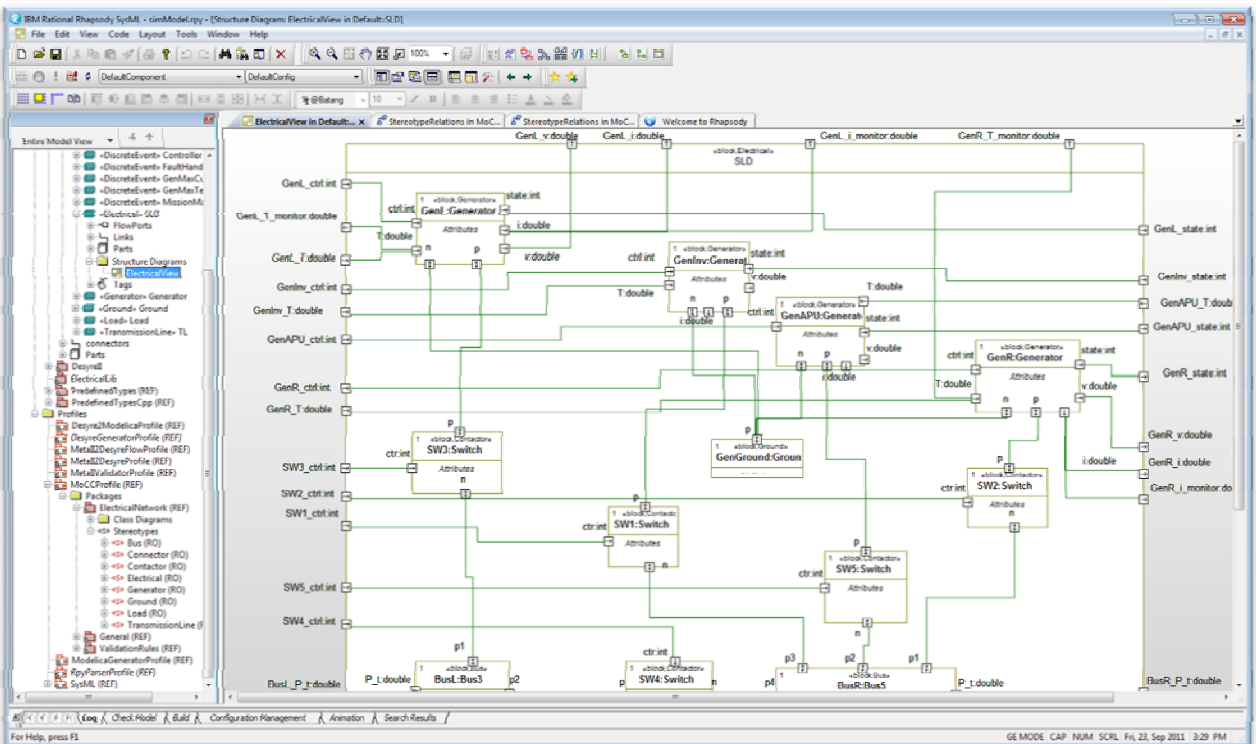


Figure 14: System's Electrical View

4.1.5 META II Tool Flow for Hybrid Simulation

The META II integrated model is built in the target analysis tool, DESYRE, for hybrid simulation purposes using an automated tool flow, shown in Figure 15. The automated tool flow consists of the following tool components:

- *META II internal model builder*: reads the Rhapsody integrated model and builds an internal META II representation of it;
- *META II validator*: validates that all models of computation specified in the integration model be consistent, in particular, for each connector, it validates that the input end be associated with a model of computation that is equal or more general than the model of computation associated with the output end, so to ensure that no information is lost in the connector; the generalization relations between the models of computation are defined in the META II language profile;
- *META II to DESYRE translator*: translates the META II integrated model to a corresponding DESYRE internal representation for hybrid simulation; this representation also contains information of the tools from which models are to be imported;
- *DESYRE to Modelica translator*: the electrical network structure, specified in Rhapsody, is translated into an Modelica internal model;
- *Modelica code generator*: Modelica code is generated out of the Modelica internal model;
- *DESYRE code generator*: DESYRE code for simulation is generated out of the DESYRE internal representation of the integrated model;
- *Modelica importer*: a Modelica FMU corresponding to the electrical network is exported using a suitable modelica tool (e.g., jModelica) and imported as a DESYRE FMI blackbox;
- *Simulink importer*: a Simulink S-function corresponding to the thermal component is exported using RTW code generator and imported as a DESYRE S-funciton blackbox.

All components above, except for the Simulink importer, are accessible from the Rhapsody application as plug-ins.

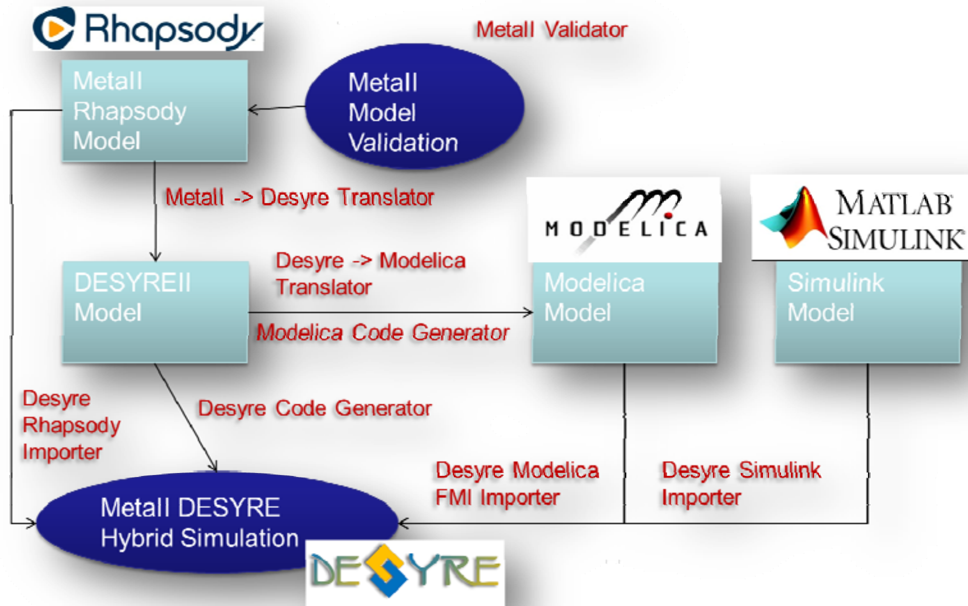


Figure 15: Automated Tool Flow for Hybrid Simulation

4.2 Black Box Integration

4.2.1 Overview

This is the simplest of all integrations and is useful for hybrid simulation. In this approach each of the integrated models is co-simulated with the rest of the system using the original solvers of the tool it was modeled in. The TSM based simulation framework connects to the black box model using a specially defined API that satisfies the selected operational semantics, number of operational semantics will be supported by the simulation framework.

For example the model side API may have methods like get(port), set(port, value), setNextEventTime(time), getTime(), sendEvent(port, time,value), getNextEvent(port), isUpdated(inputPort /inputOutputPort), isChanged(inputPort/inputOutputPort), initialize() etc. The analysis side API may have methods like getTime(), sendEvent(port, time, value), isUpdated(outputPort/inputOutputPort), isChanged(outputPort/inputOutputPort), etc. Examples for such APIs are Ptolemy actor based API, Functional Mockup Interface (FMI), etc.

4.2.2 Modeling

In a design ecosystem where a black-box multi-physics approach is applied several roles can be identified:

- A Systems Engineer is building the architecture that defines how blocks are interconnected – top level model.
- The top level model also contains information needed for the hybrid simulator to function correctly. This information includes:
    - MoCCs of components and their ports
    - Parameters required by the respective MoCC. These parameters (or constraints) will generally have default values so that the Systems Engineer will not

    - necessarily need to understand the intricacies of the MoCC integration. She will have the option to modify them, should her expertise permit.
    - Source of functional block – what tool/language is the block implemented an and the means to access it (filename, URL)
- The individual blocks are modeled in their respective environments. Wrappers are automatically generated for each of the blocks satisfying the API vs. the simulator.
- In addition to the regular blocks there are special "Algebra" components. These components are generic computing tools that can accept an architecture with necessary parameters and simulate or compute a set of values (signals) or metrics. The addition of such a component into a model will incur a pre-processing phase in which a generic Algebra component is transformed into a model specific Algebra component by using information from the top level architecture as modeled by the Systems Engineer. These components are then interconnected with model elements, also as a pre-processing act.

4.2.3 Integration Process

A model from any tool can be integrated into to the framework by:

- Defining the operational MoCC for the tool (see above)
- Defining the API between the MoCC and the analysis tool
- Implementing the API on the modeling tool side as a wrapper around the model
- Implementing the API on the analysis tool side – essentially wiring the API to TSM
- When modeling, augmenting the model with additional information required by the integration mechanism (wrapper, API, analysis) to function correctly

4.2.4 Pros and Cons

The relative simplicity of using this approach is in that:

- There is no need to define the entire source modeling language semantics, since it is embedded in the modeling tool solver.
    - The actual semantics of the tool may be undocumented and hidden making their formal definition a very difficult task.
- APIs can be reused with or without modifications for various tools.
- The API on the modeling side is implemented by the tool vendor who is intimately familiar with the tool.
- Models are running within their native environments making integration easier.

The cons are that:

- The approach is only useful for simulation.
- Semantics of the original modeling tools is not always the best.
- Less functional traceability.

4.2.5 EPS Usecase

See the description of black box integration of Rhapsody component in Appendix C.

4.3 Contracts

4.3.1 Specification Layer Overview

Lingua Franca specification layer is defined above Lingua Franca modeling layer. It consists from asserts, contracts and property process. In this section we provide short description of the basic concepts.

4.3.1.1 Asserts

We use assets as basic building blocks of the contracts. We separate between structural and behavioral asserts. Structural asserts define constraints on model structure, an example of structural asserts is *"Block EPS system should include not more than 2 Generators"*. Behavioral asserts deal with system behavior, i.e. attributes and ports values of the model. Here are some examples behavioral asserts "*weight is below 10 kg*" or "whenever event E1 is received event E2 is sent with 10ms" .

4.3.1.1.1 *Structural asserts*

Structural assert represent a set of structural models. For this propose any language that can constraint structure of the models can be used, for example OCL. Lingua Franca developed special concise modeling graphical language to specify such a structural constrains. For more detailed description see Section 0. Figure 16 is an example of a concise modeling structural assert. This assert accepts any architecture that can be resulted from expand of this concise model.

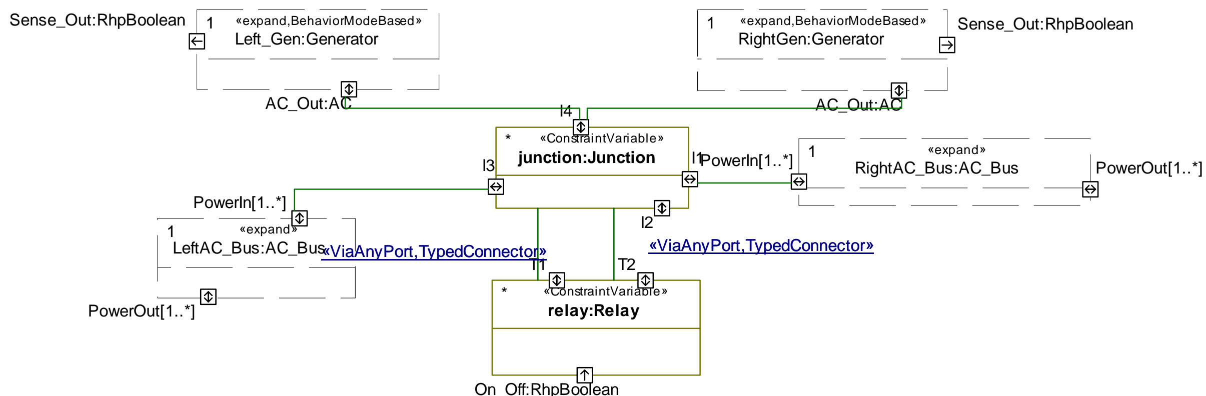


Figure 16 - Concise Structural Assert

Here are the main usages of structural asserts

- check validity of a design model
- provide constraints for design space exploration and optimization procedure for model synthesis
- provide conditions for other (static or dynamic) types of asserts

4.3.1.1.2 *Behavioral assert*

Behavioral asserts provide constraints on system behavior. Below we provide precise definition of the system behavior using Tag Signal Model. Informally system behavior consists of the evolution of all attributes and ports values over the time of system performance. It is convenient to define subset of behavioral assert which is defined above attributes/ports with fixed values. So the previous example of the behavioral assert "*weight is below 10 kg*" is a static assert for

constant attribute weight or if system has only a single time tick, which is the case when different “snapshot” scenarios are analyzed.

Behavioral asserts can be expressed in convenient for a developer languages like PSL, CSL and PRISMATIC. The only requirement for such specification languages is that their semantics need to be defined above system behaviors.

Here are the main usages of behavioral asserts:

- monitors for verification by simulation
- provide constraints for design space exploration and optimization procedure for model synthesis – mainly static asserts
- formal verification

4.3.1.2 Contracts

We use contracts framework as a formal framework for requirements specification and define contracts operators to enable a requirements processing flow [6][7].

Contracts are built from three categories of asserts: strong assumption asserts, weak assumption asserts and guarantee asserts. Not formally, strong assumption asserts must hold, otherwise the component is be not valid. The weak assumption asserts are the condition for which the guarantee asserts holds.

Contracts can be associated to a component or to a component library (platform). Contracts associated to a component library enable effective reuse of the library contracts. This is done by importing all contracts which are related to these components into the containing component.

An example of such platform contract guarantee is “*TRU should be connected to AC bus through AC_port and to DC bus through DC_port*”. So if TRU part is used in EPS block, then EPS block automatically imports above mentioned platform contract guarantee and wiring of TRU to AC and DC buses. In this way platform specifications are imposed on the blocks which use components from the platform.

4.3.1.3 Property processes

Given a specific assert, there might be a need for an additional auxiliary process to compute the required properties. Such a process is called property process. The main requirement for the property process is that its outputs are connected only to other property processes or asserts. Namely no output of property process should be connected directly to ports or attributes of the design model. A property process can be viewed as a part of the corresponding assert and belongs to the specification layer.

Events and conditions in real systems are rarely plain, but rather they are a complex logical combination of many factors and parameters. The specific processes that together assemble a condition in a contract monitor are called Property Processes. In Figure 17 we can see a complex monitor combined from several property processes.

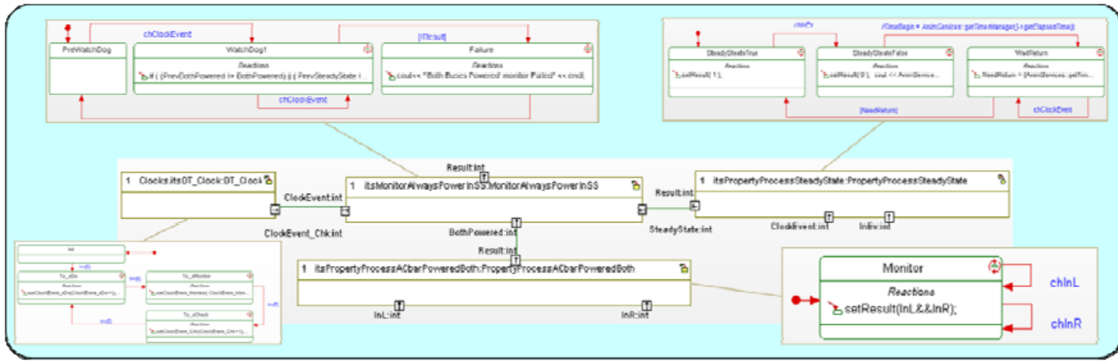


Figure 17 - Complex Monitor

4.3.2 Contracts Meta-model

Meta-model in Figure 18 describes relation between "design" block, contracts and asserts. Meta-classes ContractBlock and AssertBlock are used to define classes of contracts and asserts. Meta-classes ContractPart and AssertPart are used to define instances of contracts and asserts classes.

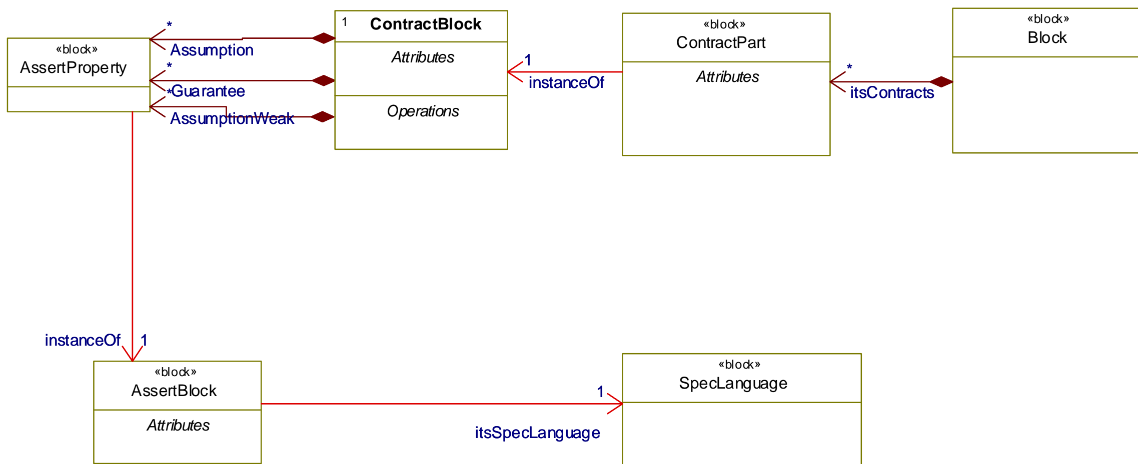


Figure 18 - Contracts Meta-model

Lingua Franca specification layer was developed as a profile of SysML. Each contract is built from three assertion formulas: a (strong) assumption, a weak assumption and a guarantee. The formulas language is specified as parameter of the assertion, examples of the languages are PSL, CSL and PRISMATIC.

There are two main usages of behavioral guarantees for simulation: monitoring and execution. In case of monitoring, guarantee "monitors" component behavior, and signal whenever implementation doesn't satisfy contract (this of course also depends on the assumptions asserts). In case of the execution mode, a guarantee synthesize behavior according to the guarantee asserts. Of course behavior synthesis from contracts is a hard problem, but for simple asserts it can be performed.

Contracts flow ports should be connected to same direction flow ports of the contract's block or its public parts. In case the contract is used in monitor mode, all contract ports become input ports (the original directions indicate the type of block port). In a case the contract is used in execution mode, the original directions of the ports are used.

There should be no port which output defined by both an implementation part of the block and “executable” contract.

Local variable in asserts/contracts is defined as attributes. An external variable in asserts/contracts can be bounded to external variable using constrained parameter, while a constrained parameter name used internally in assert/contract as a variable. In the case that a contract guarantee specified in the constraint section of the block then any variable in scope of the block can be used by name.

Figure 19 is example of a ContractBlock, which includes strong assumption and guarantee asserts. The assumption is in OPL language, which constrains StrikePhaseDuration variable. The “MaxStrikeDuration” of the assumption AssertProperty is bound to “maxStrikeDuration” of the ContractBlock (note that the names of the bound variables need not to be the same). Note that in guarantee AssertProperty there is a usage getAllPartsByType model interrogation function.

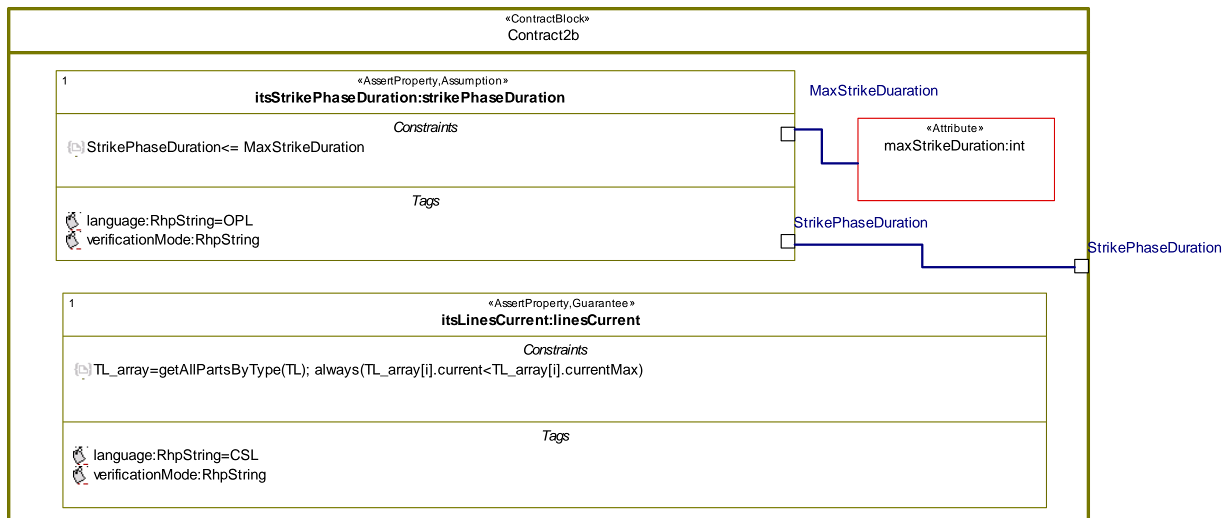


Figure 19 – Contract Block Example

Figure 20 is an example of usage of a contract and a property process which computes StrikePhaseDuration. Note that in ContractPart a concrete value (10000) of maxStrikeDuration is set.

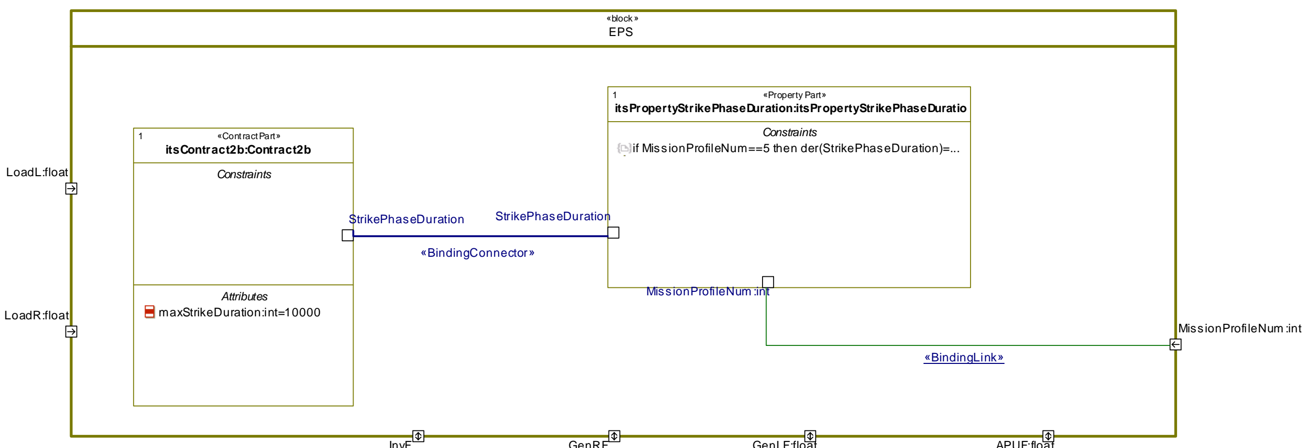


Figure 20 - Contract Usage Example

To simplify the syntax, a single assert is defined to be a contract with both strong and weak assumption equal to TRUE and the assert guarantee equals to the single assert.

4.3.3 Semantics of Dynamic Asserts

4.3.3.1 Model behaviors

Semantic of the dynamic layer is based on Tagged Signal Model (TSM) [1] and its refinement [2][4] where more detailed definitions can be founded. A ***behavior*** is defined as set of events, where an event is a tuple of ***signal name***, ***value, and global tag***. Each signal has an associated ***local tag*** (similar to Clock concept of MARTE [3]) and all events of a specific signal should have a unique local tag. ***A global tag*** is a tuple of local tags $\tau_g=(\tau_{l1}, \tau_{l2,}\ldots)$. A ***process*** defines constraints on signals associated with it.

We map the basic concepts of TSM and Lingua Franca as following. All ports and attributes of a block in Lingua Franca are signals in TSM, where corresponding clocks are defined for each port/attribute. An event tuple for a port/attribute defined as the port/attribute full name including the context, the value of the port/attribute and the current clock tick of the port/ attribute. Block can be viewed as a process that constraints the behavior of related signals. When we connect ports together there are two options:

Direct connection – in this case signals are equal, i.e. whenever event occurs on one port it simultaneously (have same global tag) occurs on the connected port. This is equivalent to SysML flow ports semantics.

Adapted connection – in this case, the connection can be viewed as additional adaptor block/process. Library of such adaptors should be provides. Adaptors can be used to define different types of interactions (for example queued interaction) or as a way to connect different MoCCs (for example sampler to convert continuous time to discrete time MoCC).

4.3.3.1.1 *Behavior discrete steps*

Given a behavior σ we define the behavior discrete steps as following.

Let E_D be a set of discrete events in σ:

$$E_D(\sigma)=\{e|\exists s\in\sigma, \text{ s is discrete signal, } e\in s\}$$

Let T_D be a set of global tags of the discrete events of the behavior σ:

$$T_D(\sigma)=\{\tau|\exists e\in E_D(\sigma),\tau=\text{globalTag}(e)\}.$$

T_D has a total order ($\leq$), so the previous (τ-1) and next (τ+1) operators are well.

As a consequence T_D can be represented as an ordered list of the unique global tags (*discrete clocks starts from clock 0)*:

$$T_D= \{\tau_0,\tau_1,\ldots\}$$

A global tag τ includes all local tags, so we use operator localTag(τ,s) to denote the local tag of the global tag τ corresponding to signal s.

For each global tag in $\tau\in T_D$ we define value function V(τ) which sets a value for each signal in correspondence to its local tag:

$$V(\tau)(s) = v \Leftrightarrow \exists e \in s, \tau' \in T(\sigma) : e=(\tau', s, v), localTag(\tau', s)= localTag(\tau, s)$$

i.e., a value of the particular signal remains constant in intervals between corresponding local tags firings.

Active signals $S_a(\tau)$ for a global tag $\tau \in T_D$ defined as signals that their clocks are fired at τ:

$$s \in S_a(\tau) \Leftrightarrow \tau=\tau_0 \textbf{ or } localTag(\tau,s) > localTag(\tau-1,s).$$

The discrete step $\zeta(\tau)$ for $\tau \in T_D$ defined as a tuple $\zeta(\tau)=(\tau, S_a(\tau),V(\tau))$. We denote Z as set of all discrete steps. Finally, a discrete run $R(\sigma)$ for a behavior $\sigma \in \Sigma$ is defined as

$$R(\sigma)=\{\zeta(\tau), \tau \in T_D(\sigma)\}$$

with a total order imposed by T_D

$$R(\sigma)=\{\zeta_0, \zeta_1, \ldots\} \text{ where } \zeta_i=\zeta(\tau_i)$$

4.3.3.2 Behavioral assertions on behavior discrete steps

Let $R(\sigma)=\{\zeta_0, \zeta_1, \ldots\}$ be a discrete run for a behavior $\sigma \in \Sigma$, where $\zeta_i=(\tau_i, S_a(\tau),V(\tau))$ is the ith discrete step of the behavior. The ith suffix of a discrete run R is defined as

$$R_i = (\zeta_i, \zeta_{i+1}, \ldots.)$$

R_i satisfy φ_B ($R_i \models \varphi_B$), where φ_B is (not temporal) Boolean formula means that a discrete step ζ_i satisfies φ_B ($\zeta_i \in [[\varphi_B]]$)

R_i satisfy φ_{LTL} ($R_i \models \varphi_{LTL}$), where φ_{LTL} is linear temporal logic formula means that a discrete step R_i satisfies φ_{LTL} ($R_i \in [[\varphi_{LTL}]]$)

Here are some typical examples of temporal logic expression and the corresponding semantics.

Always statement example:

$R \models$ always(φ_{LTL}) iff ($R_i \models \varphi_{LTL}$) for i=0,1,…

Strictly within statement (within_s) example relative to clock C_i:

$R \models$ whenever(φ_1) occurs (φ_2) within_s (t,Ci) iff for i=0,1,… $(\zeta_i \models \varphi_1) \Rightarrow \vee_{i \le j \le k} (\zeta_j \models \varphi_2)$, $k=\max\{m: Ci(\zeta_m)-Ci(\zeta_i)<t\}$

Not-strictly within statement (within_ns) example relative to clock C_i:

$R \models$ whenever(φ_1) occurs (φ_2) within_ns (t,Ci) iff for i=0,1,… $(\zeta_i \models \varphi_1) \Rightarrow \vee_{i \le j \le k} (\zeta_j \models \varphi_2)$, $k=\max\{m: Ci(\ _m)-Ci(\zeta_i)\le t\}$

4.3.4 Contract Algebra

Contract is built from strong assumption, weak assumption and guarantee $C=(A_s, A_w, G)$. Whenever any one of them is missing, it is replaced by TRUE value. Next we define number of the relation between contracts and between contracts and design models following [6][7]. In the following relations we use contracts in canonical form $C=(A_s, TRUE, \neg A_w \cup \neg A_s \cup G)$, so we specify the relations using only canonical A and G in the canonical form.

Implementation (satisfaction) $M \vDash C = M \in G \cup \neg A$

Refinement/dominance $C_1 \preccurlyeq C_2$ iff $M \vDash C_1 \Rightarrow M \vDash C_2$

Conjunction $C_1 \wedge C_2 = \{M: M \vDash C_1 \text{ and } M \vDash C_2\}$

Product/composition $C_1 \otimes C_2 = \{M: M = M_1 x\, M_2, M_1 \vDash C_1 \text{ and } M \vDash C_2\}$

4.3.4.1 Textual languages

Sometimes it is convenient to specify behavioral asserts in textual form. Example of a widely used textual specification language is PSL. The usage of PSL is limited because of high complexity of the language

A special type of textual language is pattern based textual language. CSL [5] is an example of pattern textual language with a number of predefined temporal patterns. CSL defines 8 patterns:

P1. whenever [E] occurs [C] holds during following [I]
P2. whenever [E1] occurs [E2] implies [E3] during following [I]
P3. whenever [E1] occurs [E2] does not occur during following [I]
P4. whenever [E] occurs [S] within [I]
P5. [C] during [I] raises [E]
P6. [E1] occurs [N] times during [I] raises [E2]
P7. [E] occurs at most [N] times during [I]
P8. [C] during [I] implies [C1] during [I1] then [C2] during [I2]

CSL demonstrate "freedom-from-choice" approach and is more suitable for system engineering domain. For each of the patterns semantics above behavior discrete steps should be provided (see behavioral asserts subsection for typical examples).

4.3.4.2 Graphical languages

A graphical way to specify asserts is sometimes more convenient than a textual form. While any graphical specification language can be used to specify assets, we provide two examples of such languages: visual constraints described in Section 4.4.5.5 and LF automata described below.

LF automata **A** defined using tuple $<Q, V, Z, \delta, q_0, \{F1, F2, \ldots FN\}, G>$

- The finite set Q is called the *states* of *automata*
- The finite set V is called the *internal variables* of *automata*
- Z is a set called the *alphabet* of **A**.
- δ_t: $Q \times V \times Z \rightarrow Q \cup \{\perp\}$ x V is a function, called the *triggered-transition function* of **A**.
- δ_c: $Q \times V \times Z \rightarrow Q \cup \{\perp\}$ x V is a function, called the *conditioned-transition function* of **A**.

- q_0 is an element of Q, called the initial state.
- **F1,F2,…FN**$\subseteq Q$ are the *acceptance condition sets*.
- **G**$\subseteq Q$ is the *forbidden set*.

Automata **A** alphabet Z is set of possible discrete steps.

Automata A processing for input (ζ_0, ζ_1,…) is defined as following

```
Step 1.  Initialization: i=0, q=q0
Step 2.  ζ=ζi                                    %set next step to process
Step 3.  (tmp_q,tmp_v)=δt(ζ)                     %compute triggered transition
Step 4.  If (tmp_q == ⊥) goto 8 else q=tmp_q,v= tmp_v %perform transition if it is valid
Step 5.  (tmp_q,tmp_v)=δc(ζ)                     % compute conditioned transition
Step 6.  If (tmp_q ==    ) goto 8 else q=tmp_q,v= tmp_v %perform transition if it is valid
Step 7.  goto 5                                  %try next conditioned transition
Step 8.  i=i+1, goto 2                           %advance behavior discrete step
```

Given a behavior discrete run R=(ζ_0, ζ_1,…), the first transition of **A** will be done using the *triggered-transition function.* If δ_t returns ⊥ then **A** remains at original state and start processing a new behavior discrete step, otherwise **A** advance to new state and updates internal variables.

Next, a sequence of transitions is done using the *conditioned-transition function* till it returns ⊥.

LF automata is actually performs run to completion execution, when only first transition for the step can use a *triggered-transition function* and all following transitions are done using *conditioned-transition function*.

LF automata **A** accepts exactly those runs in which visit each *acceptance condition sets* infinitely often and never visit forbidden set.

Syntax of LF automata is similar to state-chart, where transitions annotated with trigger[condition]/action. Transitions with a trigger define *triggered-transition function*; transitions without trigger define *conditioned-transition function.* Both the trigger and the condition are predicates over a discrete step and internal variables. The actions define transformation of internal variables.

Figure 21 is an example of LF automata which represents CSL pattern #4 "Whenever [E1] occurs [C] within [Eb, Ee]". This LF automata has no internal variables, has no acceptance states and a single forbidden state is "Failure". "Idle" is an initial state. Let analyze some of the transitions:

- Transition from "Idle" to "WaitEb" state is "E1[!Eb]". It belongs to *triggered-transition function,* because it has a triggered (an expression before []). The predicate E1[!Eb] is true for all behavior discrete steps where E1 event is "active" and Eb is not "active"
- Transition from "Check1" to "AllRight" state is [C]. It belongs to *conditioned-transition function* because it has no triggered. Condition C is any predicate above behavior discrete steps.

Approved for public release; distribution unlimited

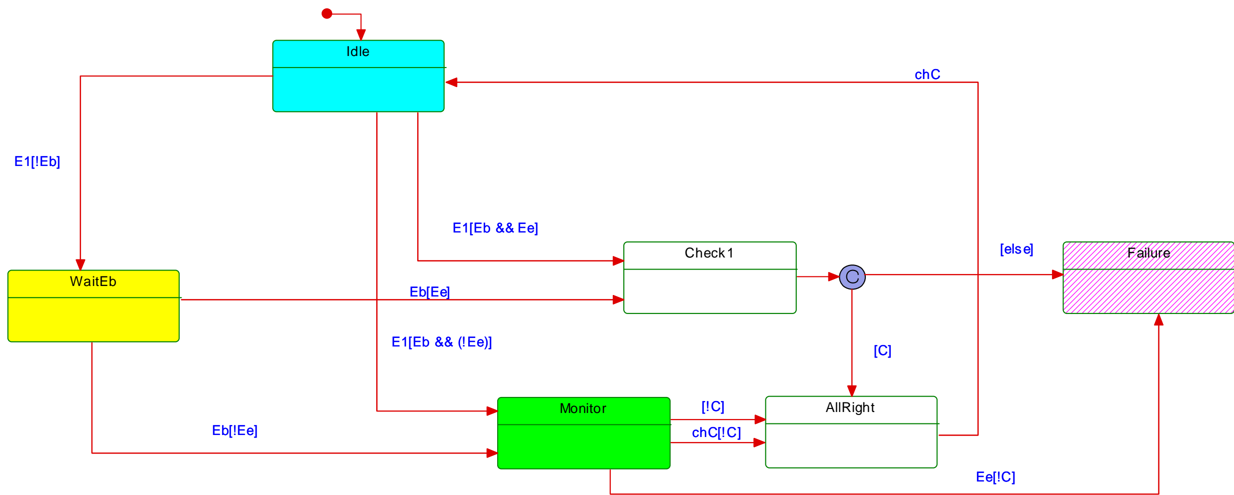


Figure 21 - Example of LF Automata

4.3.4.3 Property process on continuous signals

In previous sections the semantics of dynamic asserts was defined for behavior discrete steps. To deal with conditions which depends on continuous signal we introduce property process which generate an event whenever the required condition starts/ends. This technique enables the translation of conditions from continuous signals to discrete events. See Figure 22 for elaboration.

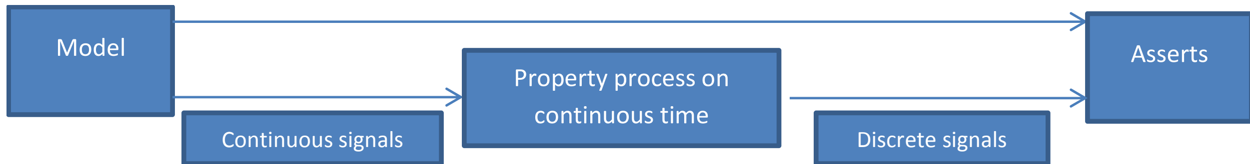


Figure 22 - Properties of Continuous Signals

4.3.5 Applications

4.3.5.1 EPS requirements as contracts

For EPS usecase the following list of requirements was defined:

1. Satisfy power quality under all conditions (voltage levels, distortion, etc) (LEVEL 1,2)
2. Load capacity of any component shall not be exceeded in steady state (LEVEL 1)
3. AC buses shall never be paralleled (LEVEL 0)
4. Do not close into a dead bus (faulted bus) (LEVEL 0)
5. Keep all buses powered in the case of 1 failure beyond Minimum Equipment List (MEL) (LEVEL 0)
6. Keep a certain set of critical buses powered in the case of two failures beyond MEL (LEVEL 0)
7. Breaks on the AC side < x ms (LEVEL 2)
8. Breaks on the DC side < y ms or No Break (LEVEL 2)
9. Only part of the aircraft shall be powered when in maintenance mode (e.g. left DC side) (LEVEL 0)
10. Do not parallel TRUs in steady state (transient conditions are OK) (LEVEL 0)
11. Minimize contactor actuations during transfers (LEVEL 0)
12. Use the APU only during take-off and landing or emergency power (LEVEL 0)
13. No single point failure shall cause the loss of all critical busses (LEVEL 0)
14. 30 minutes of power to a select few loads shall be maintained in the event of loss of all mechanical sources (LEVEL 2)

Approved for public release; distribution unlimited

The flow of transformation of natural language requirements to formal contracts is depicted in Figure 23:

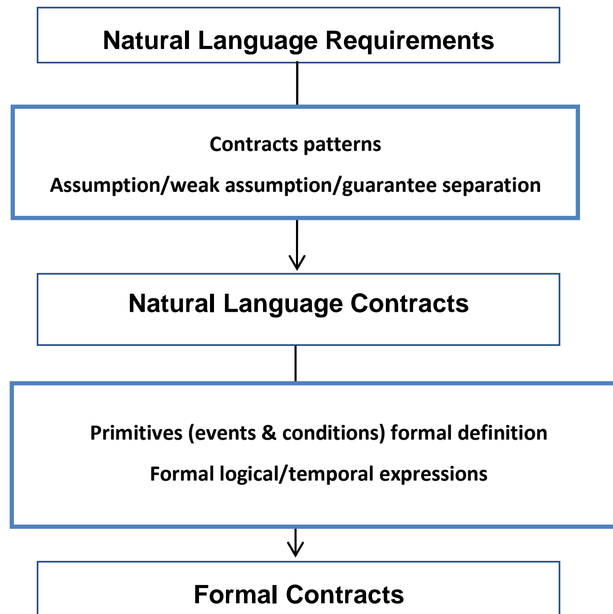


Figure 23 -Requirements Transformation Flow

We start from natural language requirements. Next, we re-express the requirements using predefined patterns and regroup them to strong assumption, weak assumption and guarantee asserts categories. We get what we call natural language contracts. Next, all used terms need to be represented using well-defined formal primitives.

We'll demonstrate this flow for "Keep all buses powered in the case of 1 failure beyond MEL" requirement. The corresponding natural language contract guarantee is "1 failure beyond MEL=>all buses are powered", while assumptions are set to be TRUE. Natural language contracts are easier for analysis, so we frequently find some inconsistencies or ambiguities will be discovered in this stage. For example, we might note that the guarantee doesn't specify behavior for cases where there are not failures beyond MEL, so the contract is not activated for this case. Another issue is that it is impossible to keep buses powered in the very moment of the failure – it takes some time to react and modify system configuration. Also there is an inconsistency with another requirement saying that some buses should not be powered during maintenance. So we go back and fix the original requirement to "Keep all buses powered in the case of less than 2 failures beyond MEL at steady state when not in maintenance". The corresponding natural language contract guarantee is "less than 2 failure beyond MEL and not maintenance and steady state => all buses are powered".

To specify formal contracts, the natural language should be replaced by well-defined formal terms. So we need to define formally all concepts that were used in the natural contracts. Here is example of such definitions:

failuresBeyondMEL - function defined on parts failure state (e.g., Generator failure),

failuresBeyondMEL = max(sum(getAllPartsByType(Generator).failed) – 1, 0)

Maintenance – state of the EPS

all_buses_powered()– functions defined on *relay* parts *closed* state and *generator* parts *failed* state and implicitly on parts structural connectivity. Figure 24 depicts a definition of all_buses_powered() function using visual expression syntax.

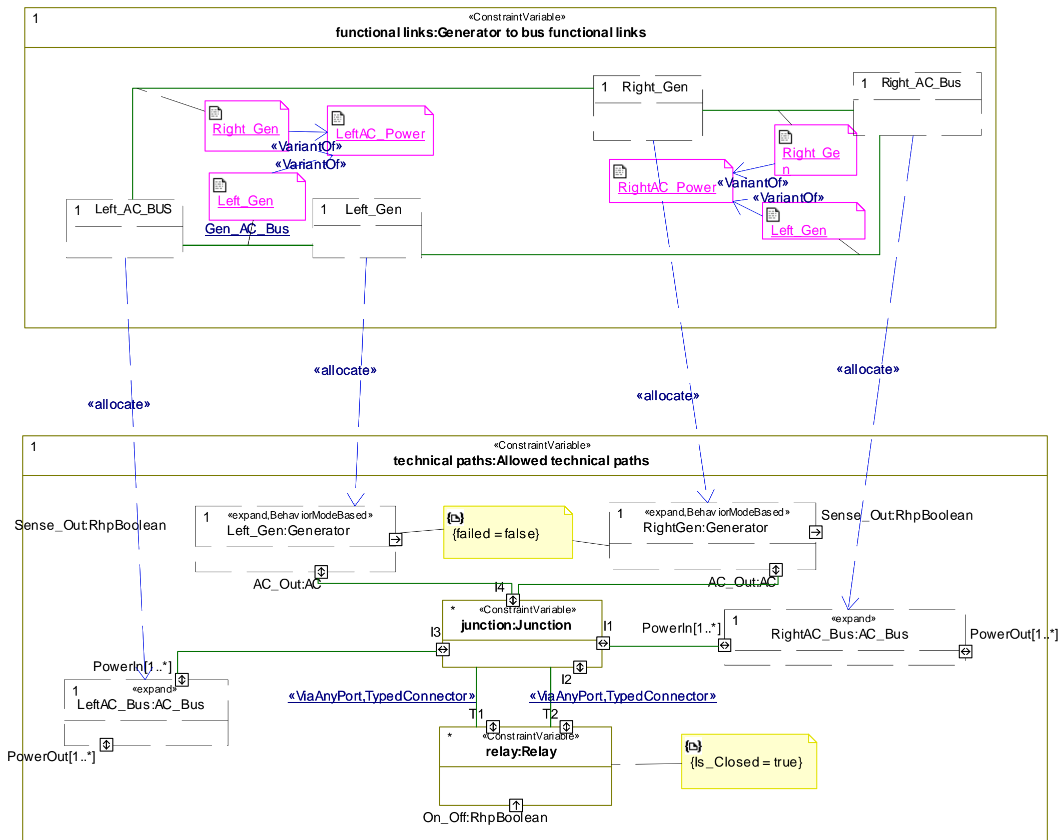


Figure 24 - Visual Expression (all_buses_powered)

Finally, the formal contract guarantee is: “**always[(C** *failuresBeyondMEL()*<**2) and (S not** *maintenance*) **and (S** “*Steady_state*”) **=> (C** *all_buses_powered()*==**true]”.**

After we formulated system level contracts, we specify sub-system contracts.

Figure 25 contains an example of EPS relay contract:

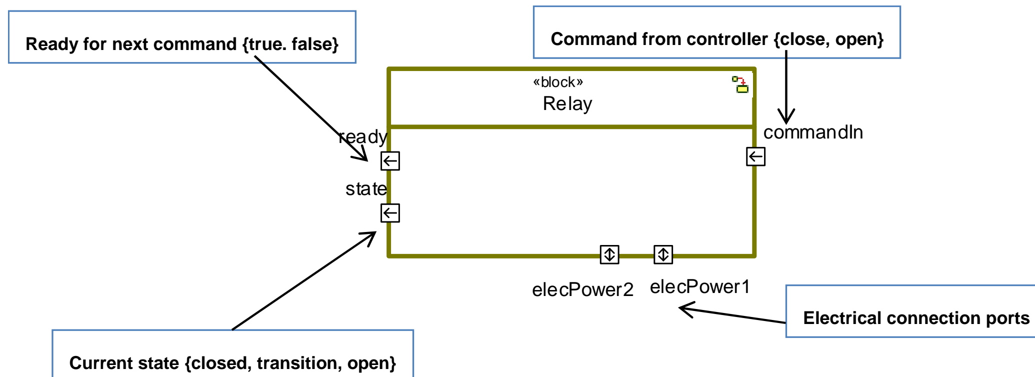


Figure 25 - EPS Relay Contract

Assumptions:

//wait till ready before new command

whenever [**E** *commandIn* com1] **occurs** [**E** *commandIn* com2] **does not occurs during following** [0,**E** *ready==true]*

Guarantees:

//Execute command and switch to ready

[**E** *commandIn* com] **implies** [**E** *state==*com; **E** *ready==true*] **within** (+*Relay_cmd_delay*]

//Change state only once after each command

[**E** *state==*com; **E** *ready==true*] **occurs at most** [1] **during** [**E** *commandIn* com, **E** *commandIn* com]

Figure 26 contains an example of contract for EPS Generator Control Unit (GCU).

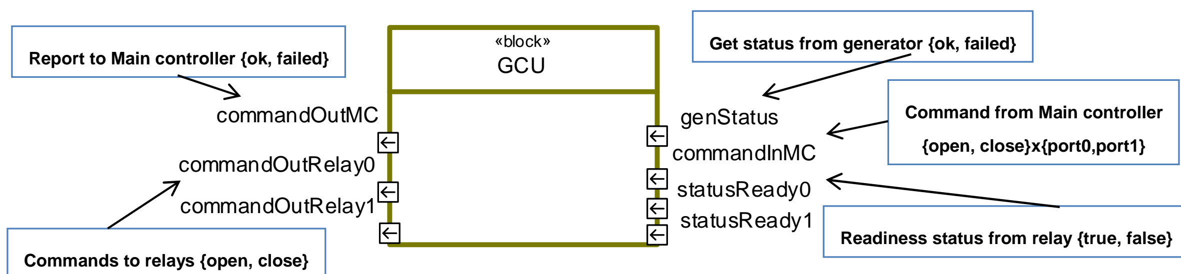


Figure 26 - Generator Control Unit Contract

Assumptions:

//generator doesn't recover

[**E** *genStatus==failed*] **implies** [**C** *genStatus==failed]* **holds forever**

//relay response time

whenever [**E** commandOutRelay[*relay*] *com*] **occurs** [**E** statusReady[relay] ==true] **within** (+*GCU_relay_delay*]

Guarantees:

//GCU response time to generator status event

whenever [**E** *genStatus==status*] **occurs** [**E** *commandOutMC==status*] **within** (+*GCU_send_delay*]]

// perform main controller commands

whenever [**E** *commandInMC* ({com, relay})] **occurs** [**E** commandOutRelay[*relay*] *com* and **C** statusReady[relay] ==true] within (+*GCU_send_delay*]] **unless** [**E** *commandInMC* ({com2, relay})]

//Change state only once after each generator status event

[**E** *commandOutMC*] **occurs at most** [1] **during** (**E** *genStatus*, **E** *genStatus*]

//Command relay only after main controller command

[**E** commandOutRelay[relay]] **occurs at most** [1] **during** (**E** *commandInMC({*,relay}*, **E** *commandInMC{*,relay}*]

4.3.5.2 Contracts as monitors in hybrid simulation

4.3.5.2.1 *Monitors for dynamic contracts*

To create a monitor for behavioral contract (behavioral contract is built from behavioral asserts), strong assumption, weak assumption and guarantee asserts output should be treated differently which we denote by $outA_s$, $outA_w$ and outG, respectively. Here is the processing scenario for the monitors:

Step 1. Initialization of the model
Step 2. Initialization of the monitors
Step 3. Execute all property processes
Step 4. Execute* strong assumption monitor => $outA_s$
Step 5. If $outA_s$ is false – report failure of strong assumption and stop
Step 6. Execute* weak assumption monitor => $outA_s$
Step 7. If $outA_w$ is false , reset guarantee assert and goto 10
Step 8. Execute* guarantee monitor => outG
Step 9. If outG is false – report failure of guarantee and stop
Step 10. Advance to next behavioral discrete step and goto 3

* Execute here means that monitor should check the asserts on the interval from the last step till the current step.

The order of evaluation of property processes should be consistent with dependencies between them. The property processes create results (variables states, or events), which are stored together with the behavior discrete step, to form together a "full configuration", so that property processes/monitors that are executed after other property processes can see / use the results the results. It important to mention that property processes events do not influence the execution of the design model.

4.3.5.2.2 *CSL Patterns translation to monitors*

To enable automatic creation of monitors from asserts specified using CSL Patterns a library of LF Automats were developed. Each LF Automata can be used as a monitor of one of the CSL Patterns (or some of its variants). Here we provide an example for CSL pattern 1 “Whenever E1 occurs [C] holds during following [I], were an interval I defined by events Ebeg, Eend. We consider different cases of the interval closure

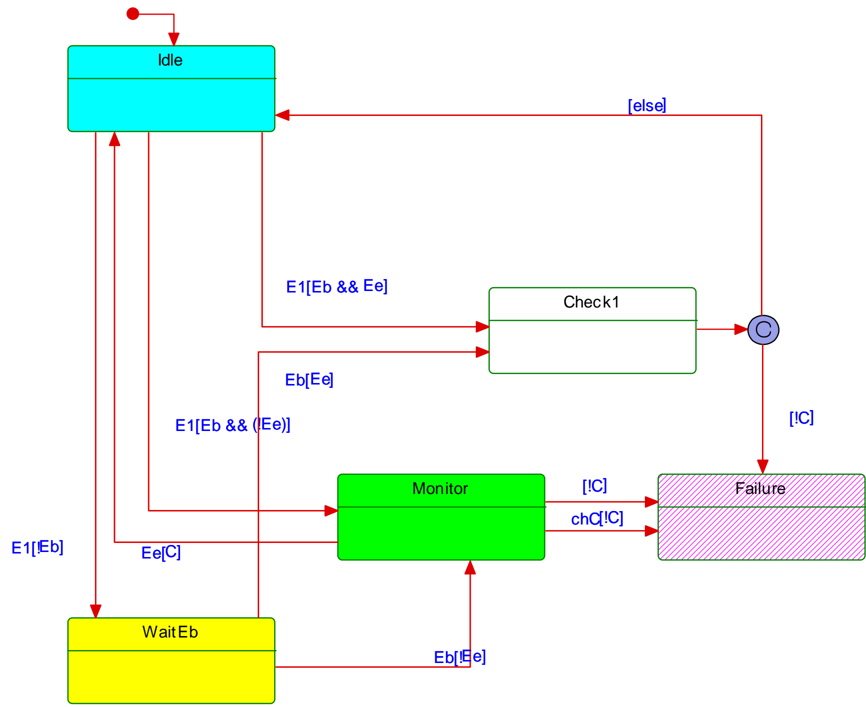


Figure 27 - Case 1 (closed,closed) [Ebeg, Eend]

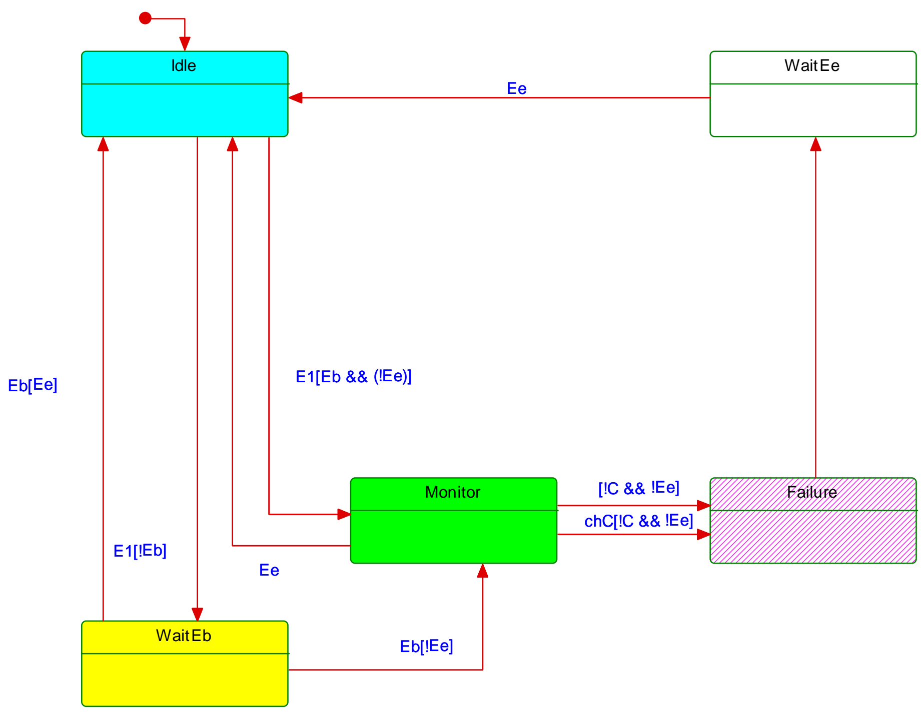


Figure 28 - Case 2 (closed,open) [Ebeg, Eend)

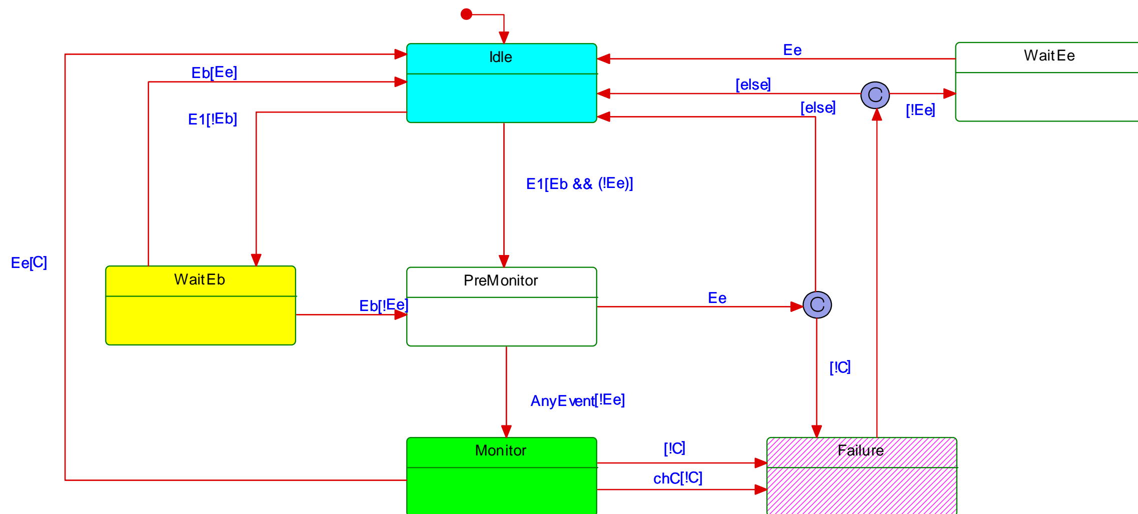


Figure 29 - Case 3 (open, closed) (Ebeg, Eend]

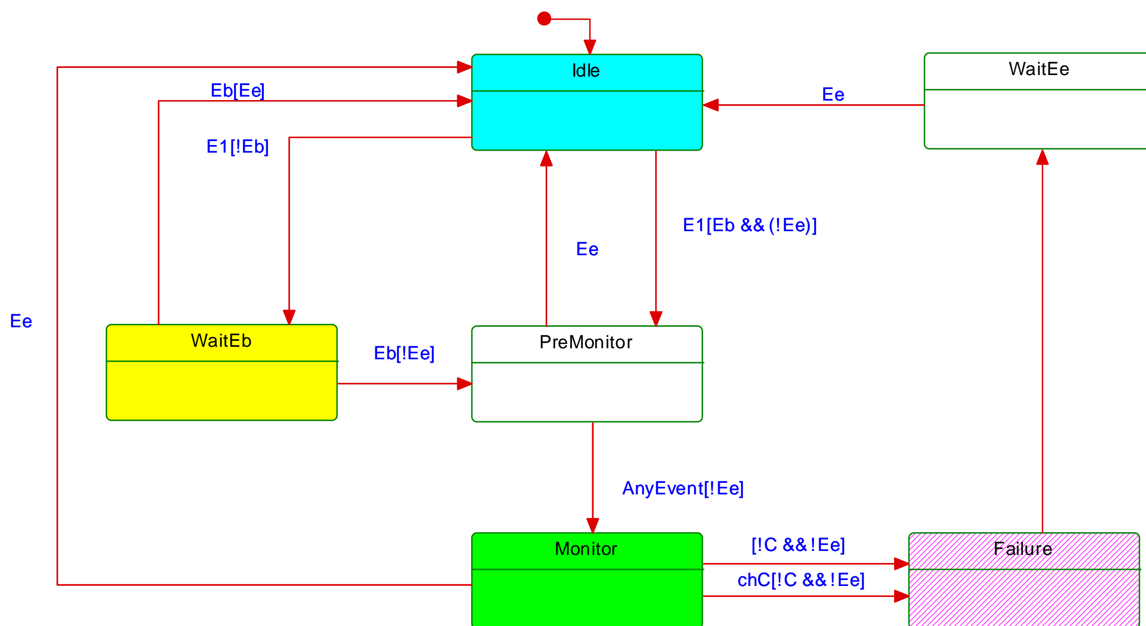


Figure 30 - Case 4 (open,open) (Ebeg, Eend)

All described LF automats has no internal variables, has no acceptance states and a single forbidden state is "Failure". "Idle" is an initial state.

Given such a library, LF automats can be synthesized from contracts. There is still a need to define property processes that generate the required events, so a library of frequently used property processes can be prepared to enable reuse.

4.4 Concise modeling

4.4.1 General

We propose a novel approach we call Concise Systems Modeling. In our approach we do not model the detailed systems, but rather the rules for their composition.

By creating a special profile of SysML with an assortment of stereotypes and tags we give additional semantics to SysML constructs, such as blocks, parts, links and associations. Thus we are able, for example, to use a SysML part to define a set of parts and similarly use a link to define a set of links. We call these sets "prototypes". Since these prototypes are legal SysML parts interconnected with ports and links, we can define the composition (or connectivity) rules for those prototypes.

The explicit prototypes are later instantiated in tables (MS Excel for example), which is much easier and less time consuming than the graphical SysML representation. Some of the fields in the tables are left unfilled and these are later filled by an architecture optimization process.

The attributes of these prototypes are also stereotyped allowing their different treatment in the design process.

An internal block diagram in the technical layer may combine prototypes and normal parts.

The model has three layers – functional, technical and an indexing layer (which in many cases represents geometry). The technical layer is the core of the concise model and all component composition rules are modeled here. The functional layer models system functions mapping them later to specific elements in the technical layer, in essence defining requirements that the technical layer needs to satisfy. The indexing layer, which is not necessarily present, is used to index the technical layer prototypes and facilitates the later expansion or optimization of the model.

In addition to the above constraints, objectives and variable algebras are defined in the model, to be used by the optimization engine. Several new sets definitions have been added to aid in the above definitions.

4.4.2 Planes / Layers

The concise modeling approach has three planes in three different model packages:

- Functional plane (Figure 31 & Figure 32)– serves as the requirements definition for the system architecture.
  - Will generally be executable.
  - May be modeled concisely in some cases, but all parts and links will be explicit (i.e. «inventory»).
  - May be a result of a higher abstraction iteration using the same approach.
  - May have links connecting ports or not.
- Technical plane (Figure 33) – architecture modeling plane. Modeling is based on the requirements of the functional layer. The objects on this plane usually represent real components (or subcomponents) and real flows between them (data, energy). The flows' media are the Typed Connectors, which are parametric and/or behavioral models of cables, shafts, ducts, pipes, wireless channels, etc.

- Indexing plane (Figure 34) – used to index the objects of the technical layer. Sometimes this layer directly represents the geometry of the system and is used as such. For example the instances of this layer may represent possible placeholders for the actual components on the technical plane with the optimization process tasked with finding the right combination of components and their locations.
  Alternatively this layer can be an abstract collection of indices bounded by constraints.
- Mapping – the way to relate one layer to the other. Mapping is done by using the SysML «allocate» dependency. An object on the functional plane can only be mapped to one object on the technical plane, as otherwise there would be ambiguity in the definition. However, any number of objects on the functional plane can be mapped to a single object on the technical plane. If a multiple mapping is indicated, the meaning is that the optimization must select the best mapping subject to constraints and rules.

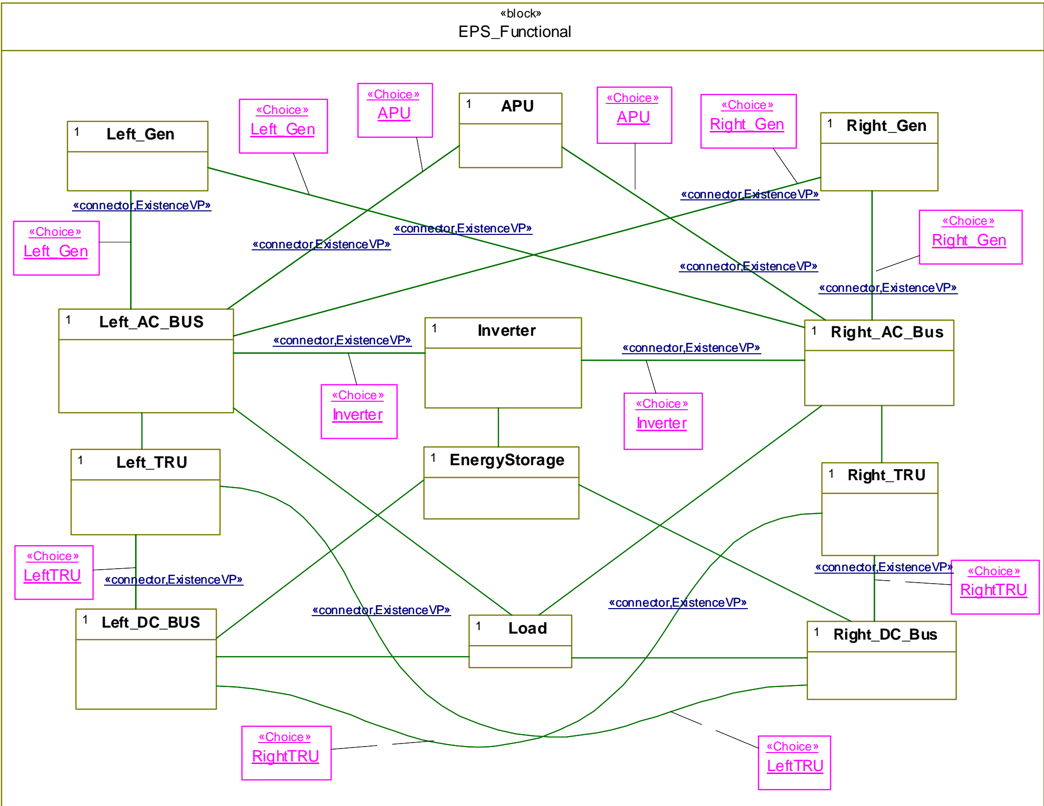


Figure 31 – Primary EPS Functional view

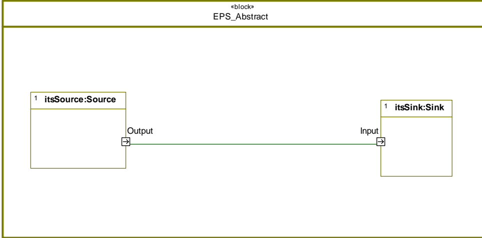


Figure 32 – Secondary EPS Functional view

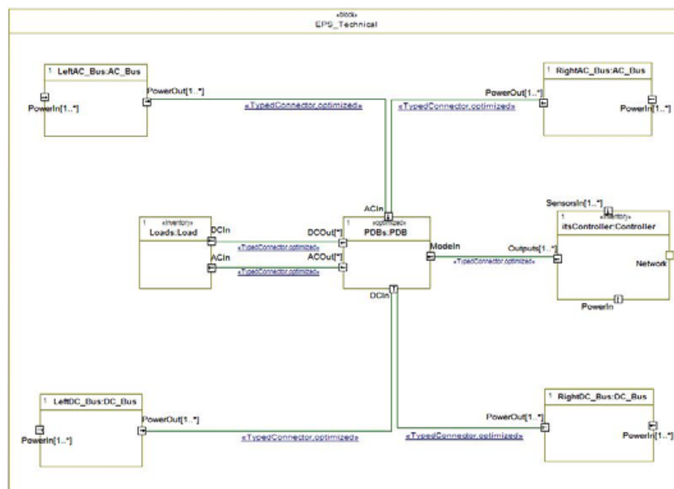


Figure 33 – Secondary EPS Technical View

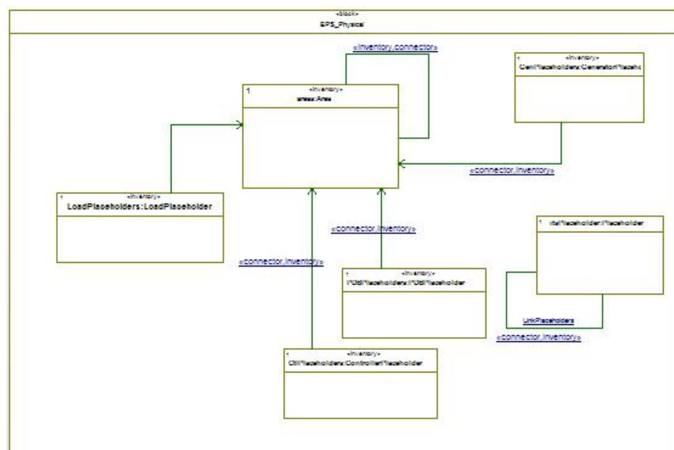


Figure 34 - Index (geometry) View

4.4.3 SysML Extensions Profile

4.4.3.1 «typedConnector»

This stereotype is used to denote a link (between parts or prototypes) that represents a concrete physical object, for example a cable, bolt or shaft. The stereotype contains a tag “type” that points to the block that models the physical object.

4.4.4 Concise Profile

4.4.4.1 «catalog»

Can be applied to blocks and to attributes.

When applied to a block, «catalog» indicates that a block has several variants (subclasses in the expanded model), which are specified in an external table instead of explicitly in the model.

Blocks marked with «catalog» may have attributes marked similarly. These attributes will have different initial values in each sub-class. These values are specified in the table – i.e. each variant will have a row in the table, and each catalog attribute will have a column.

When a normal (non-inventory) part instantiates a «catalog» block, its type will be replaced in the expanded model with a specific variant, as specified in the table. The table [of variants] may be filled from an external source or manually.

4.4.4.2 «inventory»

Can be applied to parts (which are then called prototypes), attributes, tags, connectors (including typedConnectors) and dependencies. In essence this construct removes the need to model each and every part, connector or dependencies, using external tables instead (filled from an external source or manually)

- Parts, connectors and dependencies marked with «inventory» represent sets of elements in the expanded model.
    - In case a marked «inventory» part or typedConnector instantiates a block marked with «catalog», the specific type (variant) of each element in the set will specified in the external table.
    - In case of a connector or dependency, «inventory» also indicates that the end points for each element in set are specified in the table.
- When applied to an attribute or a tag, indicates that the value of that attribute in each instance in the set (part or typed connector) should appear in the instantiating tables.

4.4.4.3 «optimized»

Can be applied to parts (prototypes), attributes, tags, connectors (including typedConnectors) and dependencies.

Much like «inventory», «optimized» allows to define sets of instances and to suggest that values for attributes and tags, instance type selection from catalog (where relevant), and endpoints (for connectors and dependencies) are defined in a table. The difference lies only in the source of the data – with «optimized», the data is to be generated by the optimization engine, not supplied manually or automatically from external tools.

Tags:

- max – the maximum number of elements to be created in the set. When applied on an attribute - this tag indicates its maximum value.
- min – the minimum number of elements to be created in the set. When applied on an attribute - this tag indicates its minimum value.

4.4.4.4 «expand»

Can be applied to any object or link. Indicates that this object/link is to be present in the expanded technical model.

4.4.5 Constraints Modeling Stereotypes

4.4.5.1 «behaviorModeBased»

This stereotype is used when there is a need to model modes of behavior. The stereotype is applied to attributes to indicate that their value changes according to some predefined schema, which, when the stereotype is used, will be defined in the instantiation tables.

Since there may be various behavior scenarios with some scenarios decoupled from each other, the stereotype carries a tag "scenarioGroup" which helps to distinguish between such orthogonal behaviors.

Examples:

- In Figure 33 the "Loads" prototype represents a collection of electric devices on an aircraft. These electric devices power consumption varies with respect to the current mission phase of the aircraft (for instance a targeting system will consume only standby power during the cruise phase and will consume maximum power during the strike mission phase). Since one of our constraints in the architecture optimization process is to select PDBs that can carry enough current, all power consumption scenarios (mission phases) need to be checked. The attribute indicating the power requirements of each Load will be marked with the stereotype.
- In Figure 31 the power generating elements have finite reliability. Therefore the requirement from the relays' network is to have a configuration (in terms of open-closed) for each failure scenario of the power generating elements, so that power is supplied to the buses (indicated by the "choice tree" concept. To model this with «behaviorModeBased» stereotype we create a new "scenarioGroup" and mark the "OK" attribute of the components with the stereotype.

4.4.5.2 «derived»

This stereotype is applied to an attribute to indicate that the attribute value is calculated from other parameters. Normally when this is done that attribute will appear on the left side of a formula in a constraint owned by / anchored to the block owning the attribute.

4.4.5.3 «objective»

This stereotype marks a «derived» attribute that is supposed to be one of the objectives of the optimization.

4.4.5.4 «invalid»

This stereotype is applied to a functional link with the purpose of denoted an illegal link. When the functional layer is mapped to the technical the corresponding paths in the technical layer will be considered illegal and will not be chosen by the optimization engine.

4.4.5.5 «VisualConstraintBlock»

This stereotype converts a block into a Visual Constraint. A Visual Constraint Block is a graphical (using SysML) method of depicting an architectural rule or a regular expression. A Visual Constraint Block is specified using a Constraint Diagram, and may contain explicit parts from the architecture, constraint variables typed by blocks from the architecture (representing some architecture part – or a set of parts, according to multiplicity - with the same type), or

constraint variables typed by other Visual Constraint Blocks, representing a composite regular expression.

4.4.5.6 «constraintDiagram»

This stereotype inherits from Internal Block Diagram. The newly defined Constraint Diagram is used to depict architectural constraints in graphical terms. Often the diagram would contain both functional and technical elements in order to define how a functional constraint is mapped to the technical plane.

Figure 35 shows a Constraint Diagram with a definition of illegal path between the left and the right generators, with a mapping to the technical level (modeled concisely). A constraints diagram may also include formulae constraining the values of attributes of some components.

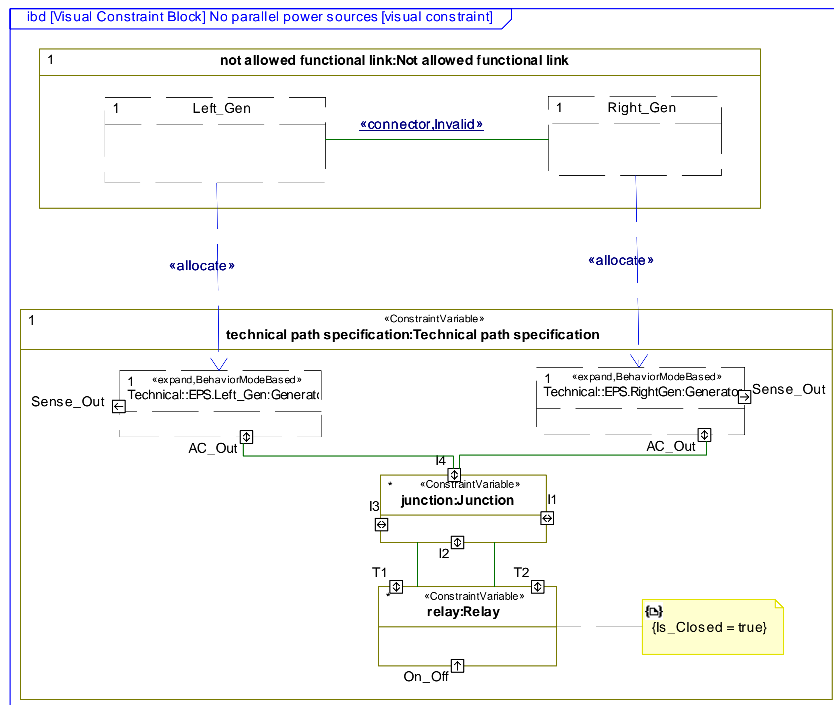


Figure 35 - Constraint Diagram

4.4.6 Variability Profile

Besides concise modeling, another useful approach to specifying architecture alternatives is to explicitly model them in a single model, and annotate them using variation points. For example, by specifying that a particular element has an "existence" variation point, the systems engineer indicates that the element may or may not exist in a given architecture. To allow for correlation of variation points (i.e. to specify that several elements go together) each variation point may be bound to a "choice" element. In order to derive a specific architecture from a model with alternatives, a decision for each choice (and unbound variation point) must be made by the optimization engine.

Choices may also be organized in a tree, which allows constraining the legal decisions for a given architecture: Each choice node in the tree defines how many of its child choices must be selected (if it is selected itself). For example, a choice may specify that exactly one of its child choices be selected. This makes all its children alternatives of one another. Respectively, their bound variation points also become alternatives. Other kinds of constraints (known as cross-tree constraints) between choices may also be used.

4.4.6.1 The VariabilityRealization package: variation points

Figure 36 outlines the hierarchy of the Variability realization stereotypes.

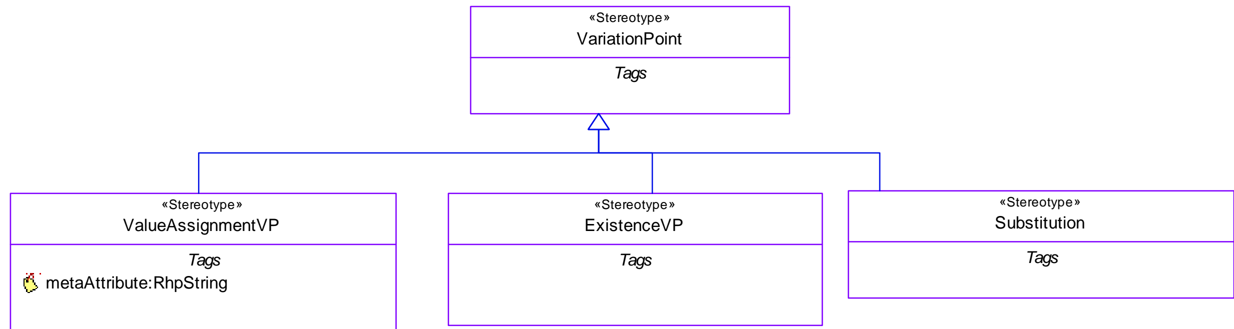


Figure 36 - Variability Realization Package

4.4.6.1.1 *«ExistenceVP»*

Specifies that the element to which it is applied may or may not exist in a particular architecture.

4.4.6.1.2 *«Substitution»*

A kind of dependency which states that the source might substitute the target in a given configuration. All connections from / to the target (e.g. connectors) will be added to the source.

4.4.6.1.3 *«ValueAssignmentVP»*

Element that allows assigning value to some field of a model element, e.g. an initial value of an attribute. The field to be assigned a value is specified by the tag "metaAttribute", which is a string, in the above example "initialValue".

4.4.6.2 The VariabilityAbstraction package: choice trees

As explained above, the choice tree (actually, variability specification tree – including choices and values) allows modeling the high-level architectural choices and their relationships. Each such choice over variable may be bound to an element marked by a variation point stereotype.

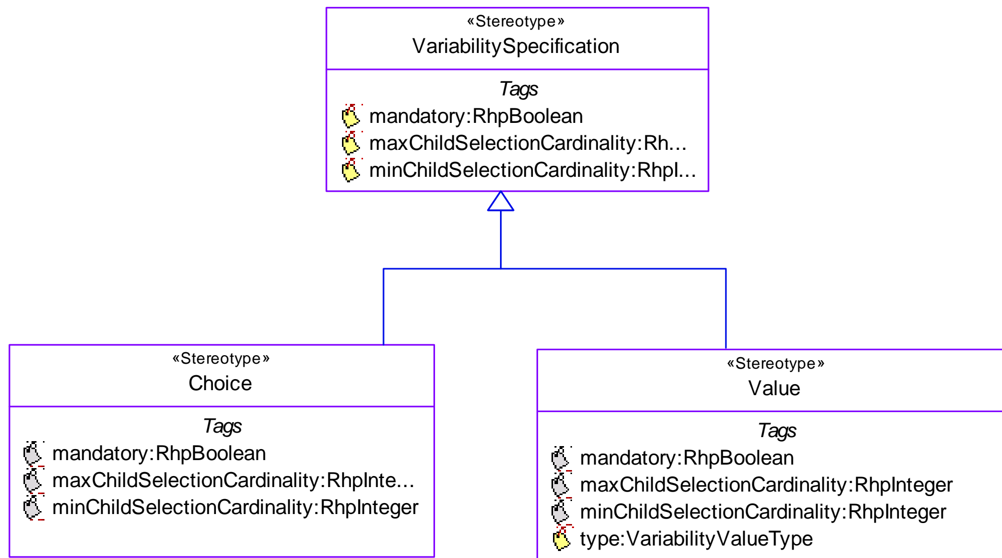


Figure 37 - Variability Abstraction Package

4.4.6.2.1 *VariabilitySpecification*

This is an abstract stereotype, which represents a node in the variability tree. Variability Specifications (VSpec) are a new kind of model element, implemented in Rhapsody as a new term based on SysML comments. This allows for their nesting, enables showing them on diagrams, and provides an easy binding mechanism to variation points, using anchors.

Each VSpec has the following tags:

- mandatory – indicates that a choice is selected together with its parent choice.
- maxChildSelectionCardinality – maximum number of child VSpecs that can be selected for a given architecture.
- minChildSelectionCardinality – minimum number of child VSpecs that can be selected for a given architecture.

The tree can have any number of levels and the leaf choices will carry no tags (the tag values will be empty).

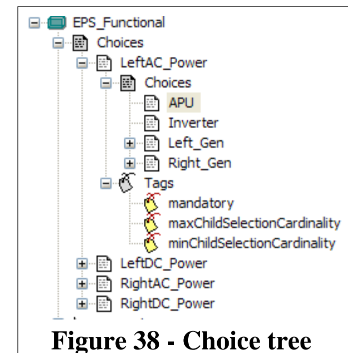


Figure 38 - Choice tree

4.4.6.2.2 *Choice*

Choices are a kind of VSpec which requires a Boolean decision to be made when specifying a particular architecture out of the possible architecture variants. It may be bound to Existence and Substitution variation points.

On the functional plane a choice anchored to an object (or a link) indicates that the object may exist and may not. Since it is a functional object the above will indicate that the function is active or not at a given point in time. When a root choice has 4 sub-choices with max and min cardinality at 2 and 1 respectively, it would mean that at any given time in the system at least one function must be active and no more than two can be active.

On a technical plane the choice would indicate an architecture option. For example whether there should be a relay between two electric power components or just a wire. The architectural choice must, in this case, be made either by a human or by the optimization engine.

4.4.6.2.3 *Value*

Values are a kind of VSpec which requires a value to be given when specifying a particular architecture out of the possible architecture variants. It may be bound to ValueAssignment variation points, which then takes the value and assigns it as explained above. The value type is specified using the "type" tag, whose type (VariabilityValueType) is an enumeration over basic data types (Integer, Float, Boolean, String, etc).

4.4.6.3 Variability constraints

Besides the tree structure, which specifies constraints regarding allowed selections, we also allow specifying cross-tree constraints. We support both simple constraints such as excludes and requires and constraint expressions, which are logical expressions over the VSpecs.

4.4.6.3.1 *Requires*

A simple constraint between two Choices. Implemented as a new kind of dependency. Denotes that selecting the source requires selecting the target as well.

4.4.6.3.2 *Excludes*

A simple constraint between two Choices. Implemented as a new kind of dependency. Denotes that selecting the source forbids the target from being selected.

4.4.6.3.3 *VariabilityConstraintExpression*

Implemented as a new kind of constraint. Allows to specify (in the constraint's Specification field) a logical condition which must occur. For example, referring to the choices in the figure above, we can write the cross-tree constraint "LeftACPower.APU || RightACPower.APU" to denote that at least one of the two APUs must be selected.

4.4.7 Objectives and Algebras

Since every optimization needs objectives, a method was required to define them and the metrics that the objectives are defined over. In the course of the development we have started with the most simple metrics of Cost and Weight, their simplicity coming from the fact that they are a basic sum function over a set of single type parameters.

4.4.7.1 Algebras

The purpose of algebras is the computation of certain values in the model that are needed for the analysis or the optimization process. These values are marked with the «derived» stereotype to indicate that a computation is required.

The definition of the computation may be done using several approaches. The first and a more immediate one is a textual definition of the computation formulae using some existing syntax. In our process, since the model is transformed into OPL code, the straightforward approach is to write the formulae in OPL. But since the automatic transformation of this part was not done yet, the representative syntax in Figure 39 is Modelica.

A complication of the above approach, introduced by the Concise Modeling profile, is the fact that at the time of the modeling we are not aware which model elements are present in the final model (which will be available after the optimization process).

For example, it is not possible to specify a weight formula – "The weight of a component is the sum of the weights of all its parts" – without explicitly referencing each part, e.g. weight = part1.weight + part2.weight + part3.weight…

To address this we have come up with a concept of Model Interrogation functions. In Modelica syntax these functions will be used in the following manner:

SystemWeight = sum (getAllParts().weight)

The function will be evaluated in the optimization process during selection of feasible solutions and used (in this case) as one of the elements of the multi-objective criteria.

An additional concept required when using Model Interrogation functions is the specification of their scope. For each such function we would like to specify the actual collection of elements that will be used in its evaluation – whether it is the entire system or a subsystem or perhaps the contents of a specific SysML diagram. Example of such scoping limitation is:

PrimarySystemCost = sum ((x.cost) for x in getAllParts() : x in getDiagramElements("Primary"))

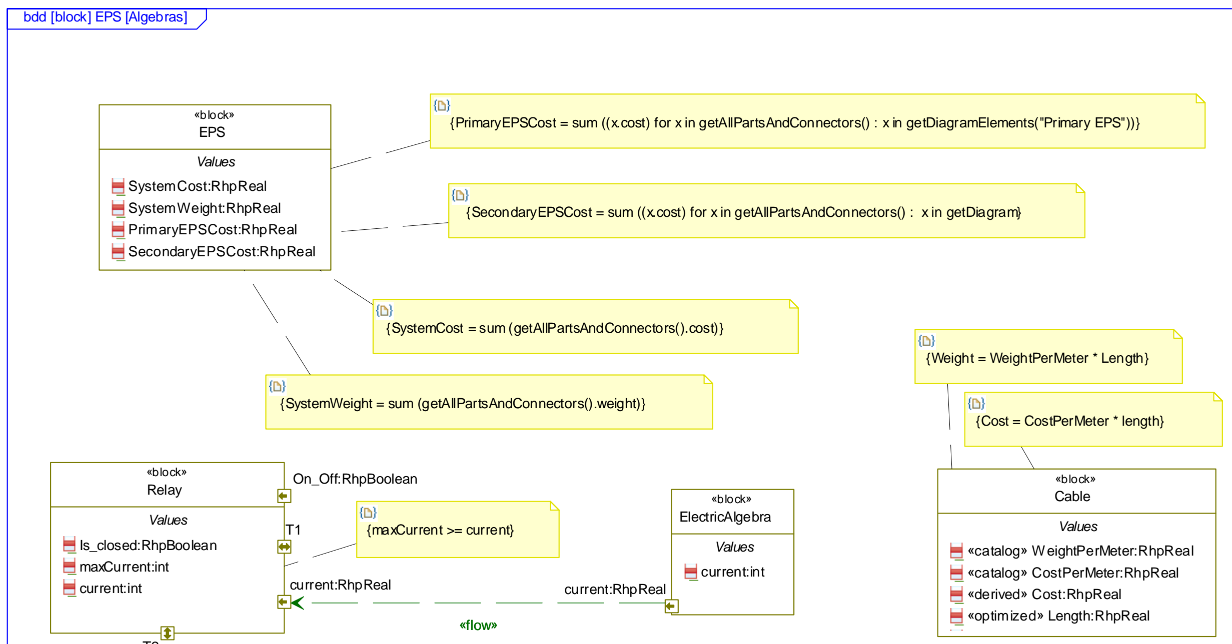


Figure 39 - Textual Approach in Algebras Definition

An alternative way of specification is to use SysML parametric diagrams as depicted in Figure 40. This approach may still require scoping limitations. The constraint may again be written in any suitable language (OPL, OCL, CSL).

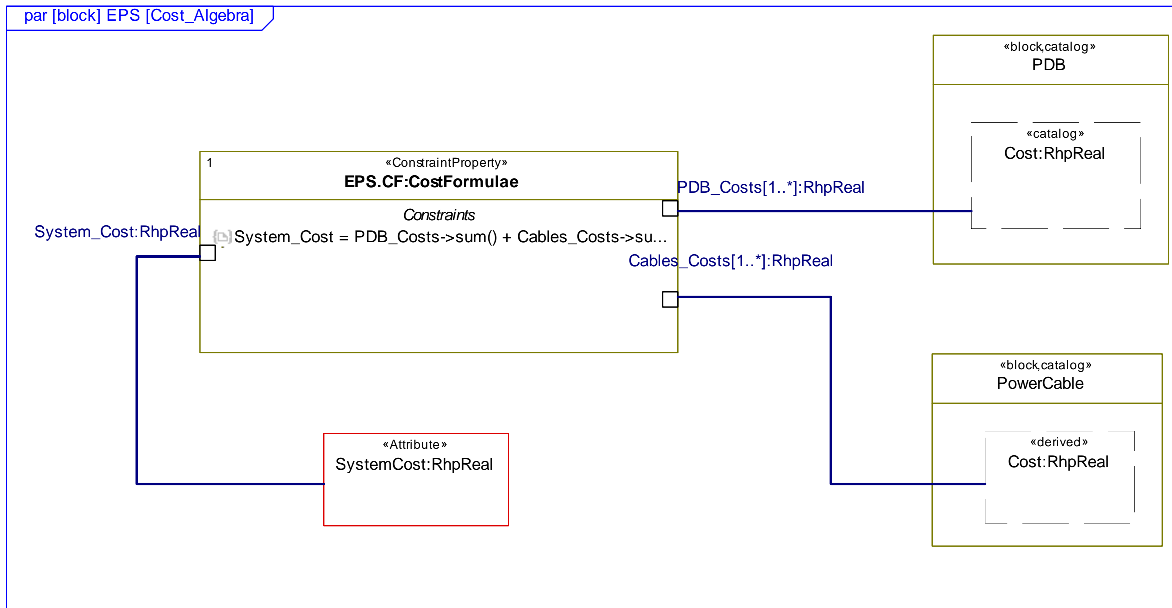


Figure 40 - Parametric Diagram as Algebra Definition

4.4.7.2 Objectives

Elements marked with «objective» stereotype will be used in the multi-objective optimization to generate the efficient frontier. In the current project a true multi-objective optimization was not done, but rather a weighted objective function was utilized.

4.4.8 Exemplary Design Process with Concise Modeling

A software mechanism is used to extract information from the model and create the tables. Created tables are filled by an external tool or manually.

In case all instantiating information has been provided in the tables, model expansion can be done. The mechanism generates an "expanded" technical layer model without prototypes, meaning that all parts are modeled explicitly. This may result in a very large and unreadable model, but its construction will be correct and a simulation or verification process can be initiated.

In case an automatic architecture optimization is needed, the software will create, based on the model and on the filled tables, an optimization program that will generate the architectures (one or more). The result is fed back into the tables from which the model expansion can be done, for visualization, simulation or else.

The entire above process is an iteration in a continuing design process from high level of abstraction down to the detailed levels. The functional layer defines the requirements, later implemented by the technical layer that in turn becomes the functional layer for the next level of abstraction.

The optimization process is limited, however, due to several reasons. First of all many times we do not possess all the information about the internals of a model, as discussed in section 3.2. In addition the optimization algorithms have difficulty dealing with dynamic behaviors. Since our objective is to have a complete and verified (to maximal possible extent) architecture, we add a verification phase. An additional tool, which may be a simulator or a formal verification tool,

Figure 41 and Figure 42 summarize the above text.

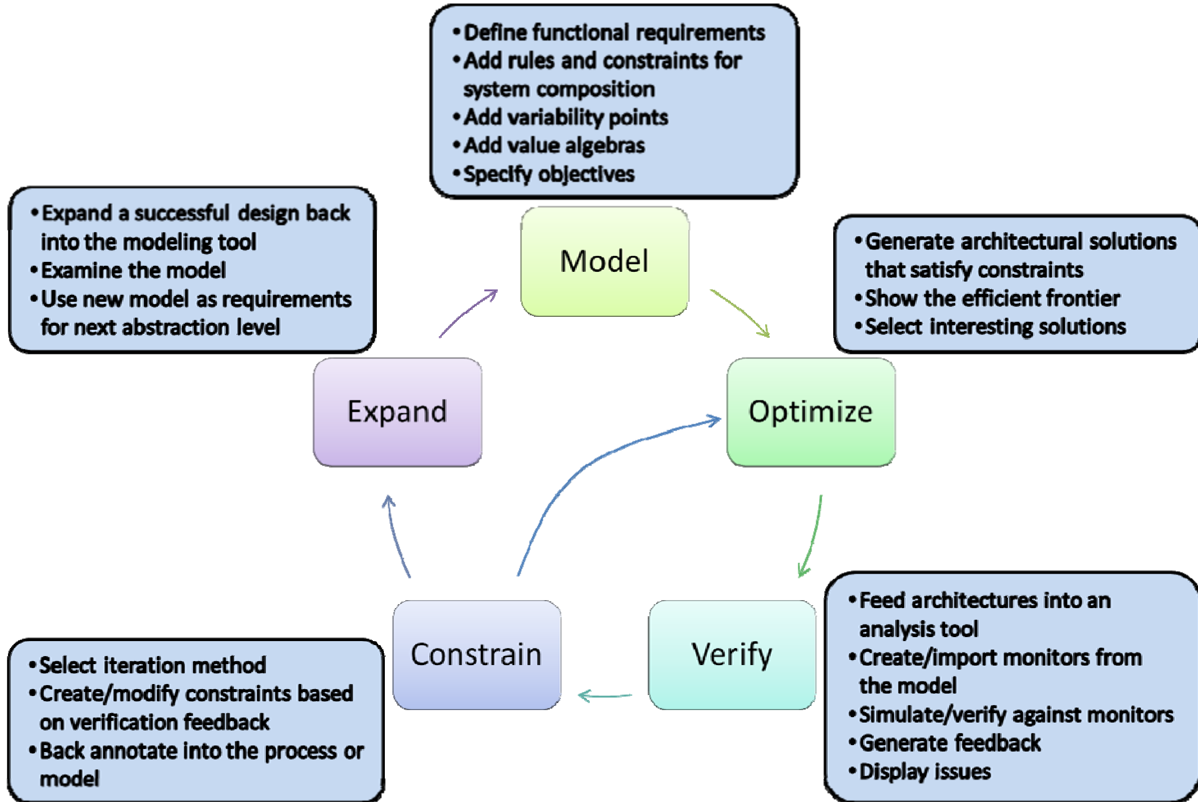


Figure 41 - IBM Design Optimization Process

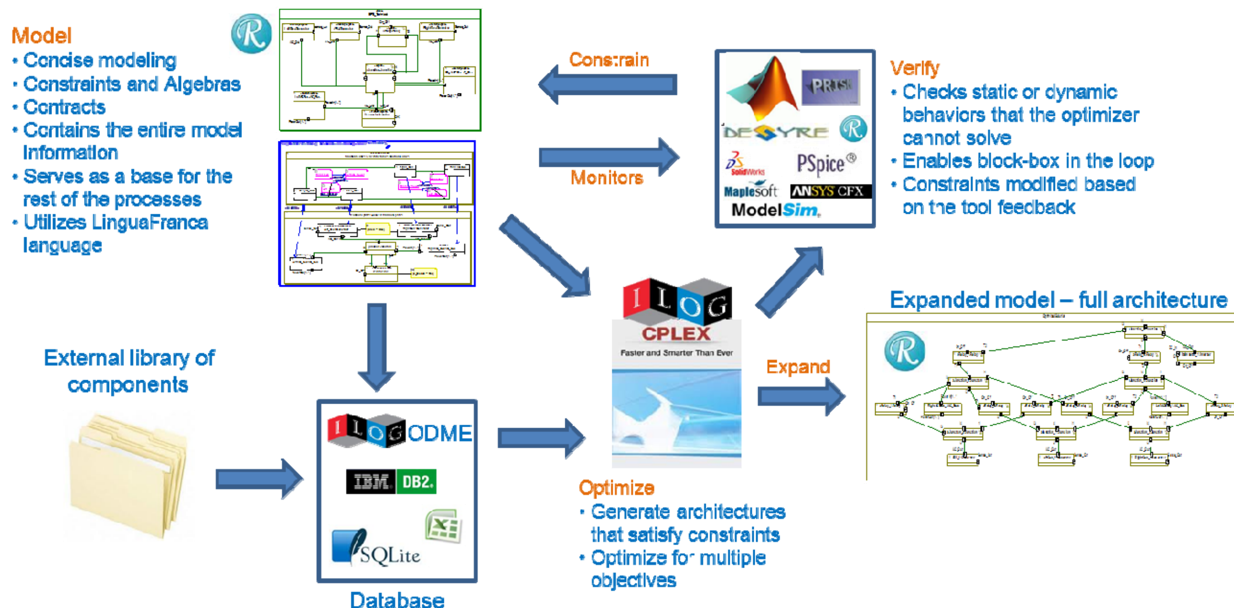


Figure 42 - Design Optimization Process in Detail

4.4.9 Concise Plug-in

In the course of the project we have developed a prototype tool build on top of Rational Rhapsody. The tool is used to facilitate the design process described in the previous section. The chain incorporates Rational Rhapsody, iLog CPlex Studio, Microsoft Excel and Mathworks Simulink.

4.4.9.1 Architecture generation

The tool works in several phases:

- Concise model structure extraction and interchange tables creation
- Generation of an optimization program
- Expansion (back-annotation) of the resulting architecture (or architectures) back into the modeling environment.
- Run additional verification processes, evaluate their output and modify optimization constraints if needed.

Once the manual modeling phase is complete the first phase of the tool traverses the model, looking for the concise modeling stereotypes. It is worth mentioning that combining concise and concrete components in a single model proved to be quite challenging and had to be given special attention after the 4th PI meeting (May 2011). All model elements that are marked are processed and a data structure in Microsoft (MS) Excel is created. The data structure is essentially a relational database and the choice of MS Excel for the prototype is based on the familiarity of most people involved with the MS Office suite and its ubiquitous nature. It is not suitable for a true product due to limitations in dataset sizes and also to its performance.

The Excel structure includes tables for components (for all layers) with columns for their parameters as well as tables linking between components. These tables are intended to be filled from an external source, such as component libraries. Speaking in terms of the META

community a component in the Excel tables is a projection of a component in a detailed component library, as will be created in the upcoming Component, Context, and Manufacturing Model Library (C2M2L) DARPA activity.

The next phase traverses the model again, this time in a more comprehensive fashion. It extracts the model structure creating an OPL (language used by the CPlex Studio environment) program that will optimize the design based on the given constraints and data. At the time of the 5th PI meeting we have managed to create only partial OPL generation, however we do not foresee a problem to have a completely automatic transformation.

Once the DSE optimization program has executed and written its results back into Excel tables, the plug-in expands the results, using the structure information (ports, topology) from the original concise model.

4.4.9.2 Optimization model creation

During the DSE process with the optimization engine a functional model of the system is mapped to a physical architecture taking into account all non-functional and mapping contracts. The generated OPL Optimization model includes the following elements:

1. The set of functional nodes and connections – based mostly on the functional model.
2. The set of physical nodes and connections – based mostly on the physical model and “inventory” databases.
3. The set of potential components that can be placed at the physical nodes and connections – based mostly on physical model and “catalog” databases.
4. Mapping sets – for each functional node it defines the set of physical nodes where the functional node could be mapped to.
5. Scenarios with appropriate parameter sets.
6. Design decision variables – mostly binary variables that define
    a. the physical architecture, including what nodes and connection are chosen and how they are instantiated;
    b. mapping from functional to physical – node to node / connection to virtual path(s) / “invalid” connection to virtual cut(s);
    c. components parameters;
    d. state variables.
7. Objective function(s).
8. Constraints.
    a. structural constraints – according to the structural (topological) contracts;
    b. mapping constraints – each functional node should be mapped to a physical node, each functional connection should be mapped to virtual physical path(s), and each functional “invalid” connection should be mapped to virtual physical cut(s);
    c. algebras;
    d. contracts defined by constraints including logical conditions.
9. DSE output.

While translation of algebras and constraints is a direct mathematical transformation

4.4.9.3 Verification phase

The verification phase was demonstrated at the 5th PI meeting (July 2011) by feeding the optimization results into a Simulink model (provided by the UTRC team). The objective of the optimization, as demonstrated, was to select the best relay topology and the most suitable relay components for the Primary EPS (see 3.1.1) in our standard use case. In the demo scenario the Systems Engineer performing the architecture optimization is not aware and does not understand the electric functionalities of the components and thus an expert (UTRC) provides him with a generic circuit model with a predefined interface. The model is capable of computing currents and voltages in an electric circuit including parasitic effects of resistance and capacitance.

Simulink output was in the form of MS Excel spreadsheet with data for voltages and currents at various circuit nodes. We have implemented a subroutine that compares these with the original requirements and, in case requirements exceed the results, modifies the optimization constraints. The method of iteration is this case is an adaptation of a common practice in mechanical engineering called Safety Factoring. In this simple and efficient method some requirements are synthetically increased by a multiplication by a safety factor, calculated from the difference between the original requirement and the unacceptable value obtained in the verification process. After several iterations of the process the design converges. This method may yield a less than optimal result, but its efficiency makes it a viable option in many design processes.

4.4.9.4 Gas Turbine Engine demo

At the 5th PI meeting the concise modeling approach and the prototype tool were tested in additional modeling effort for a completely different (from the EPS that was usually used) use case (See 3.1.2). The Gas Turbine was modeled concisely (Figure 43) and the data structure was generated. Since the objective was to validate the concise modeling language the only interesting phase was the expansion phase. The expansion subroutine was fed a synthetic optimization result and the expanded model was examined for correctness.

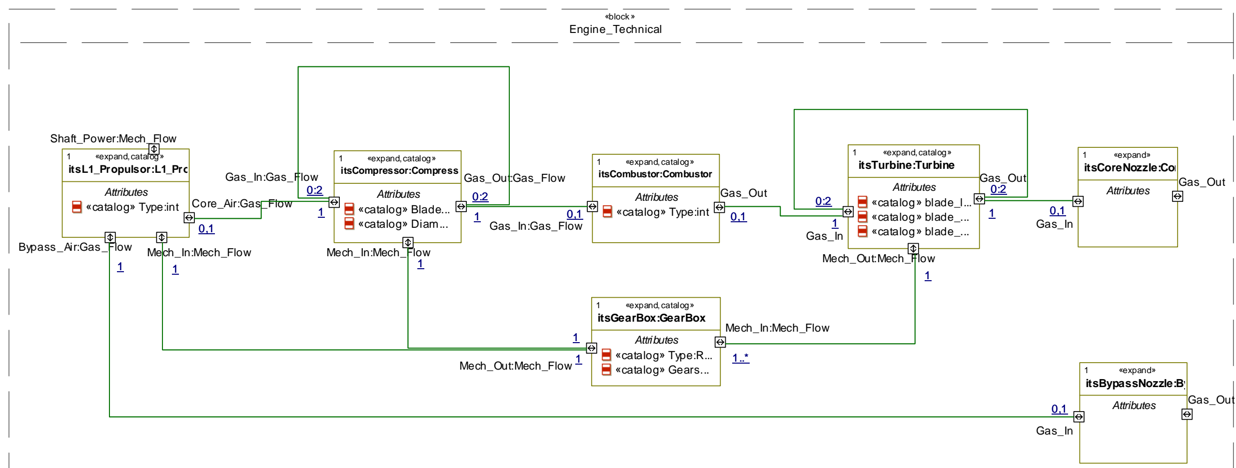


Figure 43 - Gas Turbine Engine Concise Model

5 CONCLUSIONS

5.1 Dealing With Scale

One of the challenges facing all META performers is whether the novel methodologies will prove adequate in large designs and not only in limited scope challenge problems. The truth is that the problem is not only a computational one, but human as well. Each individual possesses a finite "cognitive bandwidth" meaning that one's ability to grasp many things at once is limited.

Most of the META community, despite the diversity of approaches and different technical areas, agree that the key to scale is abstraction. Another approach that shows promise is one presented by Rockwell Collins team – complexity reducing design patterns.

Figure 44 is taken from our insight slide presented by Alberto at the 5th PI meeting (July 2011). Abstraction has two axes – quantity and quality. In quantity we reduce the number of components that the designer and his software tools have to deal with at once. Quality deals with reducing the detail level of component models, again reducing the cognitive bandwidth and computation requirements (example: a thousand components are easy to deal with if their only parameter is weight and its algebra is a simple sum). Computation complexity may further be decreased by replacing some parts of the computation with approximations.

Correctly breaking down the design process into abstraction levels is tricky. The UTRC team has made great progress in analyzing the matter. At the end, it is going to be the experience of the design engineer that will be instrumental of selecting the right abstraction at the right moment, choosing the iteration length based on available resources, deciding on the confidence level required from various analyses and so on. The designer may be aided by abstraction libraries – a collection of pre-defined and pre-verified transformations, somewhat resembling Rockwell Collins team approach.

In general a concept of a library is seen often in the community, since this allows preliminary definition of library (of any elements) elements which, because it is done by an expert, are well defined and tested. A library approach to modeling greatly reduces errors, facilitates quick design iterations and yields better overall systems, as has been demonstrated by the libraries revolution in EDA field (standard cells).

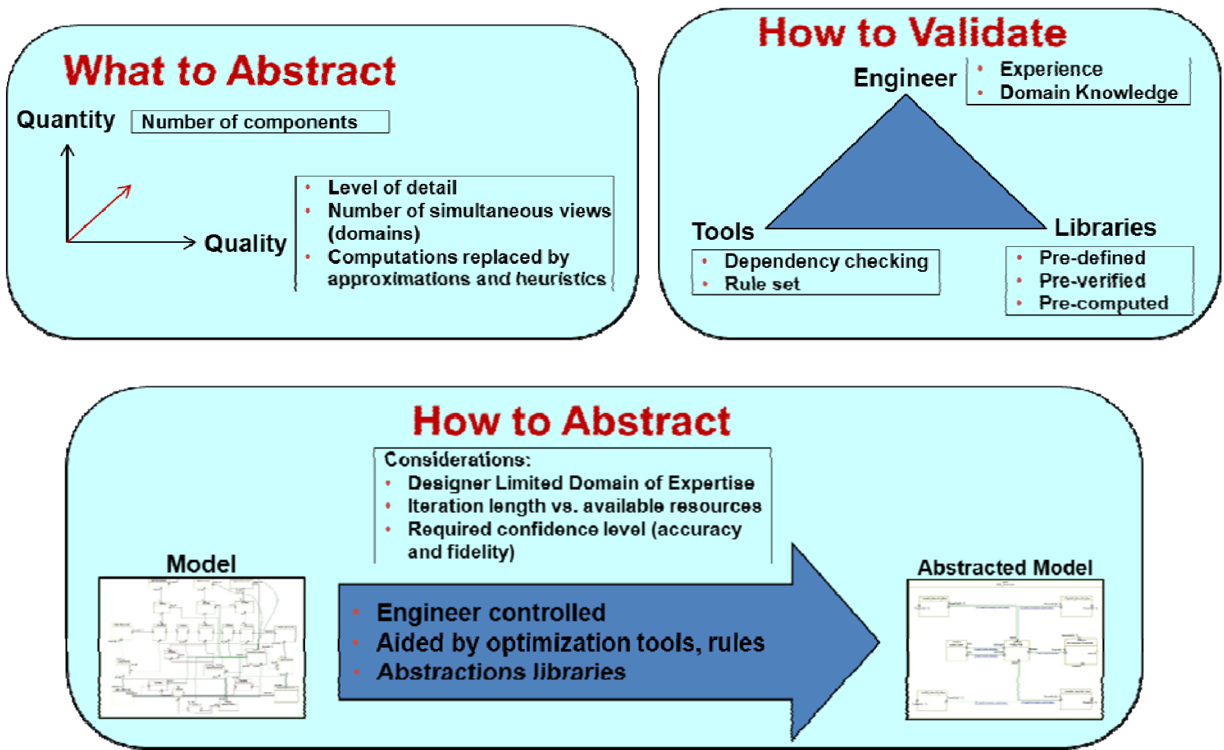


Figure 44 - Abstraction as the Key to Scale

In our concise modeling approach we use two abstraction layers – the functional and the technical (see 4.4.2) – with a specific mapping one to the other. This enables traceability of requirements down through the abstraction levels. A true multi-level design process is yet to be performed using our tool in order to fully validate its adequacy.

5.2 MoC Integration

The demonstrated MoC integration approach shows great promise. The barrier to the widespread use of it lies in the difficulty of defining the semantics of tools and languages using TSM. We have understood that an enabling concept to this approach penetration can be, similar to design abstractions, a language abstraction. It is a framework of higher language constructs, which are sufficiently fine-grained to describe the semantics of other tools and languages but are not as complex as the underlying mathematical foundation of the TSM, which few people can intimately understand.

5.3 Contract Based Design

A comprehensive library of contract patterns with rigorously defined semantics is the key to contract based design. Engineers will not use the formal contracts if they cover only a portion of the descriptions used by the engineers today in the requirements and component specifications.

6 REFERENCES

[1] E.A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation", IEEE Trans. CAD, 17(12):1217-1229, 1998.

[2] L. Mangeruca, A. Ferrari, O. Ferrante, A. Mignogna, A. Sangiovanni-Vincentelli, "The semantics of Heterogeneous Models"

[3] MARTE OMG specification

[4] X. Liu, "Semantic Foundation of the Tagged Signal Model", PhD Thesis, Technical Report, UCB/EECS-2005-31, 2005.

[5] D.2.5.4 Contract Specification Language (CSL), SPEEDS project deliverable

[6] Albert Benveniste, Werner Damm, Alberto Sangiovanni-Vincentelli, Dejan Nickovic, Roberto Passerone, and Philipp Reinkemeier. Contracts for the design of embedded systems. Part I: methodology and use cases. Submitted for publication, 2011

[7] Albert Benveniste, Jean-Baptiste Raclet, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Alberto Sangiovanni-Vincentelli, Tom Henzinger, and Kim G. Larsen. Contracts for the design of embedded systems. Part II: theory. Submitted for publication, 2011

LIST OF ACRONYMS, ABBREVIATIONS, AND SYMBOLS

ACRONYM	DESCRIPTION
AC	alternating current
ALES	Advanced Laboratory for Embedded Systems, Rome, Italy
APU	auxiliary power unit
CSL	Contracts Specification Language
DAE	differential algebraic equation
DC	direct current
DE	discrete event
DLL	dynamic link library
DSE	design space exploration
DSL	domain specific language
EDA	electronic design automation
EPS	electrical power system
EU	European Union
HRL	Haifa Research Laboratory (IBM)
IBM	International Business Machines Corporation
MEL	minimum equipment list
MoC	model of computation
MoCC	model of computation and communication
MS	Microsoft Corporation
PBD	platform based design
PDB	power distribution box
PI	principal investigator
PID	proportional-integral-derivative
PLE	product lines engineering
PSL	Property Specification Language
RAT	ram air turbine
SE	systems engineering
SLD	system level design
TDF	timed data flow
TRU	transformer rectifier unit
TSM	tagged signal model

UAV	unmanned aerial vehicle
UTRC	United Technologies Research Center
VP	variation point
VSpec	variability specification

Approved for public release; distribution unlimited

APPENDIX A – IBM Team Bios

Alberto Sangiovanni – Vincentelli

Alberto Sangiovanni Vincentelli holds the Edgar L. and Harold H. Buttner Chair of Electrical Engineering and Computer Sciences at the University of California at Berkeley. He has been on the Faculty since 1976. He obtained an electrical engineering and computer science degree ("Dottore in Ingegneria") summa cum laude from the Politecnico di Milano, Milano, Italy in 1971.

He was a co-founder of Cadence and Synopsys, the two leading EDA companies. He is the Chief Technology Adviser of Cadence. He is a member of the Board of Directors of Cadence, Sonics, and Accent. He was a member of the HP Strategic Technology Advisory Board, and is a member of the Science and Technology Advisory Board of GM, of the UTC Technology Advisory Council and of the Scientific Council of the Tronchetti Provera foundation and of the Snaidero Foundation. He consulted for many companies including Bell Labs, IBM, Intel, United Technologies Corporation, COMAU, Magneti Marelli, Pirelli, BMW, Daimler-Chrysler, Fujitsu, Kawasaki Steel, Sony, ST, United Technologies Corporation and Hitachi. He is the Senior Advisor to the President and CEO of L-Elettronica, Roma, Italy. He was an advisor to the Singapore Government for microelectronics and new ventures. He consulted for Greylock Ventures and for Vertex Investment Venture Capital funds. He is a member of the Advisory Board of Walden International, Sofinnova and Innogest Venture Capital funds and a member of the Investment Committee of a novel VC fund, Atlante Ventures, by Banca Intesa/San Paolo. He was the founder and Scientific Director of the Project on Advanced Research on Architectures and Design of Electronic Systems (PARADES), a European Group of Economic Interest supported by Cadence, Magneti-Marelli and ST Microelectronics. He is a co-founder and the President of the Advanced Lab for Embedded Systems (ALES), Roma, Italy. He is a member of the Advisory Board of the Lester Center for Innovation of the Haas School of Business and of the Center for Western European Studies and is a member of the Berkeley Roundtable of the International Economy (BRIE). He is a member of the High-Level Group, of the Steering Committee, of the Governing Board and of the Public Authorities Board of the EU Artemis Joint Technology Initiative. He is a member of the Scientific Council of the Italian National Science Foundation.

In 1981, he received the Distinguished Teaching Award of the University of California. He received the worldwide 1995 Graduate Teaching Award of the IEEE. In 2002, he was the recipient of the Aristotle Award of the Semiconductor Research Corporation. He has received numerous research awards including the Guillemin-Cauer Award (1982-1983), the Darlington Award (1987-1988) of the IEEE for the best paper bridging theory and applications, and two awards for the best paper published in the IEEE Transactions on CAS and CAD, five best paper awards and one best presentation awards at the Design Automation Conference, other best paper awards at the Real-Time Systems Symposium and the VLSI Conference. In 2001, he was given the Kaufman Award of the Electronic Design Automation Council for pioneering contributions to EDA. In 2008, he was awarded the IEEE/RSE Wolfson James Clerk Maxwell Medal for groundbreaking contributions that have had an exceptional impact on the development of electronics and electrical engineering or related fields with the following citation: For pioneering innovation and leadership in electronic design automation that have enabled the design of modern electronics systems and their industrial implementation. In 2009, he received the first

ACM/IEEE A. Richard Newton Technical Impact Award in Electronic Design Automation to honor persons for an outstanding technical contribution within the scope of electronic design automation. In 2009, he was awarded an honorary Doctorate by the University of Aalborg in Denmark.

He is an author of over 850 papers, 15 books and 3 patents in the area of design tools and methodologies, large-scale systems, embedded systems, hybrid systems and innovation.

Dr. Sangiovanni-Vincentelli has been a Fellow of the IEEE since 1982 and a Member of the National Academy of Engineering, the highest honor bestowed upon a US engineer, since 1998.

Alberto Ferrari

Director and co-founder of ALES S.r.l., he received his PhD degree in Electrical Engineering and Computer Science at the University of Bologna, Italy. In 1995, he had been a visiting fellow at the EECS Department of University of California at Berkeley and at the SGS-Thomson Berkeley Labs. In 1997, he joined the PARADES EEIG research laboratory in Rome, Italy. He has consulted several companies such as BMW, Magneti Marelli, ST Microelectronics, Cadence and UTC on design methodology and architecture for embedded real-time systems. From 2000 to 2007, he has been teaching Network for Embedded Systems (at the University of Ancona) and since 2008 he is teaching Embedded Systems at the University La Sapienza. He has been part of the ARTISTII, ArtistDesign and Hycon European Networks of Excellence. He has been the technical coordinator of PARADES in the SPEEDS and COMBEST EU projects. He is currently coordinator of ALES in the SPRINT, MBAT and DANSE EU projects. He is author of several papers on design tools and methodologies for embedded systems, safety-critical embedded controllers and hybrid systems. He has been invited as speaker on design methods and architectures for safety-critical systems to several conferences. He is co-founder of Evidence S.r.l., a company providing software for embedded real-time systems.

Leonardo Mangeruca

Leonardo Mangeruca received the Dr. Eng. degree summa cum laude in Electrical Engineering from the University of Genoa, Italy, in 1995.

In 1995 he joined the Department of Biophysical and Electronic Engineering of the University of Genoa, where he worked on the CHIPS project in cooperation with IMEC, Leuven, Belgium. Between 1997 and 1998 he was intern at the Cadence Berkeley Labs, where he worked on formal verification of FSM-based system models in POLIS in cooperation with Felice Balarin and Luciano Lavagno. In 1999 he received his PhD degree from the university of Genova with a thesis on system level specification and synthesis from behavioral VHDL. In the same year he joined PARADES GEIE, Rome, Italy, where his interests have focused on design methodologies, hardware/software architectures and formal methods for distributed embedded systems design. Currently he is senior research scientist at ALES S.r.l., Rome, Italy.

Amit Fisher

Amit Fisher is Senior Manager at IBM Research , Haifa, currently managing the Business and Systems Solutions. His main areas of expertise are Business and Enterprise Architecture modeling, business models, business strategy and IT strategy, business process management, service oriented architecture, data mining and business intelligence and operation research. Mr. Fisher gained a lot of experience in participating and leading business strategy and transformation project, and was one of the main contributors to the IBM Business a Architecture

offering. Furthermore, Mr. Fisher is involved in shaping IBM's future architecture tools, and leading several project in this realm. Amit has a B.Sc. degree in Industrial Engineering and Management and a M.SC. degree in Information System Engineering from the Technion, Israel.

Michael Masin

Michael Masin, Ph.D. 1998, M.Sc. 1992 (Industrial Engineering, Technion, Israel), M.Sc. 1987 (Mechanical Engineering, Moscow State University of Railway Transport, Russia). Michael is a Research Staff Member in Systems and Business Optimization groups at IBM Research – Haifa Lab (HRL) and has strong teaching and research ties to the Technion and Tel Aviv University, Israel. Before IBM, Michael was an analyst in the Center of Military Analyses at RAFAEL — Advanced Defense Systems Ltd. Between 1998 and 2004, Dr. Masin was a postdoctoral fellow and a visiting faculty at the Pennsylvania State University, Tel-Aviv University, and Technion. During his Ph.D. studies, he was employed by the Israeli Defense Forces for four years, working in the fields of Operations Research and Systems Analysis. Dr. Masin's research interests focus on the area of deterministic and stochastic combinatorial multi-objective optimization including (1) Systems Engineering and System of Systems design and (2) design, control, and integration of production, service, and logistics systems.

Henry Broodney

Henry Broodney is a Research Staff Member in the Systems Engineering group at IBM HRL.

Henry is an experienced Systems Engineer having dealt with wide range of disciplines in this professional career, ranging from architectural software development and electronic and chip design through complex mechanical design and heavy metal machining. He is fluent in contemporary Systems Engineering methods and tools, having both used current and implemented new methodologies in the various environments he has worked at.

He holds M.B.A and a B.Sc.EE. both from Technion, Israeli Institute of Technology, working at a chip designer at Motorola semiconductors during his studies.

He has started his professional career in the Israeli Air Force (IAF) commanding an avionics maintenance detail and later overseeing integrated avionics systems development, maintenance and upgrades at the Electronics Warfare branch in IAF headquarters. While in the military, Henry was also volunteering as a project supervisor at the Technion EE Department High Speed Digital Lab with numerous projects successfully completed under his name, mainly in the field of high speed board design for data intensive and video conferencing applications.

He later held a position of a team leader and a Systems Engineer at Rafael Systems, developing complex multidisciplinary systems and completing the formal corporate Systems Engineering training. His main subjects at Rafael, in addition to several classified subjects, included remotely controlled weapon stations and high-power laser applications.

In 2006 Henry co-founded and managed InGrid Networks, a software start-up in the field of IaaS cloud and P2P computing, thus making him intimately familiar with the world of Software development and its methodologies, such as Model Driven Development, Agile approaches and Software Quality Management. He is also the co-author of 4 patent applications in the field of P2P computing and data storage methods.

Before joining IBM Henry held a position of a System Engineer at Soltam (Elbit Land Systems company) artillery directorate, where he helped introduce novice methods for complex defense systems design, incorporating both in-house processes and sub-contractor management.

Lev Greenberg

Lev Greenberg is a Research Staff Member in the Systems Engineering group at IBM Research - Haifa Lab (HRL). Lev has an industrial experience in software engineering, algorithm development and Systems Engineering. His previous position was Senior Systems Engineer at GE Healthcare. Lev received his B.Sc. degree Cum Laude in Aerospace Engineering in 1997 and M.Sc. degree Summa Cum Laude in Applied Mathematics in 2003 both from Technion. His research areas include convex and combinatorial optimization, probability and stochastic processes, communication protocols, signal and imaging processing, medical physics and Systems Engineering.

Alexander (Sasha) Zadorojniy

Alexander (Sasha) Zadorojniy is a Research Staff Member in the Systems and Business Optimization group at IBM Research – Haifa Lab (HRL). Sasha holds a Ph.D. from Tel-Aviv University and B.Sc. and M.Sc. from Technion Israel Institute of Technology, all in Electrical Engineering. His research involved development of efficient optimization algorithms for discrete time stochastic control. He has six publications, including Informs Nicholson finalist and ORSIS winner, and was involved in developing high efficiency scheduling and routing algorithms for ad-hoc networks. He has 10 years of HW design experience at Intel IDC specializing, in addition to leading unit designs and designing complex clock networks, in verification and validation across multiple disciplines within leading edge Intel processors.

Alessandro Pinto

Dr. Alessandro Pinto is a researcher in the Embedded Systems and Networks group at the United Technologies Research Center (UTRC), Inc., Berkeley, California. His research interests are in the area of computer aided design for cyber-physical and autonomous systems. He received a Ph.D. degree in Electrical Engineering and Computer Sciences from the University of California at Berkeley in 2008, and a M.S. degree in Electrical Engineering in 2003 from the same University. He holds a Laurea degree from the University of Rome "La Sapienza". In 1999, he spent one year as a consultant at Ericsson Lab Italy in Rome, Italy, working on the design of systems-on-chips. He consulted for the same company from 2000 to 2001, developing system-level design flows for wireless access networks.
He is the developer of several model-based design tools including the Communication Synthesis Infrastructure for the automatic synthesis of communication architectures, the Stochastic Analysis of Networked Embedded Systems tool (internally funded by UTRC), and the Stochastic Analysis and Design tool (funded by DARPA under contract #FA9550-10-C-0116). He is one of the key investigators in the DARPA funded META project where he is developing tools for the design of complex system using contracts.
In the past two years, he has been contributing as principal investigator to the autonomy initiative at UTRC in the area of probabilistic analysis and planning as well as design methodologies and V&V.

APPENDIX B – THE SEMANTICS OF HETEROGENEOUS MODELS

L. Mangeruca, A. Ferrari, O. Ferrante, A. Mignogna, A. L. Sangiovanni-Vincentelli

ALES S.r.l., U.C. Berkeley

ABSTRACT

We present a novel integration language for heterogeneous model composition. The language is anchored to a mathematically rigorous denotational semantics to provide a precise meaning to the composition of heterogeneous models. Nonetheless, it is not based on a common model of computation where others models of computation can be mapped, as previous works have proposed. On the contrary, the language is fully open. Both denotational and operational models of computation can be defined and provided as libraries that can be used to integrate and design heterogeneous models with precise semantics. Component's bodies can be specified in external languages and model transformation flows of different components to executable specifications can be safely integrated in multiple back-end analysis tools.
This article is a preprint to be submitted to journals and conferences in future after further revisions.

Index Terms— Models of computation, heterogeneous systems, embedded systems, denotational semantics, operational semantics, model transformation

1. INTRODUCTION

Embedded control systems have become of common use in our daily life. The trend shows an ever-increasing complexity of such system that most of the time should guarantee high performances, safety properties, low power consumption and low costs. The design of new embedded systems requires the integration of more components in a single chip and the interaction of several devices located in different places in the space. Often, the embedded system architectures include a wide variety of heterogeneous components: processors, application specific hardware, DSPs, sensors, actuators, etc. Additionally, a large number of actors are usually involved during the different phases of the design process. Teams, spread all around the world, contribute to the overall design, each one facing a particular design problem and therefore using specific design techniques and specific tools to solve it. The final design result in a composition of heterogeneous modules based on different Models of Computation (MoCC) and characterized by aperiodic and periodic computation, event-triggered and time-triggered communication and so on. As a consequence, the capability to support heterogeneity is necessary to deal with the design of such systems. During the entire design process, and especially during the very first development steps, the heterogeneity nature of components should be considered. During the last 10 years, different methodologies, frameworks and tools have been proposed to help the designer during the entire design process. However, there is still the lack of a unique integration framework that would be able to correctly compose models based on different MoCCs and to perform some analysis on the resulting system. What is required is a standard methodology to provide interoperability between models of different nature and to cover the whole design flow, from systems requirements to system implementation.
The paper is structured as follows: in Section 2 we present a simple example that we will use throughout the document. Related work is presented in Section 3. The mathematically defined meaning of heterogeneous composition is presented in the denotation semantics described in Section 4. Section 5 shows how the denotational semantics can be used to formally define the heterogeneous composition specified by the simple introduced in Section 2. In Section 6 we discuss how denotational and operational MoCCs are defined in the integration language and their relationships, also providing examples. Additionally, we show how the definition of operational MoCCs can be used to put in place a model transformation flow from the denotational model specification to its operational representation for analysis purposes. Finally, Section 7 concludes the paper.

2. A SIMPLE EXAMPLE

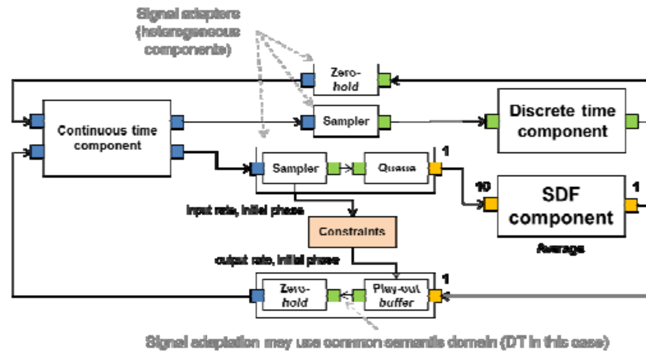


Figure 1: An example of heterogeneous composition

Figure shows an example of heterogeneous composition. We have components over four different models of computation and communication (MoCC): continuous time (CT), two periodic discrete time (DT(T_1,φ_1) and DT(T_2,φ_2), where T_1 and T_2 are rates and φ_1 and φ_2 are initial phases) and synchronous (untimed) dataflow (SDF) [1,2]. The composition between different MoCCs is achieved by means of heterogeneous processes. Informally, a process is called *homogeneous* if it operates over signals of the same MoCC, otherwise it is called *heterogeneous*. An heterogeneous process that has a single input signal and a single output signal over different MoCCs is called a *signal adaptor*. Note nonetheless that an heterogeneous process may have multiple signals over multiple MoCCs. As shown in **Figure 1**, signal adaptors can be composed to obtain new signal adaptors.

In the example of **Figure 1** we have three homogeneous components: a continuous time dynamics, a discrete time filter and an SDF component that computes the average every 10 samples coming from the continuous time component. Additionally, we have four adaptors to harmonize the different MoCCs: the sampler adapts CT signals into DT(T_1,φ_1) signals; the zero-hold converts DT signals into CT signals by holding the latest input value; the queue converts a DT signal into an untimed sequence of SDF tokens; the play-out buffer converts input SDF tokens into timed data on its DT(T_2,φ_2) output signal.

When interfacing different MoCCs several design choices might have to be taken. In the example of **Figure 1**, the play-out buffer produces a timing of the SDF tokens from its input signal. This timing is not completely arbitrary. It has in fact a relationship with the timing of the data produced by the sampler, via the rate relationship introduced by the SDF signals and component. Moreover, design constraints are enforced by the system engineer to ensure a maximum latency of the SDF computation chain.

3. PREVIOUS WORK

The problem of formally capturing the structure and behavior of heterogeneous systems has been already addressed by several authors. The tagged signal model approach proposes a theoretical framework for comparing properties of different models of computation using a denotational framework [5,6]. Based on this approach, several other solutions have been proposed to specialize the framework for an important subset of MoCCs [7]. Different specification languages and analyses frameworks have been developed to allow designers capturing heterogeneous systems. The PtolemyII and the Metropolis frameworks are modeling and simulation environments based on the tagged signal model theory [8,9]. The SPEEDS HRC language provides a common semantics and syntax to allow heterogeneous components hosted-simulation [10,11,12]. The MARTE UML profile constraints the semantics of the UML language providing a well-defined notion of time and supporting the specification of components exposing different MoCCs [13]. Other approaches uses the SystemC modeling language as glue language for the coordination and execution of heterogeneous components both using interface elements bridging components exposing different MoCCs [14] and extending the SystemC simulation capabilities to capture heterogeneous specification [15]. Most of the above approaches aim at providing a modeling and/or simulation environment for the specification and analysis of embedded systems. Exceptions are the tagged signal model, which provides a denotational framework for the definition of MoCCs, rather than models (although models can also be specified as we will show), and the SPEEDS environment that defines a language and protocol to exchange models between different tools. We focus on the integration issue of components specified in different languages, in turn defined over possibly different MoCCs.

The approach described in this document is focused on a SysML-based integration language where denotational and operational MoCCs can be defined as libraries and homogeneous (single MoCC) and heterogeneous (multiple MoCCs) processes can be defined and associated to the corresponding denotational MoCCs. Association of processes with operational MoCCs is done in a separate diagram to allow for different operational semantics of the same denotational specification.

Additionally, we show how the integration language supports multiple model transformation flows that allow mapping the denotational model specification to different analysis back-ends with maximal reuse of transformation tools and transformation artifacts.

4. MATHEMATICAL FRAMEWORK

The mathematical framework is based on the notions of tag domain (similar to the notion of clock in MARTE [3,4]), event, signal, behavior, and process (refinements from concepts defined in the Tagged Signal Model [5,6]). Informally, each signal is associated with a subset of tag domains. A tag is defined as an evaluation of a subset of tag domains. An event is an association of a tag with a value. A behavior associates each signal with a subset of events over the tag domains associated with the signal. An MoCC defines the allowed structure of tags and behaviors.

4.1. Definition of the SDF MoCC

Before providing the formal definition of the mathematical framework, we discuss a possible definition of the SDF model of computation. We will define a mathematical abstraction of the SDF MoCC, such that the balance equations can be derived and static scheduling analysis can be carried out, as described in [1]. In other words, in this section we abstract away actor functions, while in Section 5 we will show how we introduce them in our formalization. The SDF model of computation is defined in terms of a firing rule for each actor. The firing rules are expressed in terms of attributes associated with the signals, namely the token produced, the token consumed and the initial tokens (also called delay). The firing rules can be regarded as constraints on the allowed firings of the actors. Hence, to define the model of computation we need to express such constraints. To achieve this result we introduce a *tag domain* for each actor, representing the firings of the actor. Let I be a set of indices, one index for each SDF actor, for example the subset of the natural numbers $I = \{1, \ldots, M\}$. We consider a set of tag domains $K[I] = \{\kappa_i \mid i \in I\}$, each one taking values over $\mathbb{N}$, the set of natural numbers. The assignment of a value, a natural number in this example, to some tag domains is called a *tag*, i.e. in our example a tag is a function $\tau: K[I] \rightarrow \mathbb{N}$. Let $T[I]$ denote the set of tags over the tag domains $K[I]$. Since the firing rules are defined in terms of the attributes of the signals, we have to introduce a set of signals $S[I\times I] \subseteq \{s_{i,j} \mid i, j \in I\}$ and associate with signal $s_{i,j}$ the pair of tag domains $\{\kappa_i, \kappa_j\}$, corresponding to the tag domains of the actors that produce and consume tokens on that signal. In the SDF MoCC each signal is associated with three attributes: $s_{i,j}.p$ (number of tokes produced on the signal at each firing of actor i), $s_{i,j}.c$ (number of tokes consumed on the signal at each firing of actor j), $s_{i,j}.d$ (number of initial tokens). A *behavior* is a subset of events. An event associated with signal $s_{i,j}$ in the SDF MoCC is a triplet $(\tau,s_{i,j},v)$ associating a tag and a value to a signal, such that $v = (n, \nu) \in V$, where n is the number of tokens on the signal, i.e. $n = s_{i,j}.p*\tau(\kappa_i) - s_{i,j}.c*\tau(\kappa_j) + s_{i,j}.d$, and ν is a function that assigns a value to each token on the signal, i.e. $\nu: \mathbb{N} \rightarrow W\cup\{\perp\}$ such that $\nu(k) = \perp, \forall k \geq n$, where W is the set the token take value in and is no further specified. In the definition of the SDF MoCC, the token's value plays no role, so we can leave it no further specified and dependent on the particular SDF instantiation. Let $E[I,I\times I] = T[I]\times S[I\times I]\times V$ denote the set of events, then a *behavior* can be defined as a subset of events, i.e. $\sigma \subseteq E[I,I\times I]$. Not all subsets of events are behaviors. The SDF MoCC further constrains the allowed behaviors. To define such constraints recall that each signal is associated with three attributes, $s_{i,j}.p$, $s_{i,j}.c$, and $s_{i,j}.d$. The allowed set of behaviors can therefore be defined by $SDF[I,W] = \{\sigma \subseteq E[I,I\times I] \mid (\tau,s_{i,j},v) \in \sigma \Rightarrow \tau(\kappa_j)*s_{i,j}.c \leq \tau(\kappa_i)*s_{i,j}.p + s_{i,j}.d \wedge n = s_{i,j}.p*\tau(\kappa_i) - s_{i,j}.c*\tau(\kappa_j) + s_{i,j}.d \wedge v = (n,\nu)$, where $\nu: \mathbb{N} \rightarrow W\cup\{\perp\}$ s.t. $\nu(k) = \perp, \forall k \geq n\}$, in other words tokens must be produced on the signal before being consumed. Note that these constraints are a form of causality relationship. The synchronous dataflow MoCC over the set of indices I will be denoted SDF[I,W]. Note that the SDF MoCC can be instantiated with different token value sets.

4.2. Definition of the continuous time MoCC

To define the CT model of computation we only need a single real-valued tag domain representing real time, say κ_t, where t is a tag domain index that we reserve to represent real time. The set of tag domains is therefore $K[\{t\}] = \{\kappa_t\}$. The set of tags consists of functions $\tau: K[\{t\}] \rightarrow \mathbb{R}$, selecting a real value in the real-time domain. The set of event values is some N-dimensional Euclidean space. Given a set of signals $S[J] = \{s_j \mid j \in J\}$, defined over the set of signal indices J. Informally, a behavior is allowed in the CT MoCC if all events (τ,s_j,v) in it identify points in the curve of some piece-wise continuous function f. Formally, the set of allowed behaviors is restricted to $CT[J] = \{\sigma \subseteq E[\{t\},J] \mid \exists f_j: \mathbb{R} \rightarrow \mathbb{R}^{N_j}$, piece-wise continuous, such that $(\tau,s_j,v) \in \sigma \Rightarrow f_j(\tau(\kappa_t)) = v\}$. The continuous time MoCC over the set of signal indices J will be denoted by CT[J].

4.3. Definition of periodic discrete time MoCCs

Periodic discrete time models of computation form a family of MoCCs distinguished by two attributes: the period T_i and the initial phase φ_i. We only need a single tag domain per MoCC representing the clock, say $\kappa_{(T,\varphi)}$, taking values in the set of natural numbers. The tag domain index consisting of the pair (T,φ) is used for the periodic discrete time MoCC. The set of tag domains is therefore $K[\{(T,\varphi)\}] = \{\kappa_{T,\varphi}\}$. The set of tags consists of functions τ: $K[\{(T,\varphi)\}] \rightarrow \mathbb{N}$, assigning a natural number to the clock. The set of event values is some N-dimensional Euclidean space. Given a set of signals $S[J] = \{s_j \mid j \in J\}$, defined over the set of signal indices J. The set of behaviors $DT[T,\varphi,J] = \{\sigma \subseteq E[\{(T,\varphi)\},J]\ \}$ is not restricted in this class of MoCCs. The periodic discrete time MoCC over the set of signal indices J will be denoted by $DT[T,\varphi,J]$.

4.4. Formal definition of the mathematical framework

Let $K[I] = \{\kappa_i \mid i \in I\}$ denote a set of *tag domains* and let $V[I] = \{V_i \mid i \in I\}$, where V_i denotes the partially or totally ordered set, where the tag domain κ_i takes up value in. A *tag* is defined as a function τ: $K[I] \rightarrow \bigcup_{i\in I}V_i$, that associates each tag domain $\kappa_i \in K[I]$ with a value $\tau(\kappa_i) \in V_i$. Let T[I] denote the set of tags over K[I]. T[I] has an induced partial ordered defined by $\tau_1 \leq \tau_2 \Leftrightarrow \forall\kappa_i \in K[I], \tau_1(\kappa_i) \leq \tau_2(\kappa_i)$. Let $S[J] = \{s_j \mid j \in J\}$ denote a set of *signals* and let $V[J] = \{V_j \mid j \in J\}$, where V_j denotes the set, where the signal s_j takes up value in. An *event* is defined as a triplet (τ,s_j,v), such that $v \in V_j$. Let E[I,J] denote the corresponding set of events over tag domain indices in I and signal indices in J.

A *behavior* $\sigma[I,J]$ is defined as a subset of events such that it is deterministic, i.e. there is a unique event for every tag on each signal. Formally, $\sigma[I,J] \subseteq E[I,J]$ such that $(\tau,s,v_1), (\tau,s,v_2) \in \sigma[I,J] \Rightarrow v_1 = v_2$. To simplify notation, we will omit the indices specification [I,J] whenever it is clear from the context. Let $\Sigma[I,J]$ denote the set of behaviors over the set of events E[I,J]. A behavior is called *complete* if on each signal it has an event for every value of each tag domain. Formally, $\sigma[I,J]$ is complete if $\forall\kappa_i \in K[I], \forall s_j \in S[J], \{\tau(\kappa_i) \mid (\tau,s_j,v) \in \sigma[I,J]\} = V_i$. A behavior is called *incomplete*, if it is not complete. A behavior $\sigma[I,J]$ is *left-tag-bounded* if there exists $\tau' \in T[I]$ such that $(\tau,s_j,v) \in \sigma[I,J] \Rightarrow \tau' \leq \tau$. Similarly, $\sigma[I,J]$ is *right-tag-bounded* if there exists $\tau' \in T[I]$ such that $(\tau,s_j,v) \in \sigma[I,J] \Rightarrow \tau \leq \tau'$. A behavior is said to be *tag-bounded* if it is both left- and right-tag-bounded.

A *model of computation* over the set of tag domain indices I and signal indices J is defined as a subset of $\Sigma[I,J]$. For example, following sections 4.1, 4.2 and 4.3, the synchronous dataflow MoCC is represented by a set of behaviors $SDF[I,W] \subseteq \Sigma[I,I\times I]$, where I is a subset of $\mathbb{N}$, the set of natural numbers, $\forall V_i \in V[I], V_i = \mathbb{N}, \forall V_j \in V[J], V_j = \{(n,\nu) \mid n \in \mathbb{N} \wedge \nu: \mathbb{N} \rightarrow W\cup\{\perp\} \text{ s.t. } \nu(k) = \perp, \forall k \geq n\}$. The continuous time MoCC is represented by a set of behaviors $CT[J] \subseteq \Sigma[\{t\},J]$, where t is the tag domain index representing real-time, J is a subset of $\mathbb{N}$, $V_t = \mathbb{R}$ and $\forall V_j \in V[J], V_j = \mathbb{R}^{N_j}$, where $N_j \in \mathbb{N}\setminus\{0\}$. The periodic discrete time MoCC is represented by a set of behaviors $DT[T,\varphi,J] \subseteq \Sigma[\{(T,\varphi)\},J]$, where (T,φ) is the tag domain index of the discrete time with period T and initial phase φ, $V_{(T,\varphi)} = \mathbb{N}$, and $\forall V_j \in V[J], V_j = \mathbb{R}^{N_j}$, where $N_j \in \mathbb{N}\setminus\{0\}$.

A *process* is defined by introducing additional constraints on behaviors. A process in general constrains behaviors with respect to a subset of the signals, i.e. $P[I,J'] \subseteq \Sigma[I,J]$, where $J' \subseteq J$. We define process composition as follows: $P_1[I_1,J_1]\|P_2[I_2,J_2] = \{\sigma \subseteq E[I_1\cup I_2,J_1\cup J_2]\ \mid (\tau,s,v) \in \sigma \Rightarrow (s \in S[J_1]\cup S[J_2] \wedge (s \in S[J_1] \Rightarrow \exists(\tau_1,s,v) \in \sigma_1 \in P_1[I_1,J_1] \text{ s.t. } \tau|_{K1} = \tau_1) \wedge (s \in S[J_2] \Rightarrow \exists(\tau_2,s,v) \in \sigma_2 \in P_2[I_2,J_2] \text{ s.t. } \tau|_{K2} = \tau_2))\}$. Note that if $I_1 = I_2$ and $J_1 = J_2$, then $P_1[I_1,J_1]\|P_2[I_2,J_2] = P_1[I_1,J_1]\cap P_2[I_2,J_2]$. Note that the formula $P[I_1,J_1]\|\Sigma[I_2,J_2] = P[I_1\cup I_2,J_1\cup J_2]$ provides the extension of process P to the tag domains with indices in I_2 and the signals with indices in J_2. If $I_1 = I_2$ and $J_1 = J_2$, then $P[I_1,J_1]\|\Sigma[I_2,J_2] = P[I_1,J_1]$.

MoCCs are also defined as subsets of behaviors. Hence, composition of MoCCs is defined in the same way as process composition. An MoCC M is said to be *compositional* if $M[I_1,J_1]\|M[I_2,J_2] = M[I_1\cup I_2,J_1\cup J_2]$. Note that CT, DT and SDF are compositional MoCCs. A process is called *homogeneous* if it is a subset of allowed behaviors of a given MoCC. For example, $P[I,J',W] \subseteq SDF[I,W]$, where $J' \subseteq I\times I$. is an homogeneous SDF process. A process is called *heterogeneous* if it is a subset of behaviors of multiple MoCCs. For example, the process $P[T,\varphi,J_1\cup J_2] \subseteq CT[J_1]\|DT[T,\varphi,J_2]$ is an heterogeneous process defined over the continuous time and discrete time MoCCs.

A *(homogeneous) connection* is a special kind of process that constrains two or more signals to be equal. Formally, a connection is a process $C[I,J] = \{\sigma \subseteq E[I,J] \mid (\tau,s_j,v) \in \sigma \Leftrightarrow (\tau,s_k,v) \in \sigma, \forall j, k \in J\}$. Connections can be used to connect processes that are defined over disjoint sets of signal indices of the same model of computation. To connect processes over different models of computation heterogeneous connections are needed. Contrary to homogeneous connections, a general definition for heterogeneous connections does not exist, as the adaptation between signals over different MoCCs is not unique in general and depends on the user's needs. For such reasons, heterogeneous connections are also called adapters and examples will be provided in Section 5.

Note that in our mathematical framework tag domains are general structures not limited to represent temporal behaviors. Tag domains may represent other physical coordinates, such as space, velocity, etc., as well as more abstract frame of reference such as position in a graph. Hence, our mathematical framework can appropriately represent and integrate the representation

of spatiotemporal partial differential equations, positioning of geometrical objects in space, Hamiltonian and Lagrangian functions, constraints and evolutionary equations over a graph, etc.
As an example, consider the heat transfer equation:

$$\frac{\partial^2 h}{\partial t^2} - \alpha\left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} + \frac{\partial^2 h}{\partial z^2}\right) = 0$$

To represent it in our mathematical framework, we define four tag domains, three spatial tag domains x,y,z and a temporal tag domain t, so that K[I] = {x,y,z,t}. The process that represents the heat has a signal 's' defined over the four tag domains introduced above, whose value represents the heat at a given point in space and time. Let I be the set of indices associated with the tag domains x,y,z,t and J the set containing the index of signal 's', we can formally define the heat transfer process as $P[I,J] = \{(\tau,s,v) \mid \exists h: \mathbb{R}^4 \rightarrow \mathbb{R} \text{ s.t. } \partial^2 h/\partial t^2 - \alpha(\partial^2 h/\partial x^2+\partial^2 h/\partial y^2+\partial^2 h/\partial z^2)=0 \wedge v = h(\tau(x,y,z,t))\}$.

4.5. Definition of contracts

In this section we provide the semantic foundations for contracts, based on the mathematical framework introduced in Section 4.4. A contract is a pair of processes C[I,J] = (A[I,J],G[I,J]), where the process A[I,J] plays the role of the assumptions (acceptable behaviors), while the process G[I,J] plays the role of the promises or guarantees (possible behaviors). To provide a meaning to the contract, we define how to interpret it as a process. A contract is interpreted as a convenient decomposition of the process $G[I,J]\cup\neg A[I,J]$. Hence, to define a contract we need to define the operations of complementation and union of processes. The complement of a process P[I,J] is defined as $\neg P[I,J] = (\Sigma[I,J] \setminus P[I,J])$. The union of two processes $P_1[I_1,J_1]$ and $P_2[I_2,J_2]$ is defined as $P_1[I_1,J_1]\cup P_2[I_2,J_2] = (P_1[I_1,J_1]\|\Sigma[I_2,J_2])\cup(P_2[I_2,J_2]\|\Sigma[I_1,J_1])$. It can be shown that, with this interpretation of a contract as a process, the contract algebra defined in the SPEEDS project can be used to compose and relate contracts.

4.6. Extension to stochastic processes

Before providing the definition of stochastic processes in our mathematical framework, we discuss a simple example to introduce the basic concepts. Let us consider the modeling of the NOT gate with probability error p. If we abstract time, the set of behaviors associated to the non-stochastic NOT gate can be represented by the pairs {(0,1), (1,0)}. We create the model of a stochastic NOT by defining a set of conditional probability measures over subsets of behaviors, pairs in this case.
To show how the stochastic model of the NOT gate is defined, we consider one specific conditional probability measure as an example: let a and b denote the input and output signals, respectively, and consider the conditional probability measure given by p({b=0}/{a=0}) = p({b=1}/{a=1}) = p and p({b=0}/{a=1}) = p({b=1}/{a=0}) = 1 – p, where {a=0} = {(0,0), (0,1)}, {b=0} = {(0,0), (1,0)}, {a=1} = {(1,0), (1,1)} and {b=1} = {(0,1), (1,1)}. Note that, since we want to have a compositional definition of stochastic process, i.e. we want to specify local probabilities associated to the process, we need to consider conditional probability measures, rather than unconditional probability measures.
To show how composition works, let us compose two NOT gates, where a and b denote the input and output signals of the first NOT gate, while b and c denote the input and output signals of the second NOT gate. Let p and q the error probabilities of the first and second NOT gate, respectively. First we need to extend the sample space representation from pairs to triplets, so that {a=0} = {(0,0,0), (0,1,0), (0,0,1), (0,1,1)}, {b=0} = {(0,0,0), (1,0,0), (0,0,1), (1,0,1)}, {c=0} = {(0,0,0), (1,0,0), (0,1,0), (1,1,0)}, {a=1} = {(1,0,0), (1,1,0), (1,0,1), (1,1,1)}, {b=1} = {(0,1,0), (1,1,0), (0,1,1), (1,1,1)}, {c=1} = {(0,0,1), (1,0,1), (0,1,1), (1,1,1)}. Then, we need to define the conditional probability measure of the composition of the two NOT gates, which must agree with the conditional probability measure of each single NOT gate. Hence, the conditional probability measure of the composition is such that p({b=0}/{a=0}) = p({b=1}/{a=1}) = p, p({b=0}/{a=1}) = p({b=1}/{a=0}) = 1 – p and p({c=0}/{b=0}) = p({c=1}/{b=1}) = q, p({c=0}/{b=1}) = p({c=1}/{b=0}) = 1 – q. From the composition of conditional probabilities, we have that $p(\{c=1\}/\{b=1\})\ p(\{b=1\}/\{a=1\}) = p(\{c=1\}\cap\{b=1\}/\{a=1\}) = p(\{c=1\wedge b=1\}/\{a=1\}) = p*q$ and $p(\{c=1\}/\{b=0\})\ p(\{b=0\}/\{a=1\}) = p(\{c=1\}\cap\{b=0\}/\{a=1\}) = p(\{c=1\wedge b=0\}/\{a=1\}) = (1-p)*(1-q)$. Moreover, from the law of total probability, we have $p(\{c=1\}/\{a=1\}) = p(\{c=1\wedge b=1\}/\{a=1\}) + p(\{c=1\wedge b=0\}/\{a=1\}) = p*q+(1-p)*(1-q)$.

4.7. Definition of stochastic processes

In this section we extend the mathematical framework introduced in Section 4.4 to be able to represent stochastic processes. This is achieved by introducing probability spaces (see http://en.wikipedia.org/wiki/Probability_space). A probability space is a triplet (Ω,Φ,f), where Ω is the sample space, Φ is a σ-algebra over Ω, where we further require that $\Omega \in \Phi$, and f is a probability measure over Φ, i.e. $f: \Phi \rightarrow [0,1]$ with the usual axioms of probability.

In our framework we use behaviors as samples and regular (non-stochastic) processes as sample spaces. Hence, the elements of Φ are subsets of behaviors of a regular process. To simplify notation, we will represent a probability space over the set of behaviors $\Sigma[I,J]$ as a function $f[I,J]: 2^{\Sigma[I,J]} \to [0,1]\cup\{\perp\}$ such that $\Phi(f[I,J]) = \{\phi \in 2^{\Sigma[I,J]} \mid f[I,J](\phi) \in [0,1]\}$ is a σ-algebra and $f[I,J]|_{\Phi[I,J]}$ is a probability measure.

To have a compositional definition of stochastic processes, we replace probability spaces with conditional probability spaces. We represent a conditional probability space over the set of behaviors $\Sigma[I,J]$ as a function $p[I,J]: 2^{\Sigma[I,J]} \times 2^{\Sigma[I,J]} \to [0,1]\cup\{\perp\}$ such that $\Phi(p[I,J]) = \{\phi_1, \phi_2 \in 2^{\Sigma[I,J]} \mid p[I,J](\phi_1,\phi_2) \in [0,1]\}$ is a σ-algebra and $p[I,J]|_{\Phi[I,J]\times\Phi[I,J]}$ is a conditional probability measure, where $p[I,J](\phi_1,\phi_2) \in [0,1]$ is interpreted as the probability of ϕ_1 given ϕ_2. We will omit the signal specification [I,J] when it is clear from the context. Note that $p(\phi,\Sigma[I,J])$ can be interpreted as the unconditional probability of ϕ. Hence, $\forall\phi_1,\phi_2,\phi_3 \in \Phi(p)$, $p(\phi_3,\Sigma[I,J]) \in (0,1] \Rightarrow p(\phi_1,\phi_2)p(\phi_2,\phi_3) = p(\phi_1\cap\phi_2,\phi_3)$ (law of composition of conditional probabilities) and for all partitions $K \subseteq \Phi(p)$ of ϕ_1, $\sum_{\phi\in K} p(\phi_1\cap\phi,\phi_2) = p(\phi_1,\phi_2)$ (law of total probability). The set of all conditional probability measures over the set of behaviors $\Sigma[I,J]$ will be denoted $\Gamma[I,J]$.

A stochastic process is defined as a subset $\Pi[I,J] \subseteq \Gamma[I,J]$ of conditional probability measures. Given two stochastic processes, $\Pi_1[I_1,J_1]$ and $\Pi_2[I_2,J_2]$, we define their composition as follows: $\Pi_1[I_1,J_1]\|\Pi_2[I_2,J_2] = \{p \in \Gamma[I_1\cup I_2,J_1\cup J_2] \mid \exists p_i \in \Pi_i$ s.t. $\forall\phi_1,\ \phi_2 \subseteq \Sigma[I_i,J_i],\ p(\phi_1\|\Sigma[I_j,J_j],\phi_2\|\Sigma[I_j,J_j]) = p_i(\phi_1,\phi_2)$, where $i,j=1,2$ and $i\neq j\}$. Similarly to regular processes, $\Pi[I_1,J_1]\|\Gamma[I_2,J_2] = \Pi[I_1\cup I_2,J_1\cup J_2]$ provides the extension of the stochastic process Π to the tag domains with indices in I_2 and the signals with indices in J_2. Note that if $I_1 = I_2$ and $J_1 = J_2$, then $\Pi[I_1,J_1]\|\Gamma[I_2,J_2] = \Pi_1[I_1,J_1]$.

A regular (non-stochastic) process can be seen as a special case of a stochastic process as defined in the present section. A regular process can be represented by a stochastic process $\Pi[I,J]$ if $\{p(\phi_1,\phi_2) \mid p \in \Pi[I,J],\ \phi_1,\ \phi_2 \subseteq \Sigma[I,J]\} = \{0\}, \{1\}, \{\perp\}$, or $[0,1]$.

4.8. Definition of stochastic contracts

In this section we provide the semantic foundations for stochastic contracts, based on the mathematical framework introduced in Section 4.4 and the stochastic extension given in Section 4.6. A stochastic contract is a pair of stochastic processes $C[I,J] = (A[I,J],G[I,J])$, where the process $A[I,J]$ plays the role of the assumptions, while the process $G[I,J]$ plays the role of the promises or guarantees. To provide a meaning to the stochastic contract, we define how to interpret it as a stochastic process. A stochastic contract is interpreted as a convenient decomposition of the stochastic process $G[I,J]\cup\neg A[I,J]$. The complement of a stochastic process $\Pi[I,J]$ is defined as $\neg\Pi[I,J] = \Gamma[I,J]\backslash\Pi[I,J]$. The union of two stochastic processes, $\Pi_1[I_1,J_1]$ and $\Pi_2[I_2,J_2]$, is defined as $\Pi_1[I_1,J_1]\cup\Pi_2[I_2,J_2] = (\Pi_1[I_1,J_1]\|\Gamma[I_2,J_2]) \cup (\Pi_2[I_2,J_2]\|\Gamma[I_1,J_1])$.

4.9. Definition of denotational primitives

Denotational primitives represent a basic notation to simplify the mathematical definition of processes, adapters and MoCCs. The list provided in this section is not intended to be complete, but only explanatory and for the purpose of this document. Denotational primitives comprise both *constraints* and *operators*. Constraint denotational primitives are defined as sets of behaviors and as such can be regarded as a kind of processes.

Operator denotational primitives can operate on tags, events, behaviors or sets of behaviors. Below we present some examples of operators over behaviors and an example of an operator over tags. Operators over behaviors are extended to sets of behaviors by applying them element-wise. To improve readability, operator composition will be denoted $(O_1 \circ O_2)(\sigma)$, to mean $O_1(O_2(\sigma))$. To simplify notation, we will assume that whenever we use the mathematical operators $\leq$ (partial order), + (addition), – (subtraction) over tag domain values, such operators are defined in the relevant subset of the tag domain values. Additionally, we consider the following notation: $a < b$ is equivalent to $a \leq b \wedge a \neq b$.

4.9.1. Constraints

Phase constraint: $\Phi_R[\kappa_1,\kappa_2,L_B,U_B] = \{\sigma \mid (\tau,s,v) \in \sigma \Rightarrow L_B \leq \tau(\kappa_1) - \tau(\kappa_2) \leq U_B\}$;

Sampling relation: $S[T,\varphi] = \{\sigma \mid (\tau,s,v) \in \sigma \Rightarrow 0 \leq \tau(\kappa_t) < \varphi \vee \tau(\kappa_{T,\varphi})*T + \varphi \leq \tau(\kappa_t) < \tau(\kappa_{T,\varphi})*T + T + \varphi\}$;

4.9.2. Operators

Signal renaming: $\Lambda[s',s](\sigma) = \{(\tau,s,v) \mid (\tau,s',v) \in \sigma\}$;

Phase translation: $\Phi[\varphi](\sigma) = \{(\tau,s,v) \mid (\tau',s,v) \in \sigma \wedge \tau(\kappa) = \tau'(\kappa) - \varphi\}$;

Sampling: $\Psi[T](\sigma) = \{(\tau,s,v) \mid (\tau',s,v) \in \sigma \wedge \tau'(\kappa) = \tau(\kappa)*T\}$;

Signal projection: $\Pi[J](\sigma) = \{(\tau,s,v) \in \sigma \mid s \in S[J]\}$;

Tag domain projection: $\Pi_K[I](\sigma) = \{(\tau',s,v) \mid (\tau,s,v) \in \sigma \wedge \tau|_{K[I]} = \tau'\}$;

Prefix: $P_X[\tau'](\sigma) = \{(\tau,s,v) \in \sigma \mid \tau \leq \tau'\}$;

Next tag: $X[\kappa](\tau) = \tau'$ s.t. $\tau'(\kappa) = \tau(\kappa) + 1$.

We use the signal projection operator to introduce the concept of functional process. A process $P[I,J]$ is *functional* with respect to $J_O \subseteq J$ if $\forall\sigma \in P[I,J]$, $\Pi[J \backslash J_O](\sigma_1) = \Pi[J \backslash J_O](\sigma_2) \Rightarrow \Pi[J_O](\sigma_1) = \Pi[J_O](\sigma_2)$. The choice for J_O is not unique in general and J_O is maximal if it cannot be enlarged further while keeping the process functional with respect to it. If there is a unique maximal J_O such that the process is functional with respect to it, then the signals with indices in J_O are called the output (controlled) signals of the process. A functional process $P[I,J]$ with respect to $J_O \subseteq J$ is *receptive* with respect to $J_I \subseteq J$ if $\forall\sigma' \in P[I,J]$, $\{\Pi[J_I](\sigma) \mid \sigma \in P[I,J] \wedge \Pi[J \backslash J_O \backslash J_I](\sigma) = \Pi[J \backslash J_O \backslash J_I](\sigma')\} = \Sigma[I,J_I]$. The choice for J_I is not unique in general. If there is a unique maximal J_I for all maximal J_O such that the process is functional and receptive, then the signals with indices in J_I are called the input (uncontrolled) signals of the process. A functional process $P[I,J]$ with respect to $J_O \subseteq J$ that is receptive with respect to J_I is *total* if $J_O \cup J_I = J$.

5. SIGNAL ADAPTATION

Given the mathematical framework introduced in Section 4, we show how to formalize the signal adaptation processes. In particular, we show it on the example introduced in Section 2.

5.1. Definition of sampler

The sampler is an heterogeneous process that transforms a signal from continuous time to periodic discrete time. The sampler is associated with two attributes, the period T_i and the initial phase φ_i, that are also the period and initial phase of the target discrete time domain. As defined in section 4.4, the sampler can be defined by providing the constraints it imposes on the allowed behaviors.
Let $J_1 = \{1\}$ and $J_2 = \{2\}$ be two sets of indices and $V_1 = V_2 = \mathbb{R}$ be the corresponding event value sets, then the sampler is a process $P_{sampler}[T_1,\varphi_1,J_1 \cup J_2] \subseteq CT[J_1] \| DT[T_1,\varphi_1,J_2]$. Formally, the behaviors allowed by the sampler can be defined as follows: $P_{sampler}[T_1,\varphi_1,J_1 \cup J_2] = \{\sigma \in CT[J_1] \| DT[T_1,\varphi_1,J_2] \mid \sigma \in S[T_1,\varphi_1] \wedge \Pi[J_2](\sigma) = (\Lambda[s_1,s_2] \circ \Psi[T_2] \circ \Phi[\varphi_2] \circ \Pi[J_1])(\sigma)\}$.

5.2. Definition of lossless queue

The lossless queue is an heterogeneous process that transforms a signal from periodic discrete time to SDF. The queue is associated with an attribute, say L_Q, which is the length of the queue. As defined in section 4.4, the queue can be defined by providing the constraints it imposes on the allowed behaviors. Let $J_2 = \{2\}$ and $I_3 = \{3,4\}$. The instantiation of the $SDF[I_3,\mathbb{R}]$ MoCC has therefore two actors, the queue and the SDF average component, which we associate with the indices 3 and 4, respectively. We assume that the attributes of the SDF signal with index (3,4) are as follows: $s_{3,4}.p = 1$, $s_{3,4}.c = 10$, $s_{3,4}.d = d$. Note that the set of token values in the SDF MoCC must comply with the event value set of the DT MoCC. The lossless queue is a process $P_{queue}[\{(T_1,\varphi_1)\} \cup I_3, \{J_2 \cup (3,4)\}, \mathbb{R}] \subseteq DT[T_1,\varphi_1,J_2] \| SDF[I_3,\mathbb{R}]$. The behaviors allowed by the lossless queue are constrained as follows: $P_{queue}[\{(T_1,\varphi_1)\} \cup I_3, J_2 \cup \{(3,4)\}, \mathbb{R}] = \{\sigma \in DT[T_1,\varphi_1,J_2] \| SDF[I_3,\mathbb{R}] \mid \sigma \in \Phi_R[\kappa_{T1,\varphi1},\kappa_3,0,L_Q] \wedge ((\tau_3,s_{3,4},v_3) \in \sigma \Leftrightarrow (\tau_2,s_2,v_2) \in \sigma) \wedge \tau_2(\kappa_{T1,\varphi1}) = j + 10*\tau_3(\kappa_4) - d \wedge v_2 = \nu(j) \wedge 0 \leq j < \tau_3(\kappa_3) - 10*\tau_3(\kappa_4) + d$, where $v_3 = (\tau_3(\kappa_3) - 10*\tau_3(\kappa_4) + d, \nu)\}$. Note that the expression $\nu(j) = v_2$ imposes the constraint that the output signal contain the values as they arrive on the input signal, taking into account the tokens consumed by the SDF average component and the delay on the signal.

5.3. Definition of play-out buffer

The play-out buffer is an heterogeneous process that transforms a signal from SDF to periodic discrete time. The play-out buffer is associated with an attribute, say L_B, which is the length of the buffer. As defined in section 4.4, the play-out buffer can be defined by providing the constraints it imposes on the allowed behaviors. Let $I_4 = \{4,5\}$ and $J_6 = \{6\}$. The instantiation of the SDF$[I_4,\mathbb{R}]$ has two actors, the SDF average component and the play-out buffer, which we associate with the tag domain indices 4, and 5, respectively. We assume that the attributes of the SDF signal with index (4,5) are as follows: $s_{4,5}.p = 1$, $s_{4,5}.c = 1$, $s_{4,5}.d = 0$. The play-out buffer is a process $P_{buffer}[\{(T_2,\varphi_2)\}\cup I_4,\{(4,5)\cup J_6\},\mathbb{R}] \subseteq SDF[I_4,\mathbb{R}]||DT[T_2,\varphi_2,J_6]$. The behaviors allowed by the play-out buffer are constrained as follows: $P_{buffer}[\{(T_2,\varphi_2)\}\cup I_4,\{(4,5)\cup J_6\},\mathbb{R}] = \{\sigma \in SDF[I_4,\mathbb{R}]|| DT[T_2,\varphi_2,J_6] \mid \sigma \in \Phi_R[\kappa_5,\kappa_{T2,\varphi 2},0,L_B] \wedge ((\tau_5,s_{4,5},v_5) \in \sigma \Leftrightarrow (\tau_6,s_6,v_6) \in \sigma) \wedge \tau_6(\kappa_{T2,\varphi 2}) = j + \tau_5(\kappa_5) \wedge 0 \le j < \tau_5(\kappa_4) - \tau_5(\kappa_5) \wedge v_6 = \nu(j)$, where $v_5 = (\tau_5(\kappa_4) - \tau_5(\kappa_5),\nu)\}$.

6. HOMOGENEOUS PROCESS DEFINITIONS

To provide a complete description of the SDF part of the example, we show how to define the SDF average component.

6.1. Definition of SDF average

The SDF average component is an homogeneous process defined over the SDF MoCC. As defined in section 4.4, the play-out buffer can be defined by providing the constraints it imposes on the allowed behaviors. Let $I_5 = \{3,4,5\}$. Since the SDF MoCC is compositional, the instantiation of the SDF for the SDF average component is the composition of the instantiations we have done in the previous sections, i.e. $SDF[I_5,\mathbb{R}] = SDF[I_3,\mathbb{R}]||SDF[I_4,\mathbb{R}]$. The SDF average component is a process $P_{buffer}[I_3] \subseteq \Sigma_{SDF}[I_3,\mathbb{R}]$. The behaviors allowed by the SDF average component are constrained as follows: $P_{average}[I_3] = \{\sigma \in \Sigma_{SDF}[I_3,\mathbb{R}] \mid \sigma \in \Phi_R[\kappa_5,c_k[T_4,\varphi_4],0,L_B] \wedge ((\tau_4,s_{3,4},v_4) \in \sigma \Leftrightarrow (\tau_5,s_{4,5},v_5) \in \sigma) \wedge j_1 + 10*\tau_4(\kappa_4) = 10*(j_2 + \tau_5(\kappa_5)) \wedge j_1 + 10 \le \tau_4(\kappa_3) - 10*\tau_4(\kappa_4) + d \wedge \nu_2(j_2) = (\sum_{j\in\{0,\ldots,9\}} \nu_1(j+j_1+1))/10$, where $v_4 = (\tau_4(\kappa_3) - 10*\tau_4(\kappa_4) + d,\nu_1)$ and $v_5 = (\tau_5(\kappa_4) - \tau_5(\kappa_5),\nu_2)\}$.

7. OPERATIONAL SEMANTICS

The operational semantics is defined in the META II language by the denotational MoCC and operational MoCC (OMoCC for short) libraries. Informally, a denotational MoCC represents a definition of a denotational integration of blocks (clear/white-box), while an OMoCC represents the definition of an operational integration of block implementations (black-box). The denotational MoCC library provides a set of MoCC definitions that can be related by the "extends" relation. Each specific MoCC can define attributes whose values are to be specified by block definitions or flows associated with that MoCC. A specific MoCC can define parameters that can be used in the definition of the attributes, such as for example some attribute's type, so that the MoCC definition is actually a template that can be instantiated with different types. A specific MoCC, that extends another MoCC, inherits the other MoCC's attributes. Attributes can be constant or variable. Constant attributes can be given value on the META II model specification. Variable attributes can be given (dynamic) value using an associated white-box language.
The META II Operational MoCC library provides a set of OMoCC definitions that can be related by the "extends" relation. A specific OMoCC can be related to specific MoCCs by the implementation relation. Each specific OMoCC can define attributes and functions that are to be specified by block definitions or flows associated with that OMoCC. The functions represent the API that the block's implementations are expected to implement to comply with the corresponding operational semantics. Additionally, an OMoCC can specify functions that the OMoCC solver is expected to implement. These functions can be used by block's implementations or by other OMoCC solvers. A specific OMoCC can define parameters that can be used in the definition of the attributes and functions.

7.1. Transformation flow for operational analysis

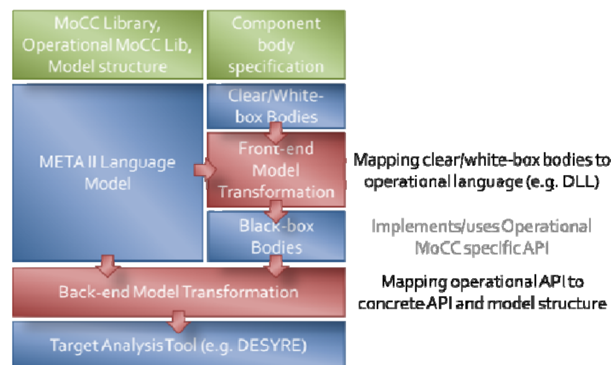


Figure 2: Transformation flow

In Figure 2 the transformation flow of the denotational specification to an operation specification is depicted. The flow consists of the following phases.

7.1.1. MoCC and Operational MoCC library definition

1. The META II integration language is provided with an MoCC library and an Operational MoCC library; both libraries internally support inheritance; moreover, an Operational MoCC can implement MoCCs;
2. Each MoCC definition in the MoCC library includes MoCC-specific attributes required of the block definitions and flows of that MoCC;
3. Each Operational MoCC definition in the Operational MoCC library includes MoCC-specific attributes and functions required of and provided to the block definition operational implementation.

7.1.2. System model definition

1. The user specifies the model structure in the META II integration language;
2. The user associates each block definition and/or flow with a specific MoCC taken from the META II MoCC library; this association enriches the block definition and/or the flow with MoCC-specific attributes;
3. The user specifies the value of the MoCC-specific attributes for each block definition and/or flow associated with that MoCC;

4. The user associates each block definition with a specific Operational MoCC taken from the META II Operational MoCC library; this association is specified on a separate diagram, to allow for multiple OMoCC associations, according to analysis needs;
5. The user associates each leaf block (a block that has no associated internal block diagram) with a body specification defined in: 1) SysML (clear-box); 2) external language (white-box); 3) external executable specification (black-box); this association is specified on a separate diagram, to allow for different block's body specifications, according to analysis needs.

7.1.3. Front-end model transformation

The front-end model transformation takes the META II model integration specification and the leaf blocks' body definitions and produces an operational representation; the operational representation will be given in some executable language such as C/C++, static lib, DLL, etc.; the operational representation expose the operational API defined in the specific Operational MoCC associated with the original block, see point 3 in Section 7.1.1.

7.1.4. Back-end model transformation

The back-end model transformation takes care of two actions:
1. Translates the structural model representation from the META II language into the target analysis tool;
2. Adapts the operational API between the black-box body and the concrete API of the corresponding specific OMoCC solver in the target analysis tool.

The API adaptation at point 2 is necessary, because different solvers may have the same API structure and semantics, but different concrete syntax. In the Operational MoCC library we can factorize several operational semantics, which differ only on the syntax and not on the structure, into a single OMoCC specification, or we can use inheritance to share common API structure among similar but not equal operational semantics.

7.2. Requirements on the META II language

To put in place the transformation flow, MoCC and Operational MoCC specificatiosn, described in Section 7.1, support from the META II language is needed. Following are a set of requirements on the language.
1. The META II language shall allow to associate a block definition with: 1) an internal block diagram (inherited from SysML); 2) a clear-box body definition (inherited from SysML); 3) a body definition expressed in an external language (white-box); 4) a body definition expressed in an external executable specification (black-box); this association shall be specified on a separate diagram, to allow for different block's body specifications, according to different analysis needs;
2. The general concept of MoCC shall be provided by the language (SemanticDomain in the current profile version of the language) and shall support the "extens" relation between MoCCs;
3. Specific MoCCs shall be defined as libraries of the language;
4. It shall be possible to associate each block definition and/or each flow with a specific MoCC; a flow that is not associated with an MoCC, shall inherit the MoCC specification from the block it belongs to;
5. Any flows shall be of kind "in", "out", or "in-out";
6. It shall be possible to connect two flows, say f_1 and f_2, with different MoCC associations if the MoCC associated with f_1 is a specialization of the MoCC associated with f_2 and f_2 is an "in" or an "in-out" flow;
7. Each MoCC shall specify block attributes and flow attributes that are characteristic of the MoCC; some attributes may depend on other attributes; each block or flow definition that is associated with the MoCC shall be allowed to customize the values of such attributes;
8. MoCCs shall be able to specify parameters, that the value or type of some attributes may depend on;
9. The general concept of Operational MoCC shall be provided by the language and shall support the "extends" relation between OMoCCs;
10. It shall be possible to define associations between an OMoCC and one or more MoCCs to represent the implementation relation;
11. Specific Operational MoCCs shall be defined as libraries of the language;
12. Each OMoCC shall be able to specify block attributes and flow attributes that are characteristic of the OMoCC; some attributes may depend on other attributes;
13. Each OMoCC shall be able to specify block functions and flow functions that are characteristic of the OMoCC;

14. Each OMoCC shall be able to specify required and provided functions that can be implemented/used by block implementations and/or other OMoCCs in a composition of OMoCCs;
15. OMoCCs shall be able to specify parameters, that the value or type of some attributes or function's arguments may depend on;
16. It shall be possible to associate each block definition to a specific Operational MoCC and such an association will be given in a separate diagram (neither in the block definition diagram, nor in the internal block diagram);
17. Adaptation between MoCCs shall be possible with block definitions (called adapters) that have flows over the different MoCCs involved in the adaptation;
18. The language shall treat the adapters as ordinary block definitions; integration between different MoCCs shall be made possible by a library of adapters.

7.3. MoCC Libraries and operational semantics

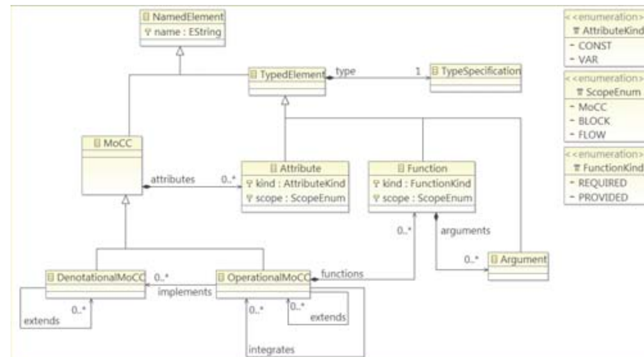


Figure 3: EMF Metamodel of Denotational and Operational MoCC Libraries

In this section we define the structure of denotational and operational MoCC libraries. Since the META II language specification is not yet able to represent such a structure, we provide a definition based on the EMF metamodeling language (see Figure 3).

Important: The actual META II definition of denotational and operational MoCC libraries is to be directly supported by the META II SysML profile, when available.

7.3.1. Denotational MoCC definition

A denotational MoCC library element is defined in terms of the characteristic attributes of the MoCC. Attributes have a *name*, a *type*, a *kind* that can be "const" or "var", and a *scope* that can be "MoCC", "Block", or "Flow". Attributes of scope "MoCC" are assigned a value when instatiating an MoCC, so that in fact the MoCC library definition is a template for a family of MoCCs. Attributes of scope "Block" are assigned a value per block associated with the MoCC. Attributes of scope "Flow" are assigned a value per flow associated with the MoCC. We assume that each block and each flow can be associated with a denotational MoCC. Attributes of kind "const" are constant, while attributes of kind "var" are dynamic and shall be specified through an appropriate white- or black-box language. Note that a denotational MoCC can "extend" another denotational MoCC, in which case it inherits all its attributes.

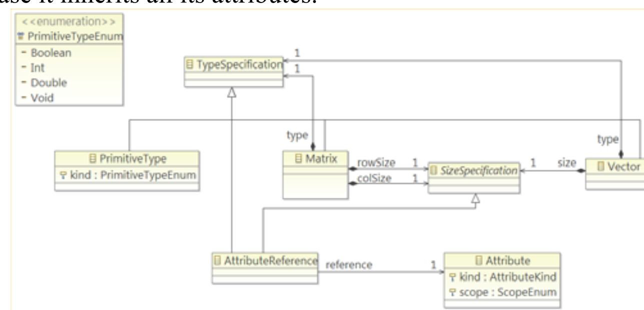


Figure 4: EMF Metamodel of Type Specification

In Figure 4, the type specification model is defined (the model is not supposed to be complete, but just as a reference example). The type specification can refer to a primitive type or a composite type, such as a vector or a matrix. Note that a type specification can also be an attribute reference and the type of an attribute can be a type specification. This allows specifying parameterizable types in MoCC specifications.

7.3.2. Operational MoCC definition

An Operational MoCC definition includes the specification of the functions that are defined in the operational semantics to carry out the computation of the block for analysis. Operational MoCCs can share part of the operational semantics specification by taking advantage of the "extends" relation. Note from Figure 3 that operational MoCCs can be declared to "implement" denotational MoCCs. Specific Operational MoCC functions are mapped to the concrete functions by the back-end model transformation (see Figure 2).

7.4. Definition of Denotational and Operational MoCCs

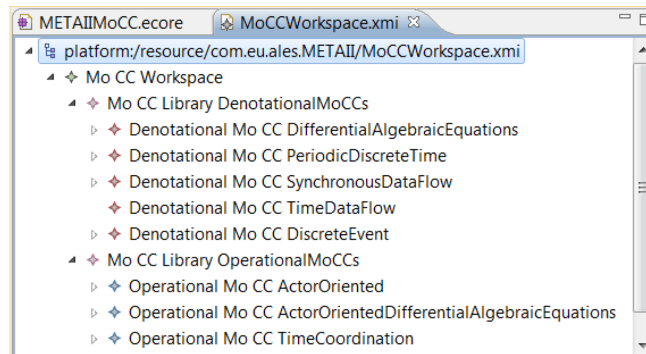


Figure 5: Denotational and Operational Libraries

In this section we use the metamodel of Figure 3 to show an example of denotational and operational MoCC libraries (see Figure 5).

7.4.1. Actor oriented OMoCC

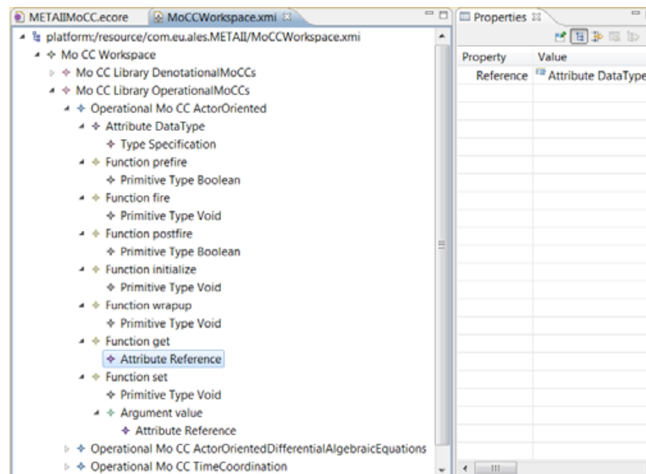


Figure 6: Actor oriented operational MoCC

In the actor oriented MoCC library element is shown. Note that this MoCC has an attribute named *DataType*. This attribute has "MoCC" scope (not visible in the picture) and is therefore fixed at each MoCC instantiation. The return type of the function get is bound to the value of the *DataType* attribute. The functions prefire, fire, postfire, initialize and wrapup are "REQUIRED" and have "BLOCK" scope, meaning that these functions must be implemented by the black-box body of each block associated with the actor oriented operational MoCC. The functions get and set are instead of "FLOW" scope, meaning

that they must be implemented by each flow directly or indirectly (via the owning block) associated with the actor oriented OMoCC.

7.4.2. Continuous Time Differential Algebraic Equation (DAE)

$$\frac{ds(t)}{dt} = \mathbf{A} \cdot s(t) + \mathbf{B} \cdot x(t - delay)$$
$$y(t) = \mathbf{C} \cdot s(t) + \mathbf{D} \cdot x(t - delay)$$

Figure 7: DAE system definition

A DAE process is mathematically defined in **Figure 7**, where *s(t)* is the state vector of size n, *x(t)* is the input vector of size m, *y(t)* is the output vector of size r and *delay* is a parameter representing the continuous time delay applied to the values at the input. The process is completely specified by the four matrices A, B, C, D, and the value of the *delay*. The matrices and the delay can be modified dynamically. These specifications are defined within the denotational library in the DifferentialAlgebraicEquations element (see **Figure 8**). Note that attribute A is of kind VAR, meaning that it may be dynamically modified according to the block's white- or black-box associated body specification. The scope of the attribute is "BLOCK", so that this attribute must be specified per block.

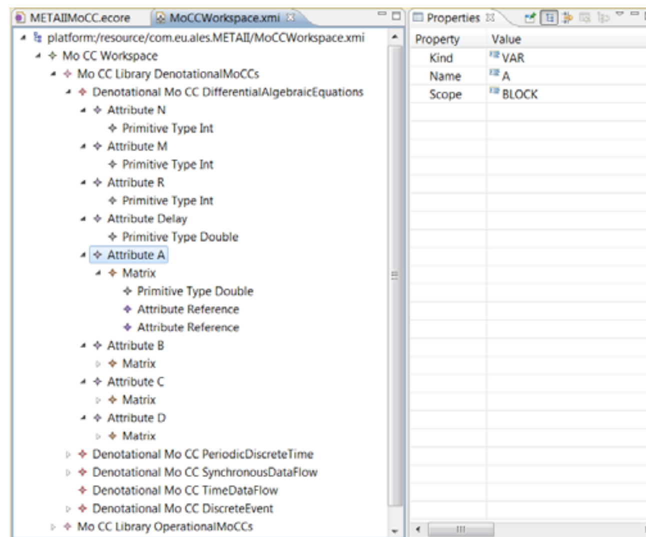


Figure 8: Denotational differential algebraic equations

To associate an operational semantics to the DAE MoCC we can use the actor oriented operational MoCC defined in Section 7.4.1. To achieve this, we can define a DAE Operational MoCC that extends the actor oriented one and implements the DAE MoCC (**Figure 9** and **Figure 10**). Moreover, the actor oriented DAE OMoCC will introduce a *TimeStep* attribute for the specification of the integration step.

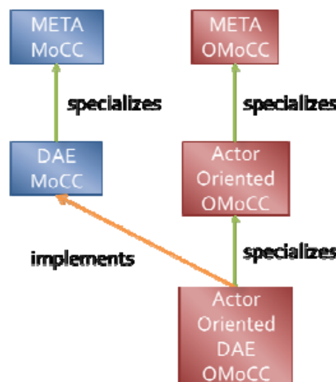


Figure 9: Relations between MoCCs and OMoCCs

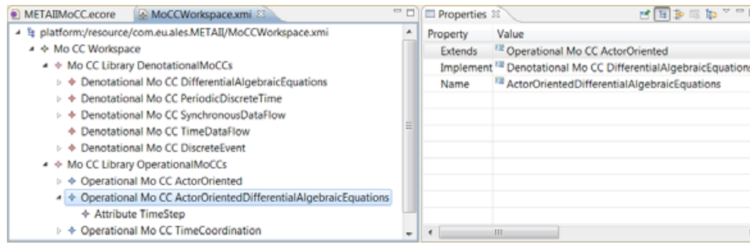


Figure 10: Operational differential algebraic equations

7.4.3. Timed Data Flow (DTF)

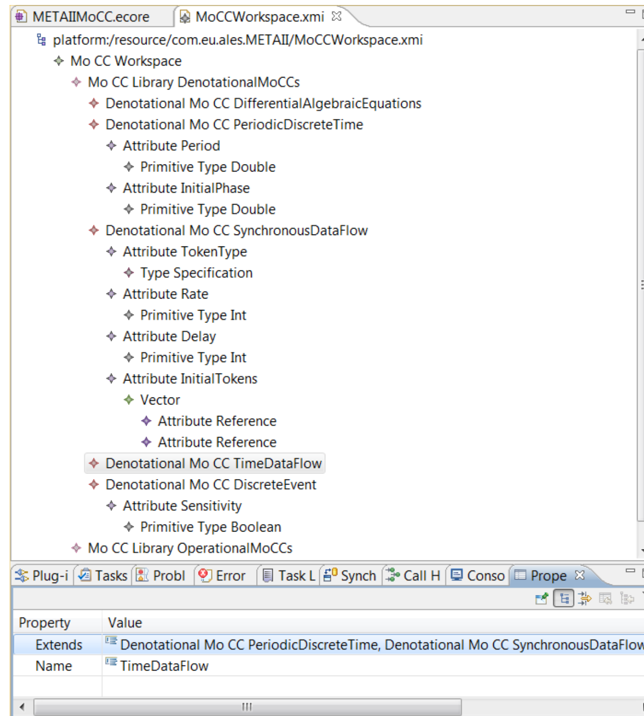


Figure 11: Timed dataflow denotational MoCC

A TDF process is an SDF process that is associated with a time step. Note that in the example library the corresponding MoCC is defined by extending the PeriodicDiscreteTime MoCC and the SynchronousDataFlow MoCC. The former is associated with the Period and InitialPhase attributes. The latter is associated with the Rate, Delay and InitialTokens. The type of the elements of the InitialToken vector is bound to the TokenType attribute, while the size of the vector is bound to the Delay attribute (not shown in the picture). While the TokenType attribute has “MoCC” scope, the Period and InitialPhase have “BLOCK” scope and the Rate, Delay and InitialTokens have “FLOW” scope (not shown in the picture). The timed dataflow MoCC can be implemented by the actor oriented operational semenatics. In the library we specify this implementation relation (**Figure 12**).

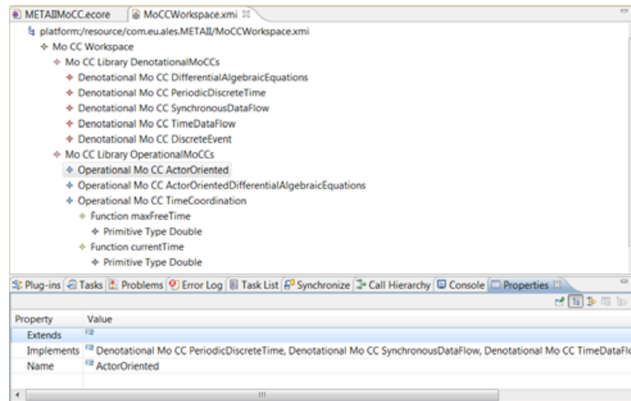


Figure 12: Implementation relation of actor oriented

7.4.4. Discrete Event (DE)

In **Figure 11** the definition of the DiscreteEvent denotational MoCC is also shown. A discrete event process is sensitive to input events on specified flows. Hence, the Sensitivity attribute is of scope “BLOCK”. As for the timed dataflow MoCC, the discrete event MoCC can also be implemented by the actor oriented operational semenatics. In the library we specify this implementation relation.

7.4.5. Time coordination operational MoCC

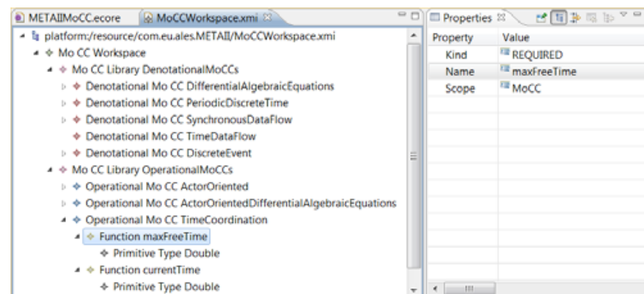


Figure 13: Time coordination operational MoCC

The operational time coordination OMoCC (**Figure 13**) is used to coordinate the execution of blocks defined over different MoCCs over time. In other words OTC allows coordinating execution of different OMoCCs. For the correct integration of different OMoCCs with OTC it is essential that each involved OMoCC is based on two phase semantics, computation phase and communication phase, and provide implementation of the “REQUIRED” function defined in the time coordination OMoCC, namely the *maxFreeTime*.
The *maxFreeTime* function semantics is that it returns the maximum interval of time from the current time instant during which the block does not change its output values. By collecting this information from all blocks, the time coordinaiton is able to compute the interval by which time can be advanced.
Implementation of the *currentTime* function is instead PROVIDED by the time coordination OMoCC.

7.5. Link to denotational semantics

As introduced in Sections 4 and 5, processes are described in terms of constraints on allowed behaviors over signals. These constraints come in addition to constraints defined by MoCCs, which the process is specified over. If the process is associated with a white-box definition, the constraints are interpreted according to the corresponding with-box language and the language is expected to be compatible with the MoCCs the block representing the process is associated with. The mathematical interpretation of the constraints the white-box language allows defining can be provided in the denotational semantics introduced in this document or in any other mathematical form that can be mapped to it. If the process is associated with a black-box definition, the constraints are interpreted according to the operational semantics of the corresponding black-box, which is expected to be compliant with the OMoCC the block representing the process is associated with. The mathematical interpretation of the constraints the black-box is imposing is provided by the mathematical interpretation of the

constraints the black-box language allows defining together with the constraints introduced by the operational semantics. Again the precise mathematical definitions can be provided in the denotational semantics introduced in this document or in any other mathematical form that can be mapped to it.

7.5.1. Extended finite state machines (EFSM)

To show how to provide the mathematical interpretation of a white-box language, we take as an example a standard mathematical interpretation of extended finite state machine languages. Any concrete EFSM language that is interpreted according to such EFSM mathematical interpretation can be formally interpreted in our framework as described below.
An extended finite state machine can be defined as a tuple $M = (R_0, A_0, R, X, Y, W, E, \Gamma)$, where R is a set of explicit states and $R_0 \subseteq R$ is the set of initial explicit states, X is a set of extended state typed variables taking values over W and $A_0 \subseteq \{f: X \rightarrow W\}$ is the set of initial assignments to the extended state, Y is a set of output typed variables taking values over W, E is a set of events, and $\Gamma \subseteq \{\gamma = (r_1, a_1, e, g, \lambda_o, r_2, a_2) \mid r_1, r_2 \in R, e \in E, g[X,W] \subseteq \{f: X \rightarrow W\}, a_1, a_2: X \rightarrow W, \lambda_o: Y \rightarrow W\}$, is the transition relation, where g is the transition guard, a_1 is the current and a_2 the next extended state assignment, and λ_o is the output assignment. Note that in a concrete language g will be expressed by some compact notation, such as $x_1 \leq x_2$, to mean in fact the set of value assignments to x_1 and x_2 such that $x_1 \leq x_2$.
To define how this mathematical model is interpreted in our framework, we do the following:

1. Associate a tag domain κ and a signal s_t with the set of explicit states;
2. Associate a signal x_i for each variable in X;
3. Associate a signal y_j for each variable in Y;
4. Associate the values $\{\perp, ⍰\}$, for absence and presence, and a signal e_k with each event in E;
5. Introduce the value set $V = R \cup W \cup \{\perp, ⍰\}$;
6. Associate a set of behaviors with the set of initial explicit states;
7. Associate a set of behaviors with the set of initial assignments to the extended state;
8. Associate a set of behaviors with each transition defined in the transition relation;

To accomplish the last three point of the program above, we use the following denotational primitives:

Guard: $G[g[S,V]](\tau) = \{\sigma \mid \exists\gamma: S \rightarrow V \text{ s.t. } (\tau,s,v) \in \sigma \Rightarrow \gamma \in g[S,V]\}$, where $g[S,V] \subseteq \{\gamma: S \rightarrow V\}$;

Event presence: $E_P[s](\tau) = \{\sigma \mid (\tau,s,⍰) \in \sigma\}$;

Event absence: $E_A[s](\tau) = \{\sigma \mid (\tau,s,\perp) \in \sigma\}$;

Initial state: $I[s,V] = \{\sigma \mid \exists\tau \text{ s.t. } (\tau',s,v') \in \sigma \Rightarrow \tau \leq \tau' \wedge \exists v \in V \text{ s.t. } (\tau,s,v) \in \sigma\}$;

Extended initial state: $I_{Ext}[\kappa,S,A] = \{\sigma \mid \exists\tau \text{ s.t. } (\tau',s,v) \in \sigma \Rightarrow \tau \leq \tau' \wedge \exists a \in A \text{ s.t. } \forall s \in S, (\tau,s,a(s)) \in \sigma\}$;

State transition: $T[\kappa,s,v_1,v_2](\tau) = \{\sigma \mid (\tau,s,v_1) \in \sigma \Rightarrow (X[\kappa](\tau),s,v_2)\}$, where $X[\kappa](\tau)$ is the next tag operator defined in Section 4.5.2;

Extended state transition: $T_{Ext}[\kappa,S,a_1,a_2](\tau) = \{\sigma \mid (\tau,s,a_1(s)) \in \sigma, \forall s \in S \Rightarrow (X[\kappa](\tau),s,a_2(s)) \in \sigma, \forall s \in S\}$, where $a_1, a_2: S \rightarrow V$;

Signal assignment: $S_A[S,a](\tau) = \{\sigma \mid \forall s \in S, (\tau,s,a(s)) \in \sigma\}$, where $x: S \rightarrow V$.

The extended finite state machine M can then be interpreted in our framework as the process $P(M) = I[s,R_0] || I_{Ext}[\kappa,X,A_0] || T_R$, where

1. $t_R[\gamma](\tau) = T[\kappa,s_t,r_1,r_2](\tau) || T_{Ext}[\kappa,X, a_1,a_2](\tau) || E_P[e](\tau) || G[g[X,W]](\tau) || S_A[Y,\lambda_o](\tau)$;
2. $T_R = \{\sigma \mid (\tau,s_t,r_1) \in \sigma \wedge (\tau,x_i,a_1(x)) \in \sigma, \forall x_i \in X \Rightarrow (\cup_{\gamma \in \Gamma} t_R[\gamma](\tau))\}$.

7.5.2. From denotation to operational

To bridge the gap between the denotational and operational semantics, it is required to mathematically represent the analysis process the operational semantics is defined for. In this section we show how this can be done in the case of operational semantics for simulation.

A *simulation* of a process $P[I,J]$ is a process that is a subset of the extension of $P[I,J]$ to the simulation tag domain indices. Let I_{sim} denote the tag domain indices associated with the simulation process. Formally, $Sim[I_{sim}](P[I,J]) \subseteq P[I,J] || \Sigma[I_{sim},J]$. Similarly, we define a tag-bounded simulation as $Sim[I_{sim},\tau](P[I,J]) \subseteq P_X[\tau](P[I,J]) || \Sigma[I_{sim},J]$, where $P_X[\tau]$ is the prefix operator defined in Section 4.4. Note that $Sim[I_{sim}]$ and $Sim[I_{sim},\tau]$ can be regarded as operators over sets of behaviors (processes).

7.5.3. Operational time coordination (OTC)

In this section we show how to use the concept of simulation process introduced above to formally link the operational API to the denotational semantics, by defining the meaning of the time coordination functions introduced in Section 7.4.5. Such functions will be defined over simulation processes.

currentTag: $C_T(Sim[I_{sim},\tau](P[I,J])) = \tau$;

maxFreeTag: Let $P[I,J]$ be a total reactive functional process with respect to J_O and J_I, then $M_{FT}(Sim[I_{sim},\tau](P[I,J])) = \max\{\tau' \mid \tau \leq \tau' \wedge \forall \sigma_1, \sigma_2 \in P[I,J], \Pi[J_I] \Box P_X[\tau](\sigma_1) = \Pi[J_I] \Box P_X[\tau](\sigma_1) \Rightarrow \Pi[J_O] \Box P_X[\tau'](\sigma_1) = \Pi[J_O] \Box P_X[\tau'](\sigma_1)\}$, where $P_X[\tau](\sigma)$ is the prefix operator defined in Section 4.4; in other words it is the maximal tag interval by which the simulation of the process proceeds independently from new inputs.

The currentTime and maxFreeTime functions of the time coordination semantics are specializations of the C_T and M_{FT} operators, respectively, applied to simulation processes $P[I,J]$ defined over time, i.e. such that $t \in I$.

8. CONCLUSIONS AND FUTURE WORKS

In this document we have presented a innovative integration language where denotational and operational MoCCs can be defined and used to give precise denotational and operational semantics to the integration of heterogeneous models. The language supports a model transformation flow towards multiple analysis back-ends, integrating implementation artifacts corresponding to models specified in different languages and complying with their specific denotational and operational semantics. We have shown that integration semantics can be easily specified in the integration language to allow coexistence of different operational semantics in the back-end analysis tools.
Future works include the definition of a complete SysML profile for the language and the definition of denotational and operational MoCC libraries of relevant models of computation and operational semantics.

9. REFERENCES

[1] E.A. Lee and D.G. Messerschmitt, “Synchronous Data Flow”, *Proceedings of the IEEE*, 75(9):1235-1245, 1987.

[2] E.A. Lee, T.M. Parks, “Dataflow Process Networks”, *Proceedings of the IEEE*, 83(5):773-799, 1995.

[3] The ProMARTE Consortium, “UML Profile for MARTE”, beta 2. Object Management Group, Number: ptc/08-06-08, 2008.

[4] F. Mallet, C. André, R. de Simone, “CCSL: specifying clock constraints with UML/Marte”, *ISSE*, 4(3):309-314, 2008.

[5] E.A. Lee and A. Sangiovanni-Vincentelli, “A Framework for Comparing Models of Computation”, *IEEE Trans. CAD*, 17(12):1217-1229, 1998.

[6] X. Liu, “Semantic Foundation of the Tagged Signal Model”, *PhD Thesis, Technical Report*, UCB/EECS-2005-31, 2005.

[7] A. Benveniste, B. Caillaud, L. P. Carloni, P. Caspi, and A. L. Sangiovanni-Vincentelli, "Composing heterogeneous reactive systems", *ACM Trans. Embed. Comput. Syst.* 7, 4, Article 43, August 2008.

[8] F. Balarin, A. Davare, M. D'Angelo, D.Densmore, T. Meyerowitz, R. Passerone, A. Pinto, A.Sangiovanni-Vincentelli, A.Simalatsar, Y. Watanabe, G. Yang, Q. Zhu*, "Platform-Based Design and Frameworks: Metropolis and Metro II"* in Model-Based Design for Embedded Systems, Boca Raton, Fl: CRC Press, Taylor and Francis Group, 2009, p. 259-289

[9] J. T. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, "*Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems*", Int. Journal in Computer Simulation, 1994

[10] L. Benvenuti, A. Ferrari, L. Mangeruca, E. Mazzi, R. Passerone, and C. Sofronis, *“A contract-based formalism for the specification of heterogeneous systems”* in FDL. IEEE, 2008, pp. 142–147.

[11] O. Ferrante, G. Codella, C. Sofronis, L. Mangeruca and A. Ferrari, *"Verify Contract-Based Designed Discrete Systems by Simulation"*, INCOSE EuSEC May 2010.

[12] SPEEDS project, http://www.speeds.eu.com/.

[13] Gerard, Sebastien, Selic, Bran, *"The UML MARTE Standardized Profile"*, Proceedings of the 17th IFAC World Congress, 2008.

[14] Herrera, F., Sánchez, P., and Villar, E. *"Heterogeneous system-level specification in SystemC"*. In Advances in Design and Specification Languages for SoCs, P. Boulet, ed. Springer

[15] J. Zhu, I. Sander, and A. Jantsch. HetMoC: Heterogeneous modeling in systemc. In *Proceedings of the Forum on Design Languages (FDL)*, Southhampton, UK, September 2010

[16] S. Edwards, E. Lee, *“The Semantics and Execution of a Synchronous Block-Diagram Language”* in Science of Computer Programming, 2003, vol. 48, no. 1, pp. 21–42.

Appendix C - EPS usecase

Tools

We have developed our Wrapper plug-in in order to enable "black-box" integration between a simulation framework and a specific modeling tool. We strive to be able to simulate blocks that come from different modeling environments inside a simulator, so that we benefit from two worlds – we have a simulator that serves our purposes well and we also can use a very advanced modeling tool, without remodeling manually the blocks we need for our simulator. We "wrap" the chosen block from the modeling tool automatically into a dll that can then be plugged-in directly into a simulation environment.
We used DESYRE as a simulation framework and Rhapsody as UML/SysML modeling tool. Our plug-in "wraps-up" the chosen block that is a part of some Rhapsody model into a stand-alone project that compiles into a dynamic-link-library. The plug-in builds the new stand-alone project automatically, from the chosen block and predefined elements, like the package that contains the definition of the used APIs.
The API is a list of functions that define the interface between simulation framework and a black box. In our case we have DESYRE interface (the relevant function calls that the wrapped block might call in the DESYRE environment); and we have the wrapper interface (the functions that the DESYRE simulation framework calls in the rhapsody dll with a block wrapped in it).

Rhapsody model

The general look of the system is in the below figure. EPS system consists of the generators and relays that supply the power to two power buses of the airplane. Our wrapper wraps the main controller block of this system for simulation in the DESYRE environment.

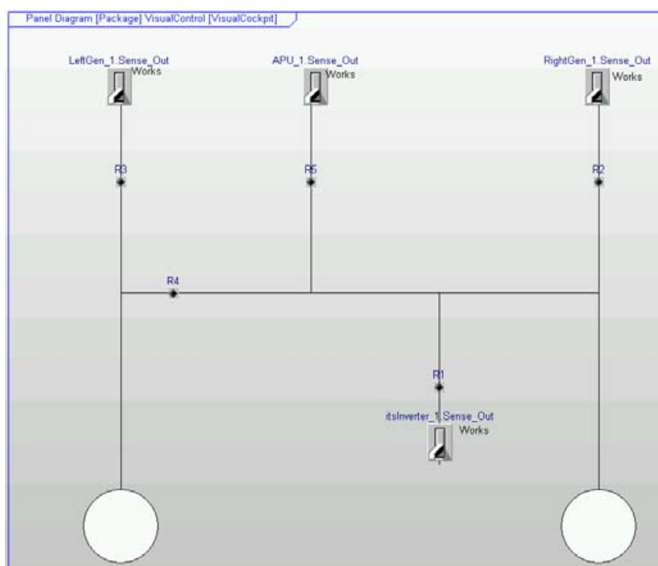


Figure 1 – EPS Use Case Rhapsody Animation Panel

Interface

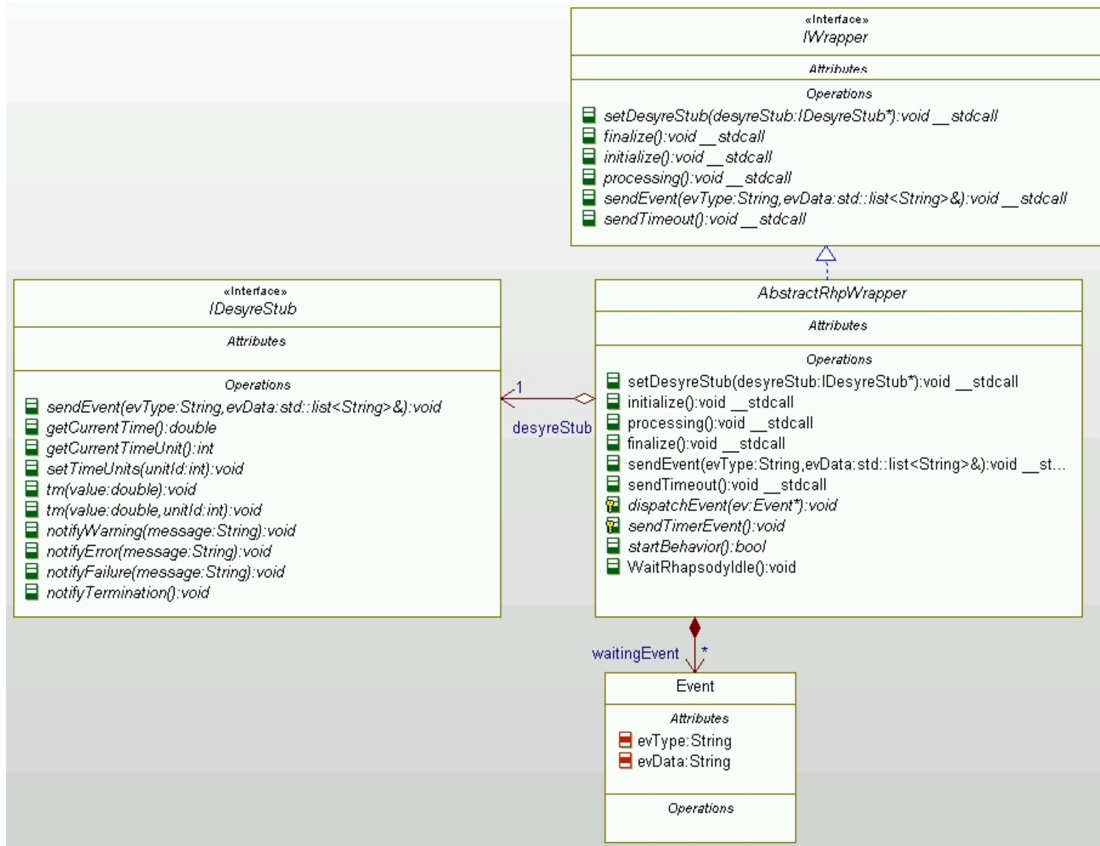


Figure 2 - Wrapper Interface

We can see the AbstractRhpWrapper Interface, which inherits from IWrapper. The functions in Interface are available for DESYRE Environment to interact with Rhapsody block. Initialize is called in the very beginning of the simulation run; sendEvent adds events into the queue that our block accumulates, and the processing() function actually "activates" the rhapsody thread and performs processing of the events in the "wrapped" block. (Rhapsody "includes" inside the dll all its internal routines that are required to "run" the behavior of the block, including creation of the separate thread in the initialization function).

In the wrapped project we build we do not use some predefined functions of the interface, our block is quite basic in this sense. Our wrapper is suitable to wrap the "Discrete Time" MOCC, meaning we have a "clock" input signal that is called every equal period of time repeatedly. We do not invoke processing of the block between the clock signals, however, accumulating events of input changes is possible using the "sendEvent" routine between the clock signals. But, these "change" events will only be processed during the next call to "processing" routine, which is associated with a clock signal.

Detailed Interface

DESYRE API for Rhapsody Statecharts

- void **setDesyreStub**(DesyreStub *desyreStub)
  - Semantics: this member function is used to pass the DesyreStub object to the component before initialization.
- void **initialize**()
  - Semantics: the member function **initialize** shall provide a context to set initial values to member variables and ports. This member function can use the services provided by the DesyreStub (see previous slide).
- void **processing**()
  - Semantics: the member function **processing** shall provide a context to define the time-domain behavior of the block. This member function can use the services provided by the DesyreStub (see previous slide).
- void **finalize**()
  - Semantics: the member function **finalize** shall provide a context to define the termination of the block. This member function can use the services provided by the DesyreStub (see previous slide).
- void **sendEvent**(...)
  - Semantics: the member function **sendEvent** is used to notify an event to the block. The event shall not be consumed by this function. The event shall be consumed by the next call to the **processing** function.
- Void **sendTimeout**()
  - Semantics: the member function **sendTimeout** is used to notify an event to the block. The event shall not be consumed by this function. The event shall be consumed by the next call to the **processing** function.

DESYRE stub for Rhapsody Statecharts

- void **sendEvent**(...)
  - Semantics: the member function **sendEvent** is used by the block to notify an event to DESYRE.
- double **getCurrentTime**()
  - Semantics: the member function **getCurrentTime**() returns the current simulation time. The units by which time is measured are obtained by calling the **getCurrentTimeUnit**() member function.
- int **getCurrentTimeUnit**()
  - Semantics: the member function **getCurrentTimeUnit**() returns the time unit of the time value returned by the **getCurrentTime**()

member function. Possible values are 1 (“fs”), 2 (“ps”), 3 (“ns”), 4 (“us”), 5 (“ms”), 6 (“s”). Default time unit is “s” (seconds).

- void **setTimeUnits**(int unitId)
    - Semantics: the member function **setTimeUnits** is used by the block to set the time units for successive calls to the **tm** member function. Allowed values are 1 (“fs”), 2 (“ps”), 3 (“ns”), 4 (“us”), 5 (“ms”), 6 (“s”). Default time unit is “s” (seconds).
- void **tm**(double value)
    - Semantics: sets a timer to expire at **getCurrentTime**()+value. When the timer expires, the DESYRE environment calls the **sendTimeout**() function on the block (see next slide).
- void **tm**(double value, int unitId)
    - Semantics: same as calling **setTimeUnits**(unitId) and **tm**(value) in sequence.
- void **notifyWarning**(std::string message)
    - Semantics: passes a warning message to the DESYRE stub.
- void **notifyError**(std::string message)
    - Semantics: passes an error message to the DESYRE stub.
- void **notifyFailure**(std::string message)
    - Semantics: passes a failure message to the DESYRE stub.
- void **notifyTermination**()
    - Semantics: the member function **notifyTermination**() notifies the termination of the behavior of the block.

Wrapper Project

The Wrapper Project consists of several packages:

- DESYREAPI package, that defines the elements required to make this project work with DESYRE Environment. This package is included as reference.
- The "Controller" Package, that contains the block we have wrapped-up
- The "Wrappers" Package, that Contains auxiliary blocks for wrapping-up the "controller" block. This package is built by a plugin according to the flow ports that the wrapped block has. It also has several functions that have their body build automatically according to the flow ports the wrapped block has.

Wrapper Package Structure

The typical structure of blocks/parts in the Wrapper Package is the following:

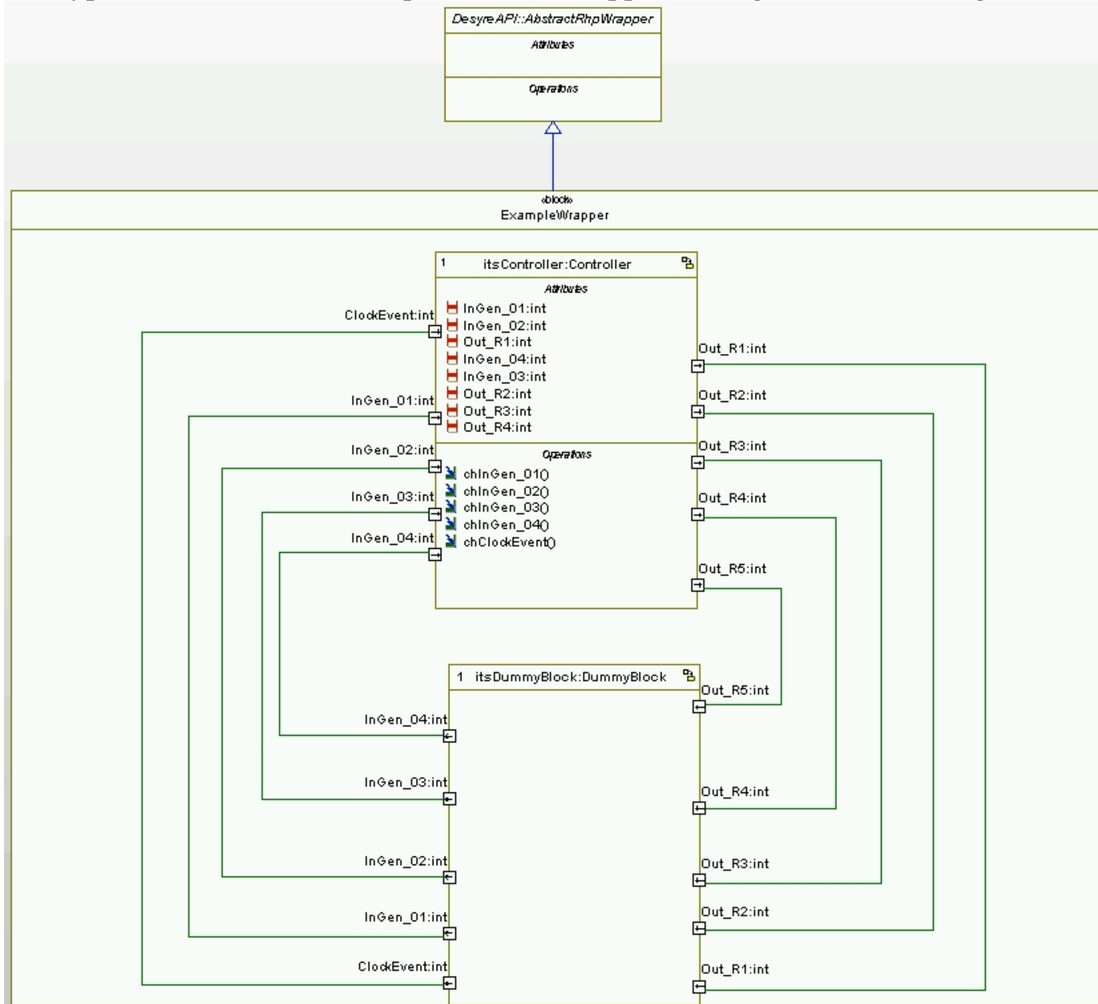


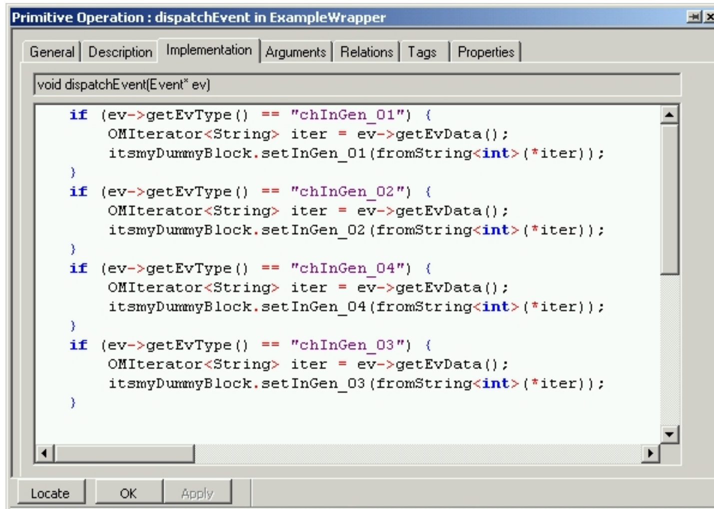
Figure 3 - Wrapper Package Structure

The ExampleWrapper Block is a realization of the AbstractWrapper Block. It is a block that "communicates" with DESYRE. It passes the events that are received from DESYRE to the controller via the "DummyBlock" block, and it passes the events from the controller, obtained via the "DummyBlock" back to DESYRE.

"DummyBlock" and Controller are parts inside the ExampleWrapper Block. Their flow ports are connected, so that the controller block is wrapped in a regular Rhapsody environment where it is connected to the outer world via input and output flow ports. The Dummyblock, in turn, has methods to pass the events from the ExampleWrapper block (which communicates with the DESYRE environment), to the Controller Block, and vice-versa.

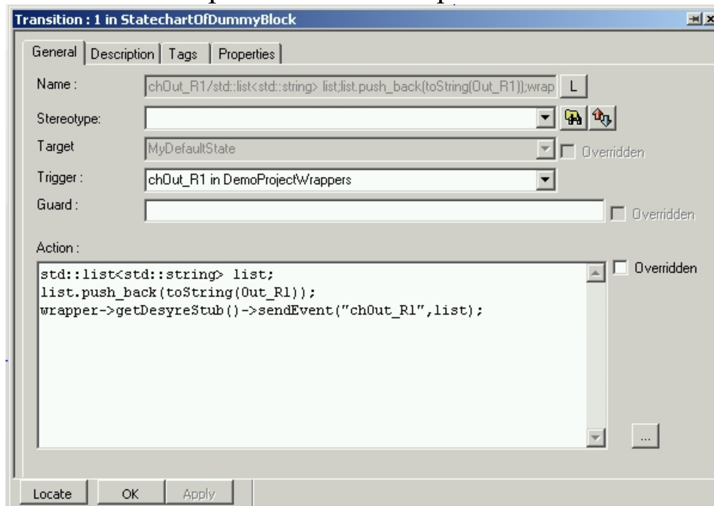
DispatchEvent operation of the ExampleWrapper Block

Once ExampleWrapper block gets a function call (an event), it calls "set" function of the DummyBlock, so that the rhapsody mechanism updates the value of the flow port and associated attribute in the "Controller" block. This is implemented in the following way:



Passing changes of output flow ports of the controller block to DESYRE

Once the controller uses "set" command to change the value of one of its output flow ports, DummyBlock reacts to this event by "passing" the command to the DESYRE Environment. An example for one of the ports is below:



"chOut" is an event automatically generated by rhapsody in the DummyBlock, when the controller calls "setout_R1" function to change the value of its outgoing flow port.

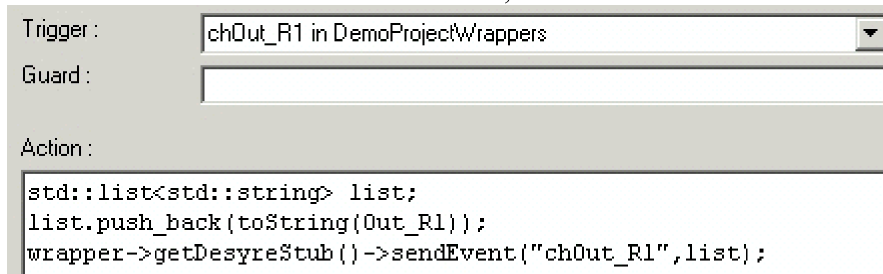
Using this methodology we obtain an environment that wraps-up the controller and enables a convenient communication with the DESYRE Environment.

Approved for public release; distribution unlimited

Wrapper Algorithm

This section describes how the Plug-in works, so this is the receipt on how to build such wrappers in the future. Note: This description shows only principal steps, without getting into details.

- In the beginning, a new project that would be a target project is created, it is defined to be a SysML project and the RhpPlugin Profile is inserted. (RhpPlugin Profile contains the definition of the stereotype that we use to mark the clock signal).
- The DESYREAPI Package is added by reference to the project. This Package has definition of the APIs that make it possible for the wrapped block to interact with the simulation framework
- Then, the Package containing the wrapped block is created, and the Package that contains ExampleWrapper is created.
- RhpDESYREPlugin Project is loaded (RhpDESYREPlugin Project contains some of the basic building blocks for the target project).
- The global function "create" and ExampleWrapper Block are copied into a target project, under DemoProjectWrappers Package; Compilation "Configurations" are copied into the target project and the correct Configuration is set as a default one.
- RhpDESYREPlugin Project is closed
- DummyBlock is created, ExampleWrapper block is made a generalization of AbstractRhpWrapper; The statechart of DummyBlock is created.
- For each flow port in the block we are wrapping:
    - We clone a flow port (with a reverse direction) and associated attribute into a dummy block.
    - We add a transition to the statechart of the DummyBlock, that reacts to the change in the DummyBlock "in" port (the controller "out" port) and notifies the DESYRE stub of the event, like this:



    - If this is an "in" flow port in the controller block, we add it to a list of "in" ports for later use
    - We locate the flow port / attrib that is marked with a stereotype "TimerSignal" that serves as the main clock signal for this block
- We copy the events from the package where original block to wrap is, and block to wrap itself to the target package in a new project.

Approved for public release; distribution unlimited

- Parts of Controller and DummyBlock are created inside the ExampleWrapper Block.
- Links between Flow Ports of DummyBlock and Controller are created
- DispatchEvent of the ExampleWrapper Block is filled:
    - For each "in" port of the controller, Add a portion of code, like this:

```
if (ev->getEvType() == "chInGen_01") {
    OMIterator<String> iter = ev->getEvData();
    itsmyDummyBlock.setInGen_01(fromString<int>(*iter));
}
```

- Set some specific properties via the code to achieve correct compilation

Approved for public release; distribution unlimited