A Model-Driven Approach to Design Engineered Physical Systems

Sagar Sen
Supervisor : Prof. Hans Vangheluwe

School of Computer Science
McGill University, Montréal, Canada

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements of the degree of
Master of Science in Computer Science

Abstract

The constant growth in complexity of real-world engineered systems has led to the concurrent development of software tools to store and reuse the knowledge for simplifying the creation of such systems. Software models that encode structure and behaviour of components in the system and the system itself are currently being developed based on the techniques prescribed by Model Driven Engineering (MDE). We use concepts in MDE to develop *modelling formalisms* to create models of a target Engineered Physical System (EPS) at different levels of abstraction. Each level of abstraction presents a certain view of the EPS to a domain expert in the development team. For instance, a high-level view is suitable for a person in a managerial role. An engineer who deals with the same system at a lower level of abstraction develops a model using idealized physical components. A physicist's concern is the physical meaningfulness of the model. The physicist's model verifies if the model prescribed by the manager via the engineer adheres to the laws of conservation of energy and momentum. Finally, a mathematician or a computer scientist obtains a solution to the dynamical system (which is usually a set of Differential Algebraic Equations provided by the physicist) by solving it analytically or numerically.

We also present transformations to automatically transform the model of an EPS in a high level of abstraction to one at the behavioral level. Therefore we have two phases in the MDE based framework. The first phase is the development of a *modelling language* at each level of abstraction. We specify modelling languages, to constrain modellers, using visually expressed *meta-model*s and textually expressed constraints. The high-level description of an EPS is specified using the High-level Physical System Model (HLPSM) modelling language. The ideal physical components in the EPS and their interconnection with each other is modelled using the Idealized Physical Model (IPM) modelling language. We specify the Hybrid Bond Graph (HBG) modelling language for developing a physical domain-independent modelling language for encoding the energy flow structure in the EPS. We then use the Modelica physical modelling language to represent the set of equations obtained from a HBG model. Finally, a model in the Trajectory modelling language represents the behaviour of the high-level EPS model. The *abstract syntax* of models in all the visual modelling languages, namely, HLPSM, IPM, and HBG are represented using *hierarchical labelled graphs*.

In the second phase we specify *model transformation*s to automatically transform models from high to low abstraction levels. The transformations are performed via *graph rewriting* on the *abstract syntax graph* representation of models. Graph Grammar (GG) rules with pre-actions, post-actions, and pre-conditions are used to define the transformations between models of visual languages. The final transformations involve code generation and simulation/execution of low-level models to obtain the behavior of the model. The model transformation MT_HLPSM_2_IPM maps a model conforming to HLPSM to one that conforms to IPM. The transformation MT_IPM_2_HBG maps IPM models to those that conform to HBG. An internal transformation MT_HABG_2_HCBG assigns causality to an *acausal* HBG (HABG) model which results in a *causal* HBG (HCBG).

This transformation gives us some insight into the physical meaningfulness of the initial model. Modelica code is generated from the HCBG via the transformation MT_HABG_2_HCBG. The Modelica code is an object-oriented representation of a system of Differential Algebraic Equations (DAE). The DAEs are solved via the transformation MT_Modelica_2_Trajectory which is nothing but the simulation of the model. The result is a set of plots in the Trajectory language that describes the behavior of the system.

Many times the complexity of a target model is so high that it is difficult for a team of modellers to come up with an optimal model for a specific task. Under such circumstances the need for automatically modifying an existing basic model to obtain optimal models becomes useful. We extend this study to explore the *model design space* created by the meta-models for each modelling language. A point in a model design space is a model represented as the abstract syntax graph. Therefore, a model design space is a space of graphs. A modification to an *embryonic* model in the graph form is done via the application of *mutation operators* to the model. If a mutation operator results in a graph that does not conform to the meta-model or does not satisfy its constraints then the model is outside the model design space. We present a set of mutation operators MT_Heuristics_HBG described as GG rules to transform HBG models for exploration of their design space. A genetic algorithm is executed to construct optimal plans that comprise of these mutation operators. We present our results for the evolution of the model of simple hoisting device.

Our MDE based framework is an introduction to a new approach for automating a large part of the design and development of EPSs. This framework is based on visual languages and graph rewriting making it viable to quick model and transformation specification due the graphical nature of the models as opposed to textual programs. This framework although currently describes a complete methodology to model electro-mechanical EPSs, can be extended to other physical domains such as hydraulics, chemical and thermodynamics.

Acknowledgements

My supervisor, Hans Vangheluwe, introduced me to meta-modelling and domain specific modelling which opened a whole new world of possibilites and ideas. Exploring these ideas will last us atleast a lifetime. I am very grateful to him for this. I am also grateful to my lab members Marc Provost, Denis Dube, Hongyang Song and Ximeng Sun for their selfless help with regard to providing tool support and ideas for my thesis.

I am deeply indebted to the Commonwealth Fellowship programme, hosted by the Govt. of Canada, which generously supported me for a period of 2 years. I would like to specifically thank Diane Cyr who selflessly helped me throughout my masters education.

My job is incomplete if I do not thank Brice Kolko, my coach in the McGill university rowing team who helped me keep my mind and body in top shape.

Finally, I would like to thank my parents for their support in the past, present, and the future in all my endeavors.

Contents

List of Figures

List of Tables

Introduction

Science and engineering have evolved hand in hand, one complimenting the other's development. Everyday there is new knowledge and new technology. With the growth in technology there is also a steady growth in complexity of engineered systems in the real-world. A desktop computer and the Internet are examples of a very complex real-world systems. The need to reuse and store the art and the science of developing complex devices of today has led to the development of software that encode engineering principles and scientific laws.

Modelling languages are used to represent software models or simply models of real-world systems. Practical examples of modelling tools include high-level programming langauges for software and visual languages to model domain-specific systems. Modelling Engineered Physical System (EPS) especially for its application to the construction of embedded systems has led to development of several modelling tools. Notably, MATLAB Simulink [Mat] and Modelica [Mat97] are widely used for modelling and simulation of plant-controller systems. Simulink now is packaged with several libraries with visual modelling elements to build very sophisticated and accurate models. Simulink libraries to model the *signal* domain for electrical circuits and controllers have been under extensive use in both industry and academia. Modelling libraries such as SimMechanics [GW03] and SimHydraulics [sim] are new additions to Simulink's repertoire for modelling mechanical and hydraulic systems. Modelica is an object-oriented language for modelling physical systems. A Modelica model is represented as a set of state variables and equations (or laws). This allows the modeller to encode the behavior as physical law equations and constraints instead of functions or operations in traditional object-oriented languages. Therefore, the modeller does not have to specify the *causality* of operation. The causality is automatically assigned via computer algebra. The *non-causal* nature of Modelica is the key feature that distinguishes it from Simulink. Modelica is rapidly gaining popularity for its application to modelling EPSs and embedded systems.

The existing modelling languages seem to provide domain-specific libraries are written in a *Turing complete* high-level programming language and the modeller is unconstrained. This makes it easy for an *experienced* modeller to encode knowledge in his/her domains in the same language, such as Matlab m-scripts. Same is the case with Modelica. The programmer will have to minimize the number of errors he commits for writing a tool for a specific domain. He/she can achieve this by writing several test cases and by using an implicit style of programming to write modelling elements. This implicit style or *pattern* is nothing but a mix of the modeller's experience with the domain and his prior experience with writing equivalent libraries. This approach leaves the domain knowledge in subjective form either in the modeller's mind, in text, or as a high-level program that knows nothing about the domain. The only responsibility of the program is to execute logically. These are the problems plaguing most existing tools. A domain expert with such a tool in his hand will have the freedom to build anything without any conceptual feedback (simple errors are always reported) about the meaningfulness of his model. Therefore we ask, can the modeller be explicity constrained given the structural and behavioral knowledge we have about the target domain?

To answer this question we present Model Driven Engineering (MDE) based techniques [Ken02] to develop domain-specific modelling languages for EPSs. These languages constrain a modeller to his/her domain allowing the creation of mostly valid models. A *meta-model* with textually

expressed *constraints* is first constructed to specify the set of valid models. A *domain-specific modelling language* is then synthesized from the meta-model plus constraints specification. We use meta-modelling to synthesize *visual modelling languages* [CLOP02] that allow the creation of models at a visual abstraction level. A meta-model is analogous to a *grammar* used to specify the syntax for textual programming languages. A model specified in one visual language is transformed to a model specified in another visual language via *graph rewriting* [HER99]. This process in general can also be called *model transformation* Using this framework of meta-models, modelling languages, and graph rewriting we separate the process of engineering a physical system into several steps.

The automatic synthesis of a modelling language from a meta-model specification and the graphical nature of model transformations presents a key difference with respect to existing modelling tools such as Simulink and Modelica. Further, the constrained nature of a modelling language makes it very easy for a domain expert to master the language and minimize development errors.

We finally present a set of Graph Grammar (GG) rules to modify physical system models. These rules are designed to be automatically executed by a genetic algorithm. A plan that comprises of a set of GG rules is executed to evolve a model to meet specific design criteria or a fitness function. This is an attempt to investigate answers the long-standing question " Can a computer replace or augment human invention ?". The constrained space of models specified by a meta-model is the search space for exploration by various artificial intelligence planning techniques. Our preliminary attempts to search the space of EPS models is presented in this thesis.

In Chapter 1 we present the development of modelling languages used to develop EPS models at different abstraction levels. This is followed by Chapter 2 that presents the model transformations to transform models between visual languages. In Chapter 3 we present a set of GG rules for exploring the design space of EPS models. In the same chapter we present a genetic algorithm and preliminary results for design space exploration for a simple example. We conclude in Chapter 4.

1

Modelling Languages for Engineered Physical Systems

1.1 Introduction

Today, *modelling* is the first step taken toward realizing a complex real-world system. This said, we ask, how do we express a model ? We need a *language* or a *modelling language* to express a model. A modelling language specifies the *syntax* that all its model must adhere to. The syntax of a modelling language is specified by a *meta-model* (for visual languages) or *grammar* in *Backus-Naur Form* (for textual languages) [BJV60]. An immediate next question is, how do we express the syntax of the meta-model itself? Obviously, we need to express this too in another modelling language. Such a language is specified using a *meta-meta-model*. But, does this hierarchy of languages that express models of other languages not stop? The answer is, yes it does. A modelling language specified by a meta-meta-model is usually expressive enough to express itself. As, we can see there are three levels of models: model, meta-model, and meta-meta-model. A model is an *instance of* a modelling language specified by a meta-model. A meta-model is an instance of a modelling language specified by a meta-meta model. The meta-model for a meta-meta-model can be expressed using the modelling language specified by the meta-meta-model. This is called *bootstrapping*.

Before we start discussing the modelling languages used to model an Engineered Physical System (EPS), it is important that we understand the role of a meta-meta-model in the MDE framework. The industry standard for the meta-meta-model is Meta-object Facility (MOF) [OMG03]. The architecture for MOF and Essential Meta-object Facility (EMOF) are shown in Figure 1.1. The EMOF is used to describe meta-models and hence is our focus. Other parts of the MOF are implementation details that are not of conceptual concern and are not used to describe our meta-models.

The EMOF suggests that a meta-model comprises of *classes* and these classes are associated with *properties*. A property can be an *attribute* or a *reference*. An attribute is of *primitive type* and can either be a Float, Integer, Boolean, or a String. Although, the official EMOF does not contain a definition for Float, we have introduced it in our meta-models to express real variables. A reference is a relationship end between two classes. The reference in a class to an object of another class is used to express complex types in a class. A reference is associated with a *multiplicity* that constrains the number of objects that could be referred to.

An *inheritance* relationship between classes can exist where the properties of a super-class are inherited by its sub-classes. An inheritance between the super class and a sub class is represented using a arrow with a white triangular head at the super-class end of the inheritance relationship. For instance, the class State inherits properties of class AbstractState as shown in

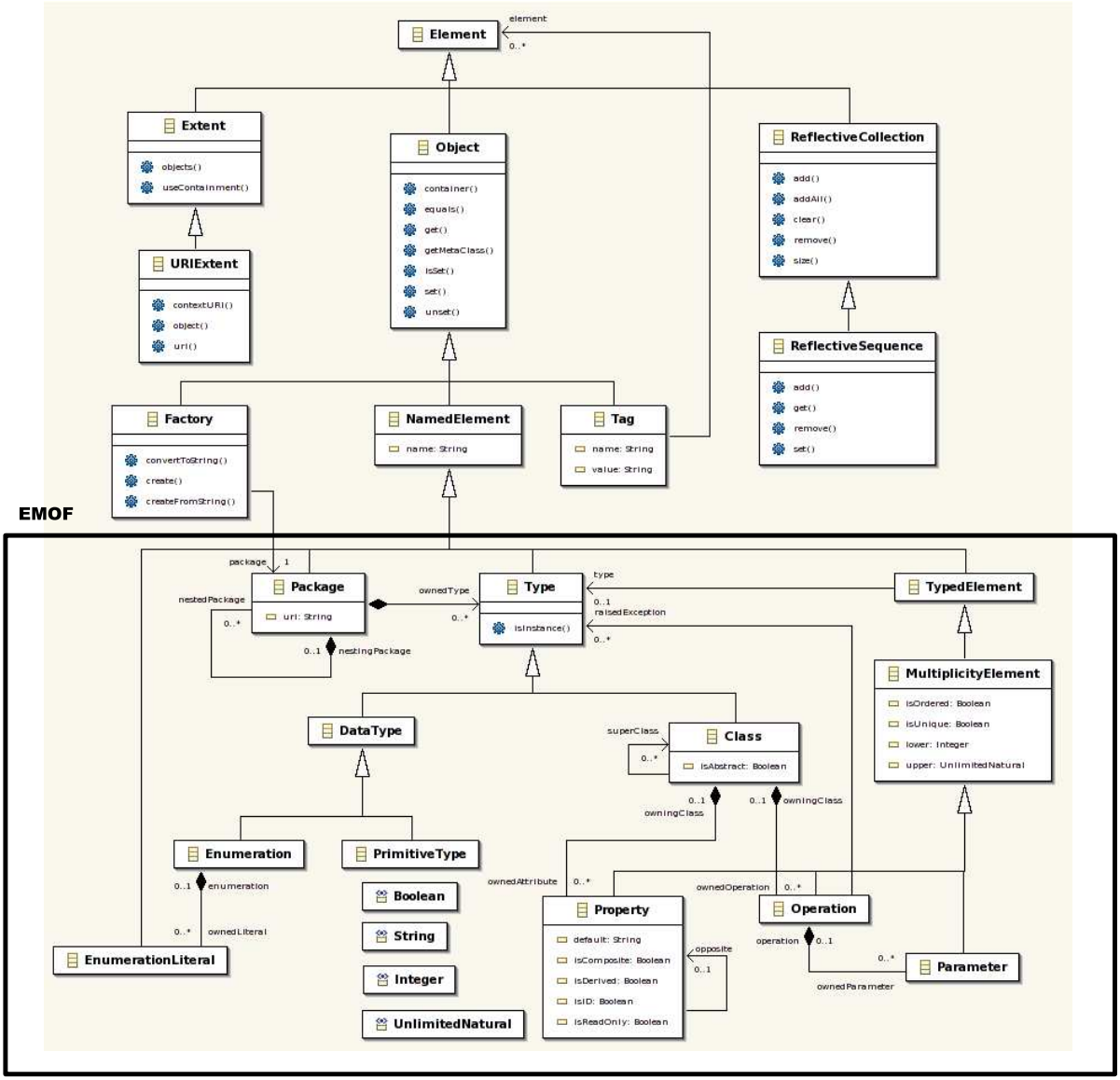


Figure 1.1: The Meta-object Facility

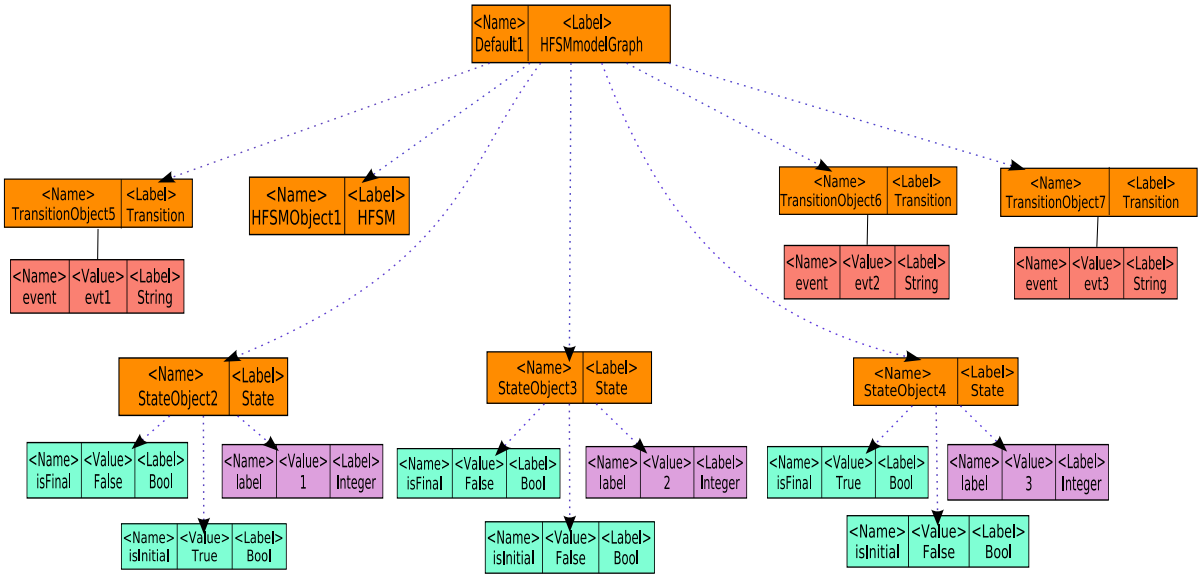


Figure 1.2: Attributes View of Abstract Syntax Graph of HFSM Model

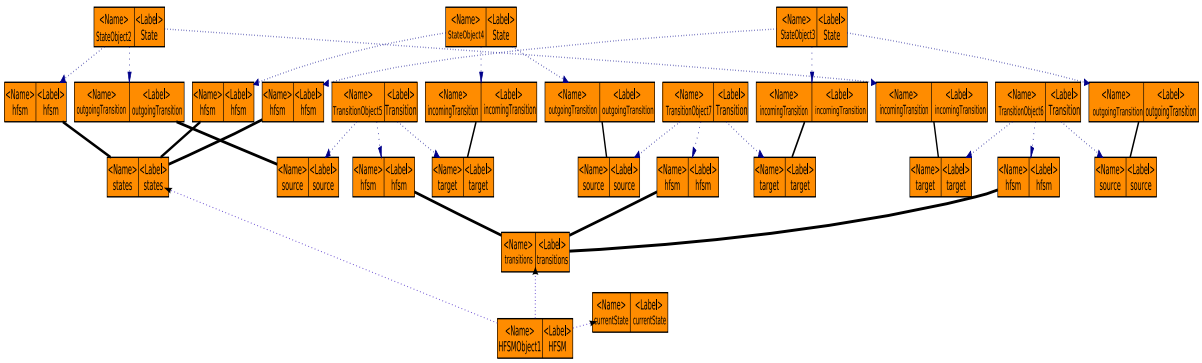


Figure 1.3: References View of Abstract Syntax Graph of HFSM Model

the Figure 1.4. A containment relationship or a *composition* is a special kind of relationship which enforces the constraint that the container class strictly contains a certain number of objects of an other class. An attempt to create a model where a contained class attempts to contain its container will result in an error when the meta-model is processed. A containment relationship between a container class and the contained class is expressed as an arrow with a black rhombus head. For instance, the class HFSM is a container for the contained classes AbstractState and Transition as shown in the Figure 1.4.

The meta-model, which is a model of EMOF, is not expressive enough to incorporate arbitrary constraints that are invariants, pre-conditions, and post-conditions during model generation or transformation. These constraints that restrict the properties of a model are expressed using a constraint language such as Object Constraint Language (OCL) [OMG] or a high-level programming language such as Python. We express constraints as Python code.

We present the meta-models of three visual languages, used to model the same Engineered Physical System (EPS), as instances of EMOF: High-level Physical System Model (HLPSM) in Section 1.2, Idealized Physical Model (IPM) in Section 1.3, and Hybrid Bond Graph (HBG) in

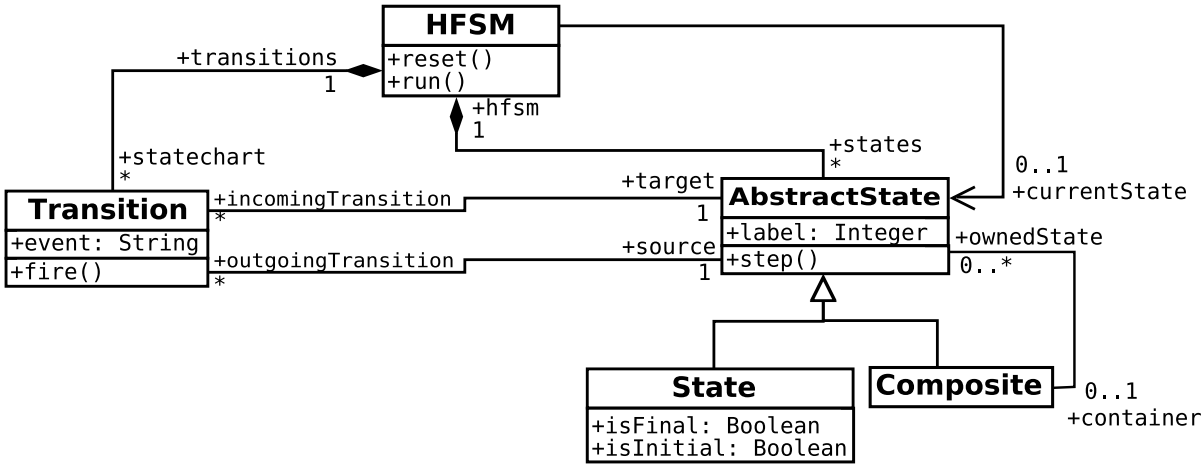


Figure 1.4: EMOF Meta-model for Hierarchical Finite State Machine

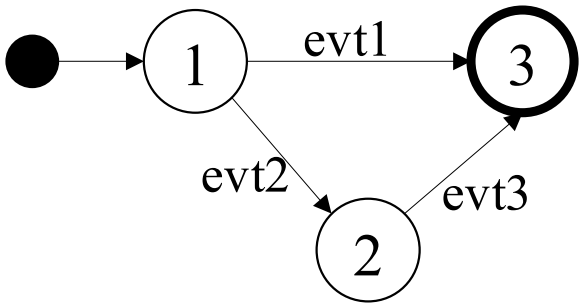


Figure 1.5: An Example HFSM Model in Concrete Visual Syntax

Section 1.4. Visual modelling languages are synthesized from these meta-models. A model specified using a visual language is expressed in two ways. The model has a *concrete visual syntax* and an *abstract syntax graph* representation. The concrete visual syntax of a modelling language is a *domain-specific* visual notation used to describe the entities in the model. The abstract syntax graph is a *hierarchical labelled graph* as implemented in the Himesis sub-graph matching kernel [Pro05].

An Himesis graph comprises of nodes and connections. There are two types of nodes, either just a *node* or a *primitive node*. Each node is associated with a *label* and a unique *name*. Primitive nodes have an extra property called the *value*. The *value* in a primitive node stores the value of a primitive data type such as String, Float, Boolean, and Integer. Everything in a valid graph is a graph in its own right as a single node is a graph too. There are two kinds of connections. A *parent-child* edge between nodes is used for representing hierarchy. A *connection* edge between nodes is used to represent a link between graph nodes. An object that is an instance of a class can be constructed using a graph node representing an empty object (referred to as object node) and its attributes and references created as nodes that are linked to the object node using a parent-child edge where the object node is the parent and the attributes and references are its children. A relationship between related objects is created by a connection edge between the associated references in two classes.

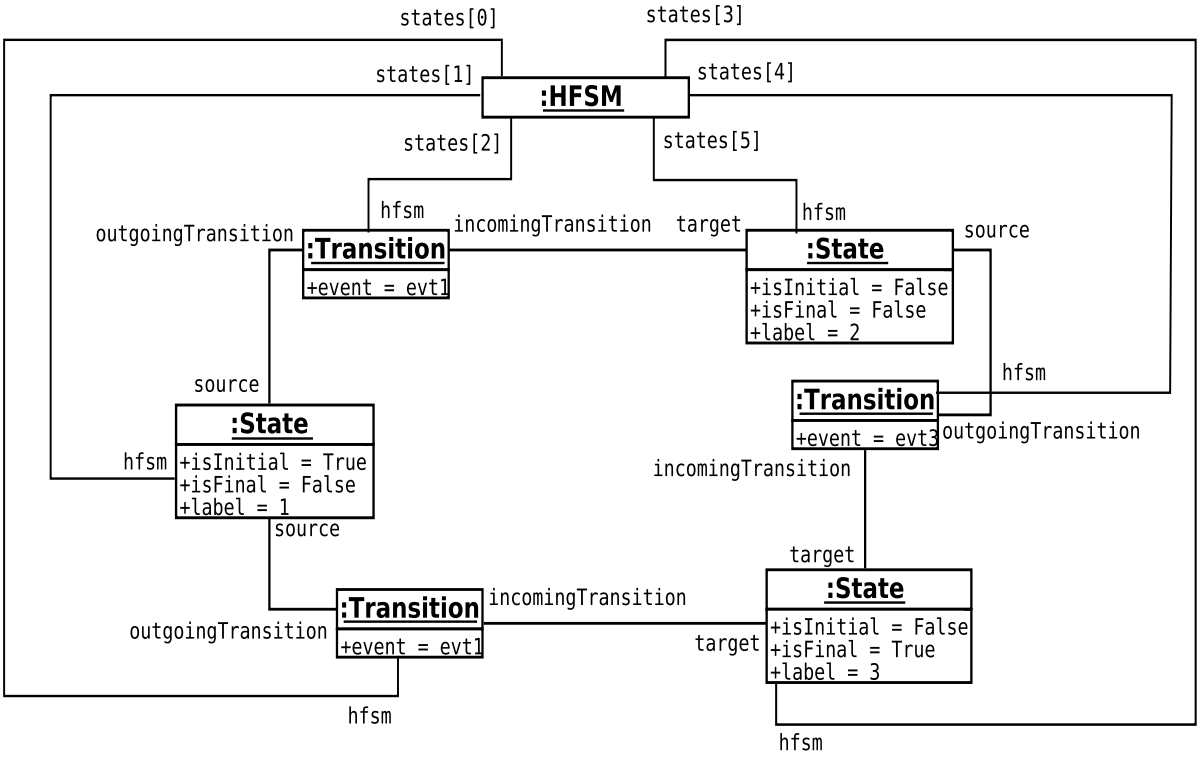


Figure 1.6: Object Diagram for the HFSM Model

We use the Hierarchical Finite State Machine (HFSM) modelling language as a toy example to illustrate the types of model representation. The meta-model for HFSM is shown in Figure 1.4. A model HFSM which is an instance of the meta-model is shown in Figure 1.5. In the MDE and software engineering community it is common practice to represent the model in computer memory as an object diagram as shown in Figure 1.6. The abstract syntax graph for the model is split into two figures. In Figure 1.2 we see the attributes view of the abstract syntax graph of the example model. In Figure 1.3 we see the references view of the abstract syntax graph of the example model. The attributes are connected to the parent object using a parent-child edge which a directed dotted blue line. For instance, in Figure 1.2, isFinal is an attribute of `StateObject2` which is an object of the State class. A relationship is a black connection edge between two graph nodes. For instance in Figure 1.3 we see that outgoingTransition reference of `StateObject2` is connected to source reference of `TransitionObject5`. The label, name and value of each attribute node or primitive node is specified in the abstract syntax graph. Similarly, the label and name for a node is also specified in the visual representation.

The representation of an EPS model, as code that describes the Differential Algebraic Equations for the system, is expressed in a Modelica program. We briefly discuss the grammar for Modelica in Section 1.5, which is in essence the meta-model for Modelica. The behaviour of a model is expressed as an instance of the Trajectory language. The Trajectory language is a visual language consisting of plots that show the time-dependent behaviour of the state variables in the model. A meta-model for Trajectory is described in Section 1.6.

The running example of a Hoisting Device [Bro] is used to illustrate the models of the same system in all the modelling languages.

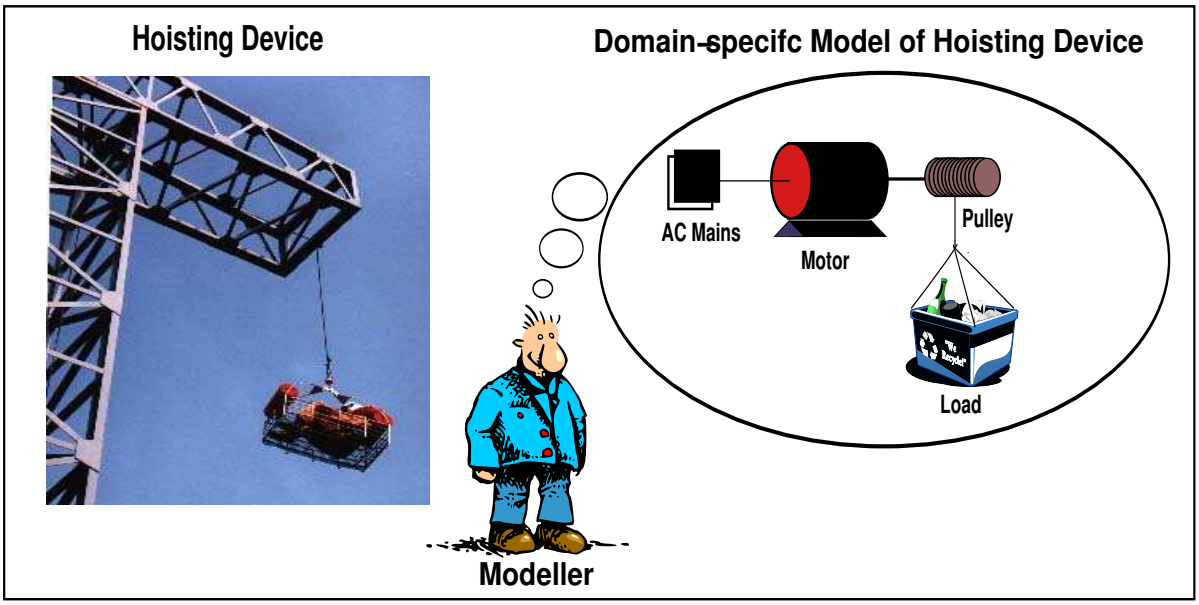


Figure 1.7: The First Model of a Hoisting Device

1.2 High-Level Physical Model Modelling Language

A modeller begins the modelling process by visualizing the system at a high-level of abstraction where he/she sketches the top-level components of the system. This form of modelling is typically done by someone in a managerial role. In Figure 1.7 we show an actual hoisting device and a possible high-level physical model as imagined by a modeller. The high-level model comprises of visual syntax to represent the electrical mains, the motor, the pulley and the load. The high-level visual notation implies that the high-level model is domain-specific.

Our goal is to provide the modeller with domain-specific modelling tools to realize his imagination in the form of a model with syntax and semantics. A meta-model is first developed to specify the properties of the objects we see in a high-level view of the physical system. An extension to the meta-model are special constraints one may observe in the system. For example a power outlet can be only connected to a compatible power inlet, a mechanical device cannot be connected to an electrical device without an intermediate transducer. An EMOF based meta-model for the HLPSM modelling language is shown in Figure 1.9. Note that the HLPSM meta-model is given only to consist of components of a hoisting device. It can be extended to other domains simply by changing the classes in the diagram according in the system.

The top-level container class in the meta-model is the HLPSM class. There always exists one `HLPSM` object in any model as indicated by the multiplicity in the meta-model. A `HLPSM` object can contain 0 or 1 `Plant` objects and is associated with a name. A `Plant` object consists of 0 to any number (represented by a * in the EMOF model) of `PlantEntity` objects. The classes that inherit from PlantEntity are the domain-specific components in the system. According to the meta-model the Mains class is connected via a Wire to Motor. The Motor class is connected to a CableDrum class via a Shaft class. The CableDrum class subtends a Load class using a Rope class. It can be observed that due to the multiplicities the modeller is highly restricted and can only build structurally valid hoisting devices.

A visual modelling environment is automatically synthesized from the meta-model and con-

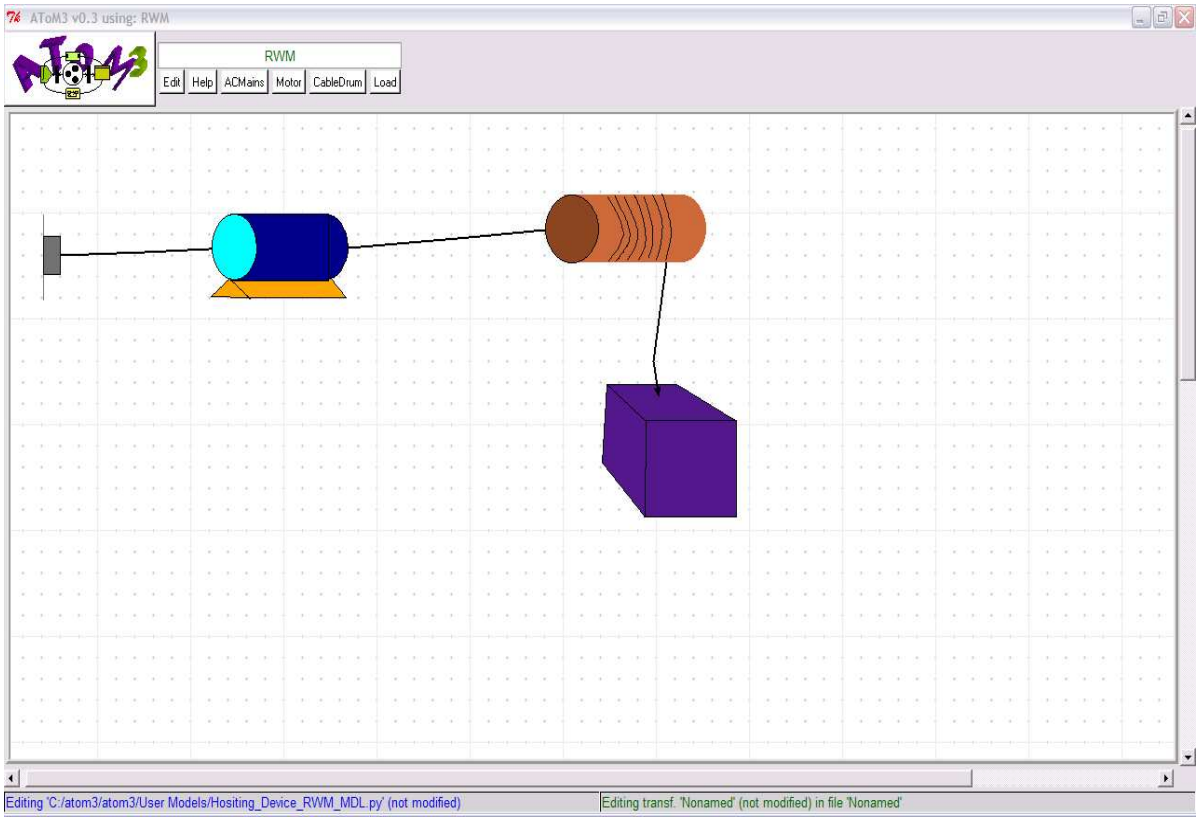


Figure 1.8: Screenshot of HLPSM Visual Modelling Environment

straint specification of the modelling language. We use the tool AToM^3 [VdL04] to achieve this transformation where a visual editor is synthesized from the meta-model. In Figure 1.8 we see the visual editor for HLPSM models. The visual editor performs syntax directed checking implying that it checks if the modeller, as he/she is constructing the model, is violating a local constraint such as going outside the multiplicity bounds.

The underlying Abstract Syntax Graph of a HLPSM model is expressed as a hierarchical labelled graph. Any transformation to the model is performed on the abstract syntax graph. The meaning or the semantics of the HLPSM language is given by transformation to the Idealized Physical Model modelling language as discussed in Chapter 2, Section **??**.

In Figure 1.10(a) we show an example model of the HLPSM for building electro-mechanical hoisting devices.

The HLPSM model is given to an engineer who constructs an Idealized Physical Model (IPM) from it. The IPM modelling language is described next.

1.3 Idealized Physical Model Modelling Language

Following the development of a high-level physical system model, the modeller associates meaning with the components of the domain-specific physical model by dissecting it into components with *idealized physical behavior*. A domain expert for creating such a model would be an engineer or a group of engineers (mechanical, electrical, chemical). An Idealized Physical Model (IPM) is constructed such that the model consists of only ideal elements with well defined

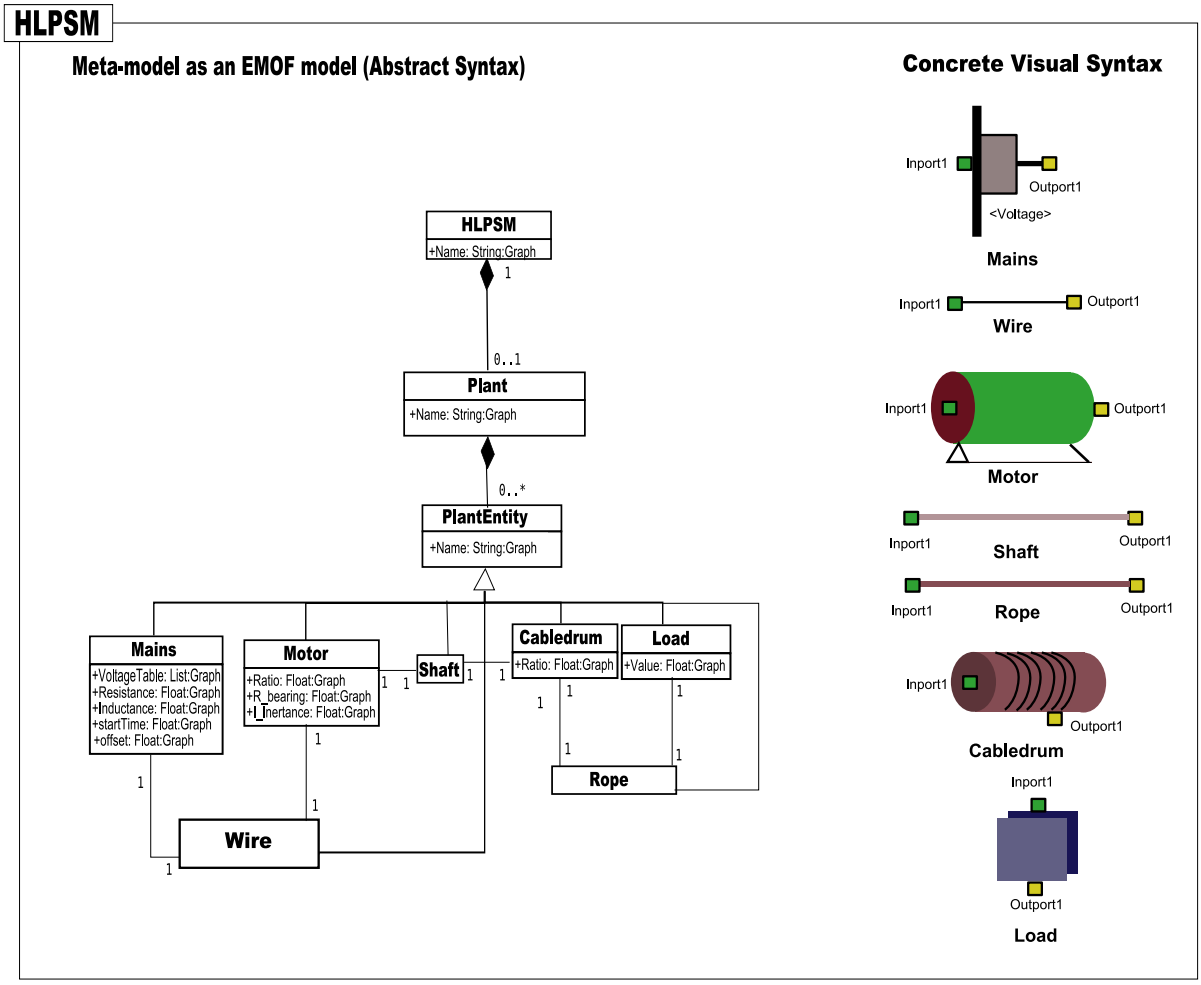


Figure 1.9: EMOF Meta-model and Concrete Visual Syntax for HLPSM to Model Hoisting Devices

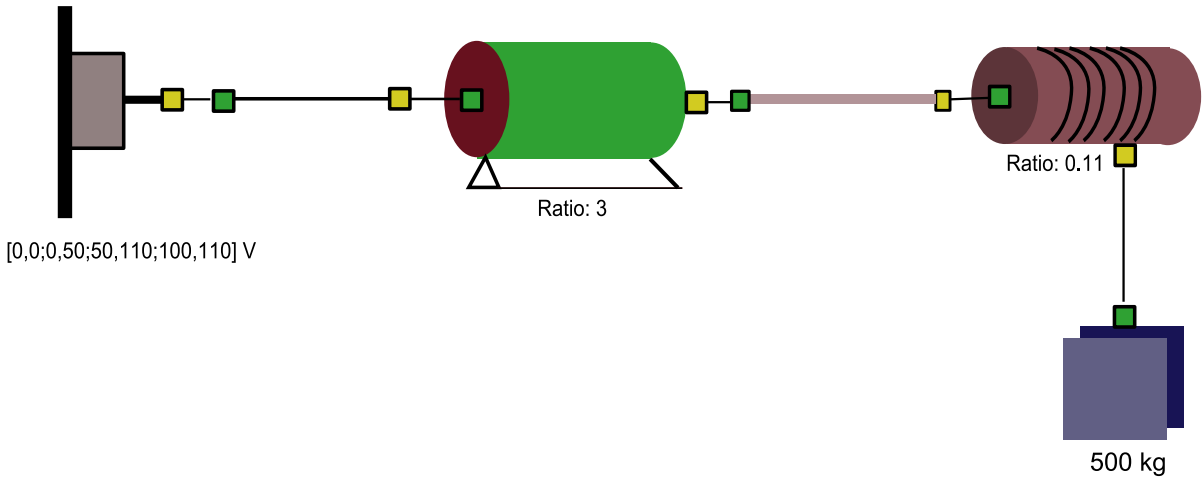


Figure 1.10: Hoisting device model in the HLPSM formalism

physics. In our case we restrict our systems to *lumped-parameter* models where aggregate phenomena is described using classical physics [Mac03]. An EMOF based meta-model is specified for the Idealized Physical Model modelling language. The IPM modelling language in general constitutes entities from different engineering domains: electrical, mechanical, hydraulic, chemical, and thermodynamic. We present the IPM meta-model to model electro-mechanical systems comprising of electrical, translational mechanical, and rotational mechanical components. An extension to the hydraulic, chemical, and thermodynamic domains is straightforward if a similar meta-modelling pattern is followed.

1.3.1 The Electrical Domain

The electrical domain in the idealized physical modelling language contains two-pin electrical components such as resistors, capacitors, inductors, voltage, and current sources. Every electrical component consists of a positive pin and a negative pin. A positive pin can be connected using a wire to a negative pin only and vice versa. An electrical circuit is a combination of electrical components connected by wires. Voltage and current are conjugate variables that carry energy in an electrical circuit.
There are many different sources (voltage or current) of electrical energy. They are distinguished based on input wave forms and functions. For instance, we can have a constant energy source, a table with a set of energy values that are interpolated over time, or a sinusoidal waveform commonly observed in alternating supplies. Resistors are energy dissipators that convert part of the input electrical energy to heat energy. Energy is lost only from a resistor since all other elements are ideal. Energy storage components are capacitors and inductors. Capacitors store current and inductors store voltage.
Electrical energy is transformed to rotational mechanical energy using a motor. Therefore, a motor has a positive pin, a negative pin and a mechanical output pin. An electrical transformer steps up or steps down input electrical energy and the output is electrical energy as well. An electrical transformer has two pairs of positive and negative pins. A generator converts rotational mechanical energy to electrical energy and has a mechanical input pin. To represent the physical concepts in the electrical domain we construct a meta-model for the electrical part of the IPM modelling language. The electrical domain meta-model is shown in Figure 1.11.
Looking at the meta-model we see that the IPMElement class is the super-class for all IPM components. The IPMElement class consists of two properties, Element and Type. In the electrical domain the value of Type is "Electrical". ElectricalElement inherits from IPMElement and contains the property Value. The content of Value depends on the nature of the classes inheriting ElectricalElement. For instance, a resistor will store the resistance in Value and a capacitor will store the capacitance.
There are two types of components in the electrical domain. The first is a TwoPin class of objects that inherits from the ElectricalElement class. The second is the ElectroMechEnergyTransform class which acts as an interface between electrical and mechanical devices and is a super-class for devices that act as transducers.

Every object of type TwoPin contains the properties PositivePin1 and NegativePin1 which are graph nodes used for connecting two objects in the electrical domain. In the model they represent *ports* of connection. Two sub-classes of TwoPin are Resistor and Earth. The TwoPin class is further categorized as TwoPinStorage and TwoPinSource. The sub-classes of TwoPinStorage include the electrical Capacitor and Inductor. The InitialValue property for TwoPinStorage elements contain the initial amount of energy stored in the device. The sub-classes of TwoPinSource in-

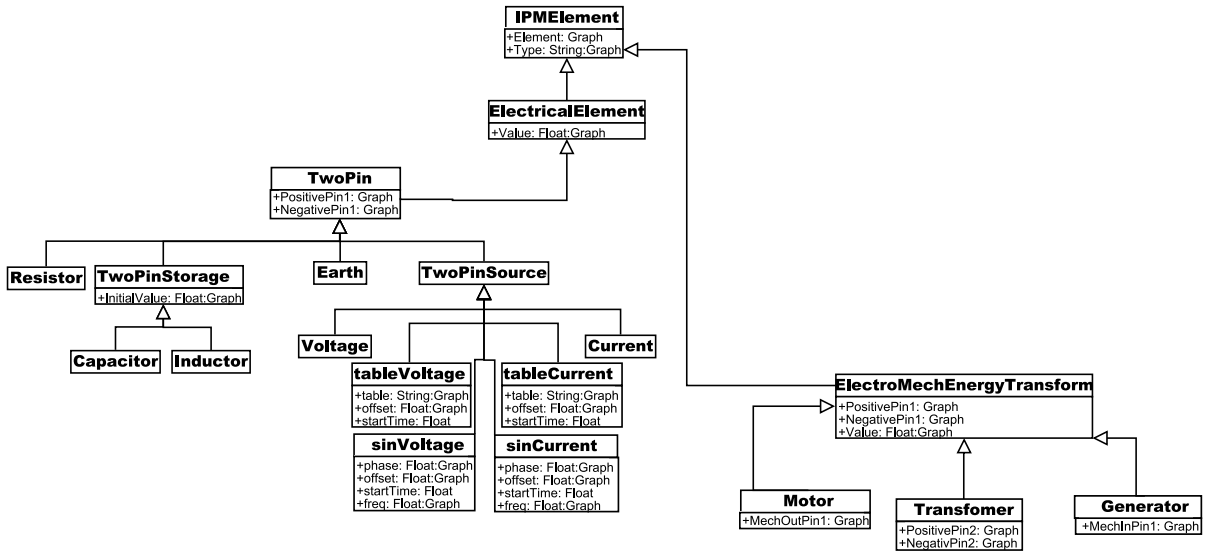


Figure 1.11: IPM Modelling Language Meta-model for the Electrical Domain

clude voltage sources and current sources. The voltage sources are Voltage, tableVoltage, and sinVoltage. The current sources are Current, tableCurrent, and sinCurrent. The class tableVoltage contains three attributes which have a graph representation. The properties are table, offset, and startTime. Similarly, other TwoPinSource classes have their own attributes as shown in the meta-model.
The ElectroMechEnergyTransform class is inherited by three kinds of energy transformers namely, Motor, Transformer, and Generator. The ElectroMechEnergyTransform class has the properties PositivePin1 and NegativePin1 as ports. The Motor class has MechOutPin1 for electrical to mechanical energy conversion. The Generator class has a MechInPin1 node for mechanical to electrical energy conversion. The Transformer class represents an electrical transformer that steps up or steps down voltage and has additional port nodes PositivePin2 and NegativePin2.

The concrete visual syntax for the concrete classes presented in the electrical domain part of the IPM meta-model is shown in Figure 1.12. The concrete visual syntax for the concrete classes in the electrical to mechanical energy transformation meta-model is shown in Figure 1.13.

1.3.2 The Translational Mechanical Domain

The translational mechanical domain consists of mechanical devices that operate on the basis of linear/translational force and velocity applied to it. Each mechanical device has an mechanical input port and a mechanical output port. A translational mechanical damper provides resistance against an input force or a velocity. Force is stored in a translational mechanical spring while momentum is stored in a translational mechanical inertance such as mass. Sources of translational mechanical forces and velocities have a mechanical output port. These concepts are modelled in the translational mechanical part of the IPM meta-model as shown in Figure 1.14.

Like the electrical domain element a translational mechanical element TranMechElement inherits from the super-class IPMElement. The TranMechElement class has properties Value and Unit. The contents of Value and Unit depend on the sub-classes of TranMechElement. The Tran-

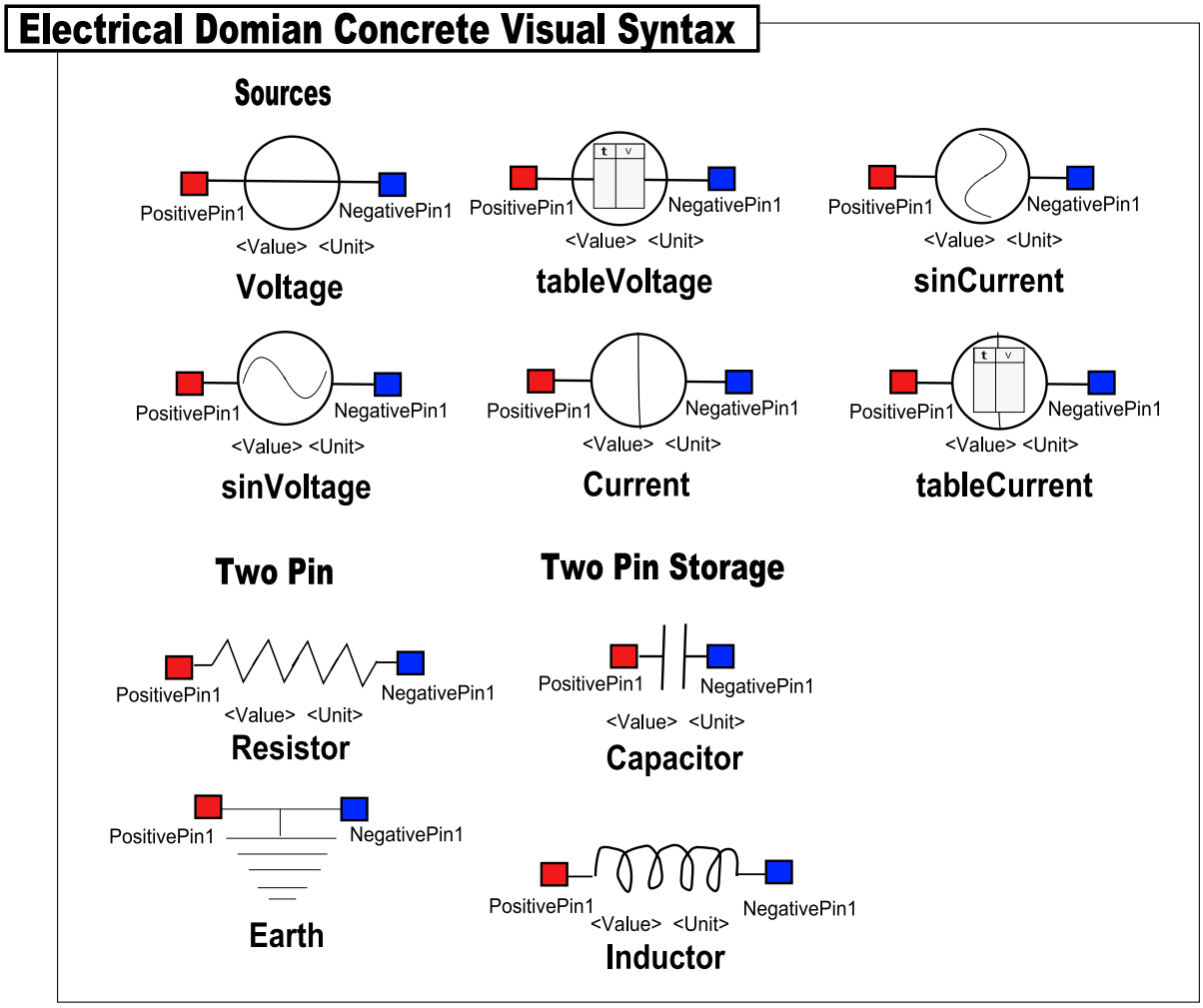


Figure 1.12: Concrete Visual Syntax for Electrical Elements in IPM

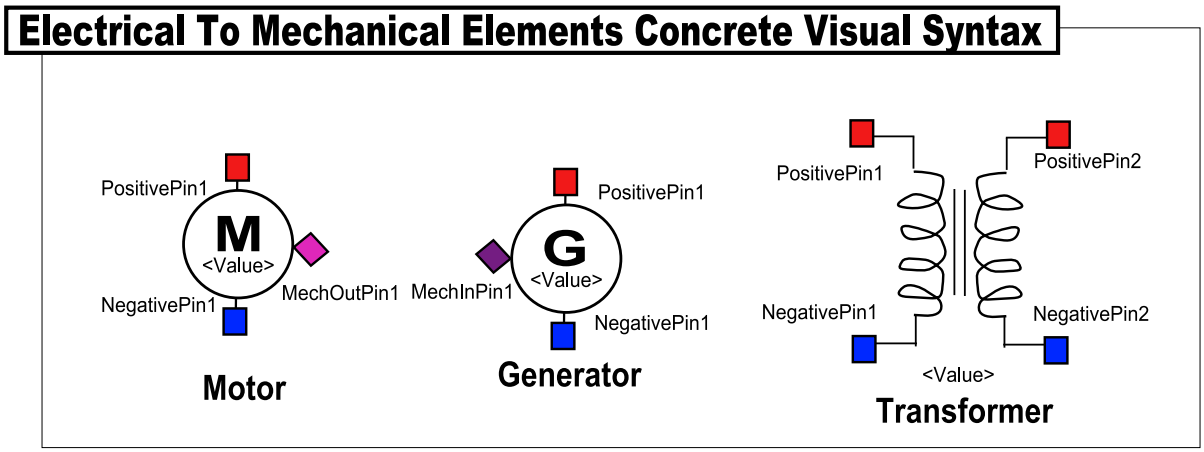


Figure 1.13: Concrete Visual Syntax for Electrical to Mechanical Elements in IPM

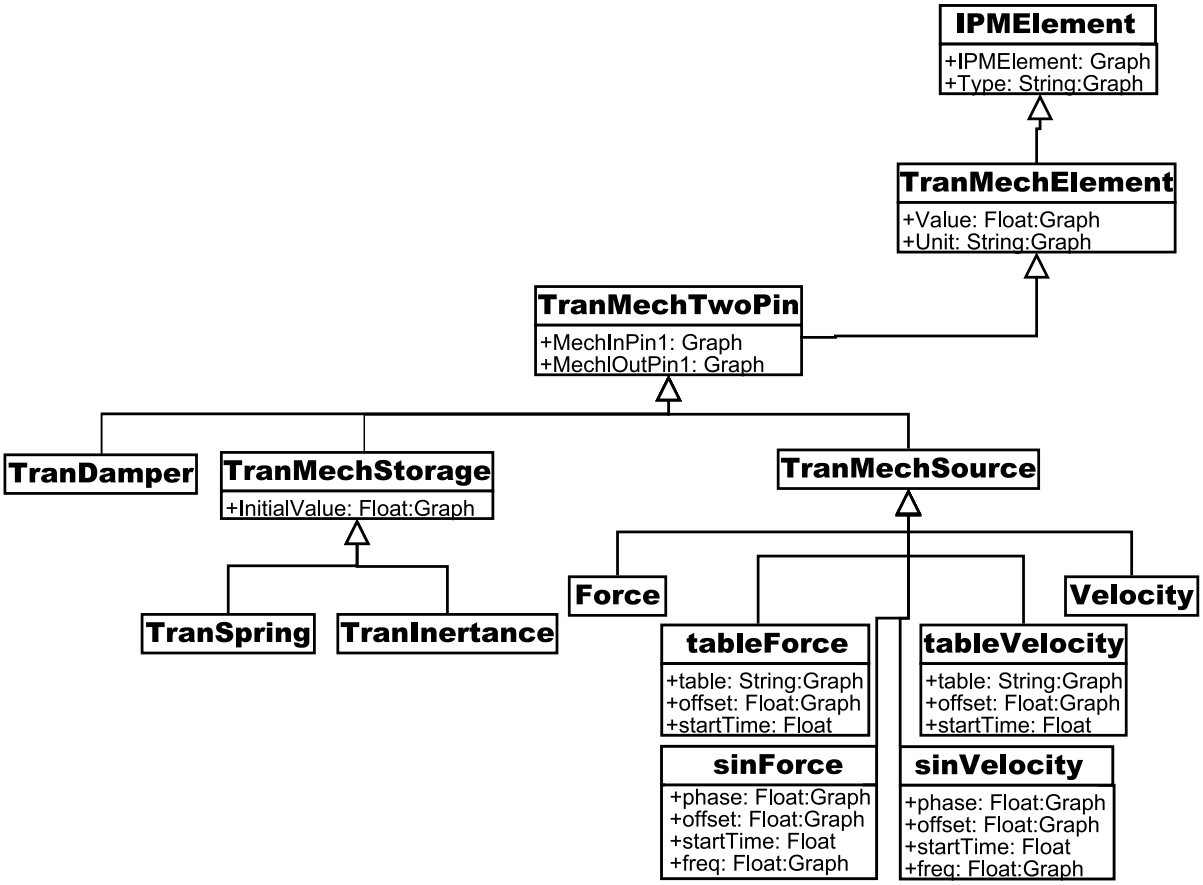


Figure 1.14: IPM Modelling Language Translational Mechanical Domain Meta-model

MechTwoPin class inherits the properties of TranMechElement. The TranMechTwoPin class and its sub-classes share the common properties MechInPin1 and MechOutPin1. The TranDamper class directly inherits from the TranMechTwoPin class.
The TranMechStorage class contains the property InitialValue. It is inherited by TranSpring and TranInertance. The TranMechSource class has sub-classes for force and velocity sources as shown in the meta-model. The properties of the sub-classes of TranMechSource sources are equivalent to those of the classes in the electrical domain and rotational mechanical domain.

The concrete visual syntax for the concrete classes in the translational mechanical domain part of IPM is shown in Figure 1.15.

1.3.3 The Rotational Mechanical Domain

The rotational mechanical domain consists of mechanical devices that operate on the basis of the torque and angular velocity applied to it. Each mechanical device has an mechanical input port and a mechanical output port. A rotational mechanical damper provides resistance against an input torque or an angular velocity. Torque is stored in a rotational mechanical spring while momentum is stored in a rotational mechanical inertance such as rotational inertia. Sources of rotational mechanical torques and angular velocities have a mechanical output port. These concepts are modelled in the rotational mechanical part of the IPM meta-model as shown in

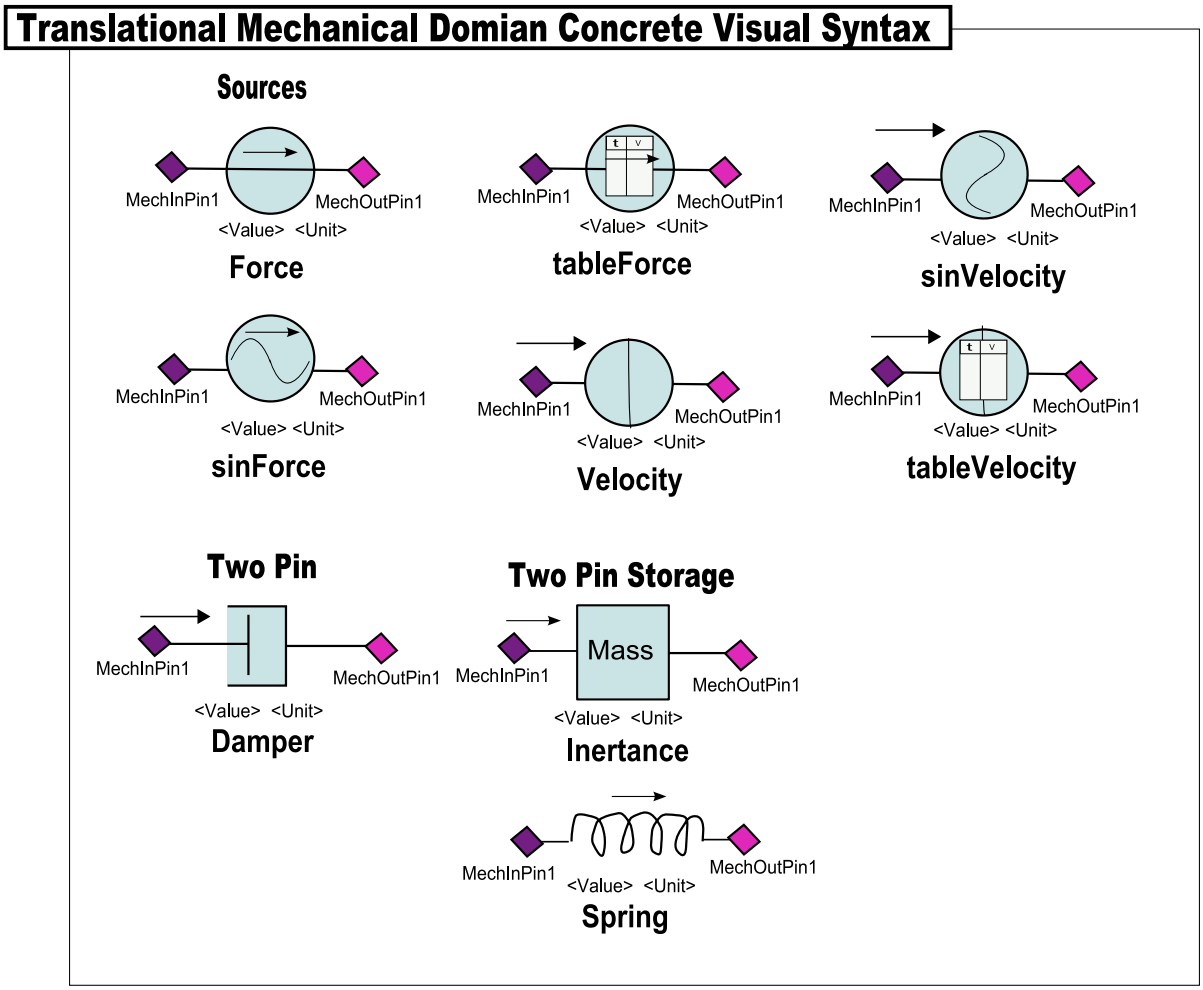


Figure 1.15: Concrete visual syntax for translational mechanical elements in IPM

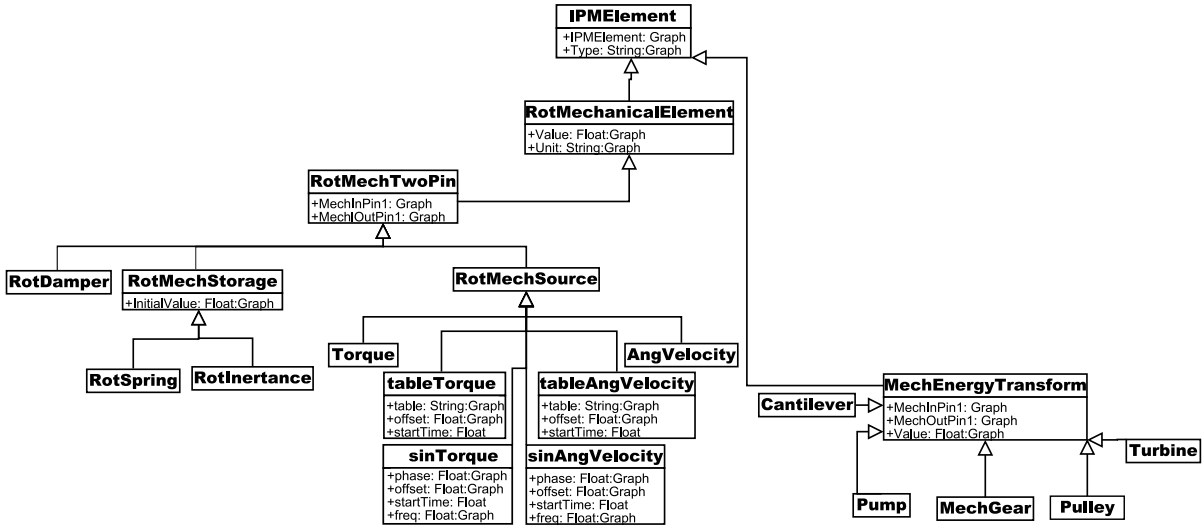


Figure 1.16: Idealized Physical Modelling formalism Rotational Mechanical Domain Meta-model

Figure 1.16.

The RotMechElement inherits from the IPMElement class. The RotMechElement class has properties Value and Unit. The contents of Value and Unit depends on the sub-classes of RotMechElement. For instance, a RotDamper object will have the units of $Nm/s¿$ The RotMechTwoPin class inherits the properties of RotMechElement. The RotMechTwoPin class and its sub-classes have share the common properties MechInPin1 and MechOutPin1. The RotDamper class directly inherits from the RotMechTwoPin class.

The RotMechStorage class contains the property InitialValue. It is inherited by RotSpring and RotInertance. The RotMechSource class has sub-classes for torque and angular velocity sources as shown in the meta-model. The properties of the sub-classes of the RotMechSource are equivalent to those of the classes in the electrical domain and translational mechanical domain.

The concrete visual syntax for the concrete classes in the rotational mechanical domain part of IPM is shown in Figure 1.17.

Transducers for transforming mechanical energy to mechanical energy are inherited from the MechEnergyTransform class. Unlike the electromechanical transducers the mechanical to mechanical transducers have an input port MechInPin1 and an output port MechOutPin1. The Cantilever scales up or down translational mechanical energy. The MechGear scales up or down rotational mechanical energy. The Pulley transforms rotational mechanical energy to translational mechanical energy. The Pump transforms rotational mechanical energy to pneumatic energy. The Turbine is responsible for transforming pneumatic energy to rotational mechanical energy. The concrete visual syntax for the concrete classes for the mechanical to mechanical transducers is given in Figure 1.18.

A visual modelling environment is synthesized from the meta-model using our tool AToM^3. A screenshot is shown in Figure 1.20. The model of the hoisting device in IPM is shown in Figure 1.19.

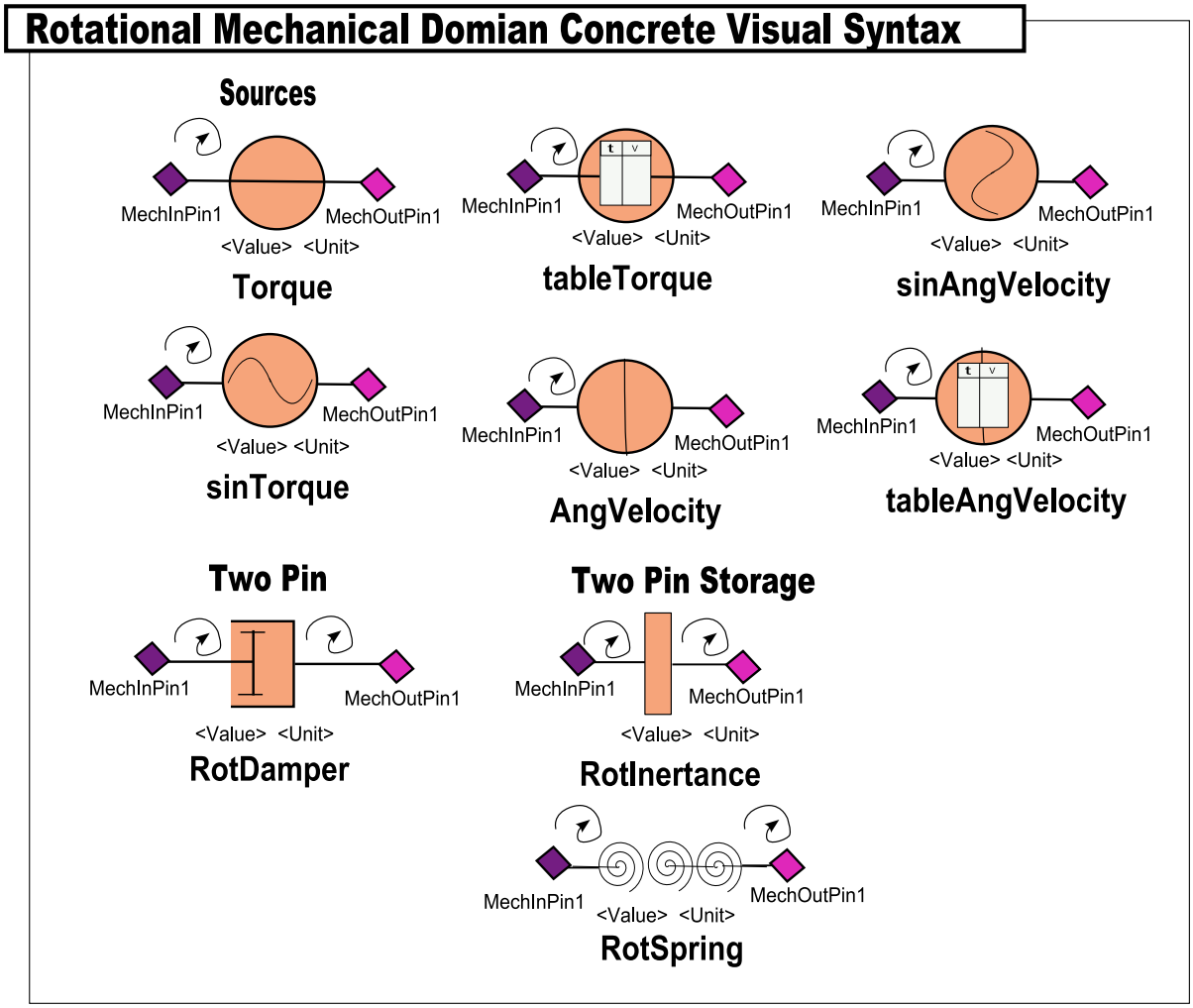


Figure 1.17: Concrete visual syntax for rotational mechanical elements in IPM

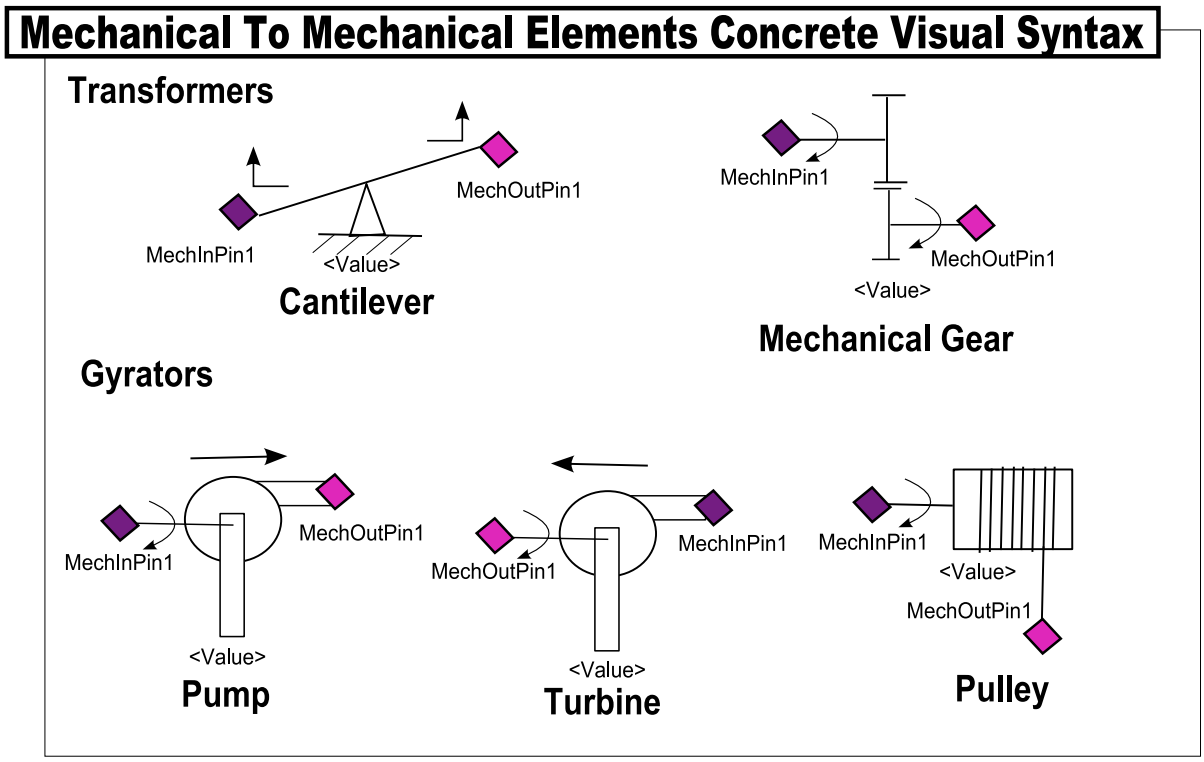


Figure 1.18: Concrete Visual Syntax for Mechanical to Mechanical Elements in IPM

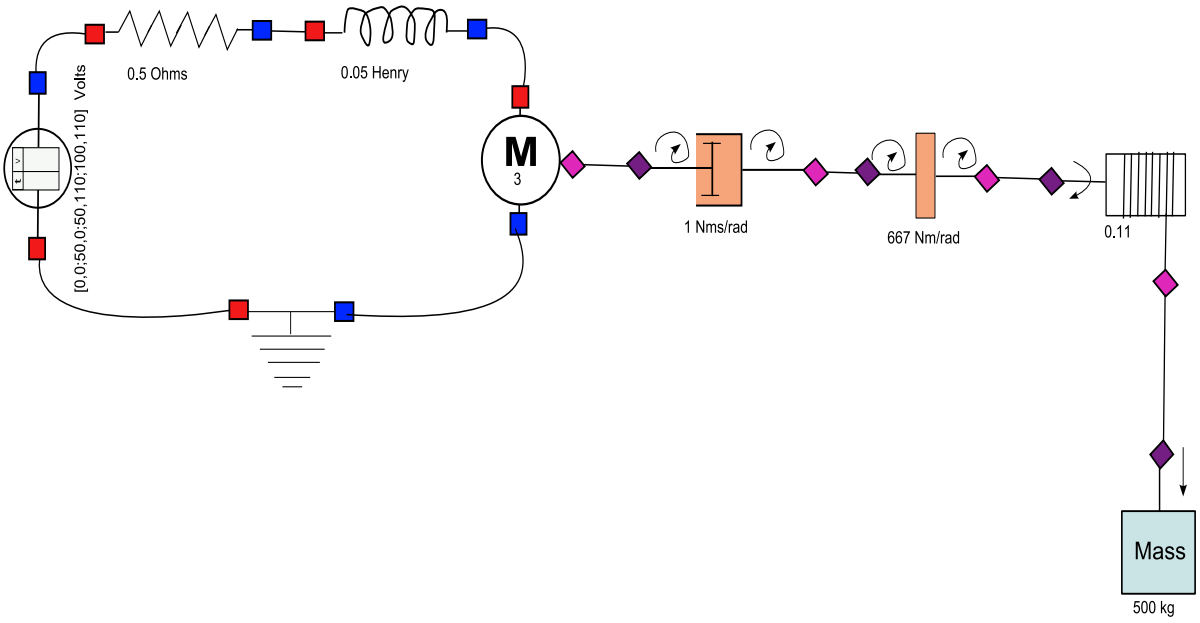


Figure 1.19: IPM Model of Hoisting Device

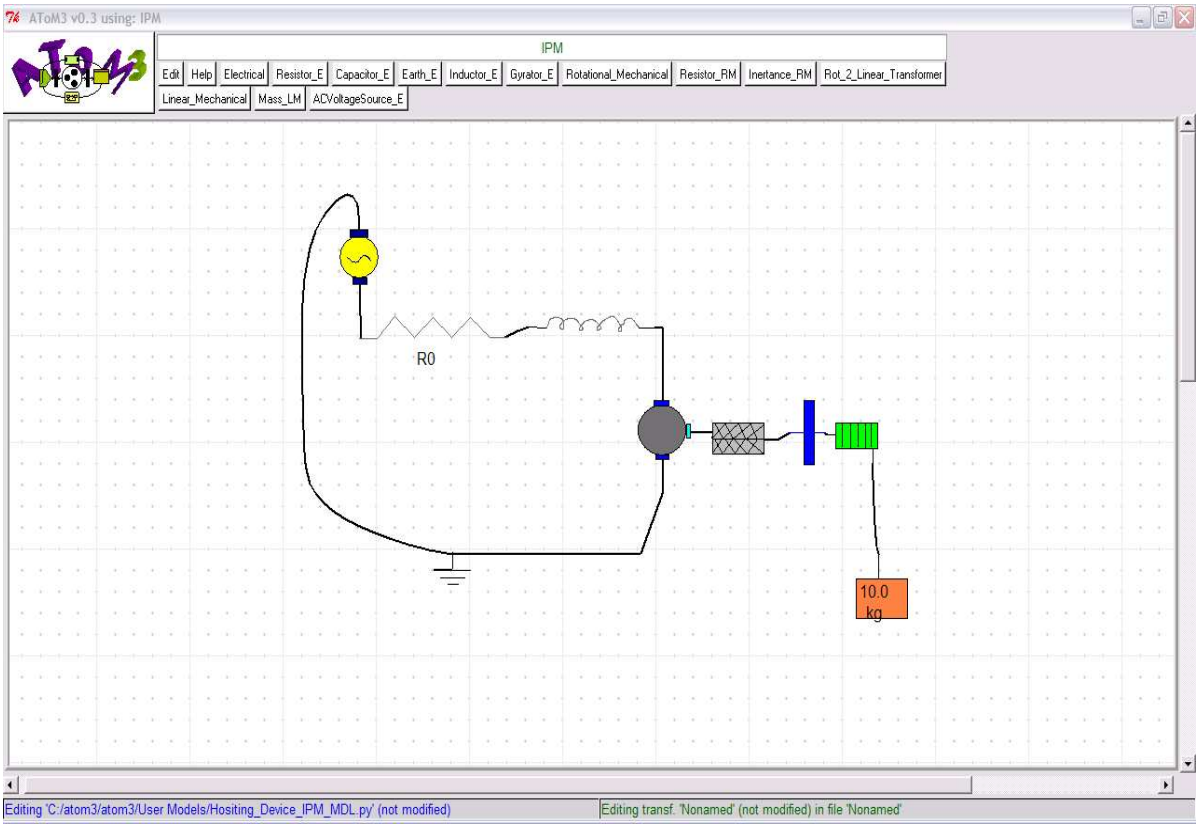


Figure 1.20: A Visual Modelling Environment Synthesized for IPMs in AToM^3

1.4 Hybrid Bond Graph Modelling Language

Models in the IPM modelling can directly be given concrete mathematical meaning in the form of Differential-algebraic Equations (DAE) or Ordinary Differential Equations (ODE). An alternative we take is to transform the IPM first to the Bond Graph (BG) modelling language. The BG modelling language can be used to verify the model's physical meaningfulness. This is done by verifying the laws of conservation of energy and momentum via *causality assignment*.

The Hybrid Bond Graph (HBG) modelling language comprises of the Bond Graph (BG) modelling language for plant modelling and the Causal Block Diagram(CBD) modelling language for controller modelling. Therefore, in general a HBG model can comprise of a BG sub-model and a CBD sub-model or just either one of them. The hybrid in HBG is due to the combination of CBD and BG elements in one language. The interface between the CBD sub-model and the BG sub-model is due to two possibilities. Certain components of the BG plant model are either sensed (diagnostic BG elements) and processed by the CBD model or controlled (modulated BG elements) by it. A HBG without causality assignment is called the Hybrid Acausal Bond Graph (HABG). The hoisting device example in HABG is shown in Figure 3.9. After causality assignment it is called the Hybrid Causal Bond Graph (HCBG). The hoisting device in HCBG is shown in Figure 1.24.

The EMOF based meta-model for the BG part of the HBG modelling language is shown in Figure 1.21. The CBD part of the meta-model is shown in Figure 1.27. A visual modelling environment is synthesized from the meta-model specification. A screenshot is shown in Figure 1.25. The visual syntax for HBG is shown in Figure 1.23. We start our discussion on the HBG modelling language by explaining the BG modelling language. This is followed by an explanation of the CBD modelling language.

1.4.1 The Bond Graph Modelling Language

The Bond Graph modelling language is a domain-independent graphical representation of energy flow structure in a physical system. The domain independence means that physical systems from different domains such as electrical, mechanical, hydraulic, chemical, and thermodynamics are all modelled using the same notation and give rise to equivalent dynamical equations. The BG formalism was first developed by Paynter [H.M61]. The idea was further popularized by Karnopp and Rosenberg [DKR00]. Today, it has evolved into a systems theory under the name of *Port Hamiltonian Systems* [Mac03] which uses *Generalized Hamiltonian Formulation* to model plant and controller systems.

The BG modeller is given the Idealized Physical Model[1] of a physical system. The modeller first identifies the elementary physical *concepts* from the IPM. This process is called *reticulation*. A *concept* or BG element manipulates energy. Energy, is an aggregate phenomena [Zun94], which can be calculated in many domains: electrical, mechanical, chemical, hydraulic, and thermodynamic. A BG element can be an energy source, energy store, energy transformer, or an energy dissipator irrespective of the domain. Also, the mathematical equations describing the energetic behaviour of the BG elements are identical for equivalent physical concepts across domains. Energy is exchanged between the BG elements via bonds connected to a junction structure.

The BG modelling language is now explained with the help of an example. The example, shown in Figure 1.26 is designed to show how idealized physical models from different domains

[1] Idealized Physical Model (IPM)

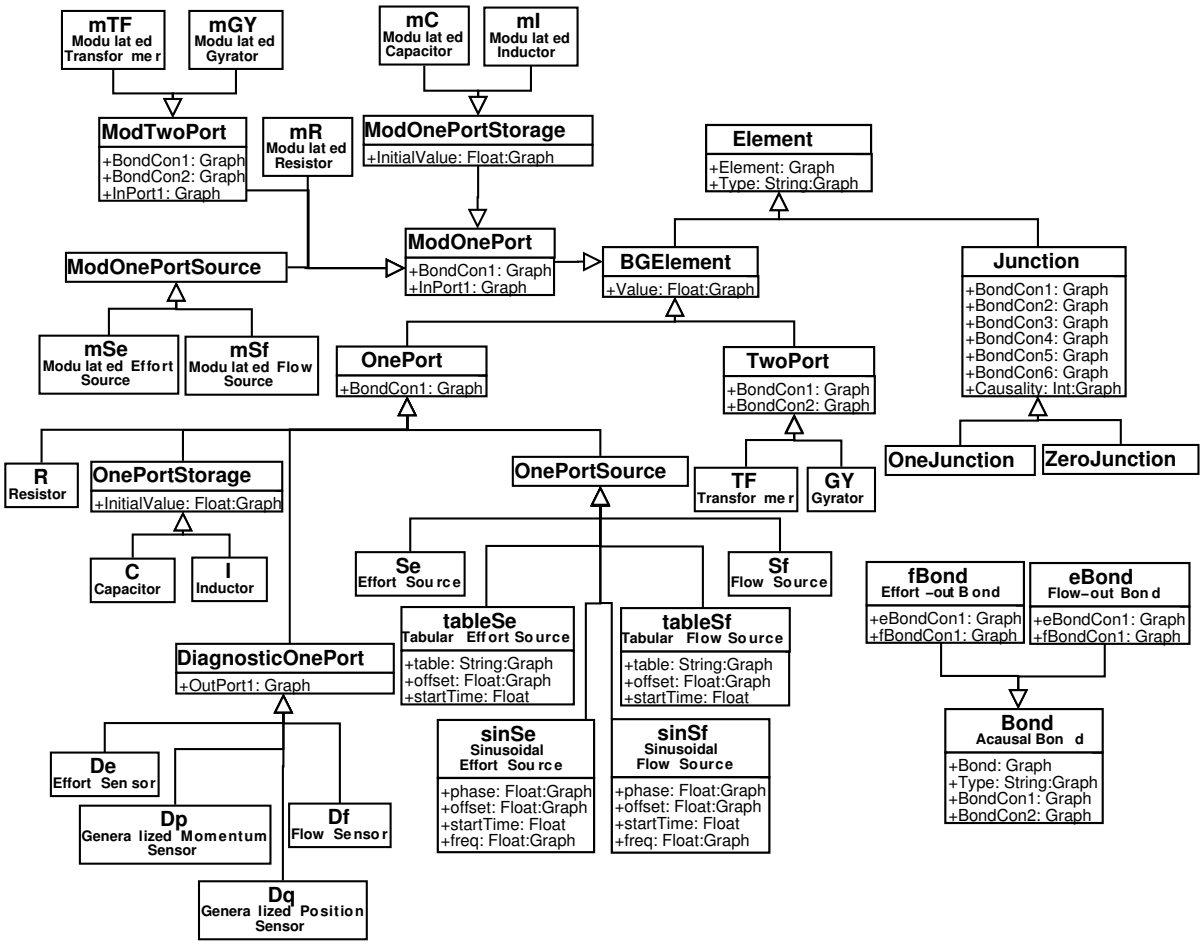


Figure 1.21: Hybrid Bond Graph meta-model for the Bond Graph formalism

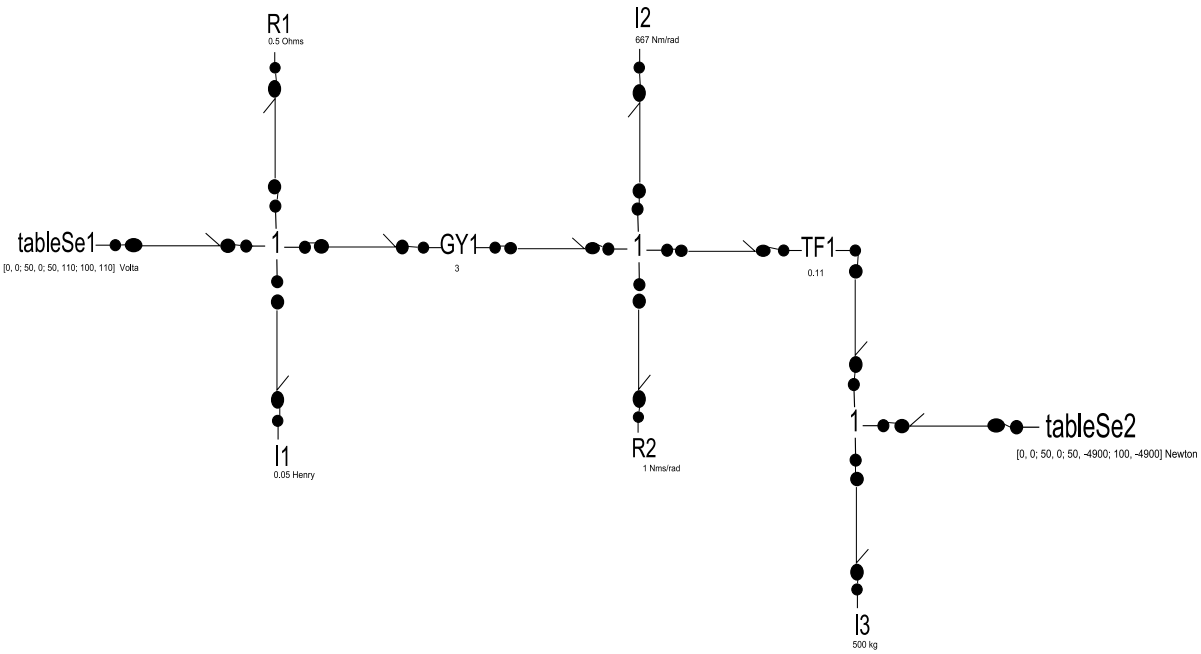


Figure 1.22: Hybrid Acausal Bond Graph

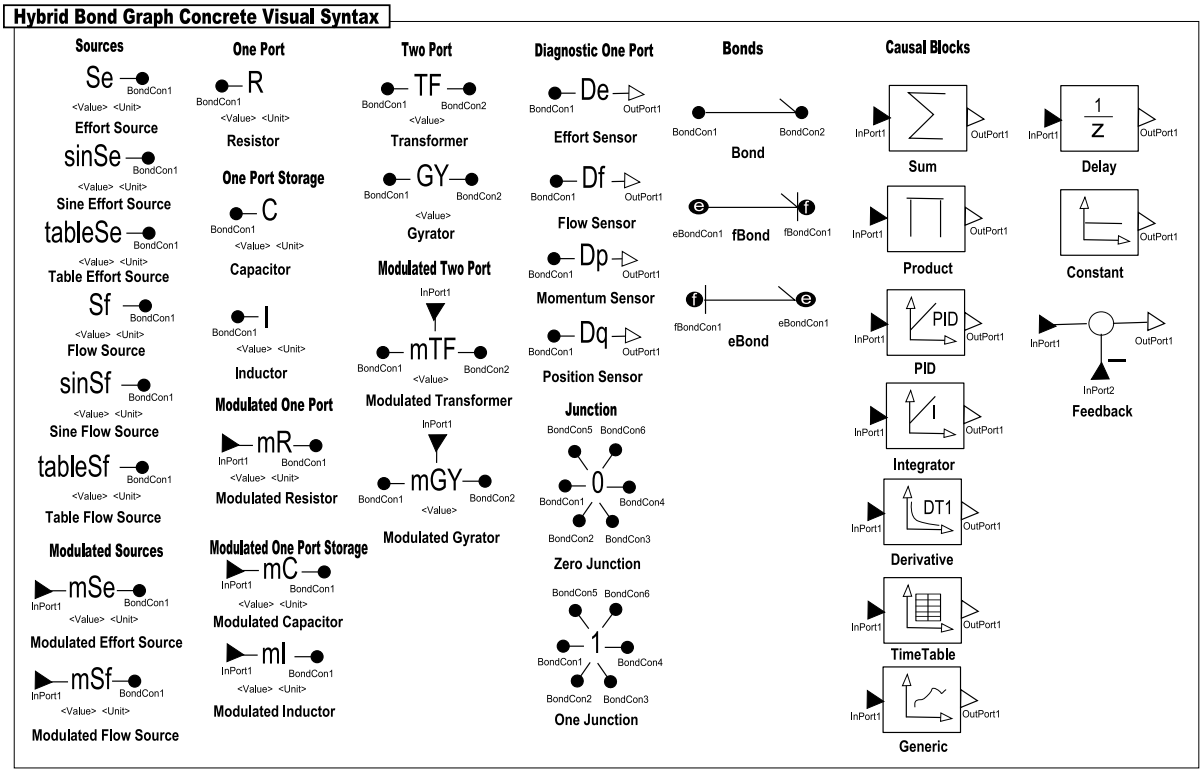


Figure 1.23: Concrete visual syntax for HBG

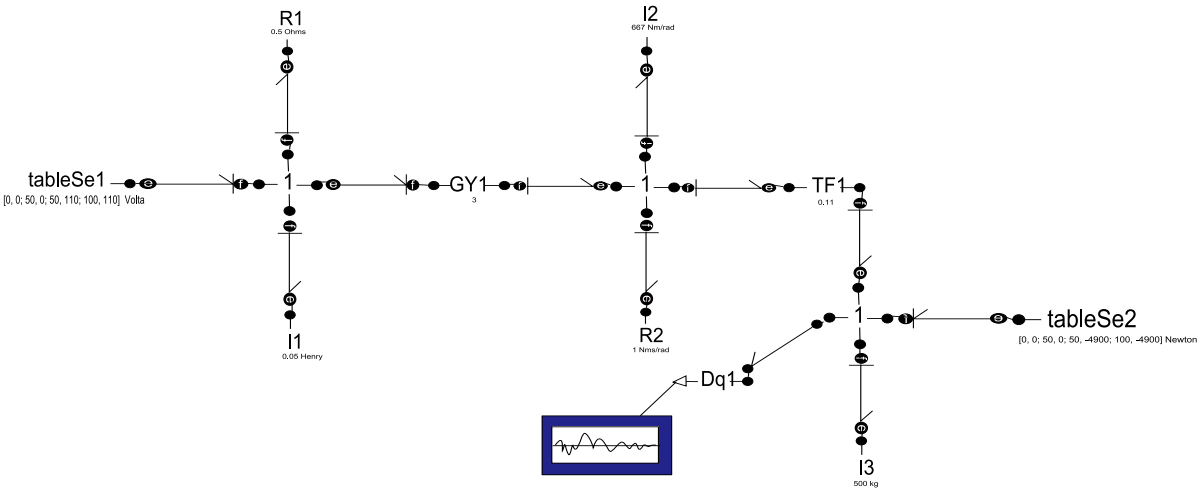


Figure 1.24: Hybrid Causal Bond Graph

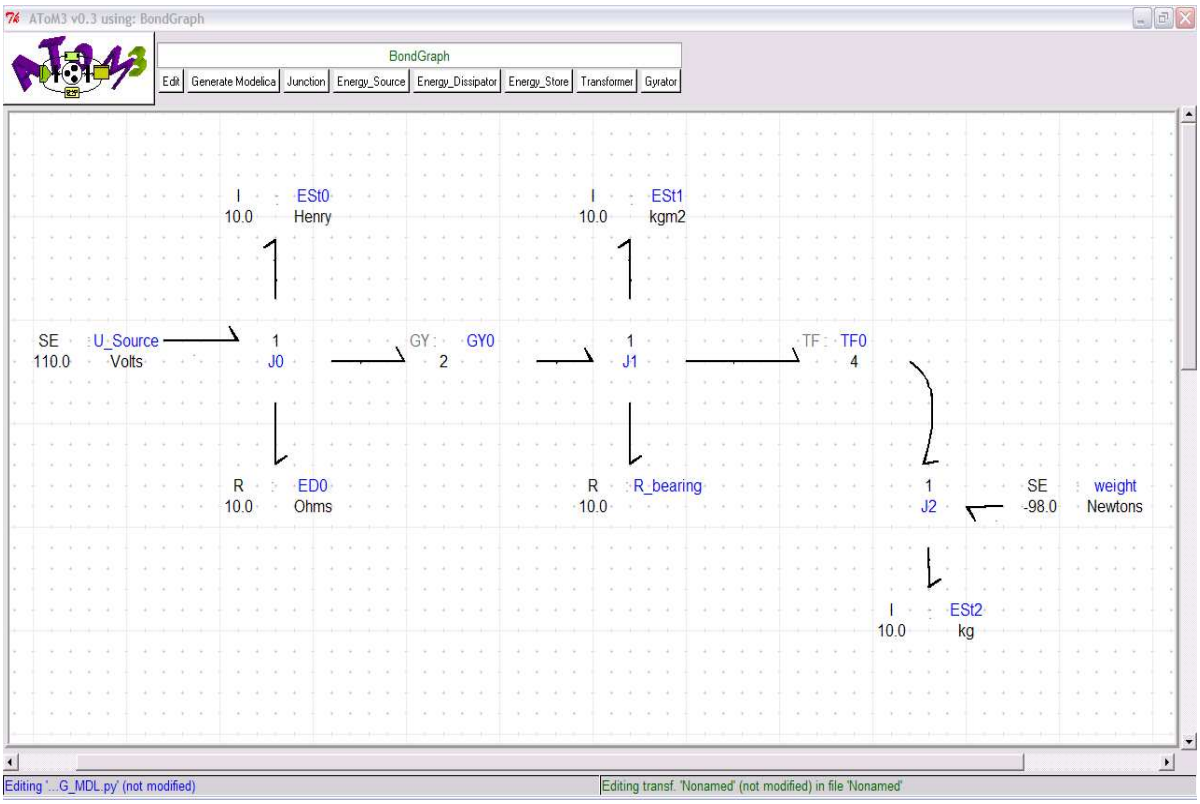


Figure 1.25: Screenshot of Visual Modelling Environment for HABG in AToM^3

(electrical and mechanical in this case) become domain-independent when transformed to the BG modelling language. Two systems are considered in the example of Figure 1.26. First, we have a purely electrical serial LCR (Inductor, Capacitor, Resistor) circuit as shown in Figure 1.26(a). The second system is a purely mechanical damped mass-spring system as shown in Figure 1.26(b).

We transform the LCR circuit to an electrical domain BG in the following steps:

1. Draw the electrical domain elements, separating them by their positive and negative pins. These elements appear in rectangular boxes as shown in Figure 1.26(c).
2. We now attach a port called a power port with each of these rectangular boxes containing electrical elements.
3. To this port we connect a bond or a power bond that denotes the exchange of energy between elements. The bond is drawn like an edge with a half arrow tip. The direction of energy flow is determined by the direction of the half arrow.
4. We now add a 1−junction to the electrical BG model. The 1−junction indicates that the current i in the serial LCR circuit is constant but the voltage across each electric element varies.
5. The voltage source is the source of energy hence it is connected to the 1−junction with the half arrow toward the 1−junction. The inductor and capacitor elements store energy hence the half arrow is in the direction of the respective electric elements. The resistor dissipates consumed energy bringing the half arrow direction from the 1−junction toward itself.

Similarly, the damped mass-spring idealized physical model from the translational mechanical domain is transformed to a mechanical domain BG. The transformation process yields an equivalent mechanical domain BG as shown in Figure 1.26 (d).

These examples serve as a good trailer to understand the modelling elements of a BG. We can see that two variables voltage and current play the role of transferring energy between elements in the electrical domain. The product of these two variables is *power*. Similarly, the product of force and velocity is power in the translational mechanical domain and the product of torque and angular velocity is power in the rotational mechanical domain. This common trend implies that quantities like voltage, force, and torque on the one hand and current, velocity, and angular velocity on the other are analogous quantities. Variables such as voltage, force, and torque are called *effort* variables. Similarly, variables such as current, velocity, and angular velocity are called *flow* variables. The energy flow between elements has the physical dimension of power which is the product of effort and flow. Therefore effort and flow are known as *power-conjugated* variables.

The analogy between domains is not just between effort and flow variables but it also exists between the basic elements of the different domains (electrical and mechanical in our example). Here is a list of analogies observed in the elements of the electrical and mechanical elements in our example:

- The damper is analogous to the resistor
- The spring is analogous to the capacitor
- The mass is analogous to the inductor
- The force source is analogous to the voltage source

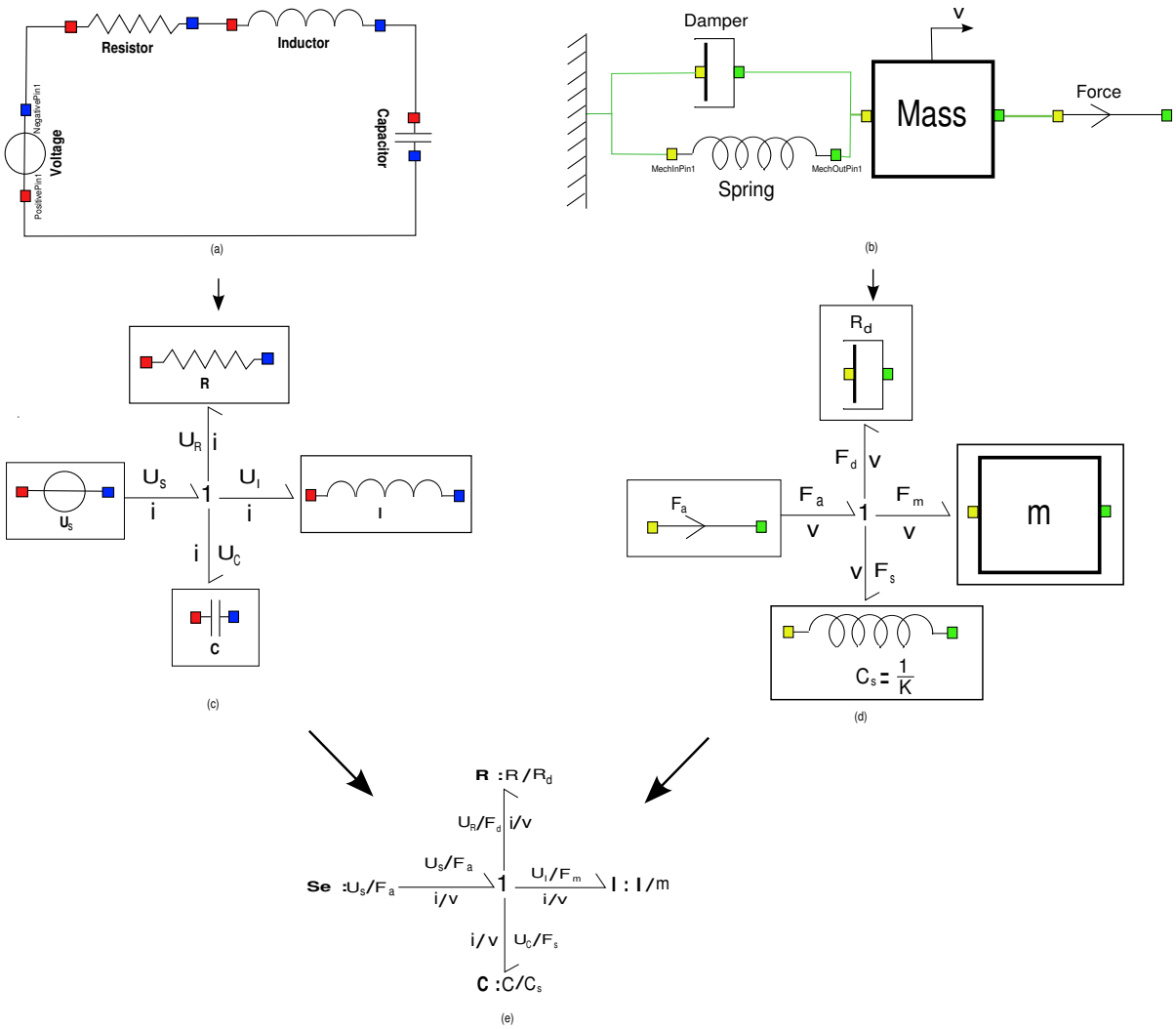


Figure 1.26: (a) Idealized Physical Model of LCR Circuit (b) Idealized Physical Model of Damped Mass-Spring System (c) Bond Graph model of LCR Circuit using Electrical Domain Notation (d) Bond Graph model of Damped Mass-Spring System using Mechanical Domain Notation (e) Bond Graph model of LCR Circuit and Damped Mass Spring System using Standard Notation

- The common velocity is analogous to the loop current

The analogies in the basic elements enables us to finally specify the standard BG model. This is shown in Figure 1.26 (e). The voltage or force source are effort sources hence the standard bond graph notation **Se** is used to represent an effort source. The **R** symbol represents an energy dissipator, **I** is the flow store, **C** is the effort store. The equations that describe the dynamics of the electrical elements are given below:

$u_R = iR$

$u_C = \frac{1}{C}\int idt$

$u_L = L\frac{di}{dt}$ or $i_L = \frac{1}{L}\int udt$

where u_R is the voltage across the resistor, u_C is the voltage across the capacitor, u_L is the voltage across the inductor, i is the current flowing in the circuit, R is the resistance , C is the capacitance, and L is the inductance of the LCR circuit. The quantity i_L is accumulated in the inductor. The inductor is a flow store, in the electrical domain it stores the current.

The equations that describe the dynamics of the mechanical elements are given below:

Instances of the BG modelling language comprising of an interconnection of elementary components itself is a component. This component has an interface that can used (reused) as a module in a parent bond graph. Further, the non-causal nature of BG components make it a hierarchically composed formalism.

The dynamical behaviour of a BG can be obtained by mapping it onto a Causal Block Diagram and simulated the CBD or by writing out Differential Algebraic Equations and solving the set of equations.

First, we describe the BG elements of the HBG modelling language. The meta-model, shown in Figure 1.21, is used to specify the properties of BG elements.

Bonds

A bond represents the flow of power, P, from one point of a physical system to another. It is represented by a harpoon. There are two physical variables associated with each bond, an effort, e, and a flow, f. The product of these two variables represents the power: $P = e \times f$.

When causality is assigned to a bond it gets a computational order. The fBond class implies that the Bond has received an effort-out causality. The eBond class implies that the Bond has received a flow-out causality. The equations for the bonds is given in Table 1.1. The ports for the non-causal Bond class are BondCon1 and BondCon2. The ports for fBond, eBond classes are eBondCon1 and fBondCon1. Each BG element with one or more ports have names BondCon1, BondCon2,...and so on. The Bond class and its sub-classes fBond and eBond connect to other BG elements via these ports.

Energy Sources

Energy sources are interfaces of the BG with its environment. In the real world examples of energy sources are: voltage and current in the electrical domain, force and velocity in the mechanical domain. In the bond graph modelling language sources of voltages, force, and torque are called *effort sources* and sources of current, velocity, and angular velocity are called *flow sources*.

In Table 1.2 we present the equations that represent the semantics of the energy sources in the meta-model for HBG. The basic effort source is Se and a basic flow source is Sf. The tableSe

Table 1.1: Equations for Bond Graph Bonds

BG Element	Equation
Bond	$BondCon2.e(t) = BondCon1.e(t), BondCon2.f(t) = BondCon1.f(t)$
	$BondCon1.d = -1, BondCon2.d = +1$
fBond	$fBondCon1.e(t) = eBondCon1.e(t), eBondCon1.f(t) = fBondCon1.f(t)$
	$eBondCon1.d = -1, fBondCon1.d = +1$
eBond	$fBondCon1.e(t) = eBondCon1.e(t), eBondCon1.f(t) = fBondCon1.f(t)$
	$fBondCon1.d = -1, eBondCon1.d = +1$

Table 1.2: Equations for Bond Graph Energy Sources

BGElement	Equation
Se	$e(t) = e0$
tableSe	$e(t) = TimeTable(table, startTime, offset)$
sinSe	e(t)=$sin(e0, freq, offset, phase, startTime)$
Sf	$f(t) = f0$
sinSf	$f(t) = sin(f0, freq, offset, phase, startTime)$
tableSf	$f(t) = TimeTable(table, startTime, offset)$
mSe	$e0 = s, e(t) = s$, s is the input signal
mSf	$f0 = s, f(t) = s$, s is the input signal

and tableSf are tabular sources. The table contains two element tuples. A linear interpolation is performed from one tuple to the next to obtain a continuous function. The sinSe and sinSf source are sources as sinusoidal wave forms. The amplitude, frequency, start time, and offset are parameters for the sinusoidal sources. The sources mSe and mSf are modulated sources. The modulated sources are controlled by the signal domain, hence they have an input s which represents the value of a signal from a controller or an other external signal source.

Energy Dissipators

Energy dissipators are responsible for consuming energy from the system model. They are called resitors in the HBG modelling language and are associated with a resistance. In the electrical domain an electrical resistor, in the translational mechanical domain a damper and in the rotational mechanical domain a rotational damper are all modelled as a BG resistor R. The semantics of an R element is given in Table 1.3.

A modulated resistor, mR, is controlled by the signal domain. The semantics of a modulated resistor is presented is presented in Table 1.3. The s signal gives variable resistance to the R element. The s variable is controlled by an external controller or an other signal source.

Energy Storage

Energy storage in the BG modelling language is of two kinds. The storage of flow takes place in the capacitor element C. The storage of effort takes place in the inductor element I. The

Table 1.3: Equations for Bond Graph Energy Dissipators

BG Element	Equation
R	$e(t) = R * f(t)$
mR	$R = s, e(t) = R * f(t)$

Table 1.4: Equations for Bond Graph Storage

BG Element	Equation
C	$f(t) = C \times \frac{de(t)}{dt}$
I	$e(t) = I \times \frac{df(t)}{dt}$
mC	$e(t) = s, f(t) = C \times \frac{de(t)}{dt}$
mI	$f(t) = s, e(t) = I \times \frac{df(t)}{dt}$

equations are presented in Table 1.4. Effort is stored in the electrical inductor of the electrical domain and flow is stored in the electrical capacitor. In the translational mechanical domain effort is stored in the inertance and flow is stored in the spring. Finally, in the rotational mechanical domain effort is stored in the rotational inertance and flow in the rotational spring elements.

Energy Transformers

Energy transformation can be formed by two kinds of elements, a transformer and a gyrator. If the input to the transformer TF is an effort the output is also an effort scaled up or down by a certain factor. A gyrator GY transforms effort to flow and flow to effort each scaled up or down by a certain factor. A transformer usually transforms energy within the same domain. Examples of transformers are the electrical transformer (electrical domain), cantilever (translational mechanical domain), and the mechanical gear (rotational mechanical domain). A gyrator transforms energy between two domains. Examples of gyrators include the motor (electrical to rotational mechanical), generator (rotational mechanical to electrical), pump (rotational mechanical to pneumatic), pulley (rotational mechanical to translational mechanical), and turbine (pneumatic to rotational mechanical). The equations that describe the semantics of TF and GY elements in given in Table 1.5. The scaling factors are m for transformers and r for gyrators.

Table 1.5: Equations for Bond Graph Transformers

BG Element	Equation
TF	$e1(t) = m \times e2(t), f2(t) = m \times f1(t)$
GY	$e1(t) = r \times f2(t), e2(t) = r \times f1(t)$
mTF	$m = s, e1(t) = m \times e2(t), f2(t) = m \times f1(t)$
mGY	$r = s, e1(t) = r \times f2(t), e2(t) = r \times f1(t)$

Table 1.6: Equations for Bond Graph Junctions

BG Element	Equation
0−Junction	$e[2:6](t) = e[1:5](t), \sum f(t) = 0$
1−Junction	$f[2:6] = f[1:5], \sum e(t) = 0$

Junctions

Junctions couple two or more BG elements in a power continuous way: there is no energy storage or dissipation in a junction. Examples are a series connection or a parallel connection in an electrical network, a fixed coupling between parts of a mechanical system. Junctions are portsymmetric: the ports can be exchanged in the constitutive equations. Following these properties, it can be proven that there exist only two pairs of junctions: the 1junction and the 0junction.

The 0junction represents a node at which all efforts of the connecting bonds are equal. An example is a parallel connection in an electrical circuit. Due to the power continuity, the sum of the flows of the connecting bonds is zero, considering the sign. The power direction (i.e. direction of the half arrow) determines the sign of the flows: all inward pointing bonds get a plus and all outward expansion pointing bonds get a minus. This summation is the Kirchhoff current law in electrical networks: all currents connecting to one node sum to zero, considering their signs: all inward currents are positive and all outward currents are negative. We can depict the 0junction as the representation of an effort variable, and often the 0junction will be interpreted as such. The 0junction is more than the (generalised) Kirchhoff current law, namely also the equality of the efforts (like electrical voltages being equal at a parallel connection).

The 1junction is the dual form of the 0junction (roles of effort and flow are exchanged). The 1−junction represents a node at which all flows of the connecting bonds are equal. An example is a series connection in an electrical circuit. The efforts sum to zero, as a consequence of the power continuity. Again, the power direction (i.e. direction of the half arrow) determines the sign of the efforts: all inward pointing bonds get a plus and all outward pointing bonds get a minus. This summation is the Kirchhoff voltage law in electrical networks: the sum of all voltage differences along one closed loop (a mesh) is zero. In the mechanical domain, the 1junction represents a force balance (also called the principle of dAlembert), and is a generalisation of Newtons third law, action = reaction). Just as with the 0junction, the 1junction is more than these summations, namely the equality of the flows. Therefore, we can depict the 1junction as the representation of a flow variable, and often the 1−junction will be interpreted as such.

The equations for BG junctions is given in Table 1.6.

Diagnostic Elements

A BG contains many diagnostic elements that can be used to read an effort, flow, the generalized position and momentum from a junction. All diagnostic elements in the BG language inherit from DiagnosticOnePort. The description for the diagnostic elements in a BG is given in Table 1.7.

Table 1.7: Equations for Bond Graph Diagnostic Elements

BG Element	Equation
De	$Outport1 = e(t), f(t) = 0$
Description	This component is used to sense the value of effort in a junction
Df	$Outport1 = f(t), e(t) = 0$
Description	This component is used to sense the value of flow in a junction
Dp	$Outport1 = \int e(t)dt$
Description	This component is used to sense the value of generalized momentum in a junction
Dq	$Outport1 = \int f(t)dt$
Description	This component is used to sense the value of generalized position in a junction

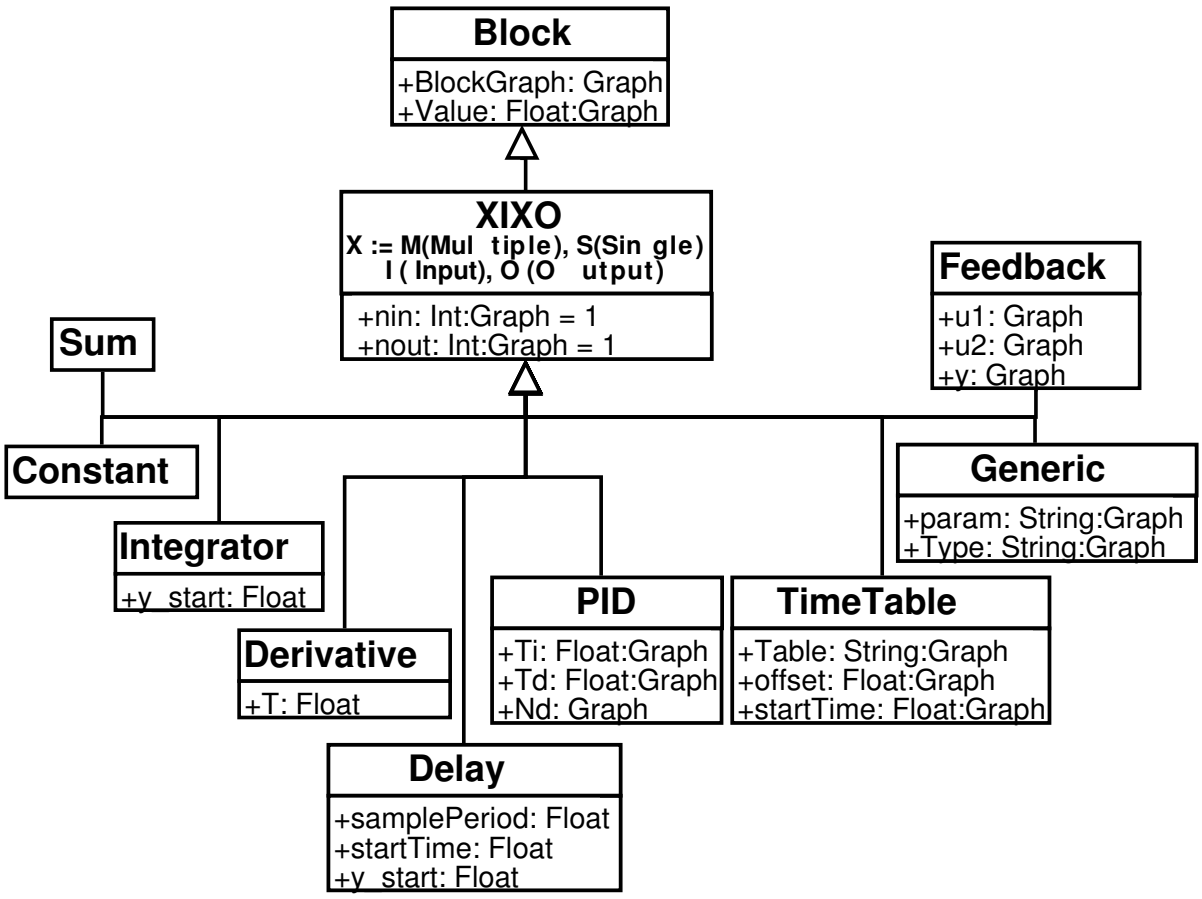


Figure 1.27: Causal Block Diagram part of the Hybrid Bond Graph Meta-model

1.4.2 The Causal Block Diagram Modelling Language

A causal block diagram processes *signals*. A causal block diagram consists of input/output *control blocks*. These control blocks are either continuous, discrete, logical, or table blocks. Each block can have zero, single or multiple inputs or outputs depending on the operation performed. Input signals are processed by a control block and the output is a signal. We present a meta-model for the Causal Block Diagram modelling langauge with a subset of operation blocks in Figure 1.27

Each block has a certain number of inputs and outputs and a output signal value based on the operation it performs. In the meta-model the class Block contains the properties BlockGraph and Value. The BlockGraph property is the root node of the abstract syntax graph that represents a CBD model. The Value property contains the signal value. The class XIXO inherits from Block and has properties that depict the number of inputs, nin, and outputs nout.

The concrete classes inherit from XIXO and represent the operation blocks that are used in a CBD. The various operation blocks and their description is given in Table 1.8.

1.5 Modelica Language

In this section we present the modelling of an EPS in the object-oriented physical system modelling language called Modelica. Modelica is based on an object-oriented textual representation of a common mathematical framework of Differential Algebraic Equations (DAE). DAE systems possess both discrete and continuous behaviour. This allows for the representation of HBG models which can contain both continuous Ordinary Differential Equations for the BG dynamics and discrete transitions due to algebraic equations obtained from the controller expressed as a CBD.

The Modelica language describes a model using the construct *model*. A model is a class that contains *state objects* and *equations*. State objects are instances of models of the components in the system. Equations either describe the DAE governing the evolution of the state objects or specify a connection between objects. The connection implies that there is flow of a physical quantity or information between objects. A Modelica model is flattened to obtain the algebraic assignment equations that are used to connect objects. For more information on Modelica refer to the citation [Fri03] [Mat97]. The BNF grammar for Modelica in essence specifies the meta-model for the Modelica langauge. For information on the Modelica language specification see [Mod05].

The Modelica language comes with a standard library which contains model elements for causal blocks. A BG modeling library written by Prof. Francois Cellier is also available [Cel05]. We use these libraries to represent HBG models in Modelica.

The concrete textual syntax for the main module for the hoisting device model in textual Modelica form is given below:

```
model HoistingDeviceHABG
  "Hoisting Device Model in Modelica using Bond Graph Library"
  BondLib.Junctions.J1p4 J1p4_1;
  BondLib.Bonds.fBond fBond1;
  BondLib.Passive.I I1(I=0.05);
  BondLib.Passive.R R1(R=0.5);
  BondLib.Bonds.eBond eBond2;
  BondLib.Passive.GY GY1(r=3);
```

Table 1.8: Equations for Causal Block DiagramElements

CBD Element	**Equation**
Constant	$y = k, k = 0$, where k is a Real Number
Description	A constant signal value is given as output. There is no input.
Sum	$y = \Sigma u$, where u is the input vector
Description	The sum of the input signals is given as a single/multiple output(s). The number of input signals is two or more.
Integrator	$y = \int_0^t u du$, where u is the input vector
Descrition	The integrator is a continuos block that integrates the incoming signal upto the current simulation time.
Derivative	$y = \frac{du}{dt}$, where u is the input vector
Description	This block outputs the derivative of a input signal with respect to the current and previous time steps
Delay	$y = pre(u)$, where u is the input vector
Description	This block outputs the value of the signal at the previous time steps
PID	$y = N_d e(t) + \frac{N_d}{T_i} \int_0^t e(t)dt + N_d T_d \frac{de(t)}{dt} + u_0$, where e is the error signal N_d is the gain, T_i is the time constant of the integrator, and T_d is the time constant of derivative
Description	This block outputs the proportional, integral, derivative control signal based on error feedback
TimeTable	$y = table, table = [0, 0; 0, 10; 0, 100]$, where $table$ is a collection of tuples
Description	This block outputs the a value at a given time from the interpolated function of the values in the table
Generic	$y = f(u, t)$, where f is a generic function
Description	This block processes the input signal using an arbitrary generic function
Feedback	$y = u_1 - u_2$, where u_1 and u_2 are input vectors
Description	This block outputs the difference in two input signals

```
  BondLib.Junctions.J1p4 J1p4_2;
  BondLib.Bonds.eBond eBond5;
  BondLib.Passive.I I2(I=667);
  BondLib.Passive.R R2(R=1);
  BondLib.Passive.TF TF1(m=0.11);
  BondLib.Bonds.eBond eBond6;
  BondLib.Bonds.eBond eBond7;
  BondLib.Bonds.eBond eBond8;
  BondLib.Passive.I I3(I=500);
  BondLib.Bonds.fBond fBond3;
  BondLib.Sensors.Dq Dq1;
  BondLib.Junctions.J1p4 J1p4_3;
  BondLib.Bonds.Bond Bond1;
  BondLib.Bonds.eBond eBond3;
  BondLib.Bonds.fBond fBond4;
  BondLib.Bonds.eBond eBond1;
  BondLib.Bonds.eBond eBond4;
  BondLib.Sources.tableSe tableSe1(table=[0,0; 50,0; 50,110; 100,110]);
  BondLib.Sources.tableSe tableSe2(table=[0,0; 50,0; 50,-4900; 100,-4900]);
equation
  connect(fBond1.fBondCon1, J1p4_1.BondCon1);
  connect(J1p4_1.BondCon3, eBond2.fBondCon1);
  connect(eBond2.eBondCon1, R1.BondCon1);
  connect(J1p4_2.BondCon3, eBond5.fBondCon1);
  connect(eBond5.eBondCon1, R2.BondCon1);
  connect(J1p4_2.BondCon2, eBond6.fBondCon1);
  connect(eBond6.eBondCon1, TF1.BondCon1);
  connect(TF1.BondCon2, eBond7.fBondCon1);
  connect(eBond8.eBondCon1, I3.BondCon1);
  connect(fBond3.fBondCon1, J1p4_3.BondCon2);
  connect(eBond7.eBondCon1, J1p4_3.BondCon4);
  connect(J1p4_3.BondCon3, eBond8.fBondCon1);
  connect(J1p4_3.BondCon1, Bond1.BondCon1);
  connect(Bond1.BondCon2, Dq1.BondCon1);
  connect(J1p4_1.BondCon4, eBond3.fBondCon1);
  connect(eBond3.eBondCon1, I1.BondCon1);
  connect(J1p4_1.BondCon2, fBond4.eBondCon1);
  connect(fBond4.fBondCon1, GY1.BondCon1);
  connect(GY1.BondCon2, eBond1.fBondCon1);
  connect(eBond1.eBondCon1, J1p4_2.BondCon1);
  connect(J1p4_2.BondCon4, eBond4.fBondCon1);
  connect(eBond4.eBondCon1, I2.BondCon1);
  connect(tableSe1.BondCon1, fBond1.eBondCon1);
  connect(tableSe2.BondCon1, fBond3.eBondCon1);
end HoistingDeviceHABG;
```

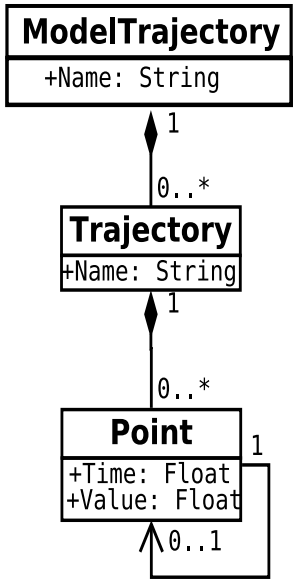


Figure 1.28: Meta-model for the Trajectory Language

1.6 Trajectory Language

We see the behavior of an Engineered Physical System as a model of it. The Trajectory modelling language is specified to represent the evolution of the state of a model. A Trajectory model is obtained by solving a set of DAE or ODE that mathematically represent the model of the EPS. The system of DAE is solved using a solver such as DASSL [AP98]. The result of solving the system is the trajectory of the state variables in the model with respect to time.

A Trajectory model consists of several trajectories. Each tracjectory consists of several state points or just points. A point is associated with a time stamp and a value. We present the meta-model for the Trajectory modelling language (Figure 1.28) to represent the syntax of Trajectory models. The ModelTrajectory class has a Name attribute which is of type String. It is the container class for 0..* Trajectory objects. Each Trajectory object has a Name attribute which is the name of a state variable in the system model. The range of values a state variable or a Trajectory object takes is a collection of 0..* Point objects. Each Point object consists of a Value attribute which is of type Float and its Time of occurence.

The concrete syntax for a Trajectory model is a set of plots. Two important plots for hoisting device example are shown in Figure 1.29. The voltage applied to the hoisting device is shown in Figure 1.29 (b). The voltage is zero until 50 seconds and then it is brought up to 110 V for the next 50 seconds. A mass of 500 kg is being lifted by the hoisting device. It is assumed to be laying on the ground. The reactive force from the ground stops the mass from plunging into the earth. Therefore, a downward force comes into play as soon as the hoisting device attempts to lift the mass off the ground. This occurs at 50 seconds. At the end of 100 seconds the hoisting device lifts the mass to a height of around 15 meters as shown in Figure 1.29 (a). The parameters for all the components in the hoisting device are given in Table 1.9.

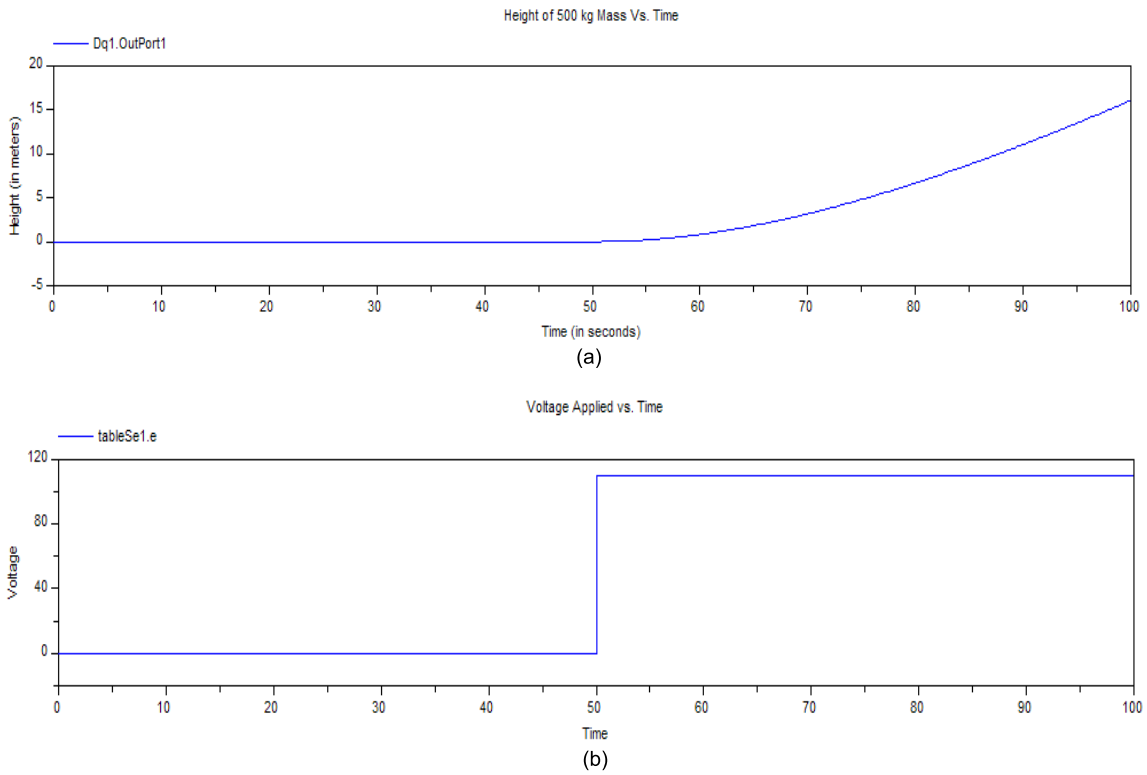


Figure 1.29: Trajectory Model of the Hoisting Device

Table 1.9: Hoisting Device Parameters

Component Name	**Parameter**	**Value**	**Unit**
tableSe1	table	$[0, 0; 50, 0; 50, 110; 100, 110]$	Volts
tableSe1	startTime	0	Seconds
tableSe1	offset	0	Volts
tableSe2	table	$[0, 0; 50, 0; 50, -4900; 100; -4900]$	Newton
$R1$	R	0.5	Ω
$I1$	I	0.05	Henry
GY1	r	3	*NA*
$I2$	I	667	Nm/rad
$R2$	R	1	Nms/rad
TF1	m	0.11	*NA*
$I3$	I	500	kg

The Trajectory model is the final model a modeller can observe for a system. It is the behavior of the EPS. Therefore, this model can also be regarded as the true semantics of the models described in all the modelling languages in the previous sections.

2

Model Transformations

2.1 Introduction

In the previous chapter we introduced several modelling languages that specify the syntax for representing an EPS at different abstraction levels. In this chapter we specify the *semantics* for each of the modelling languages. The specification of the semantics along with the syntax of a modelling language makes it a *modelling formalism*.

Most of the modelling languages described earlier are visual modelling languages. Models that are an instance of a visual language have a concrete visual syntax and an abstract syntax graph representation as we have seen for the hoisting device example. The graph data structure for the abstract syntax of a visual model makes it viable to the application of *graph rewriting* [Aga03] [HER99] for model transformation. We use the Himesis sub-graph matching kernel [Pro05] to facilitate graph rewriting.

Graph rewriting on a model is performed by an ordered set of Graph Grammar (GG) rules. A GG rule consists of an LHS graph and an RHS graph. The rule is applied on a *host graph* which is the abstract syntax graph of the current model. Sub-graph matching between the LHS graph and the host graph is performed. The result of matching is a set of matching nodes in the host graph that correspond the nodes of the LHS graph. These matching nodes have the same label (or type) as the that of the LHS graph pattern. The matched nodes can be further checked for some properties before application of the rule. A *pre-condition* is a truth statement about some properties in the match. If it is true then the rule is applied. As a result of the application the nodes in the RHS effect the change in the model. Some properties of graph node values can also be set as specifications or they may simply be copied from the LHS. The execution of the set of rules can either be programmed or be executed in a sequence.

To better understand the specification of a GG rule let us look back at the HFSM modelling language example. In Figure 2.1 we present an example rule that can be applied to HFSM model. The abstract syntax, concrete syntax, and the adjoining textually expressed parts of the rule are shown. The rule createObjectOfType_State creates a new object and adds it to the top-level model graph. The LHS of the rule contains a pattern with label HFSMlabelgraph. The sub-graph matching algorithm in Himesis looks for a node with label HFSMlabelgraph and returns a set of matches. Each match is tuple with two elements. The first element is the LHS node and the second element is the matching node in the host graph. An object of type State is now added to the host graph (or abstract syntax graph of the model) with a unique name and default values for its attributes. The specification of the default values are shown as textual specifications.

The denotational semantics of the visual modelling language HLPSM is given by transforming

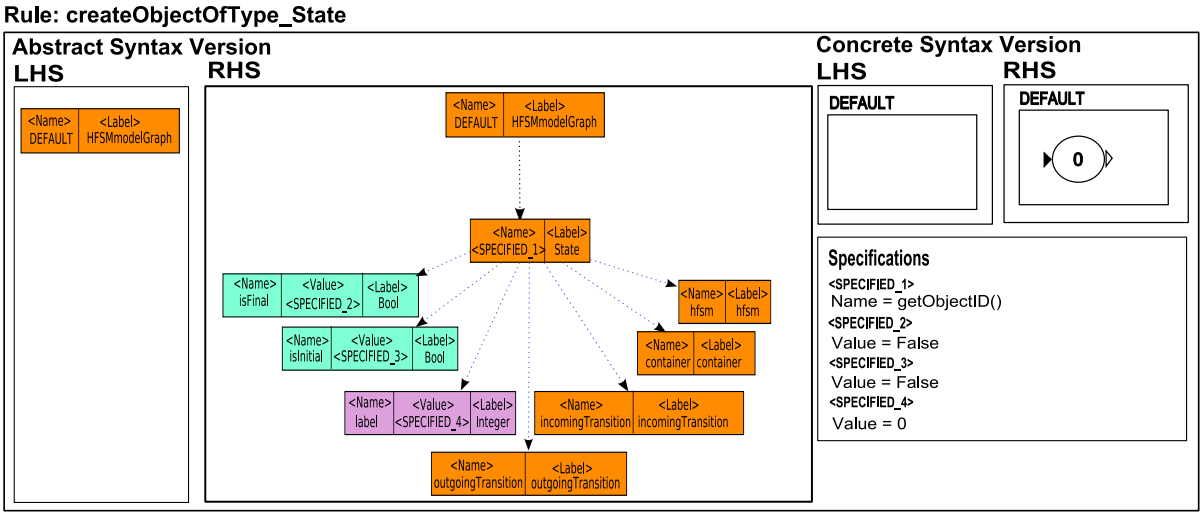


Figure 2.1: Graph Grammar Rules for HFSM

an HLPSM model to the IPM modelling language. The GG rule set MT_HLPSM_2_IPM for this transformation is given in Section 2.2. The denotational semantics for the IPM modelling language is obtained by a transformation to the HBG modelling language. Therefore, the transformation MT_IPM_2_HABG (presented in Section 2.3) transforms the IPM to an acausal HBG or HABG. The computational direction for evaluating the efforts and flows in the HABG is obtained by applying the transformation MT_HABG_2_HCBG (in Section 2.4) which performs causality assignment on a HABG model. The set of rules mentioned so far are based on graph rewriting.

The next transformation from HCBG to Modelica is a graph traversal algorithm. This transformation MT_HCBG_2_Modelica is presented in Section 2.5. The conversion of Modelica code to efficient C code that implements a DAE solver and its execution/simulation is discussed in the transformation MT_Modelica_2_Trajectory given in Section 2.6.

During formalism transformation we have a model in a source formalism that is transformed to a model in a target formalism. The model elements in the source formalism could be related to each other. The application of a GG rule may introduce the counterpart model element from the target formalism for a model element in the source formalism. Removing the source formalism model elements at this stage will destroy all its relationships and hence we have no way to find out what it connects to. Moreover, the rule may be applied to several source formalism model elements. To precisely identify which source formalism element was connected to which target formalism element we introduce a special model element called a GenericLink. In most transformations this link is described by a dotted purple line.

The GenericLink connects model elements in the source and target formalism. Consider the situation, a source model element A is connected to another model element B in the source formalism. Let X be the model element in the target formalism that ultimately replaces A and is connected to A via a GenericLink. The replacement for B in the target formalism is Y and Y needs to be related to X. A pattern that associates A to X has already been created using a GenericLink, hence we can easily formulate a rule that finds the X that corresponds to A and can be associated with Y that is the counterpart for B. Therefore, a GenericLink is a special model element that is used during formalism transformation and does not exist otherwise when

Table 2.1: Graph Grammar rules in execution order for MT_HLPSM_2_IPM

Order	Rule Name	Description
1	Mains_2_IPM	HLPSM of electrical mains is transformed to IPM electrical circuit with resistance, capacitance, voltage source, and motor
2	Motor_2_IPM	HLPSM of a motor is transformed to IPM components of the rotational mechanical domain with rotational inertance and damping and connected to the motor.
3	CableDrum_2_IPM	HLPSM of the cable drum is transformed to an IPM pulley and connected to the rotational inertance
4	Load_2_IPM	HLPSM load is transformed to IPM mass which is connected to the pulley
5	delete_HLPSM_Load	HLPSM load is deleted from the graph
6	delete_HLPSM_Cabledrum	HLPSM cabledrum is deleted from the graph
7	delete_HLPSM_Motor	HLPSM motor is deleted from the graph
8	delete_HLPSM_Mains	HLPSM mains is deleted from the graph
9	delete_GenericLink	GenericLinks are deleted from the graph

a model conforms to its modelling language.

As a note, in the rules all the elements have an unique label but labels are shown only for the nodes relevant to the transformation.

2.2 High-level Physical System Model to Idealized Physical Model

We now present the GG rules used to transform a HLPSM model to an IPM model. The rules are presented in concrete visual syntax with textually expressed parts where necessary. Each rule is executed according to an execution order. If one rule is executed the next rule to be checked is the first rule in the order. Hence, after executing every matching rule the execution starts from the first rule. The order prescribed is one of the many possible orders of execution. Some rules can be executed in parallel.

The rules are shown in Figures 2.2 and 2.3. The list of rules, their execution order and short descriptions are given in Table 2.1.

2.3 Idealized Physical Model to Hybrid Acausal Bond Graph

The Idealized Physical Model (IPM) is obtained by applying the transformation MT_HLPSM_2_IPM to the abstract syntax graph of the HLPSM model. The next step is to obtain the Hybrid Bond Graph model for the IPM. A step by step process to perform this transformation is textually described in [Bro99]. We present the GG rules to perform the transformation in Figures 2.4, 2.5, 2.6, 2.7, 2.8, 2.9. The rules for simplifying the structure of the obtained BG are given in Figures 2.10, 2.11.

The rules for transforming IPM to HABG are executed in the order described in Table 2.2. The rules for simplifying the structure of a HABG are described in Table 2.3. The order is not unique. Depending on the independence of one rule with respect to another the rules can either be executed in a different order or even in parallel.

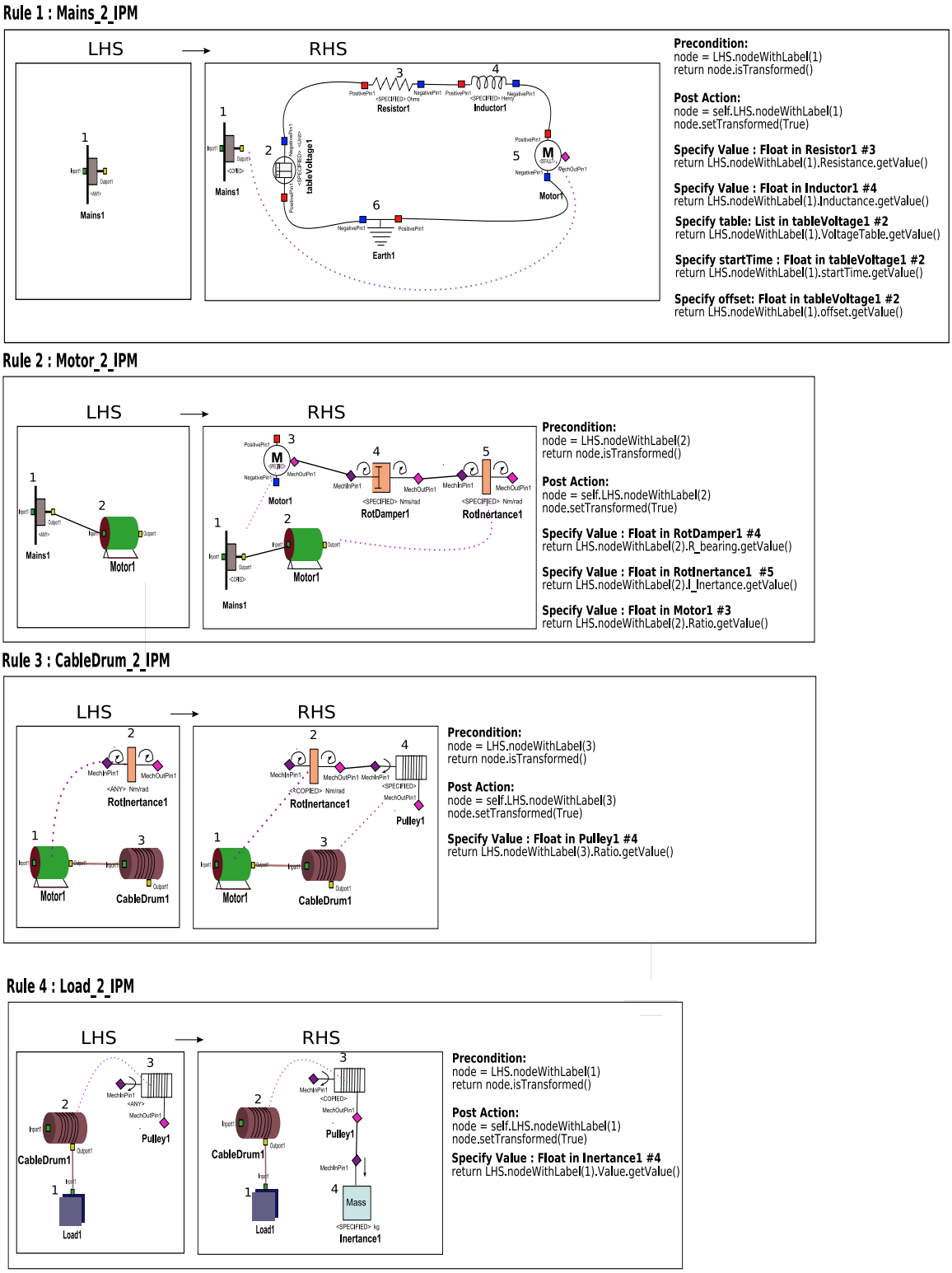


Figure 2.2: Model Transformation HLPSM to IPM: Rules 1-4

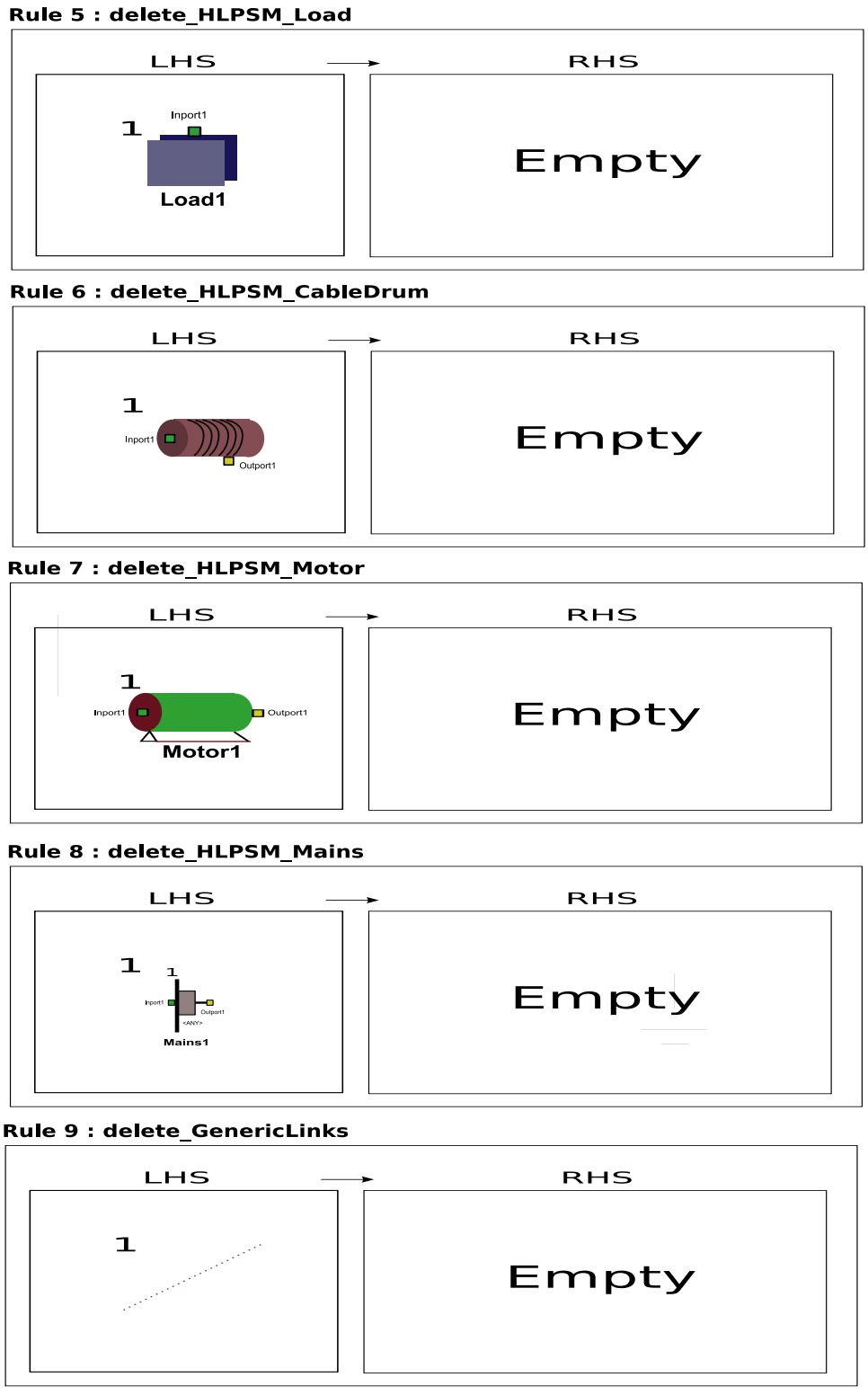


Figure 2.3: Model Transformation HLPSM to IPM: Rules 5-9

Table 2.2: Graph Grammar rules in execution order for MT_IPM_2_HABG

Order	Rule Name	Description
1	identify_efforts_E_2_E	IPM electrical link is converted to a 0-junction
2	identify_efforts_M_2_M	IPM mechanical link is converted to a 1-junction
3	effort_differences_Resistor	An R element is inserted between junctions at the ends of an IPM resistor
4	effort_differences_Capacitor	A C element is inserted between junctions at the ends of an IPM capacitor
5	effort_differences_Inductor	An I element is inserted between junctions at the ends of an IPM inductor
6	effort_differences_tableVoltage	A tableSe element is inserted between junctions at the ends of an IPM table voltage
7	flow_differences_RotDamper	An R element is inserted between junctions at the ends of an IPM rotational damper
8	flow_differences_RotInertance	An I element is inserted between junctions at the ends of an IPM rotational inertance
9	flow_differences_Inertance	An I element is inserted between junctions at the ends of an IPM translational inertance
10	motor_2_GY	A GY element is inserted between junctions at the ends of an IPM motor
11	pulley_2_TF	A TF element is inserted between junctions at the ends of an IPM pulley
12	delete_E_2_E	All IPM E_2_E links are deleted (for all matches)
13	delete_M_2_M	All IPM M_2_M links are deleted (for all matches)
14	delete_Resistor	All IPM Resistor elements are deleted (for all matches)
15	delete_Capacitor	All IPM Capacitor elements are deleted (for all matches)
16	delete_Inductor	All IPM Inductor elements are deleted (for all matches)
17	delete_Voltage	All IPM Voltage elements are deleted (for all matches)
18	delete_RotInertance	All IPM RotInertance elements are deleted (for all matches)
19	delete_RotDamper	All IPM RotDamper elements are deleted (for all matches)
20	delete_Motor	All IPM Motor elements are deleted (for all matches)
21	delete_Pulley	All IPM Pulley elements are deleted (for all matches)
22	delete_Inertance	All IPM Inertance elements are deleted (for all matches)
23	deleteEarth	All IPM Earth elements are deleted (for all matches)
24	delete_GenericLink	All GenericLinks are deleted

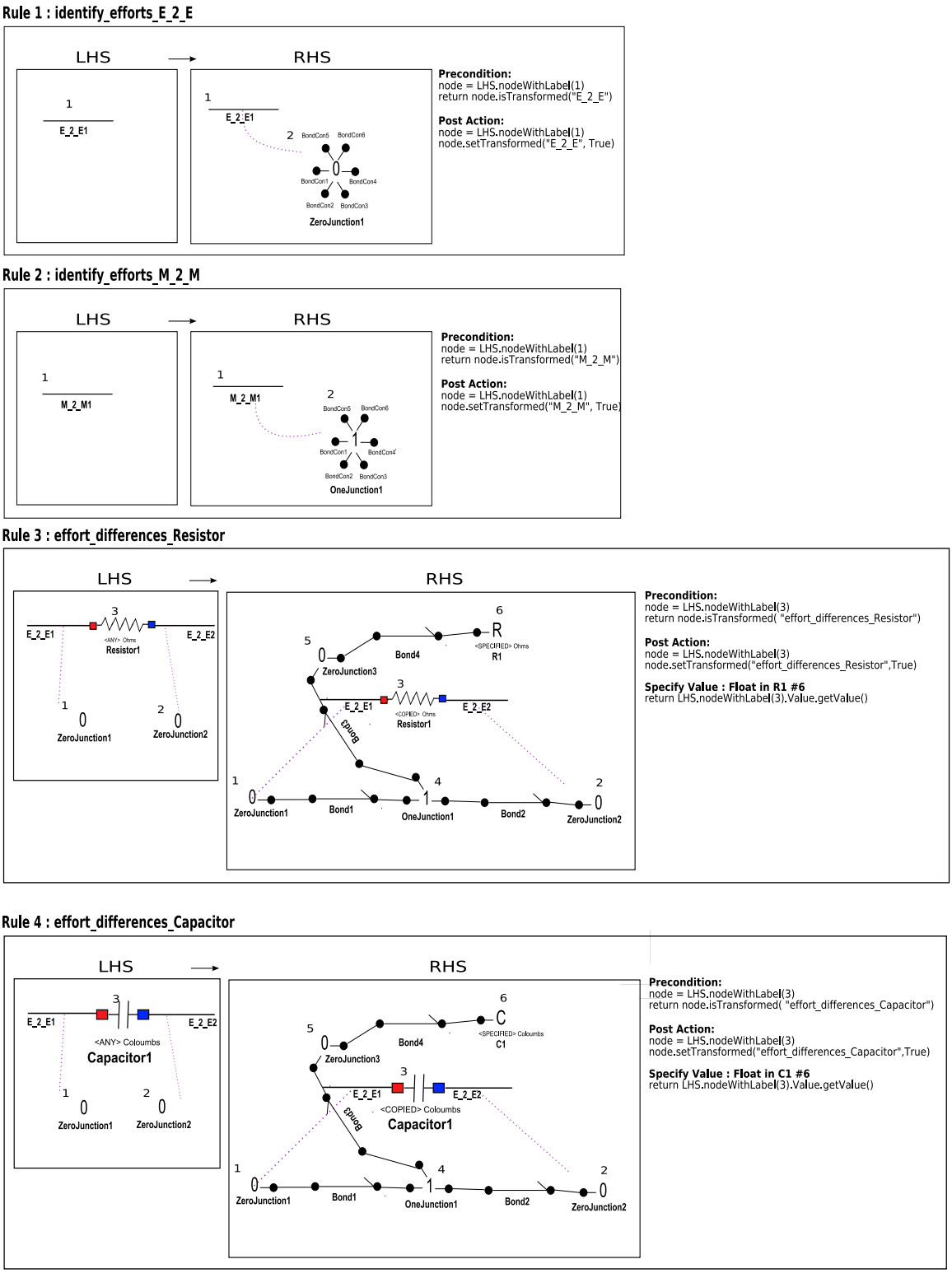


Figure 2.4: Model Transformation IPM to HABG: Rules 1-4

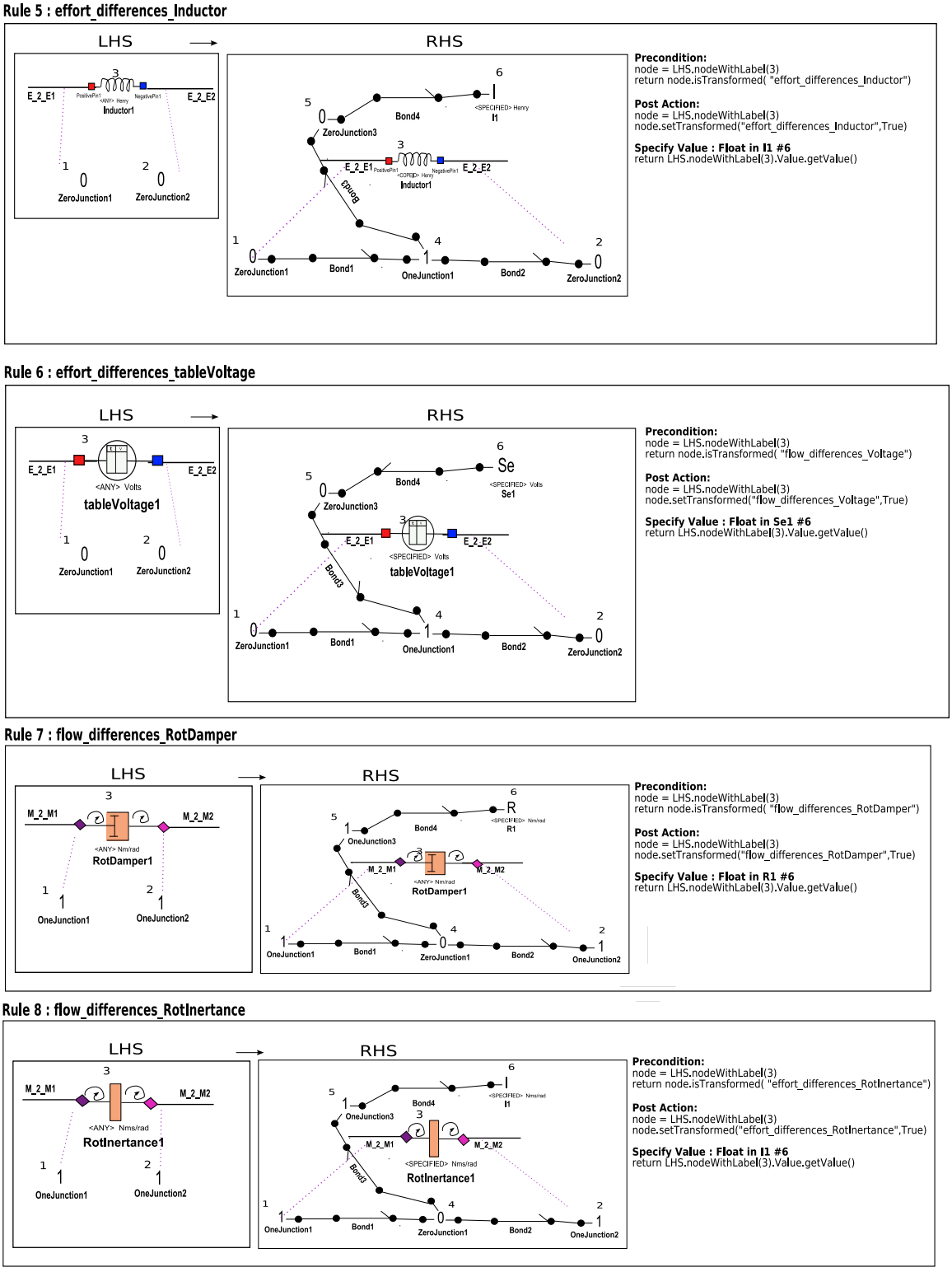
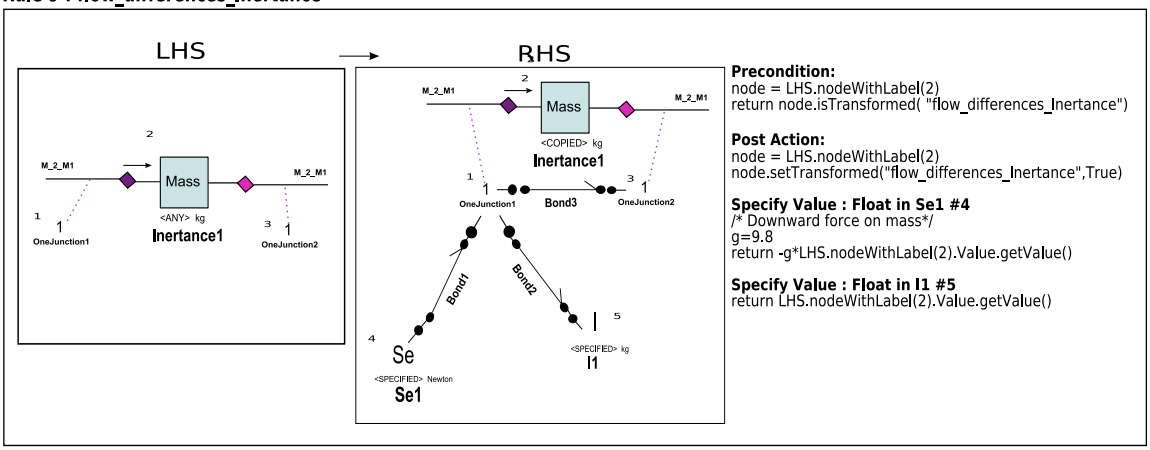
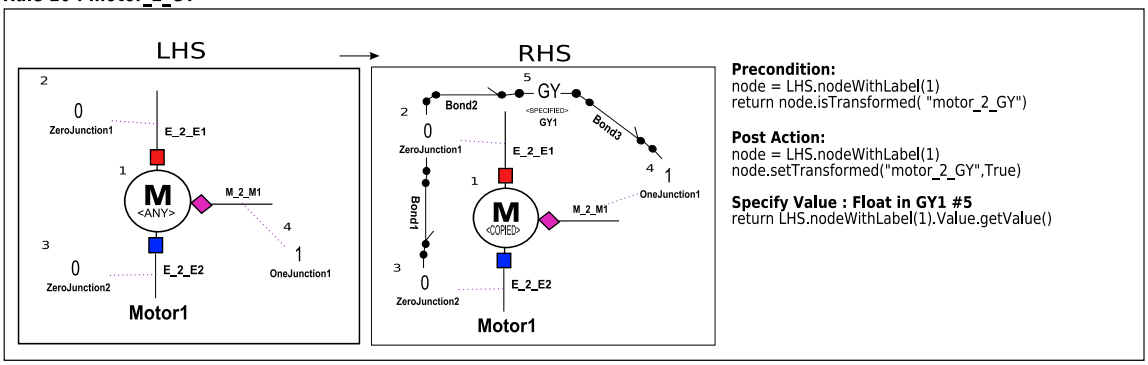


Figure 2.5: Model Transformation IPM to HABG: Rules 5-8

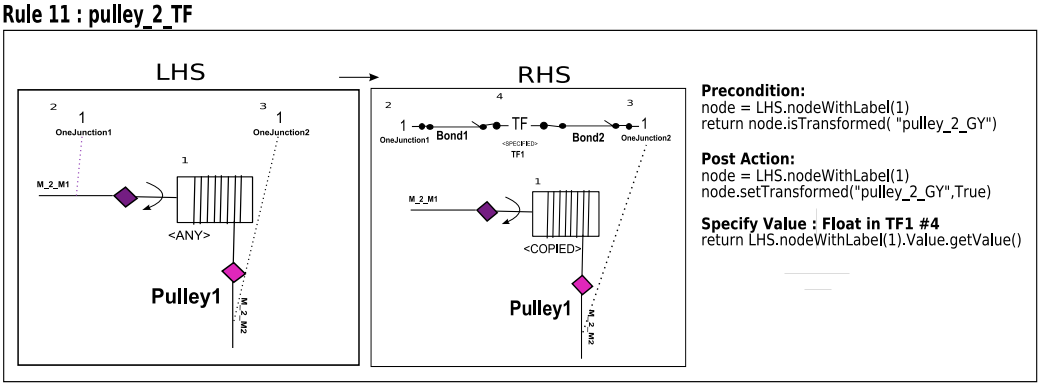
Rule 9 : flow_differences_Inertance



Rule 10 : motor_2_GY



Rule 11 : pulley_2_TF



Rule 12 : delete_E_2_E

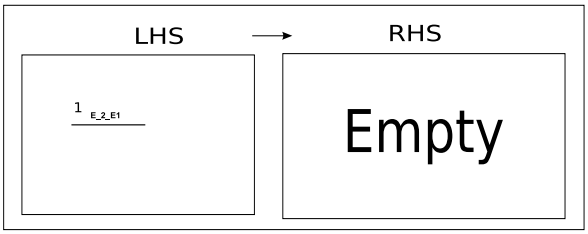


Figure 2.6: Model Transformation IPM to HABG: Rules 9-12

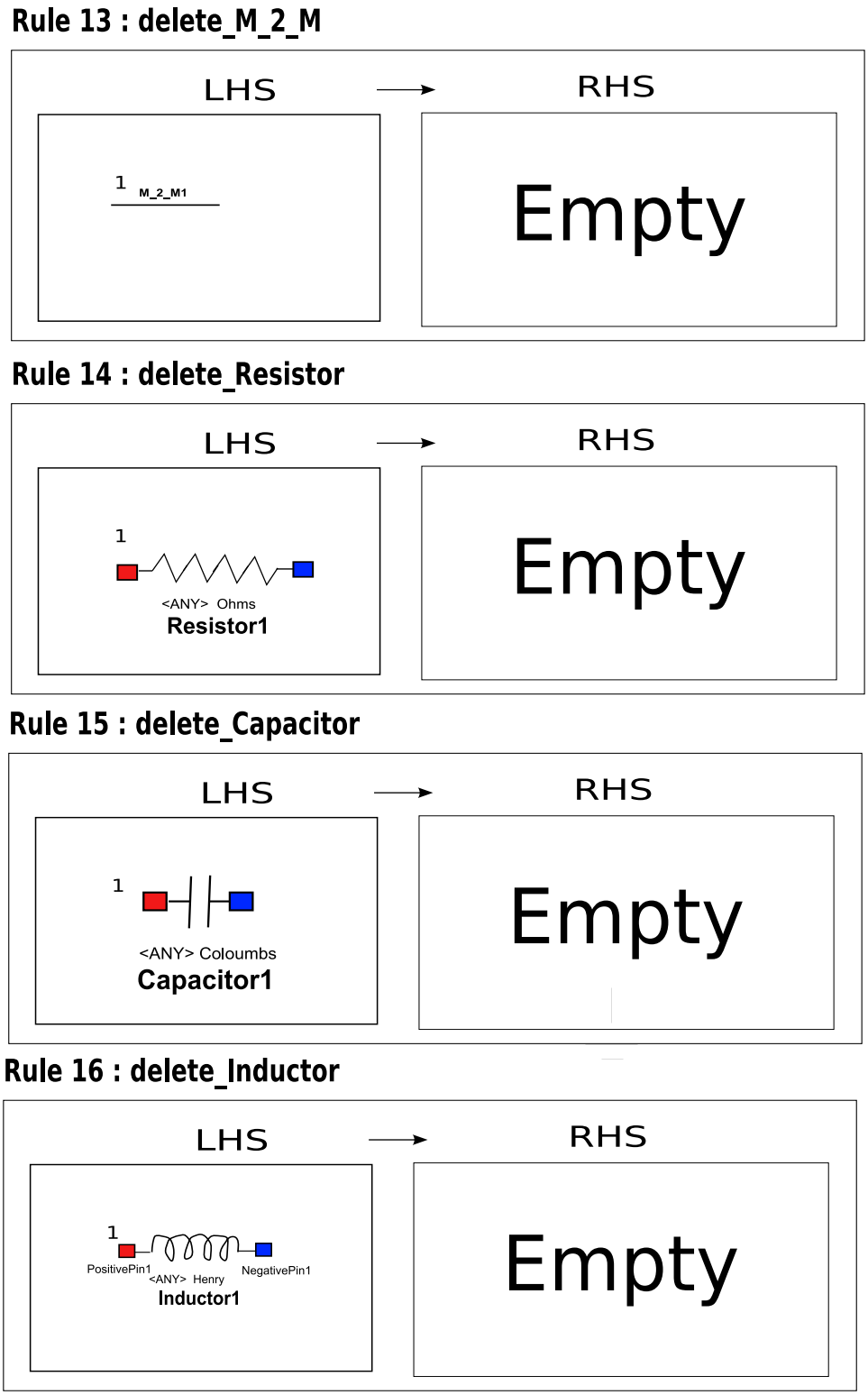


Figure 2.7: Model Transformation IPM to HABG: Rules 13-16

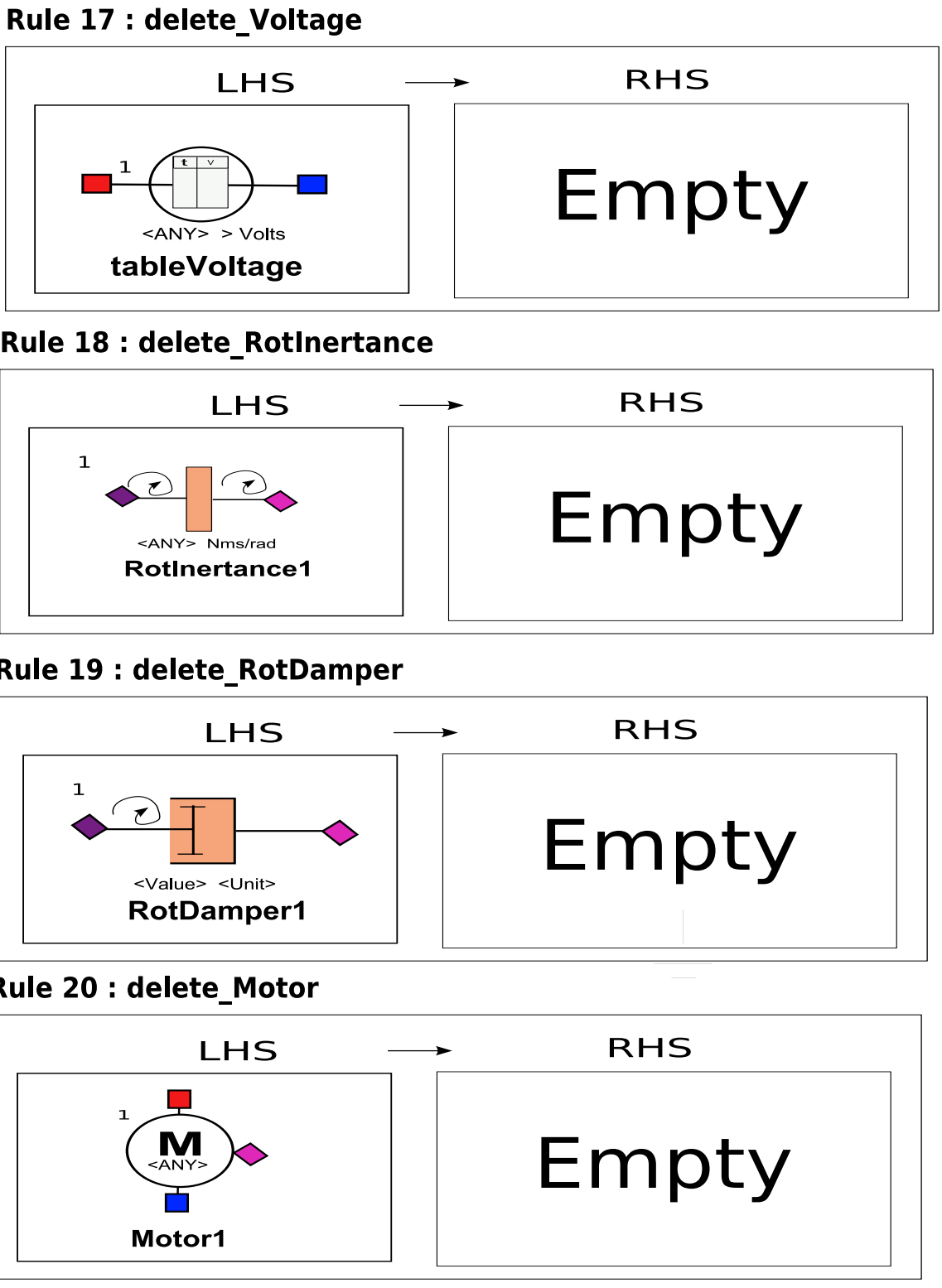


Figure 2.8: Model Transformation IPM to HABG: Rules 17-20

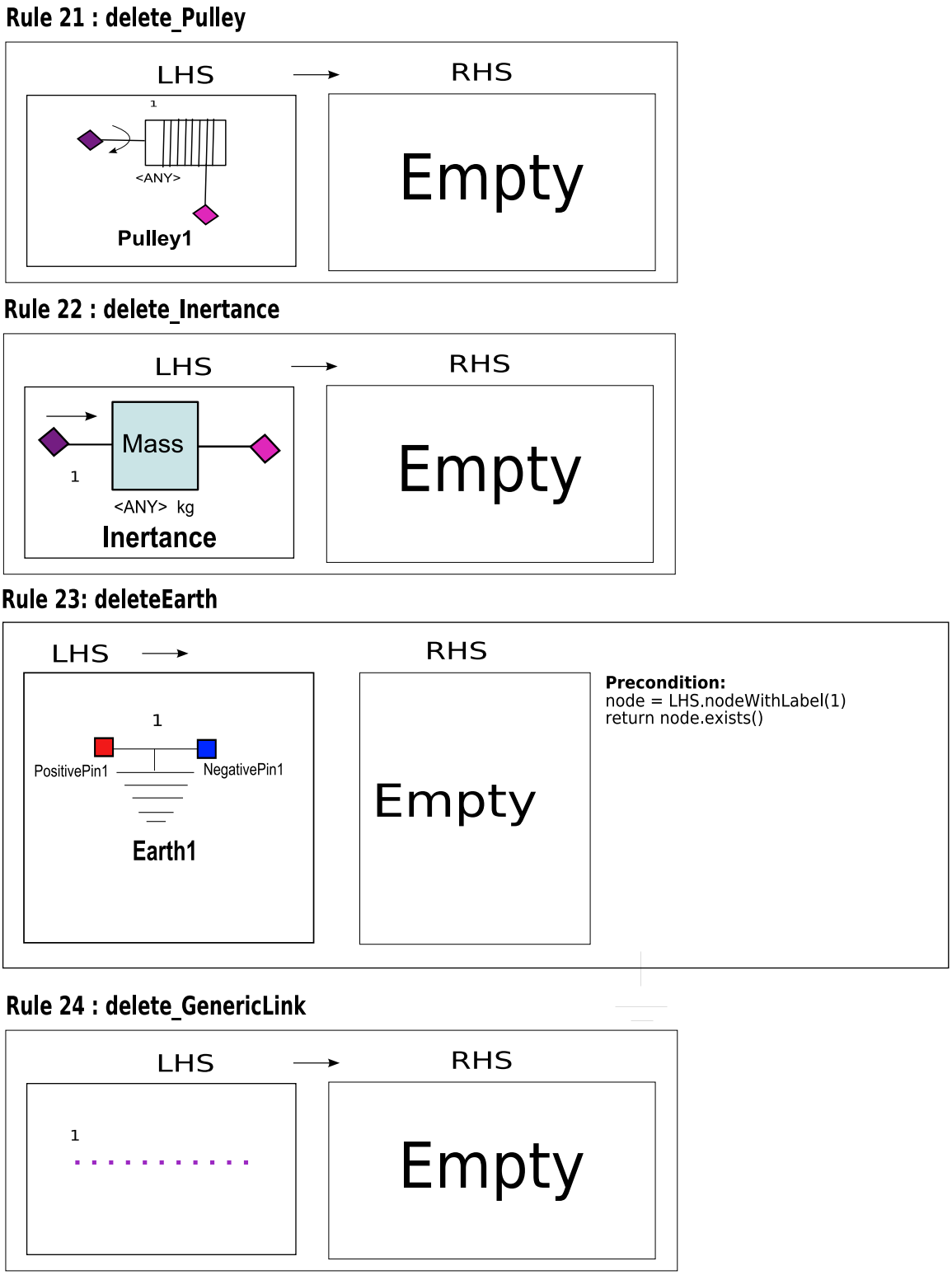
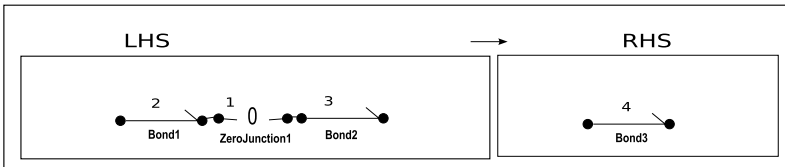


Figure 2.9: Model Transformation IPM to HABG: Rules 21-24

Rule 25: optimize_ZJ

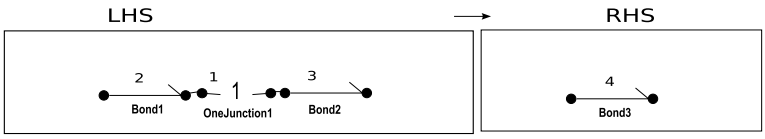


Precondition:
siz_in=len(LHS.nodeWithLabel(1)).in_connections)
siz_out=len(LHS.nodeWithLabel(1)).out_connections)
if siz_in==1 and siz_out==1:
  node = LHS.nodeWithLabel(1)
  return node.isTransformed("optimize_ZJ")
else:
  return False

Action:
Bond1=LHS.nodeWithLabel(2)
Bond2=LHS.nodeWithLabel(3)
Bond3.in_connections=Bond1.in_connections
Bond3.out_connections=Bond2.out_connections

Post Action:
node = LHS.nodeWithLabel(1)
node.setTransformed("optimize_ZJ, True)

Rule 26: optimize_OJ

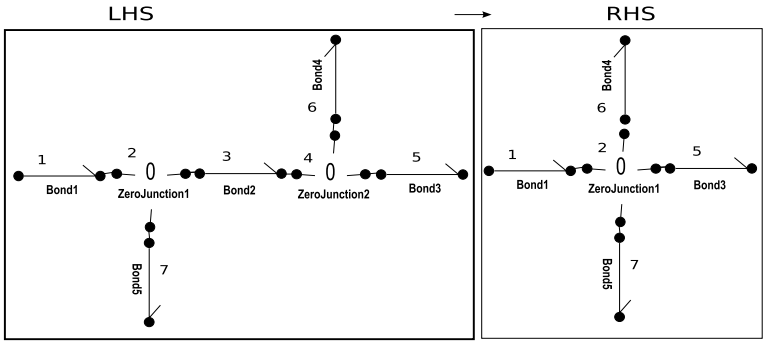


Precondition:
siz_in=len(LHS.nodeWithLabel(1)).in_connections)
siz_out=len(LHS.nodeWithLabel(1)).out_connections)
if siz_in==1 and siz_out==1:
  node = LHS.nodeWithLabel(1)
  return node.isTransformed("optimize_ZJ")
else:
  return False

Action:
Bond1=LHS.nodeWithLabel(2)
Bond2=LHS.nodeWithLabel(3)
Bond3.in_connections=Bond1.in_connections
Bond3.out_connections=Bond2.out_connections

Post Action:
node = LHS.nodeWithLabel(1)
node.setTransformed("optimize_ZJ, True)

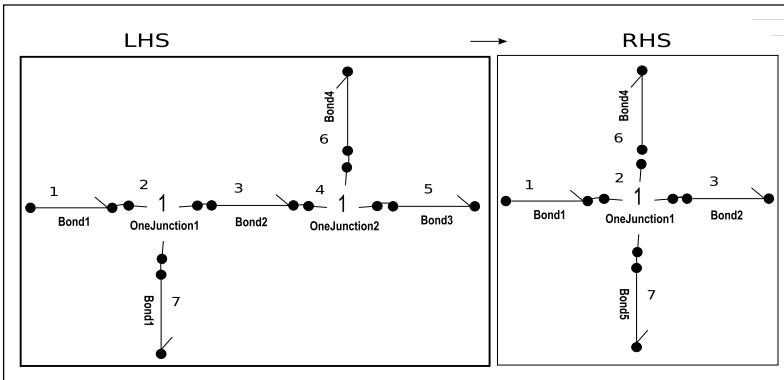
Rule 27: optimize_ZJ_ZJ



Precondition:
siz_in=len(LHS.nodeWithLabel(4)).in_connections)
siz_out=len(LHS.nodeWithLabel(4)).out_connections)
if siz_in==1 and siz_out==2:
  node = LHS.nodeWithLabel(4)
  return node.isTransformed("optimize_ZJ_ZJ")
else:
  return False

Post Action:
node = LHS.nodeWithLabel(1)
node.setTransformed("optimize_ZJ_ZJ, True)

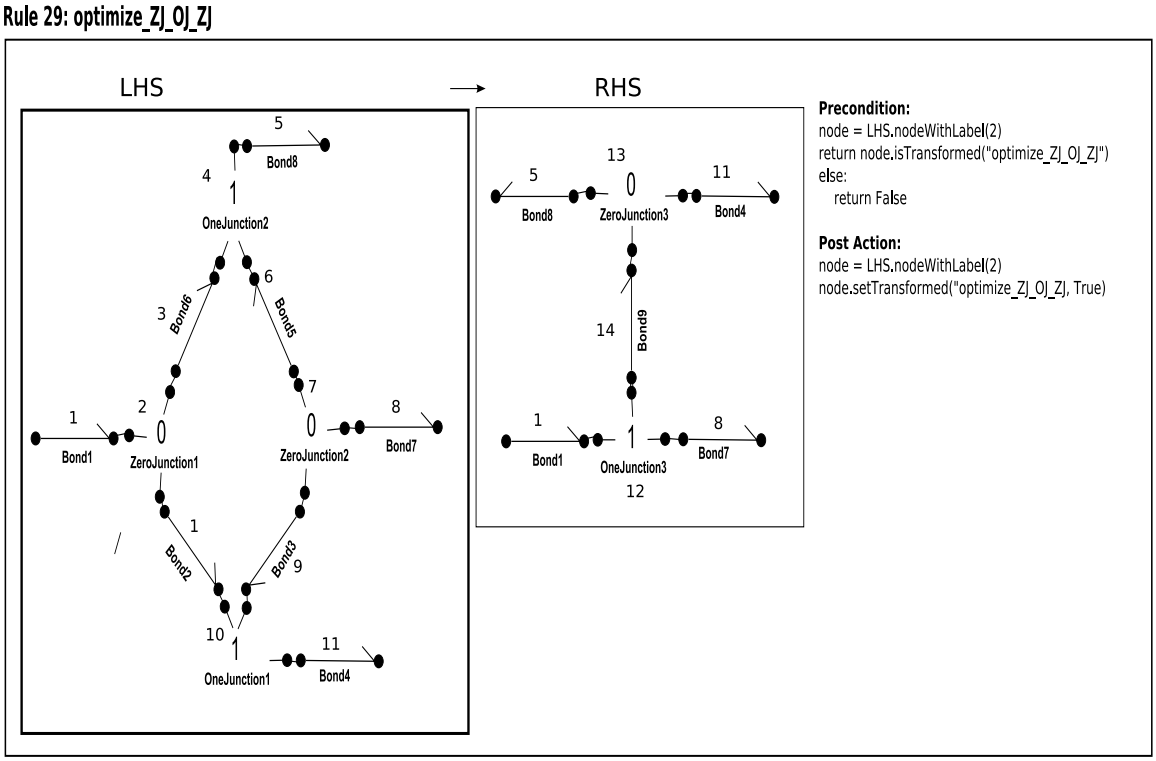
Rule 28: optimize_OJ_OJ



Precondition:
siz_in=len(LHS.nodeWithLabel(4)).in_connections)
siz_out=len(LHS.nodeWithLabel(4)).out_connections)
if siz_in==1 and siz_out==2:
  node = LHS.nodeWithLabel(4)
  return node.isTransformed("optimize_OJ_OJ")
else:
  return False

Post Action:
node = LHS.nodeWithLabel(1)
node.setTransformed("optimize_OJ_OJ, True)

Figure 2.10: Model Transformation IPM to HABG: Rules 22-28



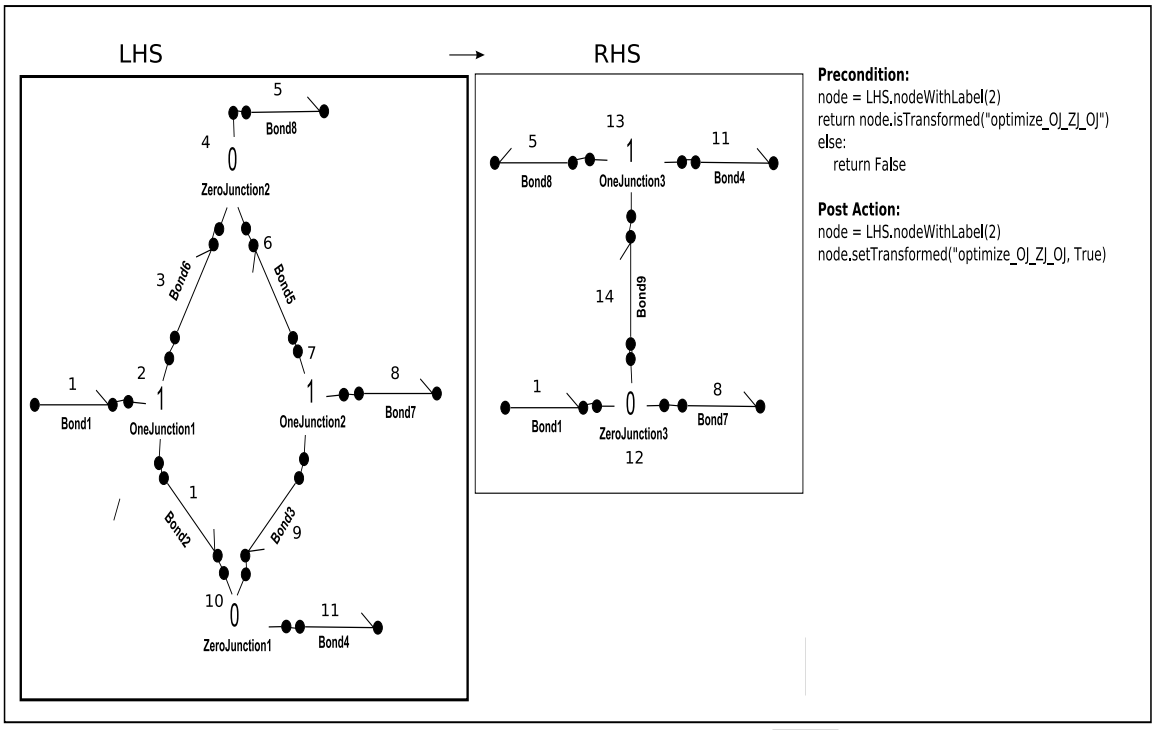


Figure 2.11: Model Transformation IPM to HABG: Rules 29-30

Table 2.3: Graph Grammar rules in execution order for optimizing BG

Order	Rule Name	Description
25	optimize_ZJ	Transform redundant LHS 0-junction pattern to a single 0-junction
26	optimize_OJ	Transform redundant LHS 1-junction pattern to a single 1-junction
27	optimize_ZJ_ZJ	Transform redundant structure on LHS to simplified RHS
28	optimize_OJ_OJ	Transform redundant structure on LHS to simplified RHS
29	optimize_ZJ_OJ_ZJ	Transform redundant structure on LHS to simplified RHS
30	optimize_OJ_ZJ_OJ	Transform redundant structure on LHS to simplified RHS

2.4 Hybrid Acausal Bond Graph to Hybrid Causal Bond Graph

We now discuss the transformation MT_HABG_2_HCBG to perform causality assignment on an acausal BG. We perform causality assignment to given computational direction the bonds in the bond graph. Each bond is either given an effort-out or a flow-out causality.

The Se type effort source elements always have an effort-out causality. The Sf element always has a flow-out causality. The fixed causalities assigned by the sources are propagated by constrained causalities to other bonds. Fixed causalities are shown in Figure 2.12.

The causality assigned by fixed causalities are propagated to the connected junction. The Causality of a junction is set to +1 if the assigned causality is an effort-out causality from an effort source to a 0-junction. The Causality is set to -1 if the assigned causality is a flow-out causality from a flow source to a 1-junction. The fixed causalities are described in Table 2.4.

This causality is further propagated to connected bonds via constrained causalities. Constrained causalities are shown in Figures 2.13, 2.14,2.15,2.16,2.17,2.18, and 2.19. The rule execution order and a short description for each rule is given in Table 2.5.

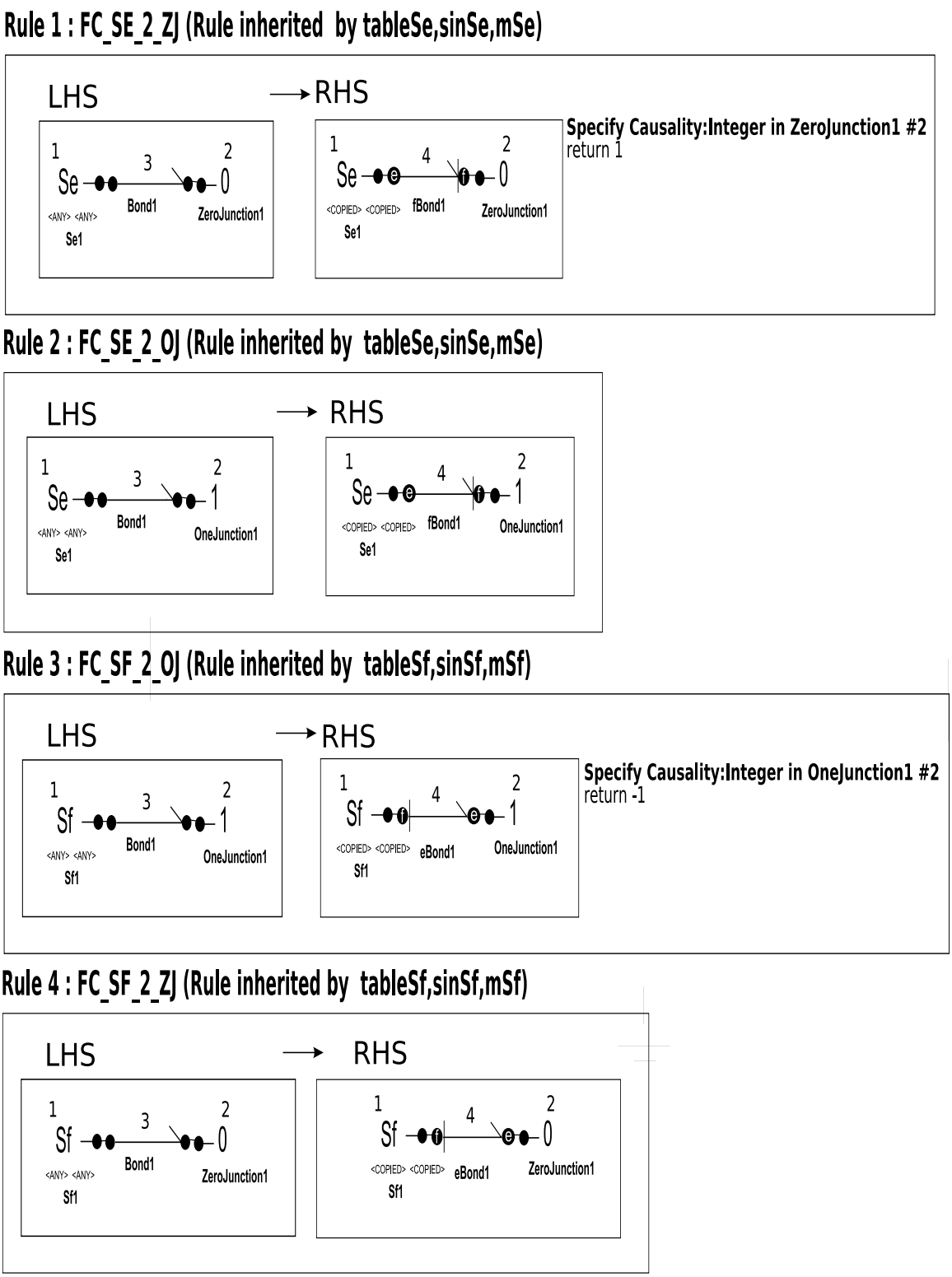


Figure 2.12: Model Transformation HABG to HCBG: Rules 1-4

Table 2.4: Graph Grammar rules for fixed causality in transformation MT_HABG_2_HCBG

Order	Rule Name	Description
1	FC_SE_2_ZJ	Transform redundant 0-junction-¿0-junction to a single 0-junction
2	FC_SE_2_OJ	Transform redundant 1-junction-¿1-junction to a single 1-junction
3	FC_SF_2_OJ	Transform redundant structure on LHS to simplified RHS
4	FC_SF_2_ZJ	Transform redundant structure on LHS to simplified RHS

If none of the constrained causalities match the BG model then a preferred causality is assigned to storage elements. The C element gets an effort-out fixed causality while the I element gets a flow-out fixed causality. The rules to assign the causality to the bonds are shown in Figures 2.19 and 2.20. The execution order of the rule in the graph grammar and a short description for each rule is given in Table 2.6.

The bond to an R element gets indifferent causality (if not constrained causality) which means that it does not matter if flow or effort comes in. The indifferent causalities are shown in Figure 2.20. The execution order for indifferent causality rules and a description are given in Table 2.6.

The execution resumes from the first rule of the GG and continues until no rule is matched. At this point the HABG has been completely assigned causality and is a HCBG. If any of the storage elements that is C or I have got a non-preferred causality due to propagation of causality constraints then we can say that there is a problem with the physical meaningfulness of the model. In other words a capacitor or an inductor element does not store energy in the sense of integration the incoming flow or effort. At this point the modeller has to change the physical model to make it causally correct and physically meaningful.

Table 2.5: Graph Grammar rules for constrained causality propagation in MT_HABG_2_HCBG

Order	Rule Name	Description
5	CC_ZJ_2_R	Effort-out causality assigned if LHS 0-junction causality is 1
6	CC_ZJ_2_C	Effort-out causality assigned if LHS 0-junction causality is 1
7	CC_ZJ_2_I	Effort-out causality assigned if LHS 0-junction causality is 1
8	CC_ZJ_2_ZJ	Effort-out causality assigned if LHS 0-junction causality is 1
9	CC_ZJ_2_OJ	Effort-out causality assigned if LHS 0-junction causality is 1
10	CC_ZJ_2_TF	Effort-out causality assigned if LHS 0-junction causality is 1
11	CC_ZJ_2_GY	Effort-out causality assigned if LHS 0-junction causality is 1
12	CC_OJ_2_R	Flow-out causality assigned if LHS 1-junction causality is -1
13	CC_OJ_2_C	Flow-out causality assigned if LHS 1-junction causality is -1
14	CC_OJ_2_I	Flow-out causality assigned if LHS 1-junction causality is -1
15	CC_OJ_2_OJ	Flow-out causality assigned if LHS 1-junction causality is -1
16	CC_OJ_2_ZJ	Flow-out causality assigned if LHS 1-junction causality is -1
17	CC_OJ_2_TF	Flow-out causality assigned if LHS 1-junction causality is -1
18	CC_OJ_2_GY	Flow-out causality assigned if LHS 1-junction causality is -1
19	CC_J_GY_J_fBond	Flow-out causality assigned from GY input effort-out causality
20	CC_J_GY_J_eBond	Effort-out causality assigned from GY input flow-out causality
21	CC_J_TF_J_fBond	Effort-out causality assigned from TF input effort-out causality
22	CC_J_TF_J_eBond	Flow-out causality assigned from TF input Flow-out causality
23	CC_ZJ_2_De	Effort-out causality assigned if LHS 0-junction causality is 1
24	CC_ZJ_2_Df	Effort-out causality assigned if LHS 0-junction causality is 1
25	CC_ZJ_2_Dp	Effort-out causality assigned if LHS 0-junction causality is 1
26	CC_ZJ_2_Dq	Effort-out causality assigned if LHS 0-junction causality is 1
27	CC_OJ_2_De	Flow-out causality assigned if LHS 1-junction causality is -1
28	CC_OJ_2_Df	Flow-out causality assigned if LHS 1-junction causality is -1
29	CC_OJ_2_Dp	Flow-out causality assigned if LHS 1-junction causality is -1
30	CC_OJ_2_Dq	Flow-out causality assigned if LHS 1-junction causality is -1

Table 2.6: Graph Grammar rules for preferred and indifferent causality in MT_HABG_2_HCBG

Order	Rule Name	Description
31	PC_ZJ_2_C	Preferred effort-out causality assigned to 0-junction to C bond
32	PC_ZJ_2_I	Preferred flow-out causality assigned to 0-junction to I bond
33	PC_OJ_2_C	Preferred effort-out causality assigned to 1-junction to C bond
34	PC_OJ_2_I	Preferred flow-out causality assigned to 1-junction to I bond
35	IC_ZJ_2_R	Indifferent flow-out causality assigned to 0-junction to R bond
36	IC_OJ_2_R	Indifferent flow-out causality assigned to 1-junction to R bond

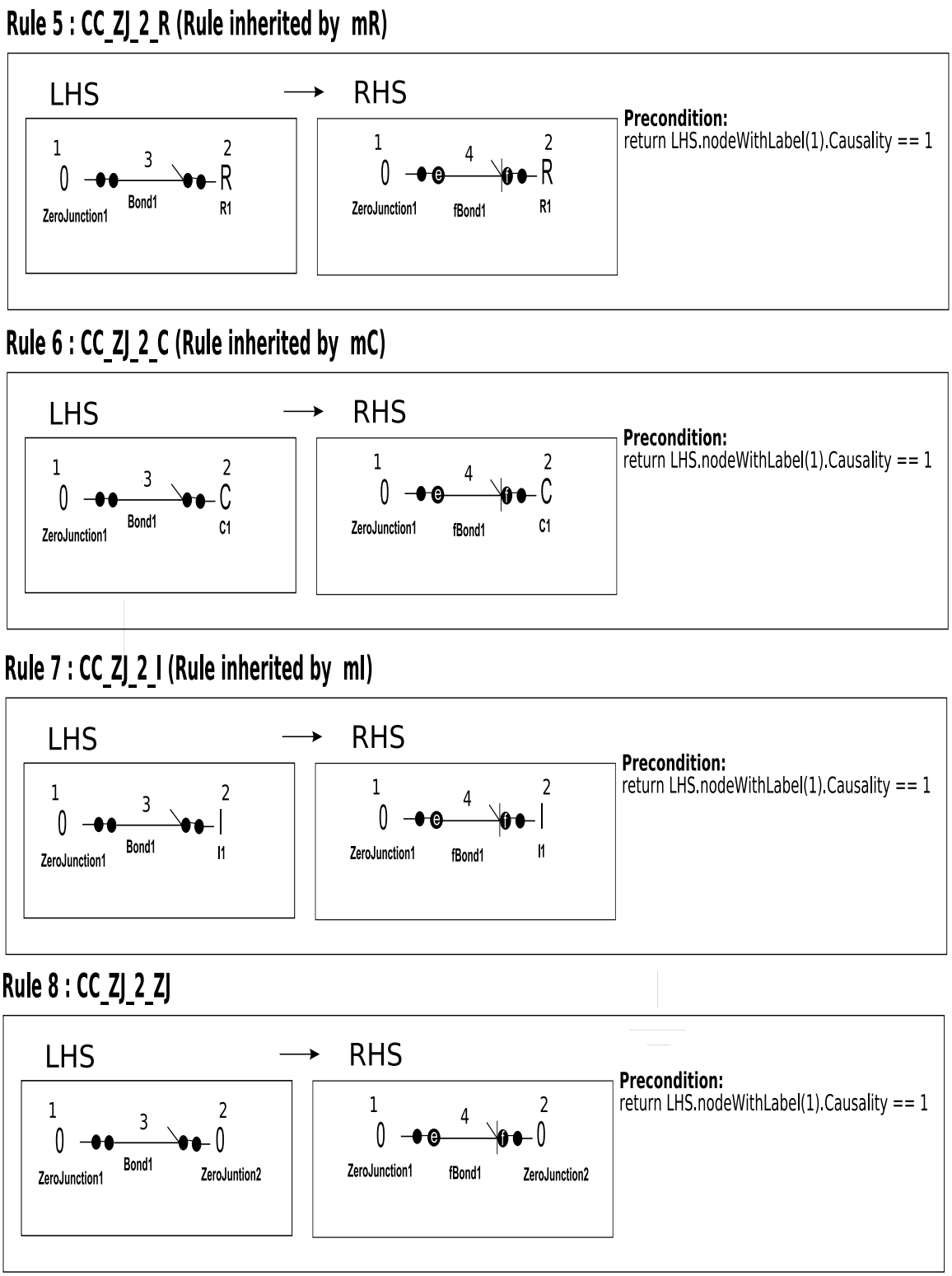


Figure 2.13: Model Transformation HABG to HCBG: Rules 5-8

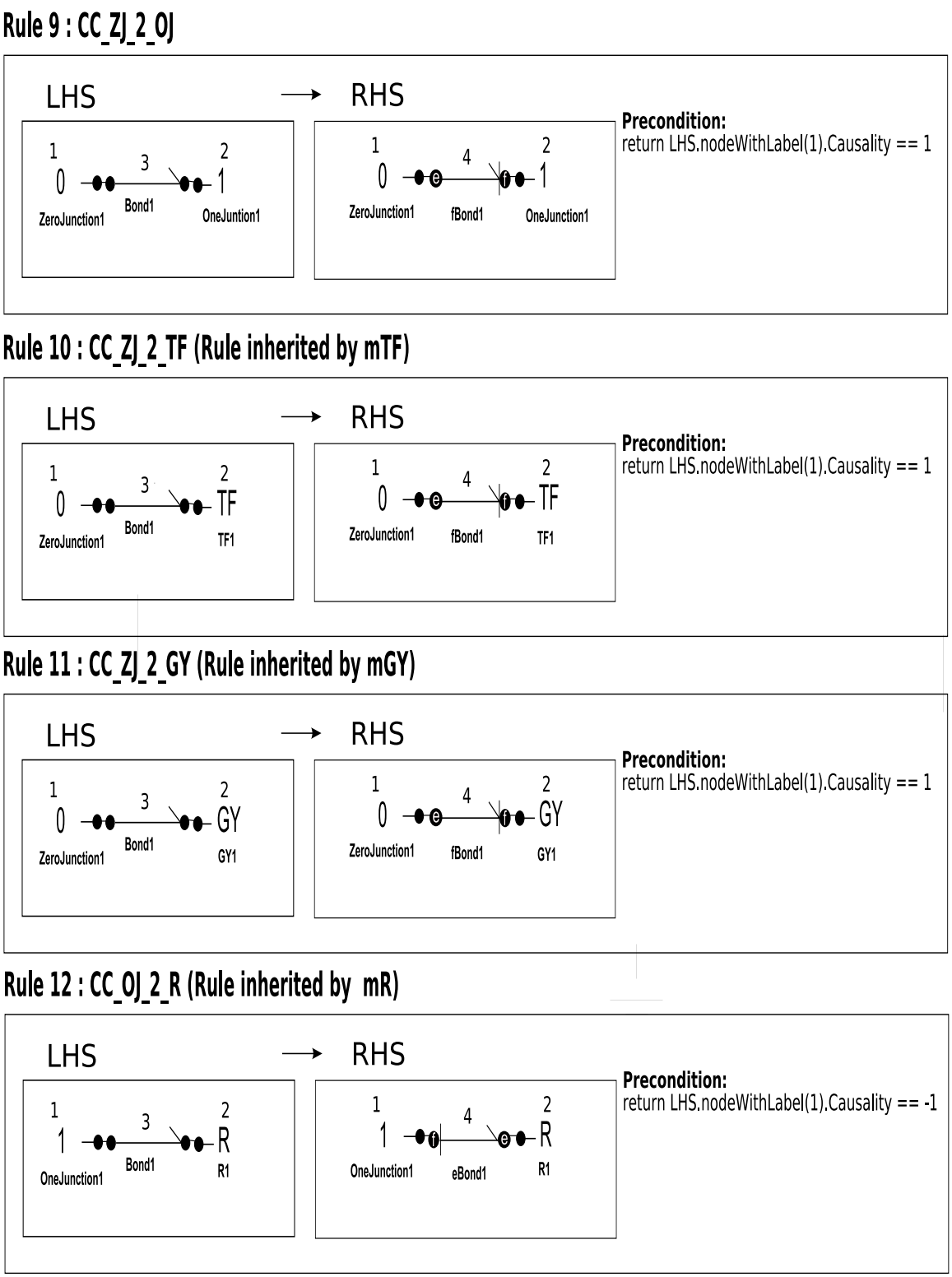


Figure 2.14: Model Transformation HABG to HCBG: Rules 9-12

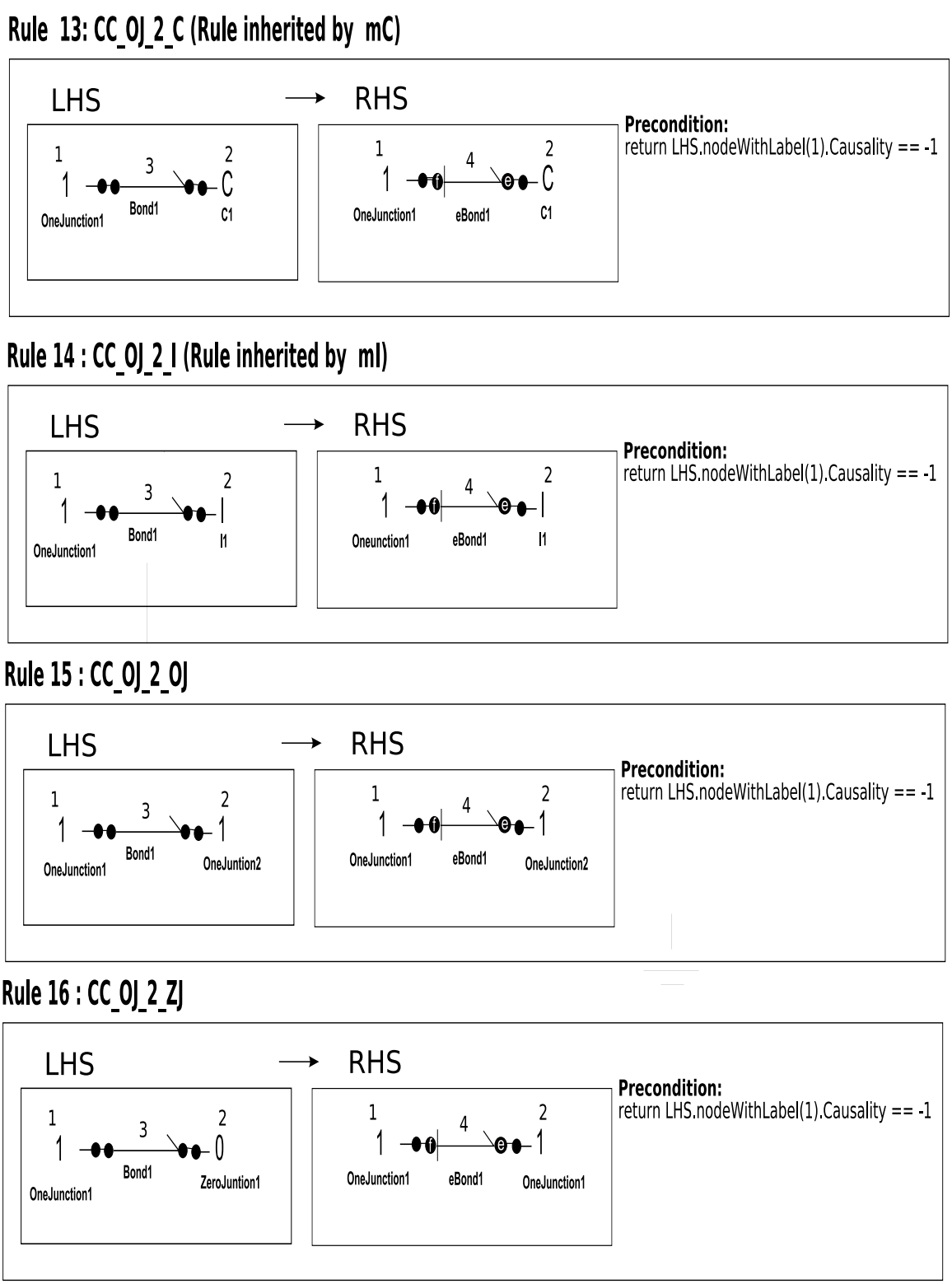


Figure 2.15: Model Transformation HABG to HCBG: Rules 13-16

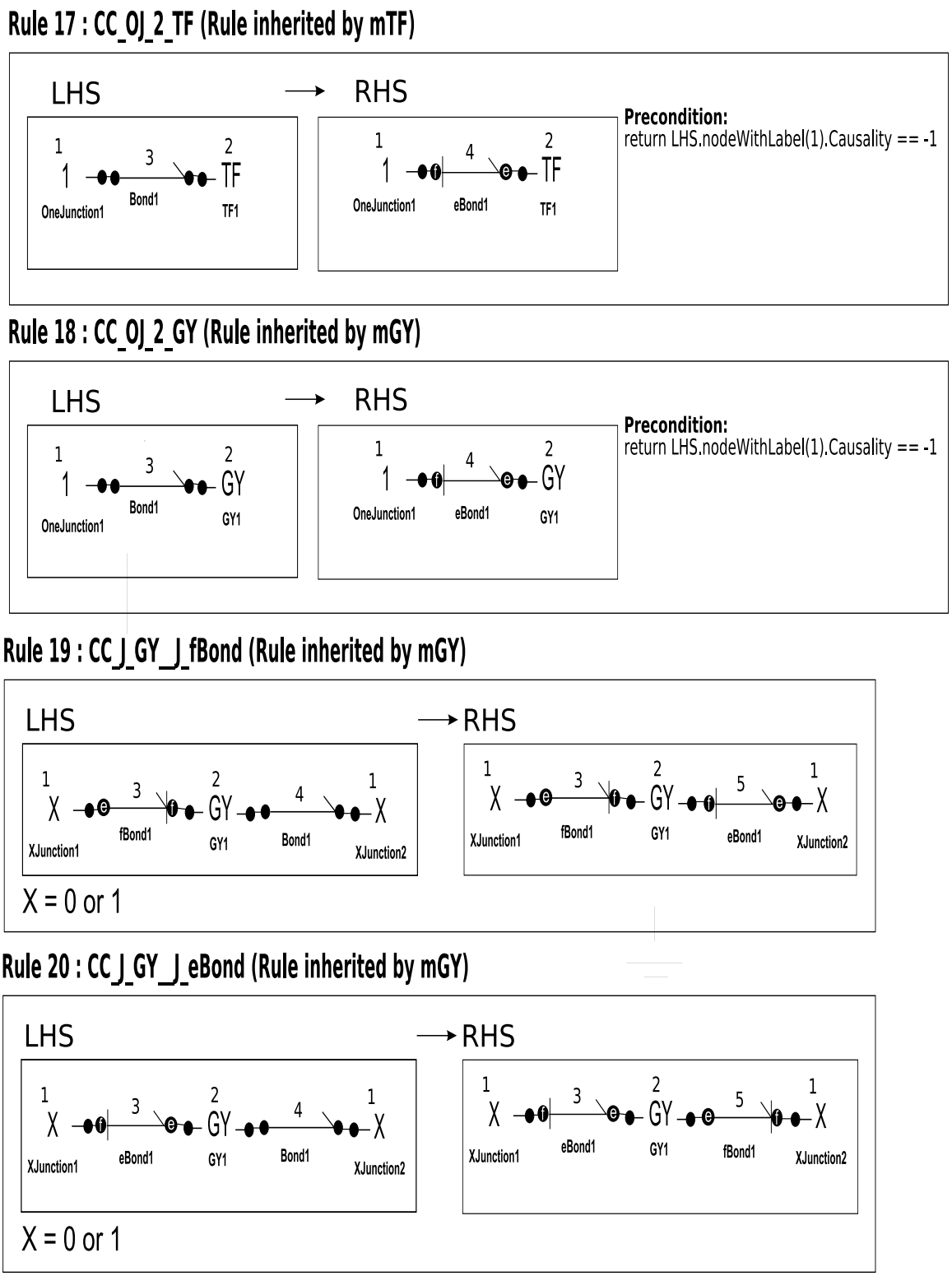


Figure 2.16: Model Transformation HABG to HCBG: Rules 17-20

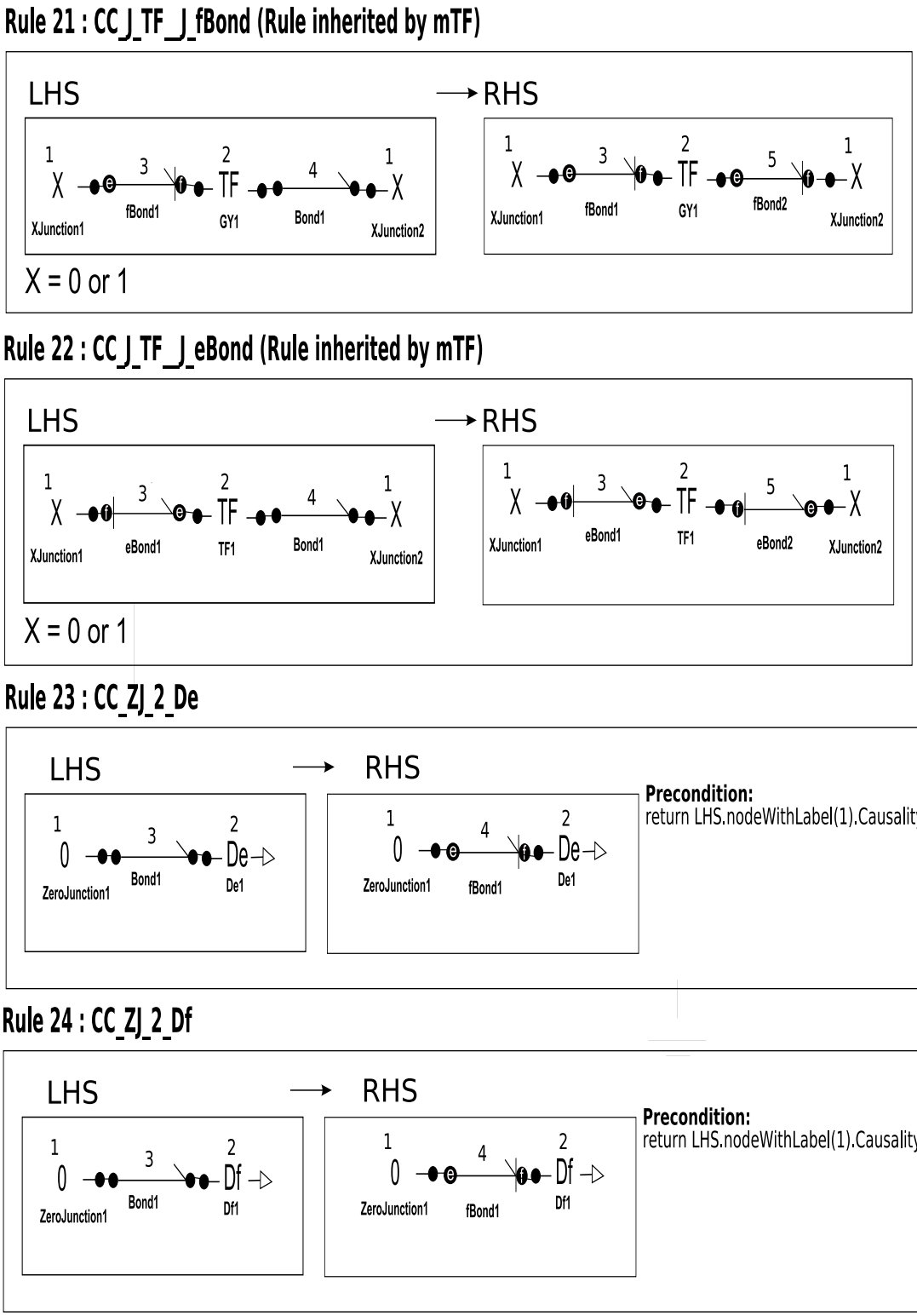
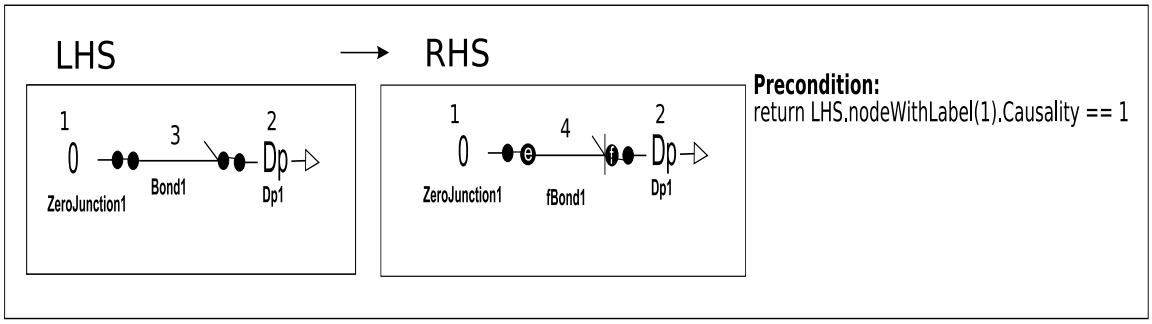
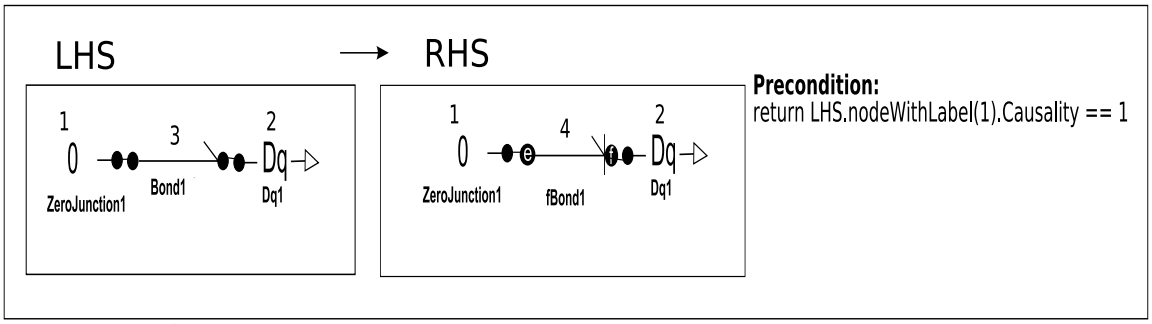


Figure 2.17: Model Transformation HABG to HCBG: Rules 21-24

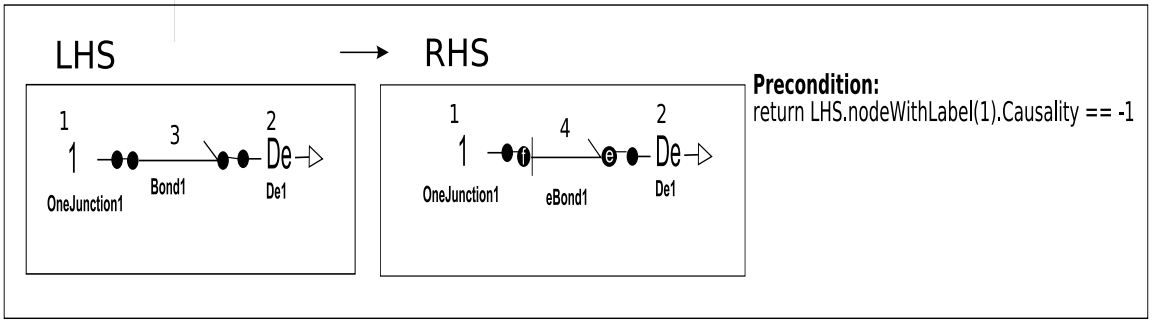
Rule 25 : CC_ZJ_2_Dp



Rule 26 : CC_ZJ_2_Dq



Rule 27 : CC_OJ_2_De



Rule 28 : CC_OJ_2_Df

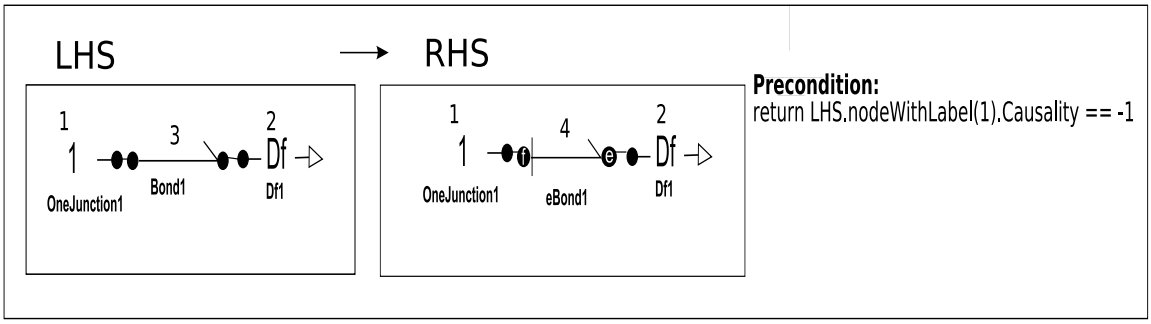


Figure 2.18: Model Transformation HABG to HCBG: Rules 25-28

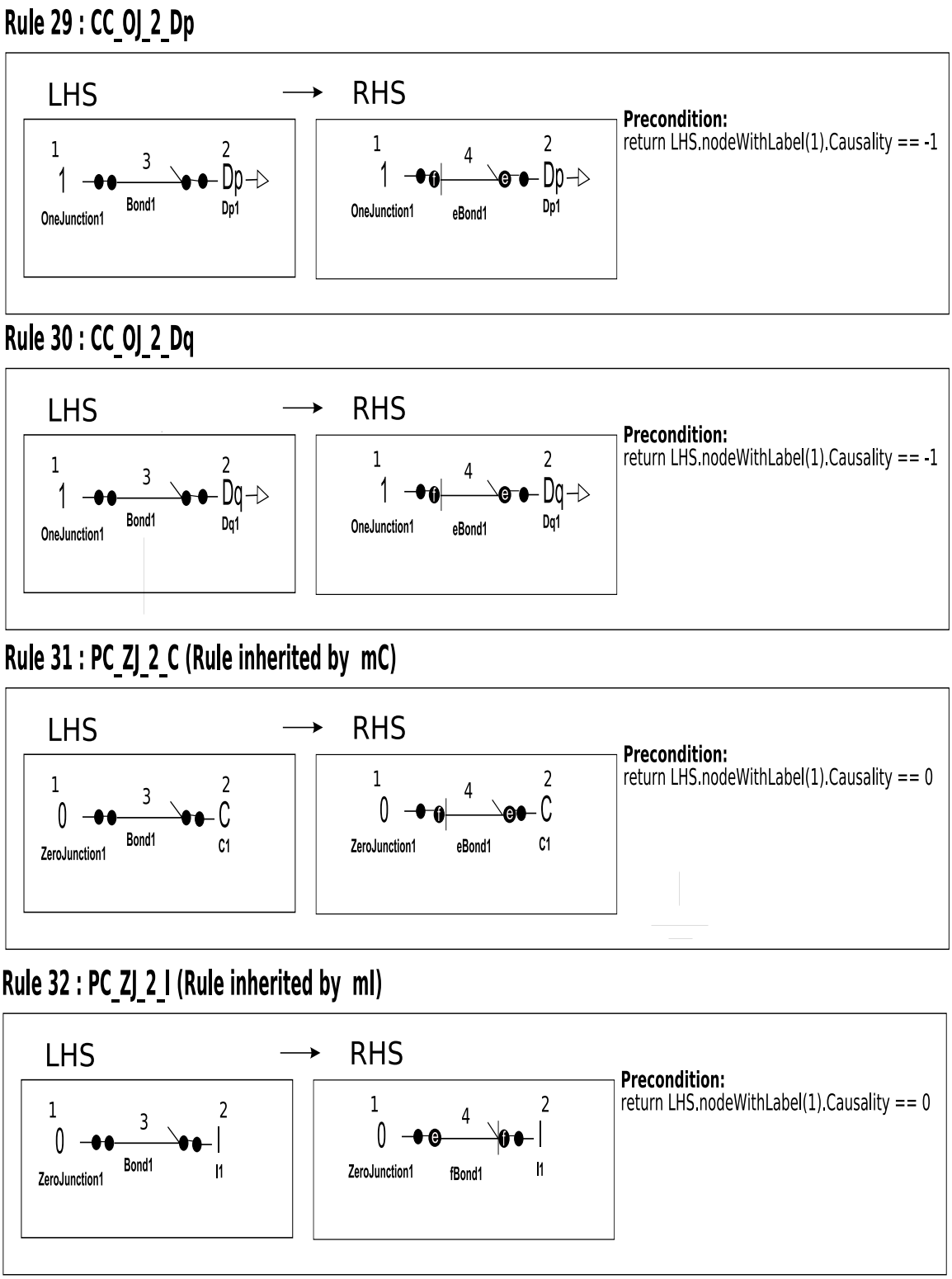


Figure 2.19: Model Transformation HABG to HCBG: Rules 29-32

2.5 Hybrid Causal Bond Graph to Modelica

The HCBG model obtained from the transformation MT_HABG_2_HCBG is now ready to be converted to a set of Differential Algebraic Equations (DAE). Instead of directly transforming a model in the differential equation form we generate Modelica code. Modelica code is an object-oriented textual representation of DAEs.

The main routine, HCBG_2_Modelica, and its helper routines to transform the abstract syntax graph of a HCBG model to Modelica are given below:

```
#Main Routine to transform HCBG to Modelica

def HCBG_2_Modelica(HCBGGraph):
      ModelicaCode="model " + HCBGGraph.getName() + "\n"
      # Define and Initialize the Physical Objects
      ModelicaCode+=getHABGObjects(HCBGGraph) + "\n"
      # Define the Equations of the Physical Model
      ModelicaCode+="equation"
      ModeliaCode+=getConnects(HCBGGraph) + "\n"
      ModelicaCode+="end "+HCBGGraph.getName()+";\n"
      return ModelicaCode

#Routine to obtain physical object  declarations

def getHABGObjects(HCBGGraph):
      Code=" "
      for aNode in HCBGGraph.iterateAll():
           if self.getModelicaObject(aNode)!=None:
              Code+="\t"+getModelicaObject(aNode)+";\n"
      return Code

#Routine to obtain physical object connections

def getConnects(HCBGGraph):
      Code=" "
      for aNode in HCBGGraph.getAllConnTuples():
            connectCode="connect("+aNode[0].Parent.Name+"."+aNode[0].Name+","
+aNode[1].Parent.Name+"."+aNode[1].Name+");"
            Code=Code+"\t"+connectCode+"\n"
      return Code

#Routine to obtain Modelica code for a physical component

def getModelicaObject(aNode):
nodeType=Type(aNode)
if  nodeType=="SE":
    return "BondLib.Sources.Se "+aNode.Name+
    "(e0="+str(aNode.Value)+");"
```

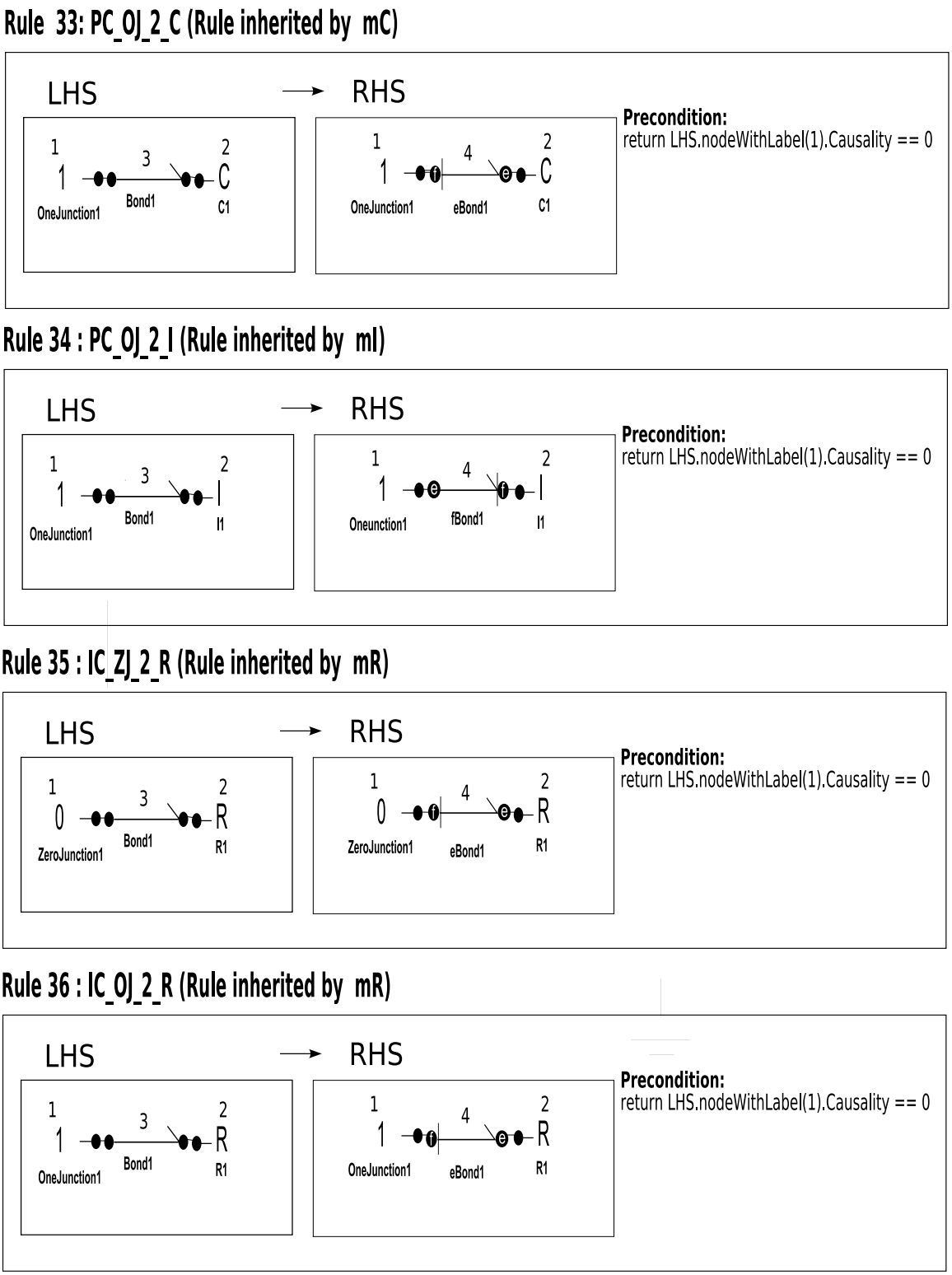


Figure 2.20: Model Transformation HABG to HCBG: Rules 33-36

```
elif nodeType=="SF":
    return "BondLib.Sources.Sf "+aNode.Name+
    "(f0="+str(aNode.Value)+");"

elif nodeType=="mSE":
    return "BondLib.Sources.mSe "+aNode.Name+";"

elif nodeType=="mSF":
    return "BondLib.Sources.mSf "+aNode.Name+";"

elif nodeType=="sinSE":
    return "BondLib.Sources.sinSe "+aNode.Name+
    "(e0="+str(aNode.Value)+",phase="+
str(aNode.PhaseAngle)+",freqHz="+str(aNode.Frequency)+
",offset="+str(aNode.Offset)+"
,startTime="+str(aNode.startTime)+");"

elif nodeType=="sinSF":
    return "BondLib.Sources.sinSf "+aNode.Name+
    "(e0="+str(aNode.Value)+",phase="+
str(aNode.PhaseAngle)+",freqHz="+str(aNode.Frequency)+
",offset="+str(aNode.Offset)+
",startTime="+str(aNode.startTime)+");"

elif nodeType=="tableSE":
    return "BondLib.Sources.tableSe "+aNode.Name+
    "(e0="+str(aNode.Value)+",
offset="+str(aNode["Offset"].getValue())+
",startTime="+str(aNode.startTime)+")"

elif nodeType=="tableSF":
    return "BondLib.Sources.tableSf "+aNode.Name+
    "(f0="+str(aNode.Value)+",offset="+
str(aNode["Offset"].getValue())+",startTime="+str(aNode.startTime)+")"

#Bond Graph Junctions

elif nodeType=="OJ":
   numberOfBondsConnected=getOutDegree(aNode)+aNode.getInDegree(aNode)
   return "BondLib.Junctions.J1p"+numberOfBondsConnected+" "+aNode.Name

elif nodeType=="ZJ":
   numberOfBondsConnected=getOutDegree(aNode)+getInDegree(aNode)
   return "BondLib.Junctions.J0p"+numberOfBondsConnected+" "+aNode.Name

elif  nodeType=="fBond":
   return "BondLib.Bonds.fBond"+" "+aNode.Name
```

```
elif  nodeType=="eBond":
   return "BondLib.Bonds.eBond"+" "+aNode.Name

elif  nodeType=="I":
   return "BondLib.Passive.I"+" "+aNode.Name+"(I="+str(aNode.Value)+")"

elif  nodeType=="C":
   return "BondLib.Passive.C"+" "+aNode.Name+"(C="+str(aNode.Value)+")"

elif  nodeType=="R":
   return "BondLib.Passive.R"+" "+aNode.Name+"(R="+str(aNode.Value)+")"

elif  nodeType=="TF":
   return "BondLib.Passive.TF"+" "+aNode.Name+"(m="+str(aNode.Value)+")"

elif  nodeType=="GY":
   return "BondLib.Passive.GY"+" "+aNode.Name+"(r="+str(aNode.Value)+")"

elif  nodeType=="mI":
   return "BondLib.Passive.mI"+" "+aNode.Name

elif  nodeType=="mC":
   return "BondLib.Passive.mC"+" "+aNode.Name

elif  nodeType=="mR":
   return "BondLib.Passive.mR"+" "+aNode.Name

elif  nodeType=="mTF":
   return "BondLib.Passive.mTF"+" "+aNode.Name

elif  nodeType=="mGY":
   return "BondLib.Passive.mGY"+" "+aNode.Name

elif  nodeType=="De":
   return "BondLib.Sensors.De"+" "+aNode.Name

elif  nodeType=="Df":
   return "BondLib.Sensors.Df"+" "+aNode.Name

elif  nodeType=="Dp":
   return "BondLib.Sensors.Dp"+" "+aNode.Name

elif  nodeType=="Dq":
   return "BondLib.Sensors.Dq"+" "+aNode.Name

#CBD Declarations
```

```
elif  nodeType=="Constant":
   return "Modelica.Blocks.Sources.Constant"+" "+
   aNode.Name+"(k="+str(aNode.Value)+")"

elif  nodeType=="Delay":
   return "Modelica.Blocks.Discrete.UnitDelay"+" "+
   aNode.Name+"(y_start="+str(aNode.Value)+")"

elif  nodeType=="Integrator":
   return "Modelica.Blocks.Continuous.Integrator"+
   " "+aNode.Name+"(k="+str(aNode.Value)+
",y_start="+str(aNode.y_start)+")"

elif  nodeType=="Derivative":
  return "Modelica.Blocks.Continuous.Derivative"+
  " "+aNode.Name+"(k="+aNode.Value+
",y_start="+str(aNode.T)+")"

elif  nodeType=="Sum":
  return "Modelica.Blocks.Math.Sum"+" "+aNode.Name

elif  nodeType=="Product":
  return "Modelica.Blocks.Math.Product"+" "+aNode.Name

elif  nodeType=="Feedback":
  return "Modelica.Blocks.Math.Feedback"+" "+aNode.Name

elif  nodeType=="PID":
  return "Modelica.Blocks.Continuous.PID"+
" "+aNode.Name+"(k="+str(aNode.Value)+",Ti="+
str(aNode.Ti)+",Td="+str(aNode.Td)+",Nd="+str(aNode.Nd)+")"

elif  nodeType=="TimeTable":
  return "Modelica.Blocks.Sources.TimeTable"+" "+aNode.Name+"(table="+
str(aNode["Table"].getValue())+",offset="+
str(aNode.Offset)+",startTime="+str(aNode.startTime)+")"

elif  nodeType=="Generic":
  return "Modelica.Blocks.Sources."+str(nodeType)+str(aNode.Parameters)
else:
  return None
```

2.6 Modelica to Trajectory

The Modelica code is now compiled to generate C code consisting of a DAE solver. We use DASSL [AP98] to solve DAEs generated from the Modelica code as C code. DASSL uses

backward differentiation formula (BDF) methods [Gea71] to solve a system of DAEs or ODEs. The methods are variable step-size variable order. The system of equations in DASSL is written in an implicit ODE form like

$F(t, y, y') = 0,$

where y' denotes the time derivatives of y. The BDF methods used in DASSL require the solution of a large system of nonlinear equations

$$F(t_n, y_n, \alpha_n y_n + \beta_n) = 0$$

on each time step. Here, α_n and β_n are scalars which depend on the method and stepsize. In DASSL, this system is solved by a modified Newton iteration. Each iteration of the Newton method requires the solution of a linear system

$Ayn(k+1) = bn(k),$

where the matrix A is given by

$$A = \alpha_n \frac{\partial F}{\partial y'} + \frac{\partial F}{\partial y}.$$

The one-dimensional PDEs generates a matrix which is block tridiagonal. In DASSL, this linear system is solved via a banded direct solver. Because the CPU cost to solve this linear system is proportional to the bandwidth of the matrix, this solver is quite efficient if the bandwidth of the matrix is relatively small. Different moving mesh strategies result in different bandwidths, which is a very important factor in considering the efficiency of the method. The reader can refer to [Pet83] for details.

The results of solving the system of DAE is a Trajectory model consisting of the time varying behavior of all the state variables in the physical system model.

3

Design Space Exploration

3.1 Introduction

Design space exploration is the search through the space of possible solutions to a problem. A solution is a model which is a point in the *model design space* defined by the meta-model of a modelling language. The model design space is further constrained by usage of OCL constraints specified along with the meta-model. Using models to guide search in design space has always been useful [AG98].

We perform design space exploration of HABG models. An overview of the procedure is shown in Figure 3.1. We present parameterized Graph Grammar (GG) rules or mutation operators in Section 3.2 to evolve the model of a physical system. The rules only modify the physical part of the HABG. Evolution of controllers described as CBD and other formalism in the signal domain are well established and a survey of techniques is available in [MC95]. Our focus is in the evolution of the physical system. Evolution of the plant or the physical system models based on genetic programming [BF97], [HGS03] is described in [JWG05]. Matching an evolving BG to a mathematical function such as a filter is discussed in [AMP05].

In our case the evolution process is guided by a genetic algorithm. We present a genetic algorithm in Section 3.3. We first develop a simple experimental setup for the evolution in Section 3.4. Finally, we briefly discuss the results of the evolution process for the experimental setup in Section 3.5.

3.2 Heuristics for Evolving Physical System Models

The heuristic rules or mutation operators for evolving/mutating a physical system are defined for evolving Hybrid Acausal Bond Graph (HABG) in Figures 3.2, 3.3, 3.4, 3.5, 3.6, and 3.7. The rules not required to be executed in any particular order. The order of rule execution is based on a *plan*. A plan will be discussed in the next section. A description of the heuristic rules are given in Table 3.1. All the rules are parameterized and obtain as input a BG element.

3.3 Genetic Algorithm

We now describe a genetic algorithm [Hol92] that evolves physical system models in the HABG modelling language. Before a genetic algorithm can be put to work on any problem, a method is needed to encode potential solutions to that problem in a form that a computer can process. The input to our genetic algorithm is an embryo model. We define a *plan* that is a sequence of paramterized heurisitic rules that are applied on the input model to mutate it. The plan is a potential solution to an evolution task.

The structure of plan is illustrated in Figure 3.8. The plan comprises of an unit which consists

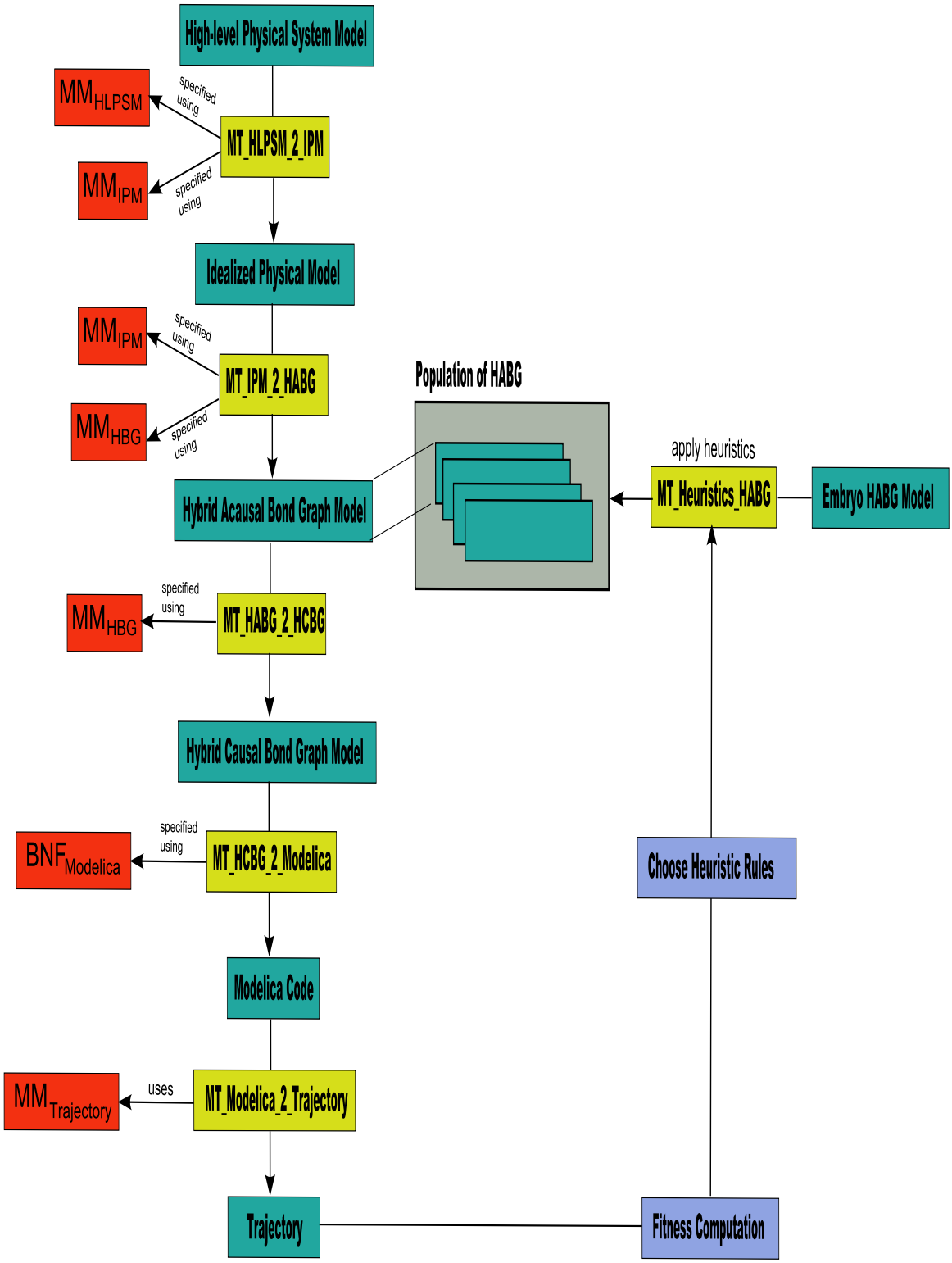
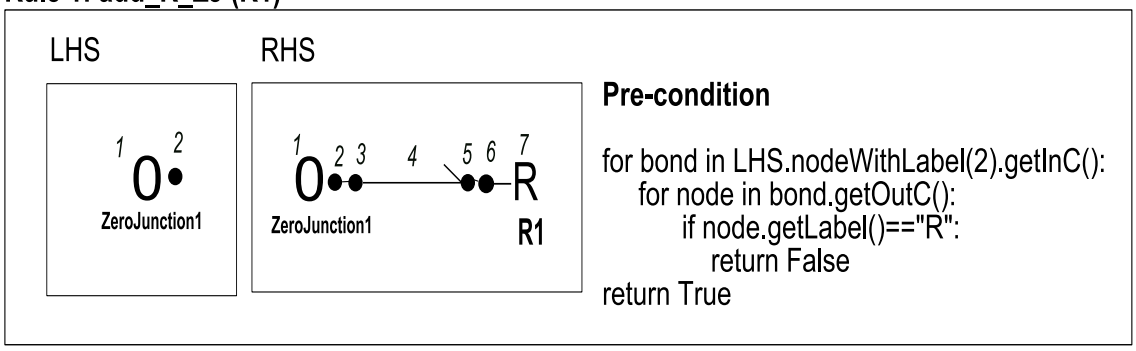
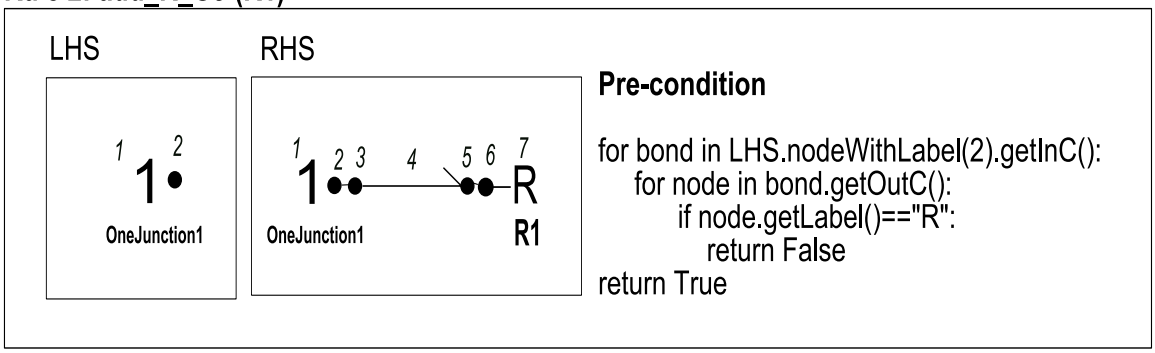


Figure 3.1: Design Space Exploration of Physical System Models

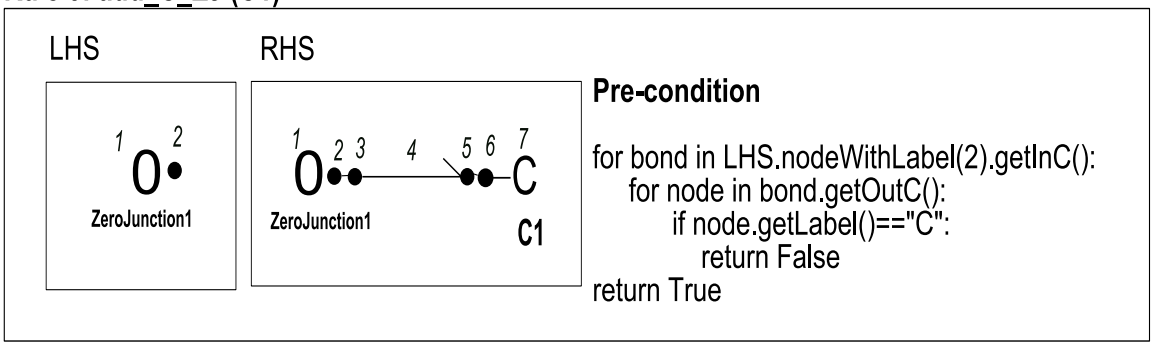
Rule 1: add_R_ZJ (R1)



Rule 2: add_R_OJ (R1)



Rule 3: add_C_ZJ (C1)



Rule 4: add_C_OJ (C1)

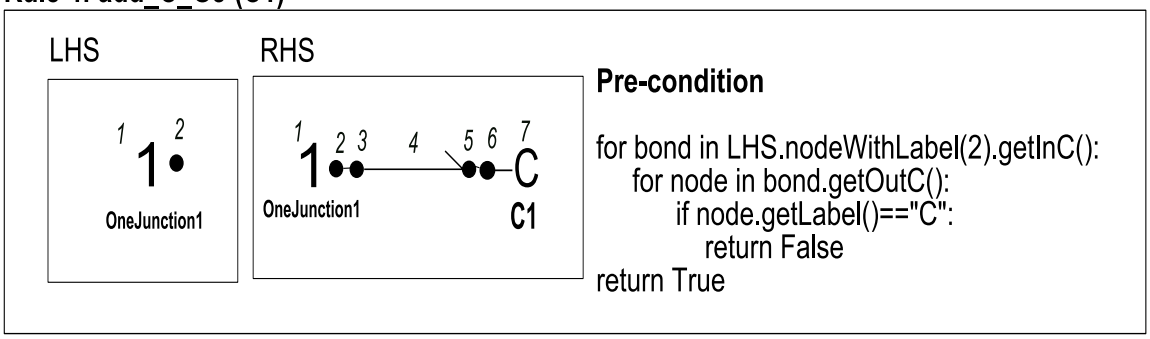
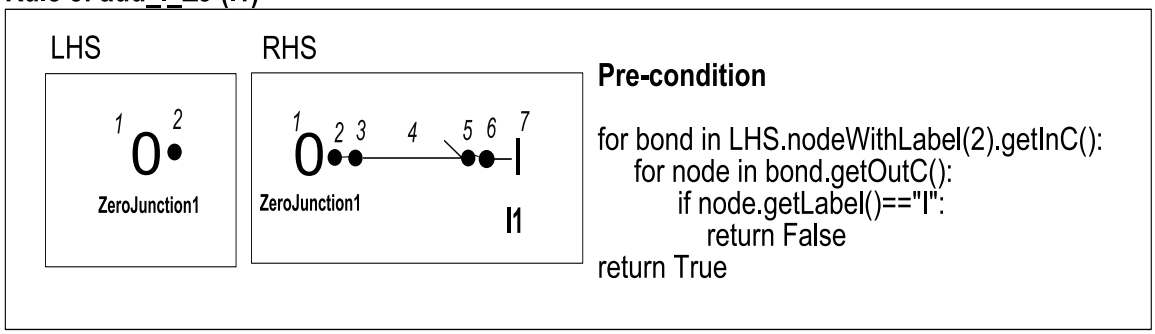
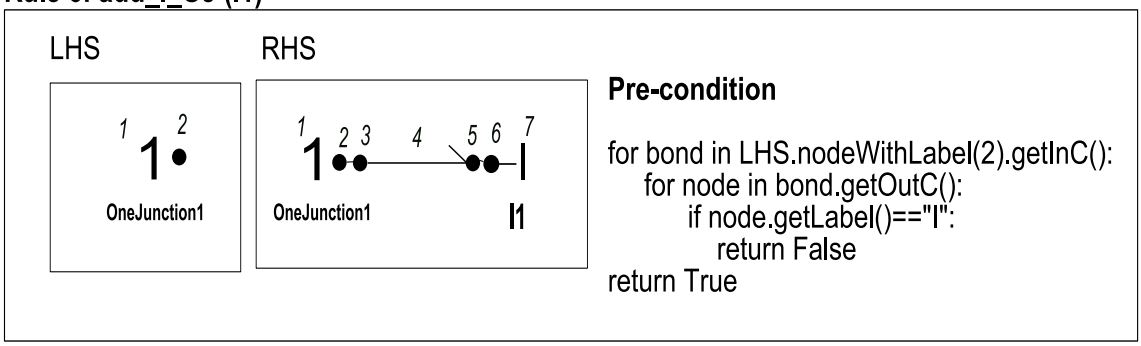


Figure 3.2: Model Evolution Heuristics: Rules 1-4

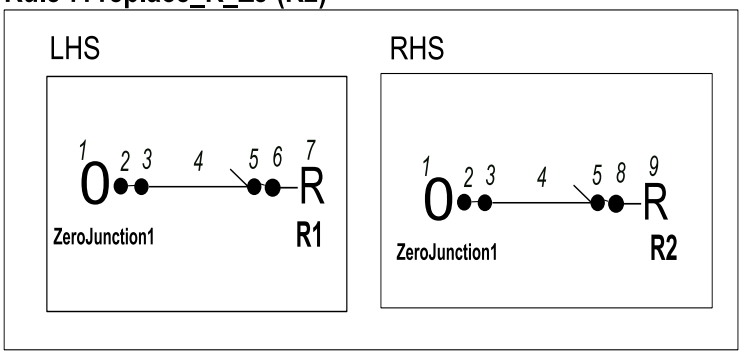
Rule 5: add_I_ZJ (I1)



Rule 6: add_I_OJ (I1)



Rule 7: replace_R_ZJ (R2)



Rule 8: replace_R_OJ (R2)

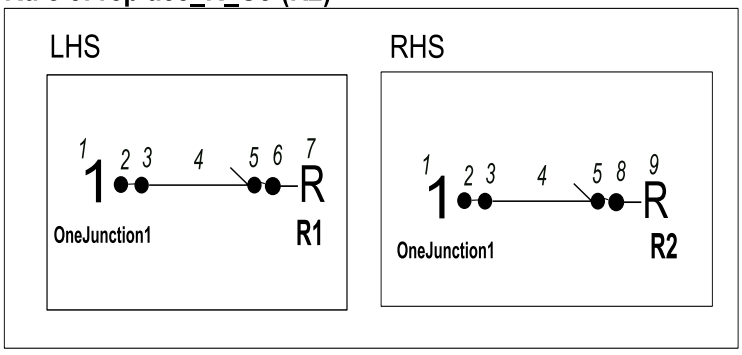


Figure 3.3: Model Evolution Heuristics: Rules 5-8

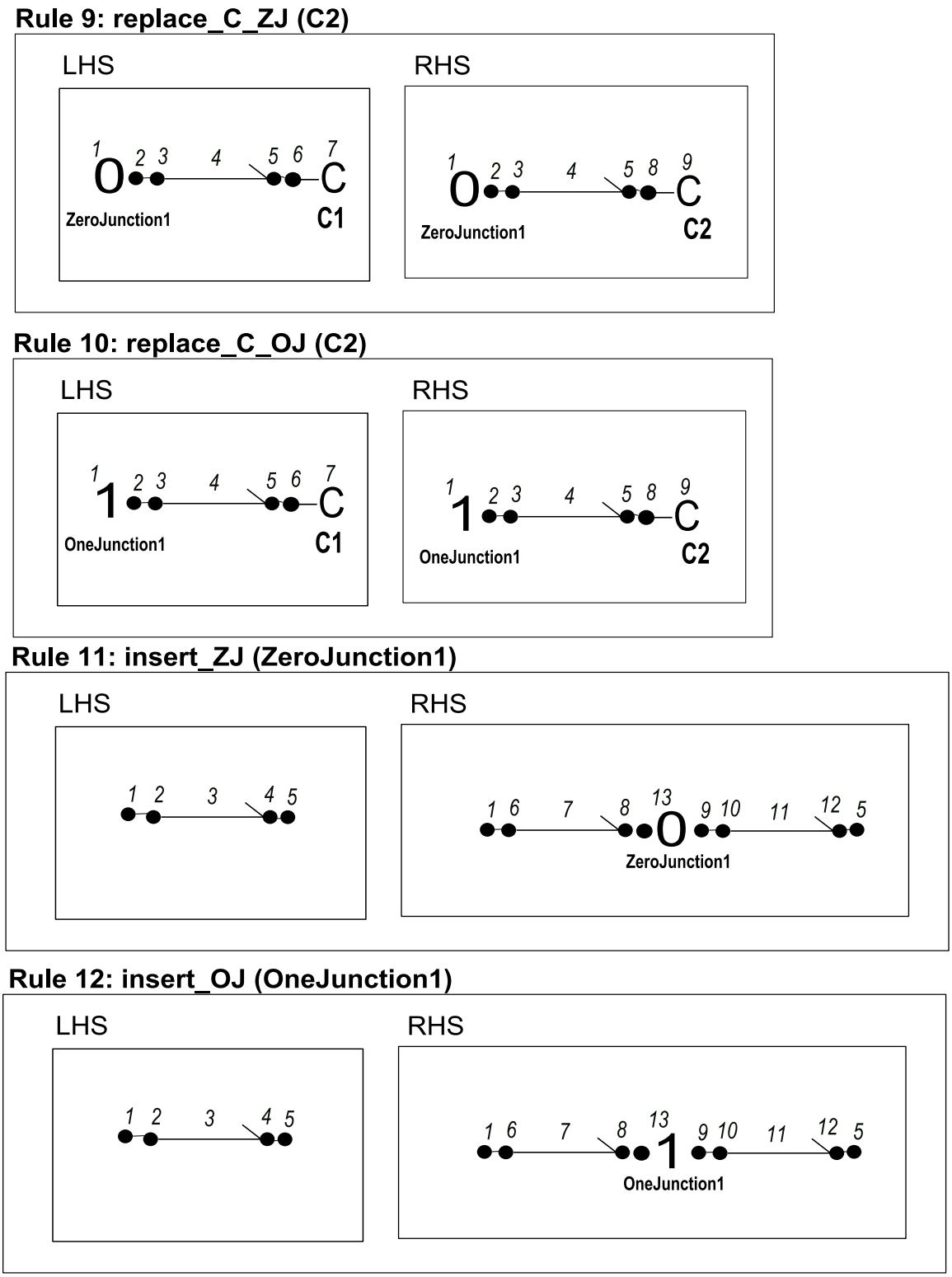
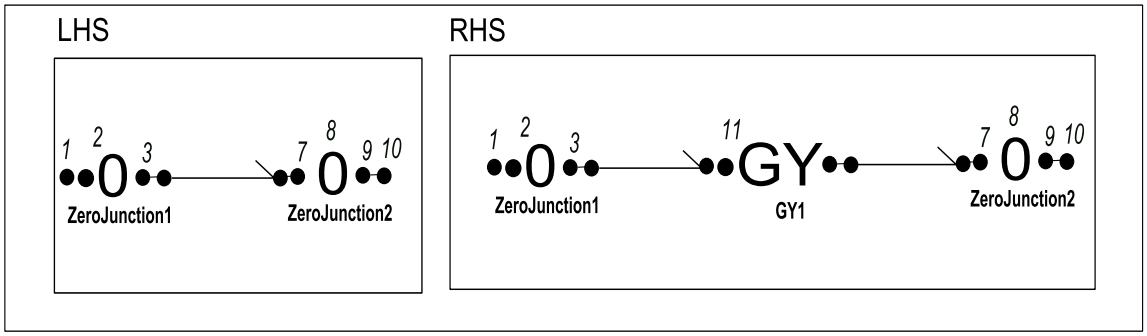
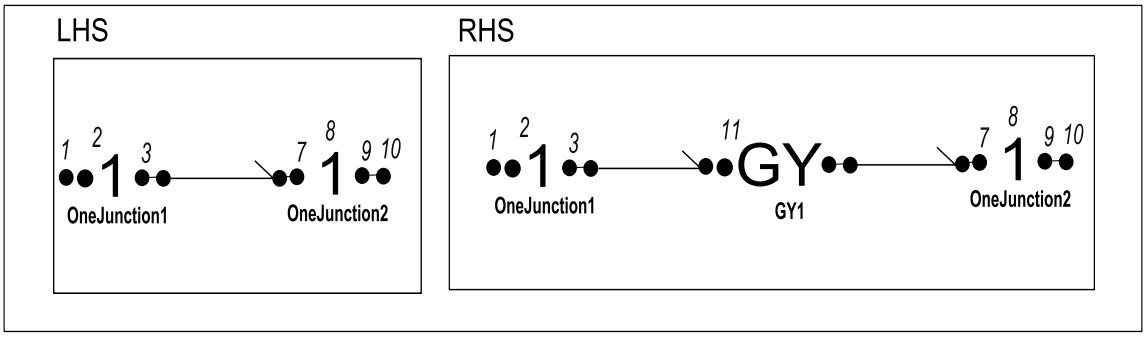


Figure 3.4: Model Evolution Heuristics: Rules 9-12

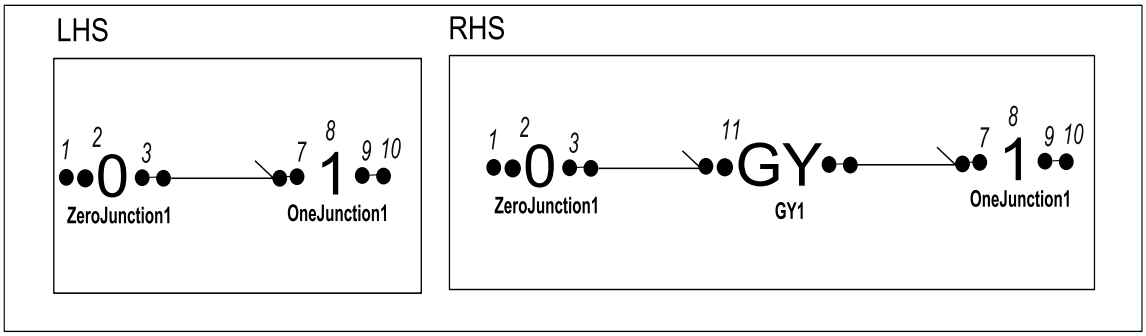
Rule 13: insert_ZJ_GY_ZJ (GY1)



Rule 14: insert_OJ_GY_OJ (GY1)



Rule 15: insert_ZJ_GY_OJ (GY1)



Rule 16: insert_OJ_GY_ZJ (GY1)

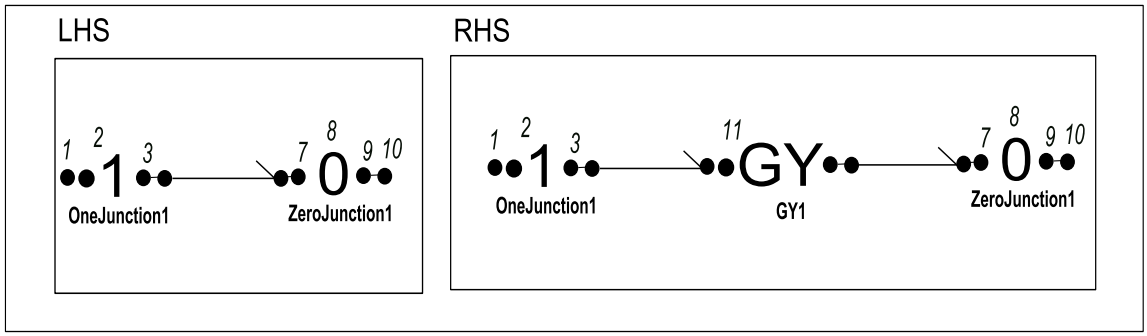


Figure 3.5: Model Evolution Heuristics: Rules 13-16

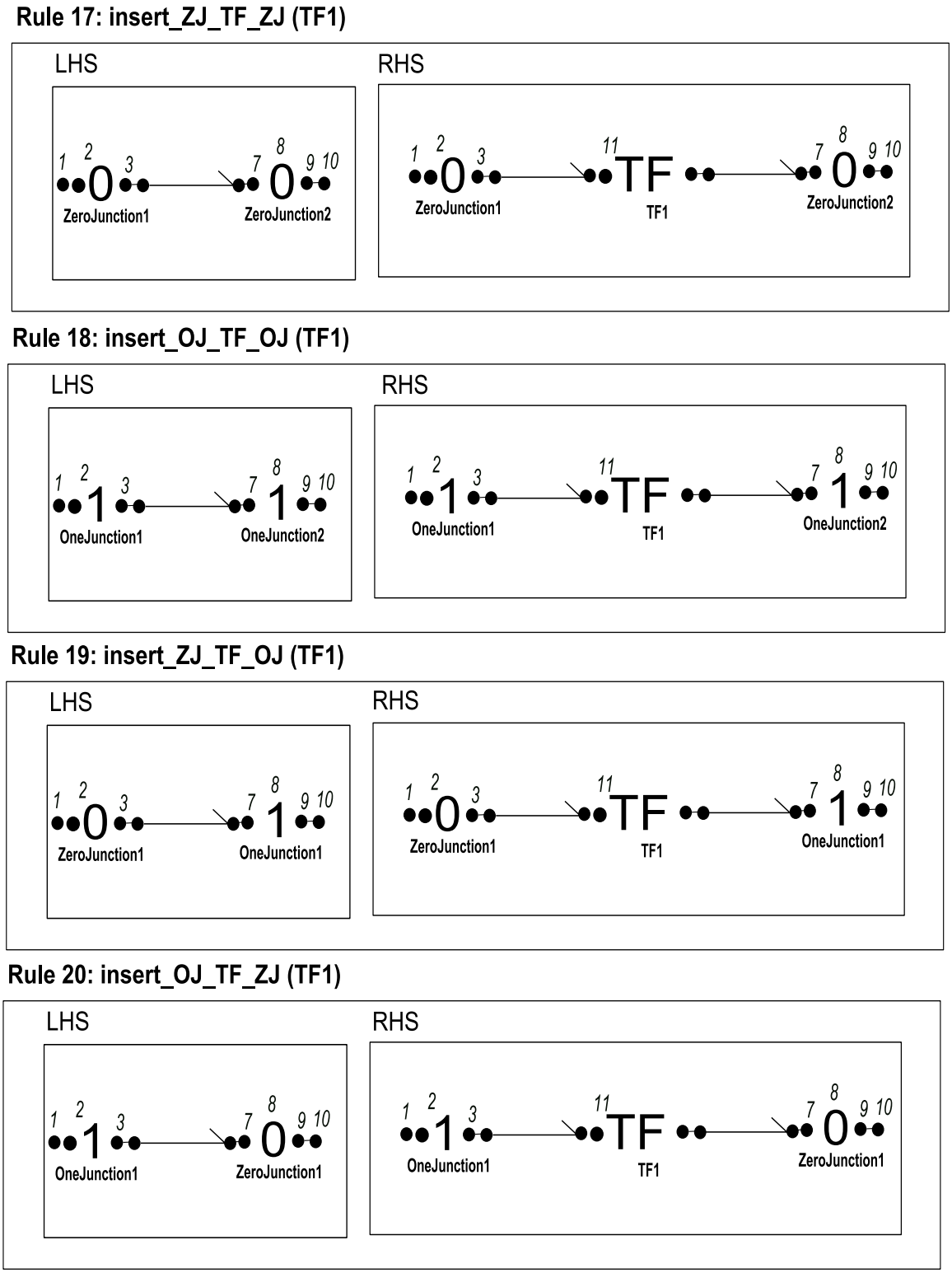


Figure 3.6: Model Evolution Heuristics: Rules 17-20

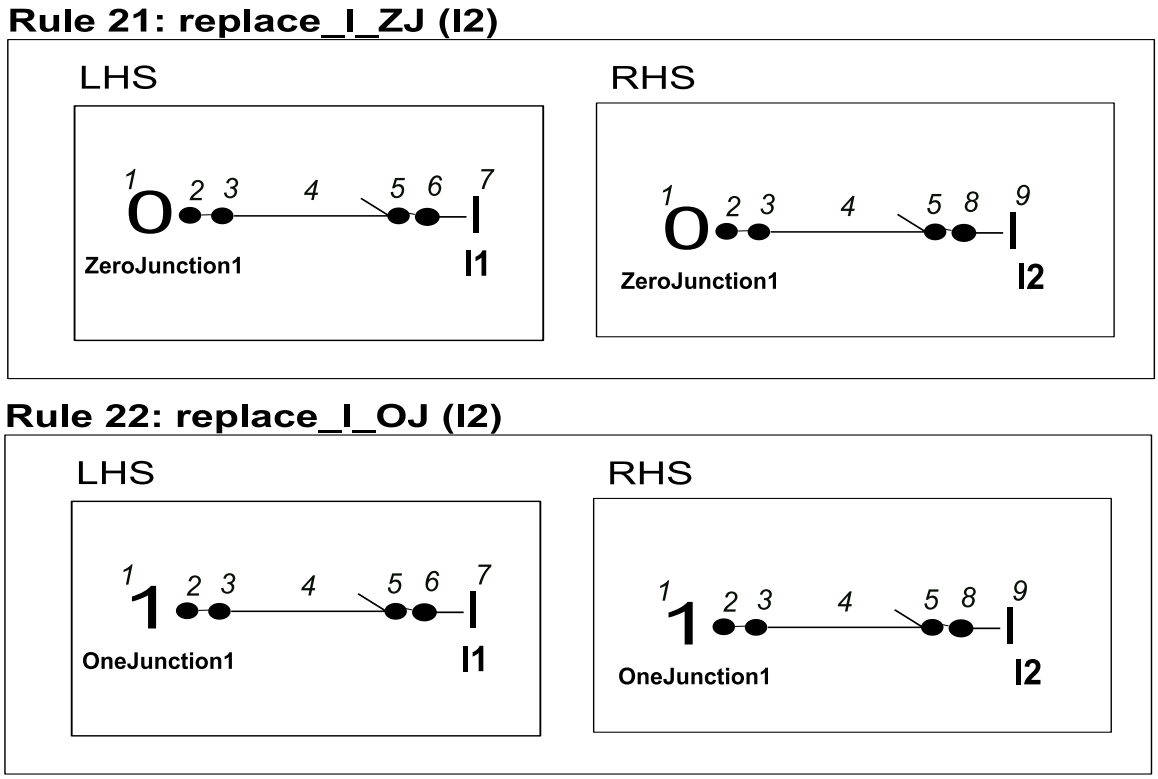


Figure 3.7: Model Evolution Heuristics: Rules 21-22

Table 3.1: Graph Grammar rules for preferred and indifferent causality in MT_HABG_2_HCBG

Order	Rule Name	Description
1	add_R_ZJ(R1)	Add a R element to a 0-junction without an R element
2	add_R_OJ(R1)	Add a R element to a 1-junction without an R element
3	add_C_ZJ(C1)	Add a C element to 0-junction without an C element
4	add_C_OJ(C1)	Add a C element to 1-junction without an C element
5	add_I_ZJ(I1)	Add a I element to 0-junction without an I element
6	add_I_OJ(I1)	Add a I element to 1-junction without an I element
7	replace_R_ZJ(R2)	Replace an R element connected to a 0-junction R1 with another R element R2
8	replace_R_OJ(R2)	Replace an R element connected to a 1-junction R1 with another R element R2
9	replace_C_ZJ(C2)	Replace an C element connected to a 0-junction R1 with another C element C2
10	replace_C_OJ(C2)	Replace an C element connected to a 1-junction R1 with another C element C2
11	insert_ZJ(ZeroJunction1)	Insert a 0-junction between a Bond element
12	insert_OJ(OneJunction1)	Insert a 1-junction between a Bond element
13	insert_ZJ_GY_ZJ(GY1)	Insert a GY element between two 0-junctions
14	insert_OJ_GY_OJ(GY1)	Insert a GY element between two 1-junctions
15	insert_ZJ_GY_OJ(GY1)	Insert a GY element between a 0-junction and a 1-junction
16	insert_OJ_GY_ZJ(GY1)	Insert a GY element between a 0-junction and a 1-junction
17	insert_ZJ_TF_ZJ(TF1)	Insert a TF element between a 0-junction and a 0-junction
18	insert_OJ_TF_OJ(TF1)	Insert a TF element between a 1-junction and a 1-junction
19	insert_ZJ_TF_OJ(TF1)	Insert a TF element between a 0-junction and a 1-junction
20	insert_OJ_TF_ZJ(TF1)	Insert a TF element between a 1-junction and a 0-junction
21	replace_I_ZJ(I2)	Replace an I element connected to a 0-junction R1 with another I element I2
22	replace_I_OJ(I2)	Replace an I element connected to a 1-junction R1 with another I element I2

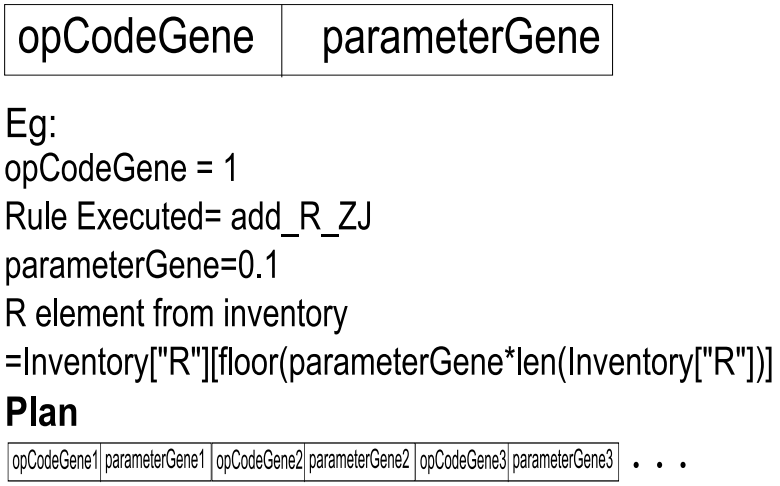


Figure 3.8: The Structure of a Plan

of two parts. The first part of the unit contains the *opCode* or the rule number of the heuristic to be applied. The second part of the unit contains a floating point number between 0 and 1. The is number is used to choose an element from an *Inventory*. Before the evolution starts we initialize an inventory containing potential building blocks that can be used to evolve the model. In an industrial setting these building elements can be considered as off the shelf resistors, capacitors, gears etc. that are available as resources. A rule removes the parameter element from the inventory and puts it into the model according to a rule.

Now that we know what a plan is and the notion of an inventory is clear we present a the pseudocode for the genetic algorithm.

1. Initialize the Emrbyo Model
2. Initialize the Inventory with Objects of Different Elements
3. Initialize PopulationSize, MutationProbability (Probability of applying the genetic mutation operator), ChildCull (Number of children to remove in each generation)
4. Initialize Population of Plans P
5. While True:
    6. P.generate() //Generate or mutate a population of plans
    7. P.applyPlansToEmbryo() //Apply the plans to the embryo model
    8. P.assignCausality() //Assign causality to all models
    9. P.getModelica() //Generate Modelica Code
    10. P.simulate() // Simulate models in the population
    11. P.computeFitness() //Compute the fitnesses
    12. best=P.best() //Obtain the fittest individual
    13. currentfitness = best.fitness() //Obtain the fitness of the fittest individual
    14. if currentfitness >= maxFitness: //If the fittess condition is satisfied the algorithm has reached its goal

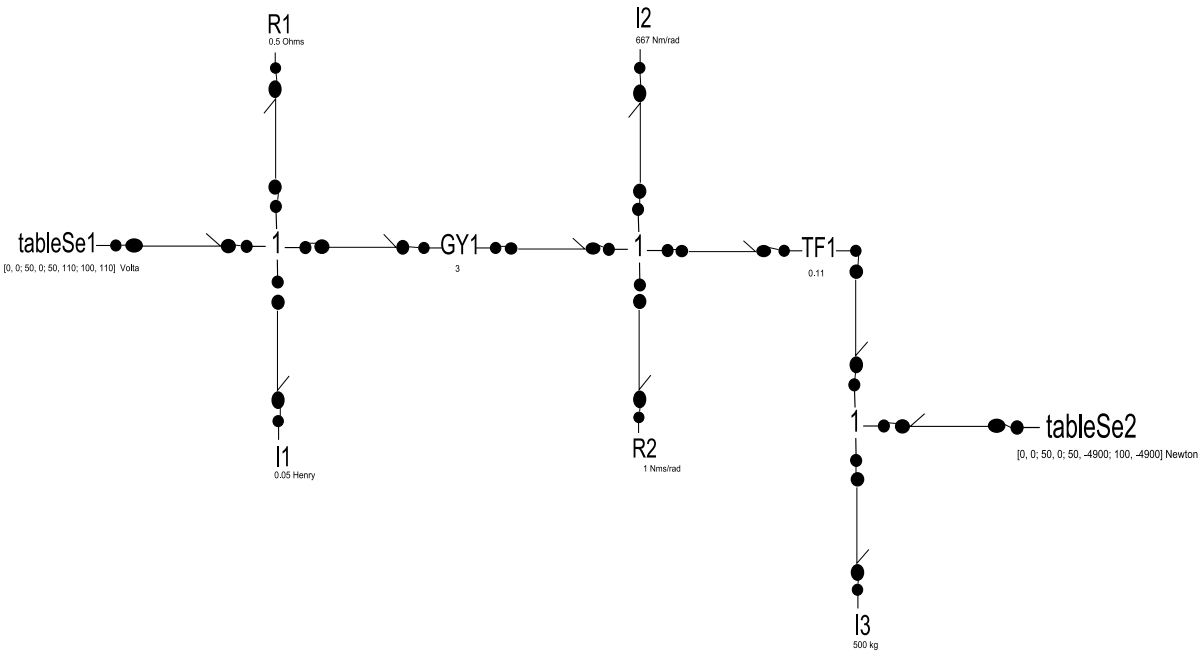


Figure 3.9: ACausal Bond Graph of the Embryo Model

15. print 'End of Evolution'
16. break()

The genetic algorithm described does not include crossover. The only genetic operation that takes place is, mutation. A fitness function is defined to evaluate the optimality of a plan and in turn a model which helps in guiding the search. In the next section we describe an experimental setup to evolve a hoisting device model using the above algorithm.

3.4 Experiment Setup

We first construct an embryo model for a hoisting device. The model is capable of lifting 500 kg of load without breaking down. The parameters for the embryo model are given in Table 3.2. The embryo model is represented in the HABG modelling language and is shown in Figure 3.9.

The maximum input voltage to the hoisting device is 110V as supplied by tableSe1. The mass is initially on the ground therefore the downward force is compensated by the reactive force. At 50 secs the hoisting device attempts to lift a mass of 500 kg. At this instant the downward force comes into play as described in tableSe2. The current model is able to lift the 500 kg mass without any problem. This is shown in Figure 3.10 (a).

We now increase the load to 1000 kg. At this point the physical system cannot handle the load anymore and it breaks down. The effect is shown in Figure 3.10 (b). The question is, keeping the input voltage fixed can we modify the model of the physical system to lift a load of 1000 kg ?

The fitness function is very simple. We simply return the height h or Dq1.OutPort1 which is the sensor value for the height of the load. The fitness improves if h goes from the negative to the positive domain. The height h is measured at 60 seconds from start time. The length of the

Table 3.2: Embryo Hoisting Device Parameters

Component Name	Parameter	Value	Unit
tableSe1	table	$[0, 0; 50, 0; 50, 110; 100, 110]$	Volts
tableSe1	startTime	0	Seconds
tableSe1	offset	0	Volts
tableSe2	table	$[0, 0; 50, 0; 50, -4900; 100; -4900]$	Newton
$R1$	R	0.5	Ω
$I1$	I	0.05	Henry
GY1	r	3	*NA*
$I2$	I	667	Nm/rad
$R2$	R	1	Nms/rad
TF1	m	0.11	*NA*
$I3$	I	500	kg

program or the number of heuristic rules applied to the model is limited to 3. The inventory comprised of 1 transformer with ratio 0.11 and a 1-junction. This limited inventory size is used to observe if the genetic algorithm can automatically detect the use of gears.

We run our genetic algorithm to see if something useful turns out. The results are presented in the next section.

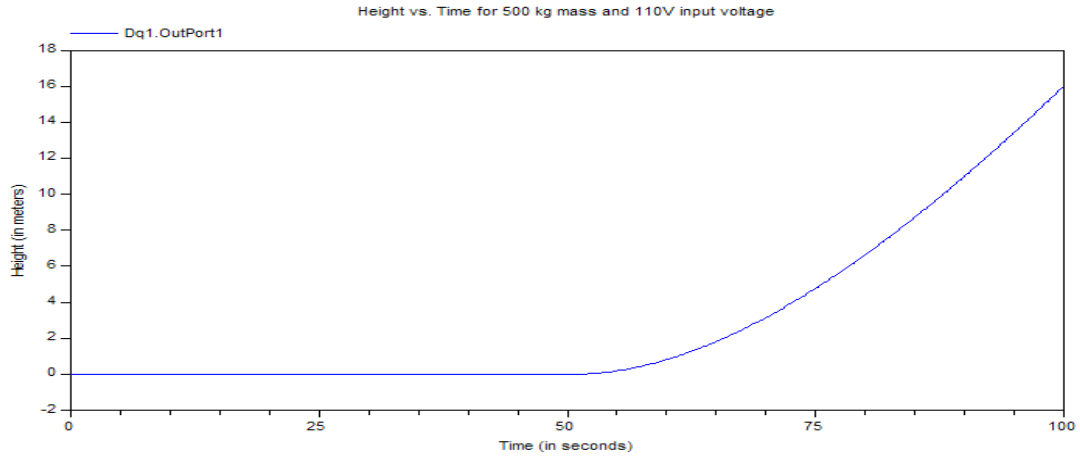
3.5 Results

The genetic algorithm is run with a maximum of 10 models in the population as performing a simulation is computationally expensive. The simualtion is run for 2 generations. Plans are applied to each model and the resulting fitness is computed.

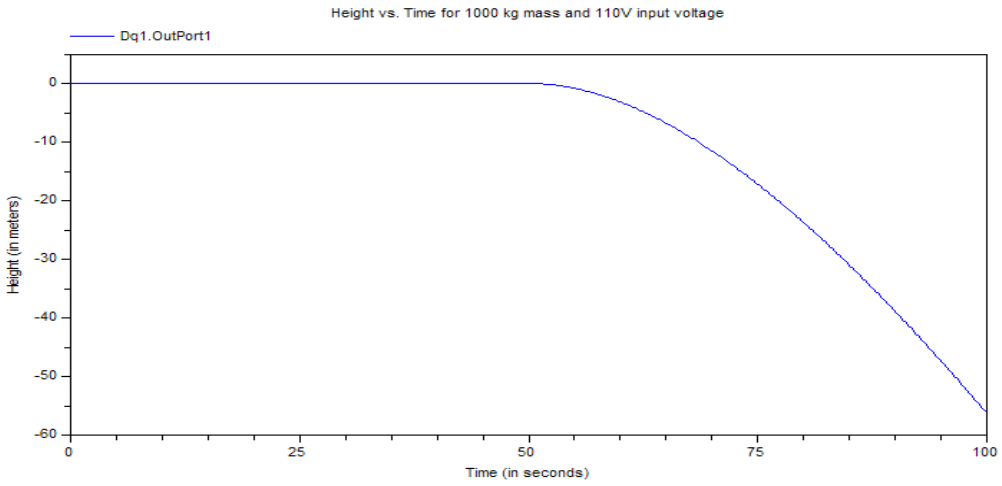
After 2 generations the fittest individual is able to lift the load of 1000 kg with the same input voltage of 110 V. The plan inserts a 1-junction in the mechanical domain by executing the insert_OJ rule. Following this the algorithm inserts a transformer TF element between the 1-junctions, by applying the heuristic rule OJ_TF_OJ.

Physically the genetic algorithm is able to discover the role of gears. The inclusion of the transformer allows the hoisting device to lift the load of 1000 kg. The structural variation on the embryo model is shown in Figure 3.11.

The graph showing the height vs. time behaviour of the new model is shown in Figure 3.12.



(a)



(b)

Figure 3.10: (a) Height attained by hoisting device for 500 kg mass (b) Hoisting device breaks down due to heavy mass

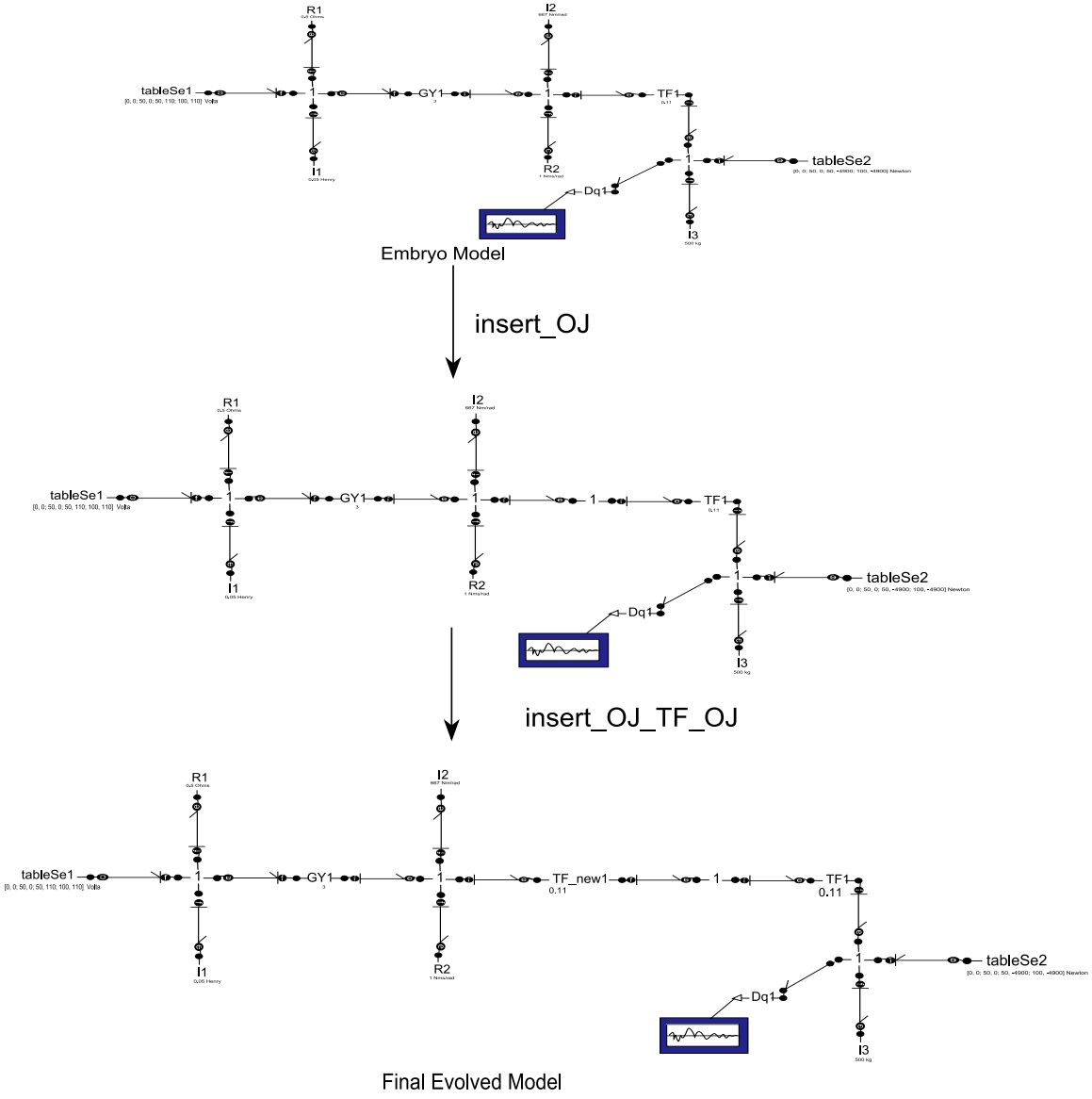


Figure 3.11: The Application of Optimal Heuristics to the Embryo Model

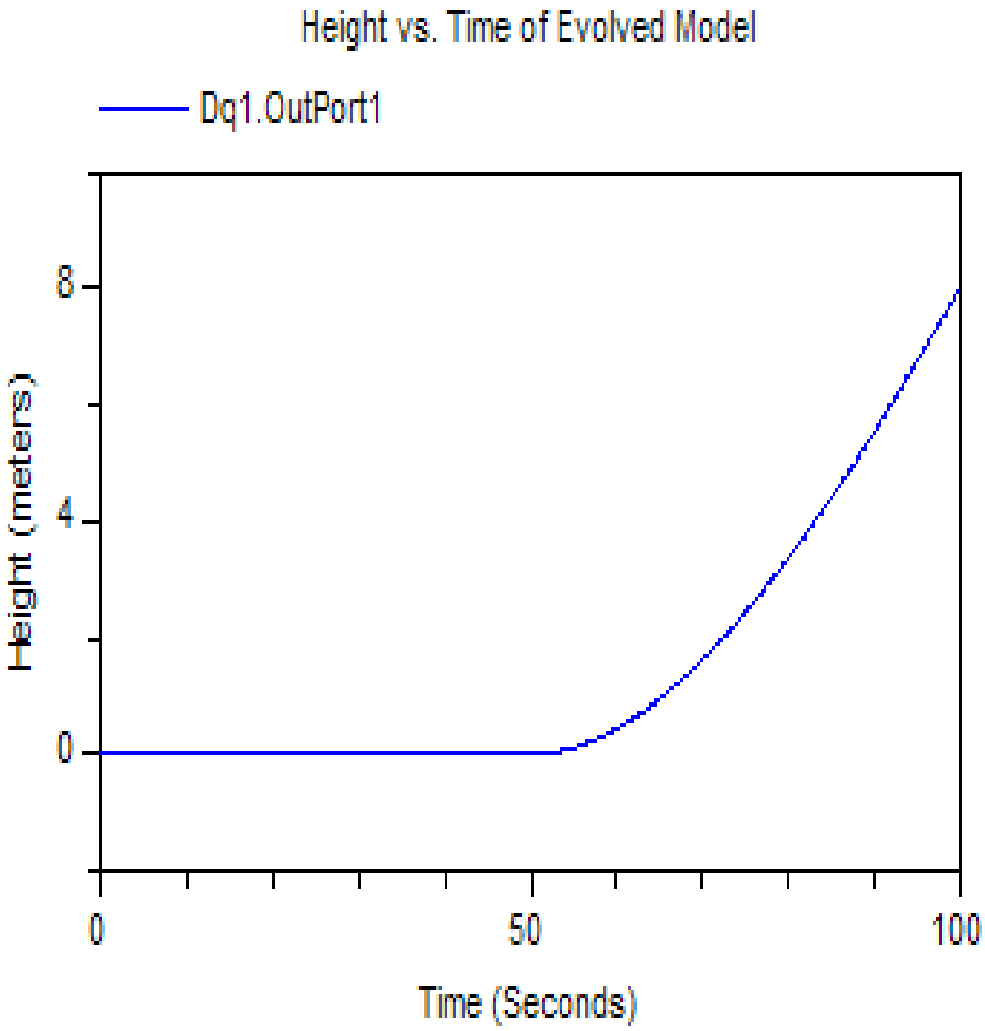


Figure 3.12: Behavior of improved Hoisting Device

4
Conclusion

In this thesis we presented a methodology for developing models of a physical system at different abstraction levels. We illustrate the importance of dividing the modelling process into different levels or viewpoints. Using MDE based ideas it is now very easy to synthesize a complete modelling environment or a modelling language from a meta-model specification. This empowers modellers of one system at different viewpoints to cast their ideas into completely executable code.

The transformations between visual languages have been specified using GG rules. The visual specification of models and also their transformations makes the process of development much faster that traditional techniques. The visual specification of rules is far less error prone than specifying rules textually. The rules are compact and give a clearer understanding of a transformation. The GG rules by themselves are usually self-explanatory.

Finally, we see with the help of a simple hoisting device example that meta-models provide a consistent way to describe a model design space. The model design space of a physical system is an ideal candidate for exploration by many existing and newly developed artificial intelligence (AI) techniques. We specify heuristic rules as GG rules. Again, we notice the power of a visual rule. The first ideas in the mind of an engineer is usually visual and also domain specific. Encoding such a heuristic as a visual rule is faster, error-free and closer to the modeller's experiencial knowledge. The rules are executed in a sequence prescribed by the genetic algorithm.

As future work it would be interesting to build two-way transformations between modelling languages. The transformation from a low-level modelling language to a high-level modelling language will usually pose many possibilities. For instance when a BG model is transformed to an IPM model a BG element can be assigned many different domains. Coming to the general notion of the model design space of a modelling language. It would be useful to come up with a method that can do efficient constrain satisfication and design space search for any given meta-model and constraints specification.

Bibliography

[AG98] Don Smith Andrew Gelsey, Mark Schwabacher. Using modeling knowledge to guide design space search. *Artificial Intelligence*, 101:35–62, 1998.

[Aga03] Aditya Agarwal. Graph rewriting and transformation (great): A solution for the model integrated computing (mic) bottleneck. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, page 364, 2003.

[AMP05] Philippe Kubiak Anca-Maria Pirvu, Genevive Dauphin-Tanguy. Automatic generation of all bond graph structures matching a given transfer function. In *Proceedings of the 2005 International Conference on Bond Graph Modeling*, 2005.

[AP98] Uri M. Ascher and Linda Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM, 1998.

[BF97] Nordin Peter Keller-Robert E. Banzhaf, Wolfgang and Frank D. Francone. *Genetic Programming An Introduction.* San Francisco, CA: Morgan Kaufmann and Heidelberg: dpunkt., 1997.

[BJV60] J.H. van Wijngaarden A. Woodger M. Bauer F.L. Green J. Katz C. McCarthy J. Perlis A.J. Rutishauser H. Samelson K. Backus J.W., Wegstein and B. Vauquois. Report on the algorithmic language algol 60. *Communications of the ACM*, 3(5):299–314, 1960.

[Bro] Jan F. Broenink. Introduction to physical systems modelling with bond graphs. http://www.ce.utwente.nl/bnk/papers/BondGraphsV2.pdf.

[Bro99] Jan F. Broenink. Introduction to physical systems modelling with bond graphs. *DRAFT (version 2, 20-5-99), to be published in the SiE Whitebook on Simulation methodologies*, 1999.

[Cel05] Cellier, F.E ,and A. Nebot. The Modelica Bond Graph Library. In *Proc. of the 4th International Modelica Conference*, volume 1, pages 57–65, Hamburg, Germany, 2005.

[CLOP02] Gennaro Costagliola, Andrea De Lucia, Sergio Orefice, and Giuseppe Polese. A classification framework to support the design of visual languages. *J. Vis. Lang. Comput.*, 13(6):573–600, 2002.

[DKR00] D.L. Margolis D.C. Karnopp and R.C. Rosenberg. *System Dynamics: A Unified Approach*. Wiley, New York, 3rd edition, 2000.

[Fri03] Peter Fritzson. *Principles of Object-oriented Modelling and Simulation in Modelica 2.1.* Wiley-IEEE Press, 2003.

[Gea71] C. W. Gear. *Numerical Initial Value Problems in Ordinary Differential Equations*. Prentice-Hall, Englewood Cliffs, NJ, 1971.

[GW03] DC Kennedy GD Wood. Simulating mechanical systems in simulink with simmechanics. Technical report, The MathWorks Inc., 2003.

[HER99] H-J Kreowski H Ehrig, G Engels and G Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[HGS03] Jianjun Hu, Erik D. Goodman, and Kisung Seo. *Genetic Programming Theory and Practice*, chapter Continuous Hierarchical Fair Competition Model for Sustainable Innovation in Genetic Programming, pages 81–98. Kluwer, 2003.

[H.M61] Paynter H.M. *Analysis and Design of Engineering Systems*. MIT Press, Cambridge, Massachusetts, 1961.

[Hol92] John Holland. Genetic algorithms. *Scientific American*, pages 66–72, July 1992.

[JWG05] Janis P. Terpenny Jiachuan Wang, Zhun Fan and Erik D. Goodman. Knowledge interaction with genetic programming in mechatronic systems design using bond graphs. *IEEE Transactions on Systems, Man, and CyberneticsPart C: Applications and Reviews,*, 35(2):172–182, 2005.

[Ken02] Stuart Kent. Model driven engineering. *LNCS , Integrated Formal Methods: Third International Conference,Turku, Finland,*, 2335:286, May 2002.

[Mac03] A. Macchelli. *Port Hamiltonian systems. A unified approach for modeling and control finite and infinite dimensional physical systems*. PhD thesis, University of Bologna - DEIS, 2003.

[Mat] The MathWorks Inc. *Simulink User's Guide.*

[Mat97] Elmqvist H. Broenink J.F. Mattsson, S.E. Modelica: An international effort to design the next generation modelling language. *Journal A, Benelux Quarterly Journal on Automatic Control*, 38(3):16–19, September 1997. Special issue on Computer Aided Control System Design, CACSD, 1998.

[MC95] Maja Mataric and Dave Cliff. Challenges in evolving controllers for physical robots. Technical Report CS-95-18, Brandeis University, 1995.

[Mod05] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling Specification ver 2.2*, February, 2 2005.

[OMG] OMG. The object constraint language specification 2.0, omg document: ad/03-01-07.

[OMG03] OMG. Meta object facility 1.4, omg document: formal, April, 02 2003.

[Pet83] L. R. Petzold. *Scientific Computing*, chapter A description of DASSL: A differential/algebraic system solve, pages 65–68. North- Holland, Amsterdam, 1983.

[Pro05] Marc Provost. Himesis: A hierarchical subgraph matching kernel for model driven development. Master's thesis, McGill University, 2005.

[sim] The MathWorks SimHyrdraulics. http://www.mathworks.com/access/helpdesk/help/toolbox/physmod/hydro/.

[VdL04] Hans Vangheluwe and Juan de Lara. Domain-specific modelling with atom3. ini, editors. In Jonathan Sprinkle Juha-Pekka Tolvanen and Matti Ross, editors, *The 4th OOPSLA Workshop on Domain-Specific Modeling*, number 4, page 8, Vancouver, Canada., October 2004.

[Zun94] L.W. Wang; A. Zunger. Large scale electronic structure calculations using the lanczos method. *Comput. Mater. Sci.*, 2(326), 1994.