Model Transformation with Hierarchical Discrete-Event Control

Thomas Huining Feng

Electrical Engineering and Computer Sciences
University of California at Berkeley

May 20, 2009

Model Transformation with Hierarchical Discrete-Event Control

by

Huining Feng

B.S. (Nanjing University, China) 2002
M.S. (McGill University, Canada) 2004
M.S. (University of California, Berkeley, USA) 2008

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering-Electrical Engineering and Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Edward A. Lee, Chair
Professor Rastislav Bodik
Professor Lee W. Schruben

Spring 2009

The dissertation of Huining Feng is approved:

_____

Chair   Date

_____

Date

_____

Date

University of California, Berkeley

Spring 2009

Model Transformation with Hierarchical Discrete-Event Control

Abstract

Model Transformation with Hierarchical Discrete-Event Control

by

Huining Feng

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Science

University of California, Berkeley

Professor Edward A. Lee, Chair

In this dissertation I present my work on a flexible and efficient model transformation technique, which includes a transformation engine and a formal and expressive control language.

I define a basic transformation with a transformation rule, which describes the relationship between input models and the result of the transformation. I invent a syntax for transformation rules, which is close to the modeling language that designers use to create the input models. This makes it convenient for the designers to use the technology. The semantics of transformation rules is defined based on graph transformation theory. To improve flexibility over traditional approaches, I introduce extensions to allow complex transformation rules with variable structures to be specified with compact higher-order descriptions.

A transformation workflow consists of basic transformations controlled in a control language. Existing control languages include finite state machines and dataflow diagrams. Realizing their limitations in expressiveness and efficiency, I create the Ptera (Ptolemy event relationship actor) model of computation based on event graphs. I show that Ptera is an appropriate control language for transformation workflows. It allows transformation tasks to be composed hierarchically. The event queue and the notion of model time enable the scheduling of future tasks. By using a shared variable to represent the model under transformation, the overhead caused by communication with transient data packages is eliminated.

The research has a wide range of practical applications. I demonstrate the power of the idea with examples including model optimization and automatic model construction.

<div style="text-align: right">

_____

Professor Edward A. Lee
Dissertation Committee Chair

</div>

To Amy and Raymond.

Contents

List of Figures

Acknowledgments

My achievement is not separable from the help I received from many people.

Let me start with my wife Amy and our adorable child Raymond. I am indebted to Amy for her sacrifice over the years and for the best gift she brought to complete our family. To Raymond, thanks for bringing us enormous fun and developing me into an early riser.

I am deeply grateful to my parents who spent all the time, effort and money on me, giving me education and every other thing I needed.

I must thank Prof. Edward Lee, my advisor, who have helped me on every aspect of my academic life. Without his advice and support, I would have accomplished nothing here. I must also thank Prof. Rastislav Bodik and Prof. Lee Schruben for going over my dissertation. The knowledge I learned from Prof. Rastislav Bodik enabled me to collaboratively create the Ptalon language. The inspiration I received from Prof. Lee Schruben and his event graphs simulation theory led me to develop an important part of my work.

I am also grateful to Prof. Hans Vangheluwe, my former advisor at McGill University. A lot of the ideas on model transformation root on my work with him. Thomas Mandl at Bosch implemented industrial applications for my work and provided me with valuable feedback.

For Ptalon, I should thank its chief developer Adam Cataldo, along with Elaine Cheong and Andrew Mihal. I am also grateful to Yang Zhao who gave me a lot of

insights into discrete-event execution. Gang Zhou and Man-Kit Leung created the code generation framework that I used for performance analysis, and Christopher Brooks and Mary P. Stewart managed the project for all of us.

I would also like to thank my other friends in graduate school, especially Jimmy Su and Lexin Shan, for making my life joyful.

Chapter 1

Introduction

Model transformation is a systematic approach to altering model structures. It can serve as a tool for automatically constructing and maintaining large-scale models. The benefits are obvious compared to manual updates. Much larger models can be handled in a short time. Errors introduced by human mistakes in the tedious editing process can be avoided. The changes can be easily redone, so the transformed models need not be preserved if they are not interesting to model designers or model users.

The idea behind model transformation is related to compiler techniques, if one considers a model as a program specifying certain behavior. Statically evaluating computation in a model is comparable to partial evaluation performed by optimizing compilers. Refactoring a model to obtain a more preferable design roots on the same idea as program refactoring. Automatic model construction and configuration are similar to code generation and linking. The list of examples goes on.

As imperative programs can be represented with program dependency graphs [25], models can be represented with attributed graphs. Model transformation studied in this work is based on graph transformation [55], which is generally applicable to attributed graphs. Therefore, even though the forthcoming discussion focuses on actor models [46], the technique can be applied to other kinds of models such as Statecharts, class diagrams and Petri nets. It can also be applied to imperative programs, for example, as a tool for clone refactoring [40, 39].

A graph transformation is specified with a transformation rule. Application of the rule takes two steps: locating a subgraph in the attributed graph representing the model, and transforming that subgraph to obtain the result. The first step is essentially to solve a subgraph isomorphism problem, which is *NP*-hard in general.

For scalability as well as flexibility, existing model transformation tools provide control mechanisms in addition to graph transformation engines [13]. A control mechanism helps to compose transformation rules to form more complex transformations.

Experiments that I performed with practical applications reveal limitations of existing model transformation tools. A simple control mechanism with fixed priorities suffers from an inability to specify run-time decisions. Finite state machines are not expressive enough and only support bounded state space. Tools using dataflow diagrams frequently create and destroy data packages containing model structures for communication, causing performance overhead.

In this work I develop a more flexible and efficient model transformation technique.

The work consists of two parts:

1. a graph transformation engine for applying transformation rules specified in a visual syntax, and

2. a formal, expressive, efficient and convenient language to model the control of the application of transformation rules.

1.1　Syntax of Actor Models

The visual syntax for actor models used in the Ptolemy project [21, 6] is adopted here. An example is shown in Figure 1.1. An *actor* is an abstraction of computation. It is visually represented with a box. It may have *ports* shown as triangles. A port pointing toward the actor itself is an *input port* and a one pointing away is an *output port*. *Connections* between input ports and output ports designate communication channels. A *single port* represented with a filled triangle accepts at most one connection. A *multi port* represented with a hollow triangle can accept multiple connections. Each connection is associated with a *relation*. On a direct connection between two ports, the relation is usually hidden. On a multi-way connection involving more than two ports, or on a dangling connection to only one port, the relation is explicitly shown as a filled diamond.

Textual and decorative *attributes* can be associated with the model as well as objects contained in the model. A *director* is a special attribute decorated with a
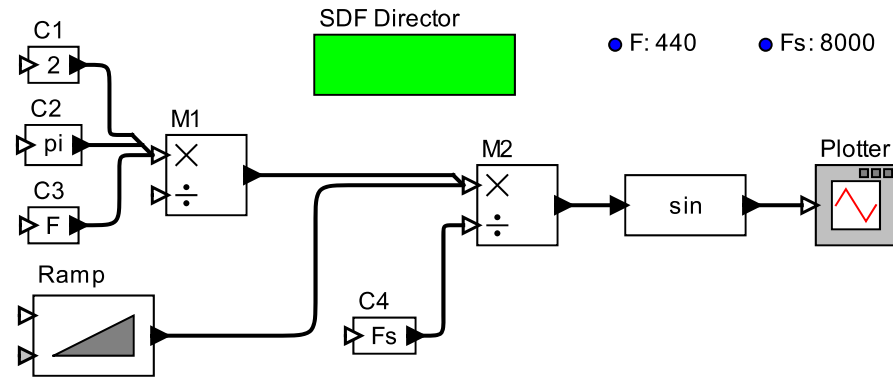
a) The actor model
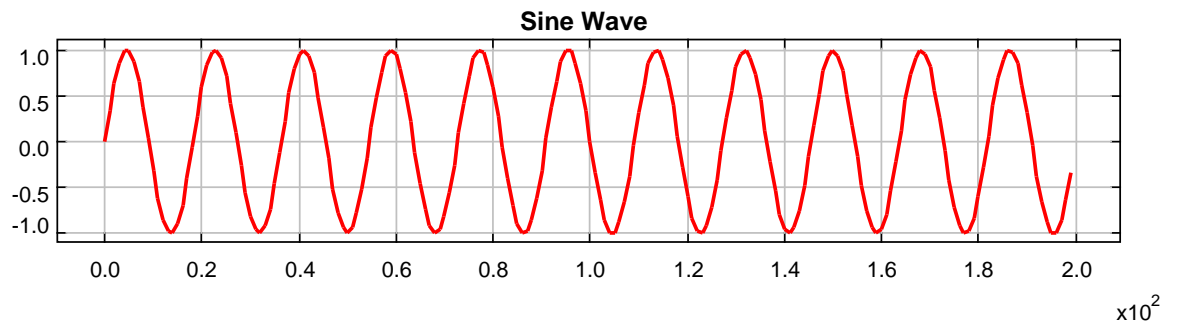
b) The generated sine wave

Figure 1.1: A Sinewave model that can be partially evaluated to improve performance.

filled green box having a name above it. It implements a model of computation for the model that it is associated with.

In a hierarchical design, actors may also be models themselves containing actors inside, in which case they are called *composite actors*. Actors that do not contain actors in them are called *atomic actors*. In a *hierarchical heterogeneous composition*, composite actors may have different directors. That leads to designs that are extremely flexible and well suited for complex software systems [4, 36, 49, 32, 29].

Figure 1.1 motivates model transformation as a tool for automatic model optimization. It is an SDF (synchronous dataflow) model using the SDF director. It generates a sine wave signal with function

$$\sin\left(\frac{2\pi F n}{F_s}\right)$$

The actors with names C1 through C4 are *constant actors*, which output sequences of constant numbers 2, π, F and F_s, respectively. The actual values of F and F_s are defined in the model as *parameters*, visually shown as name-value pairs separated with colons. The constant π (written as "pi" on the constant actor) is an internal constant and need not be explicit defined.

The Ramp actor outputs a sequence of consecutive integer numbers starting from 0. Each number is used as an n to compute a value of the function. M1 and M2 are MultiplyDivide actors that perform multiplication and/or division operations on their inputs and send the results to their outputs. The sin actor computes the sine function. Its output values are sent to the Plotter actor, which plots those values in
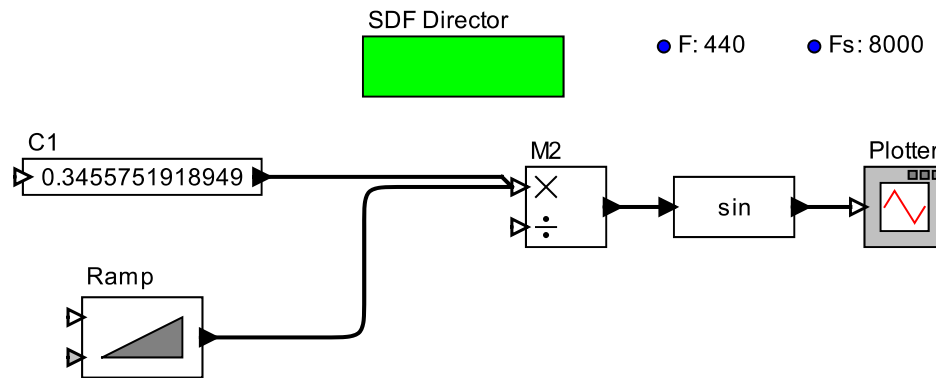
Figure 1.2: The model obtained by partially evaluating the Sinewave model.

a separate window.

Run-time performance of the model can be improved with partial evaluation. Part of the computation can be done statically to reduce the amount of computation required in an execution. Figure 1.2 shows an improved model. The four constant actors in the original model are combined into one with the precomputed value $2\pi F/F_s$.

In general, opportunities of partial evaluation in an initial design are abundant. Manually transforming the design into a more efficient execution model is usually tedious and error-prone. The difficulty also lies in the fact that when the design or the data values in it are updated, the transformation needs to be redone to keep the execution model synchronized.

The model transformation technique that I develop provides a scalable and systematic approach to solve the problem. By creating a transformation and applying it to the design with a transformation engine, a model optimized for execution per-

a) Pattern   b) Replacement

	Pattern Object	Replacement Object
1	C1	C
2	M	M
3	Relation	Relation

c) Correspondence
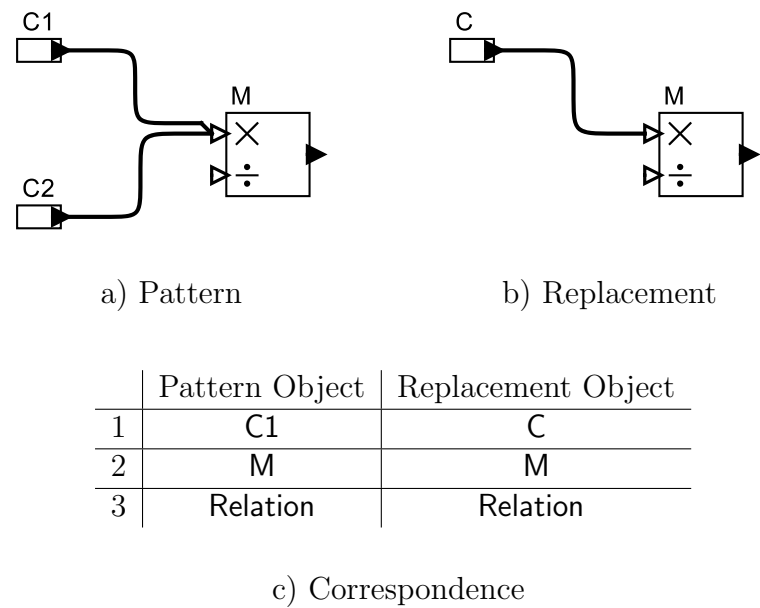
Figure 1.3: A transformation rule to eliminate a multiplication operation in the Sinewave model.

formance can be generated automatically. The transformation is reusable. When the design is changed, the transformation can be reapplied to get an updated execution model.

1.2 Specification of Transformation Rules

To transform a model, a transformation rule needs to be defined. I created a syntax for transformation rules that is close to the visual language that designers use to design actor models. This avoids the need for learning a new language and also reduces the risk of errors due to misunderstanding.

Figure 1.3 shows one of the transformation rules for optimizing the model in

Figure 1.1. It matches any two constant actors connected to a MultiplyDivide actor, such as C1, C2 and M1. The effect is to replace them with a single constant actor connected to the MultiplyDivide actor.

Part a) and part b) of the rule show the *pattern* and the *replacement*, respectively. They are similar to actor models except the following.

- In addition to actors, boxes representing *matchers* can also be included in the pattern, which are essentially placeholders to match different types of actors.

  In this rule, C1 and C2 are matchers, whereas M is an ordinary actor. C1 and C2 have *criteria* to constrain them to match constant actors only. Other criteria of C1 and C2 ensure that the matched actors have output ports.

- In the replacement, objects may correspond to those in the pattern, meaning that the matched objects are preserved in the result. An object with no correspondence would be created in the result.

  In this rule, C, M and the relation in the replacement correspond to C1, M and the relation between C1 and M in the pattern, respectively. In addition, C has an *operation* to set its value to be equal to the result of the multiplication.

Part c) of the transformation rule specifies *correspondence* between objects in the replacement and those in the pattern. Designers are not required to manually edit this table. Instead, the design environment keeps track of the copy and paste operations between the pattern and the replacement, and automatically creates entries in the

table.

Compared to model transformation tools providing separate languages for specifying transformation rules, such as AGG [58] and GReAT [1] using class diagrams, this syntax is arguably clearer and more intuitive for model designers.

1.3   Transformation Engine

Given a model and a transformation rule as inputs, the transformation engine applies the rule on the model if it finds a match to the pattern. It produces the transformed model as output. Objects that only exist in the replacement with no correspondence in the pattern are created in the result. Objects only in the pattern with no correspondence in the replacement are removed. Operations in the replacement are executed.

If multiple matches are found, some strategy is needed to make a choice among them. Without specific reasons for preferring one match over another, a common choice would be the first one that is discovered, so that the computationally expensive pattern matching process can stop immediately.

Consider the transformation rule in Figure 1.3 being applied to the model in Figure 1.1. Suppose the first match consists of C1, C2 and M1. The result would be to replace the two constant actors with a single constant actor whose value equals 2π. The same transformation rule can also be applied to the match consisting of C1, C3 and M1, and that consisting of C2, C3 and M1.

As part of this work, I adapt the single pushout approach [52, 18] in graph transformation theory to formalize transformation rules. I show that any actor model can be systematically converted into an attributed graph. I define transformation from one such attributed graph into another. I extend the classical theory with criteria and operations that can be associated with objects in the pattern and the replacement. Finally, I explore two approaches to specifying transformation rules with variable structures. I introduce selective tags to allow objects in the pattern to be optional. I enhance Ptalon, a higher-order model composition language, to specify complex parametrizable transformation rules with compact descriptions.

1.4   Event-Oriented Control Language

A variety of control mechanisms and languages have been incorporated into contemporary model transformation tools. They play an important role in the effectiveness of those tools, since designers tend to keep individual transformation rules simple and to compose them to form more complex transformations. Examples of control include priorities in AGG [58] and AToM^3 [41], story diagrams [26] (which are essentially Statecharts composed with activity diagrams) in FUJABA [53], abstract state machines [30] in VIATRA2 [2], hierarchical dataflow diagrams in GReAT [1], and imperative programs in PROGRES [57].

My work aiming for a more flexible and efficient control language is motivated by the limitations of existing approaches. Priorities and state machine based approaches

are not expressive or convenient enough. Dataflow diagrams incur run-time overhead for the creation and destruction of large data packages. Imperative programs depart from the visual syntaxes that model designers are familiar with, and their lack of abstraction makes reasoning of formal properties difficult.

I created the Ptera (Ptolemy event relationship actor) model of computation based on event graphs [56] as the control language. I operationally define the semantics of Ptera models, and show that Ptera can be composed with other models of computation such as SDF (synchronous dataflow) [44], DDF (dynamic dataflow) [61], HDF (heterochronous dataflow) [28], SR (synchronous reactive) [3] and DE (actor-oriented discrete-event) [42] in a hierarchical heterogeneous composition. Furthermore, besides applying it to model transformation, I demonstrate that Ptera as a general-purpose model of computation can also be used to model discrete-event processes.

As an example to show the use of this control language, a transformation workflow is provided in Figure 1.4. The round-corner boxes in the workflow are *events*. The edges are *scheduling relations*. They represent causality relations, which mean that the start events cause the end events to occur.

The workflow is hierarchical, because the Transform event is associated with a submodel shown in the lower part of the figure. The submodel contains three events that perform model transformation. The SimplifyMultiply event applies the transformation rule in Figure 1.3.

The transformation rules for the SimplifyDivide event is shown in Figure 1.5. In
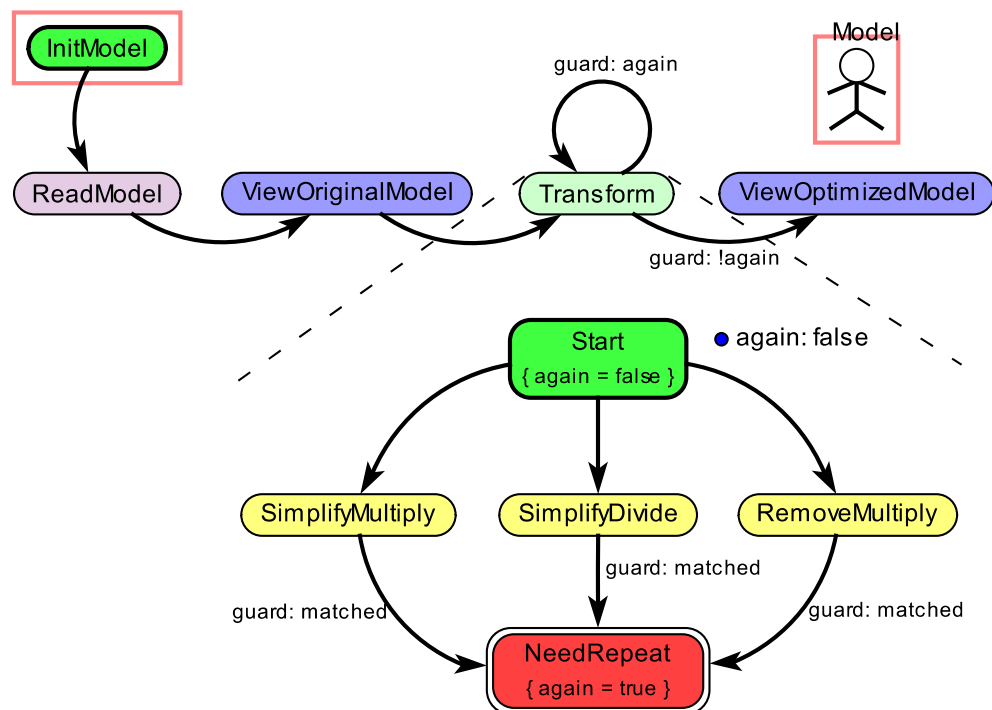
Figure 1.4: A transformation workflow to partially evaluate the Sinewave model.
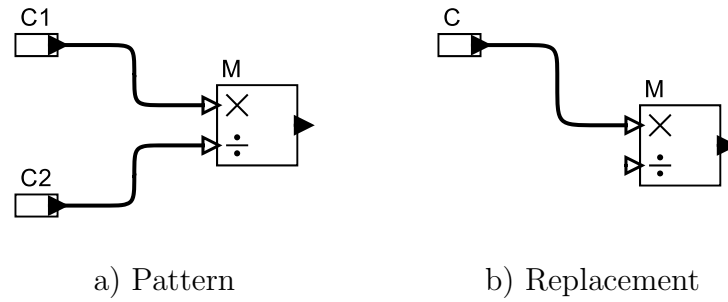
a) Pattern   b) Replacement

	Pattern Object	Replacement Object
1	C1	C
2	M	M
3	Relation	Relation

c) Correspondence

Figure 1.5: A transformation rule to eliminate a division operation.

● Constraint:
  M.multiply.getWidth() == 1 &&
  M.divide.getWidth() == 0 &&
  M.output.getWidth() == 1

a) Pattern   b) Replacement
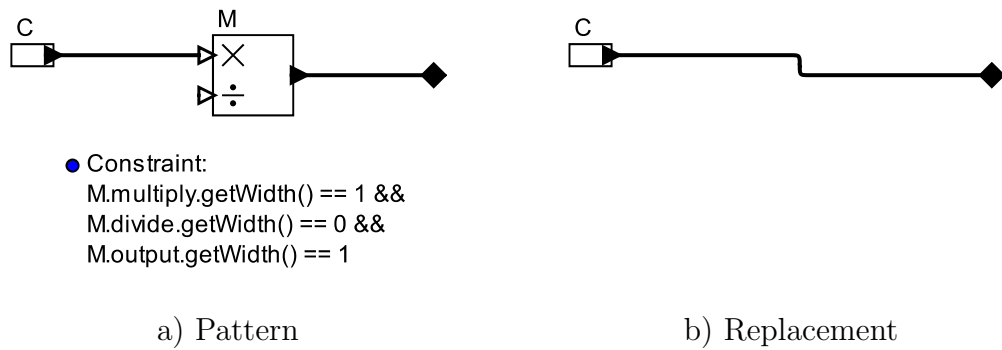
	Pattern Object	Replacement Object
1	C	C
2	Relation	Relation

c) Correspondence

Figure 1.6: A transformation rule to remove a MultiplyDivide actor.

the replacement, C has a hidden operation to compute its value, which should be equal to C1's value divided by C2's value. Figure 1.6 shows the transformation rule for RemoveMultiply. The textual constraint in the pattern further restricts pattern matching, requiring that M's multiply input port has exactly one connection, that its divide input port is not connected, and that its output port has exactly one connection. Only in that case it is safe to remove the MultiplyDivide actor and to directly connect the constant actor's output to the destination.

At the beginning of the workflow in Figure 1.4, the InitModel event occurs because it is an initial event with a thick border and green fill color. It sets the Model variable with an empty model. Due to the scheduling relation, finish of the InitModel event causes the ReadModel event to occur. ReadModel has an invisible parameter specifying the location of the model to be transformed on the disk. In this case, the model is the one in Figure 1.1. After the model is read, it is stored in the Model variable. ReadModel then triggers ViewOriginalModel, whose action causes a separate window to be opened with the untransformed model in it.

The workflow then enters a loop consisting of the Transform event. Each time Transform is processed, its submodel is executed and one of the three transformation rules is applied. The variable named "again" in the submodel records whether a transformation rule was successful. If that is the case, the Transform event schedules itself again, so the the submodel is also executed again.

The loop finishes when none of the transformation rules is successful. At that

point, the Transform event does not schedule itself but the ViewOptimizedModel event, which displays the optimized model as the one in Figure 1.2.

As a control language for model transformation, Ptera has the following advantages over existing approaches.

- The Model variable encapsulates the model to be transformed, which is commonly accessible by all the transformation tasks. There is no need to create data packages for communication.

- A workflow is similar to an activity diagram but is more expressive. Events can be scheduled in an unbounded event queue with time stamps.

- A workflow can be hierarchical, so that more complex tasks can be created by composing simpler ones. Design reuse is convenient.

- A workflow is itself a model, which can also be transformed.

Chapter 2

Basic Transformation

A basic transformation captures an atomic update on a model. To apply graph transformation theory, I convert the model into an attributed graph. The result of a graph transformation is another attributed graph that can be converted back into a new model.

2.1 Visual Syntax

I created a visual syntax for basic transformations that is close to the visual language for actor models, so model designers can easily get familiar with it.

A basic transformation is specified with a *transformation rule*, which is similar to a rewrite rule in a context-free grammar. A transformation rule has three components: a *pattern* graph, a *replacement* graph, and the *correspondence* between the vertices and edges in those two.

Figures 1.3, 1.5 and 1.6, which have been discussed in the introduction, show examples of transformation rules. Repeatedly applying those rules on the model in Figure 1.1 causes the model to be partially evaluated and a reduced-size model with equivalent behavior to be obtained as in Figure 1.2.

In the pattern in Figure 1.3a, C1 and C2 are *matchers*, which are placeholders to match distinct actors in the given model. They are shown as white boxes because they are associated with criteria (a special kind of attributes) that establish subclassing relationship from constant actors. Due to the subclassing relationship, those matchers can only match constant actors, and they acquire the default icon of constant actors. Two other criteria of C1 and C2 require them to have output ports. The object M is an ordinary actor. No criterion is needed for it. It matches only actors of the same kind.

The names of the object in the pattern are insignificant and need not be the same as those of the matched objects in the model. Names of the relations and ports are usually hidden for brevity. In this case, the name of the relation between C1 and M in the pattern is "Relation." That relation is hidden because it is on a direct connection between two ports. It would have been shown as a black diamond if there were more than two ports involved.

The connections in the pattern require that corresponding connections exist between the matched ports and relations in the model.

In the replacement in Figure 1.3b, C, M and the relation between them are pre-

served, which correspond to C1, M and the relation between the two in the pattern. Because of this, the matched objects are not removed in the result of the transformation. Objects that appear only in the pattern but not the replacement, such as C2 and the relation between C2 and M, cause the matched objects in the model to be removed. Objects that appear only in the replacement cause new objects to be created. In general, designers of a transformation rule can specify object creation and removal by simply editing the replacement starting from a copy of the pattern.

The third component of the transformation rule, the correspondence table as shown in Figure 1.3c, establishes relations between objects in the pattern and those in the replacement. For brevity, I list only entries for actors and relations, but omit those for ports.

2.2   Graph Representation of Actor Models

Figure 2.1 demonstrates how a hierarchical actor model, or the pattern or replacement of a transformation rule, can be converted into an attributed graph. In the graph, vertices represent actors, ports and relations, and edges represent either containment relations or connections. The styles of the vertices denote their kinds encoded with attributes. Actors are shown as big hollow circles, ports are shown as small hollow circles, and relations are shown as filled dots. Edges have two types, also encoded with attributes. A dashed edge represents a containment relation, where the target vertex is semantically contained by the source vertex. A solid edge represents
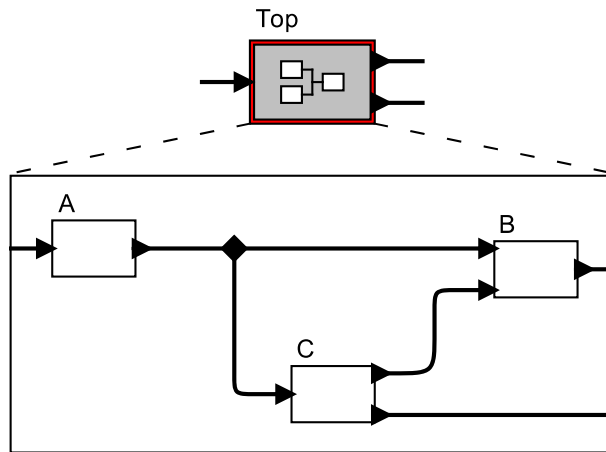
(a) An portion of a hierarchical actor model.
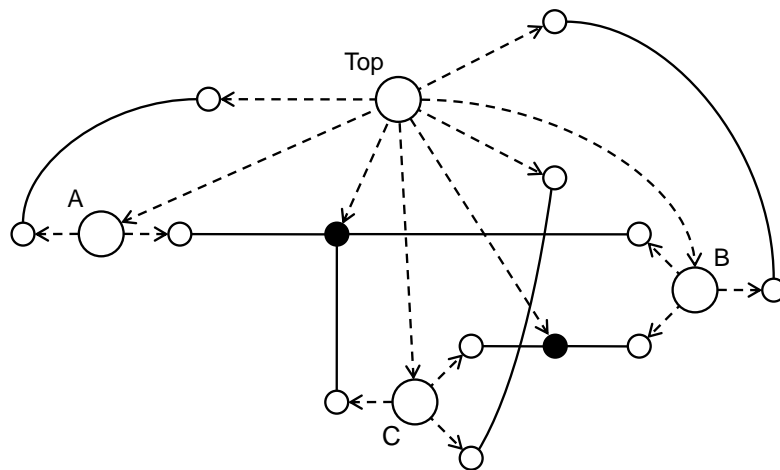
(b) Attributed graph representation.

Figure 2.1: Hierarchical actor model and its representation as an attributed graph.

a connection between a port and a relation, or that between two ports at adjacent levels of the model hierarchy. A connection is always undirected and the direction of communication is determined by the type of its port, which is input, output or both. Port types are also attributes, which are not explicitly shown in the graph.

The set of attributes can be easily extended. For the purpose of model transformation, the meaning of the attributes is unimportant because they are only used for detecting matches. Their existence is all that matters.

In general, any actor model can be converted into a graph in a similar way. This allows graph transformation techniques to be applied to modify the model structure. A transformation rule also can be converted into a graph, with a subgraph as its pattern and another disjoint subgraph as its replacement, and with edges between the two subgraphs as the correspondence relations. Therefore, transformation rules and models are unified in the graph representation.

2.3 Graph, Morphism and Transformation

Formulation of graph transformation follows the single pushout approach. This section provides a review of the theory in [12] and [18].

A *graph G* is a tuple $\langle V_G, E_G, s_G, t_G \rangle$. V_G is the set of vertices. E_G is the set of edges. $s_G : E_G \rightarrow V_G$ is a total function that maps edges to their source vertices. $t_G : E_G \rightarrow V_G$ is a total function that maps edges to their target vertices.

Compared to the work in [12] and [18], this section discusses unlabeled graphs,

where vertices and edges are not associated with labels (or are all associated with the same label). This notion of graphs is extended in the next section with attributes.

A *total graph morphism* m from graph G to H, denoted by $G \xrightarrow{m} H$, is a pair of total functions $\langle m_V : V_G \to V_H, m_E : E_G \to E_H \rangle$ that preserves graph structure, i.e., $m_V \circ s_G = s_H \circ m_E$ and $m_V \circ t_G = t_H \circ m_E$. If both m_V and m_E are bijective, then m is called an *isomorphism* from G to H. If both m_V and m_E are injective, then m is injective, and G *matches a subgraph of* H.

A *partial graph morphism* or simply *morphism* m from graph G to H, denoted by $G \xrightarrow{m} H$, is a total graph morphism from a subgraph of G to H. The *composition* of morphisms $G \xrightarrow{m} H$ and $H \xrightarrow{n} I$ is represented with $G \xrightarrow{n \circ m} I$.

Given graphs A, B, C, and morphisms $A \xrightarrow{b} B$ and $A \xrightarrow{c} C$, a *pushout* is a tuple $\langle D, B \xrightarrow{g} D, C \xrightarrow{f} D \rangle$, where D is a graph and morphisms $B \xrightarrow{g} D$ and $C \xrightarrow{f} D$ satisfy the following conditions.

1. $g \circ b = f \circ c$

2. For any graph D' with morphisms $B \xrightarrow{g'} D'$ and $C \xrightarrow{f'} D'$ satisfying $g' \circ b = f' \circ c$, there exists a unique morphism $D \xrightarrow{h} D'$ such that $h \circ g = g'$ and $h \circ f = f'$.

When there is no confusion, graph D is also called the *pushout of* $A \xrightarrow{b} B$ and $A \xrightarrow{c} C$.

It is shown in [18] that the pushout of morphisms $A \xrightarrow{b} B$ and $A \xrightarrow{c} C$ always

exists and can be constructively computed.

A *transformation rule* T is defined with a morphism $L \xrightarrow{c} R$. The graph L is called the *pattern* (or left-hand side). The graph R is called the *replacement* (or right-hand side).

Given transformation rule $T = L \xrightarrow{c} R$ and an input graph G, if an injective total graph morphism $L \xrightarrow{m} G$ exists (i.e., L matches a subgraph of G), then T *is applicable to* G. The *direct derivation* from G with T at m is a graph H, which is the pushout of $L \xrightarrow{c} R$ and $L \xrightarrow{m} G$ [52, 18].

This is called the single pushout (SPO) approach because there is only one pushout involved in the derivation, as opposed to two pushouts in the double pushout (DPO) approach [19, 12, 31]. SPO is more expressive than DPO because dangling edges are automatically removed in the derivation, whereas in DPO, deletion of a vertex is forbidden if it leads to dangling edges (known as the *dangling edge condition*).

2.4 Attributes

To transform models using graph transformation, it is necessary to categorize vertices and edges with types. (Recall that three types of vertices and two types of edges are used in Figure 2.1.) It is also necessary to take into account other attributes that further characterize vertices and edges, such as the ones that decide whether a port (represented with a vertex) is input, output, or both.

With A being a global set of attributes, graph G can be extended to be *attributed*

graph $\langle V_G, E_G, s_G, t_G, \mathcal{A}_G \rangle$, where $\mathcal{A}_G : (V_G \cup E_G) \to 2^A$ is a total function that returns a (possibly empty) set of attributes associated with each vertex or edge. Attributed graphs can be represented with E-graphs, which are introduced in [20] to formalize attributed graph transformation. An unattributed graph as discussed in the previous section can be considered as an attribute graph G where $\mathcal{A}_G$ returns empty set for all inputs.

An *attributed graph morphism* from attributed graph G to H is a morphism $G \xrightarrow{m} H$ such that for any $x \in V_G$. $\mathcal{A}_G(x) \subseteq \mathcal{A}_H(m_V(x))$ and for any $x \in E_G$. $\mathcal{A}_G(x) \subseteq \mathcal{A}_H(m_E(x))$. That is, an attributed graph morphism maps vertices and edges to the ones that have at least the same attributes.

Attributed graph transformation can be defined with a pushout of attributed graph morphisms.

Figure 1.3a provides an example of attributes. M in the pattern is a Multiply-Divide actor. It can be represented as a vertex associated with an attribute that identify MultiplyDivide actors. Any actor in the input model that matches M must also have that attribute. Such an actor can be a MultiplyDivide actor or an actor in a subclass of MultiplyDivide actors.

2.5   Criteria and Operations

As a further extension, in a transformation rule, vertices and edges in the pattern may be associated with *criteria*, and vertices and edges in the replacement may be

associated with *operations*. Criteria restrict the matching from the pattern to the input graph. After the output graph is generated, operations specified on vertices and edges in the replacement are performed on the corresponding vertices and edges in the output graph.

Take the transformation rule in Figure 1.3 as an example. C1 in the pattern has two criteria. The first one ensures that any actor matched by C1 must be a constant actor or an actor in a subclass of constant actors. (The same effect can be achieved with an attribute.) The second criterion of C1 is satisfied only when the matched actor has at least one output port.

In the replacement, C has an operation that computes the value of the constant actor remaining after transformation as a function of the values of the two constant actors before transformation. In the implementation, expression "`C1.value * C2.value`" in the Ptolemy expression language is used to specify this computation. Notice that names in the pattern can be referred to from the replacement, if those names do not exist in the latter. Moreover, C1 and C2 in the pattern are matchers and they do not have values. To evaluate the expression, a match to the model needs to be given, and the actual actors matched to C1 and C2 are used as the scopes in which the values are searched for.

Textual constraints can be associated with the pattern. One such constraint was seen in Figure 1.6a. That constraint may be considered as a criterion associated with the top level of the pattern.
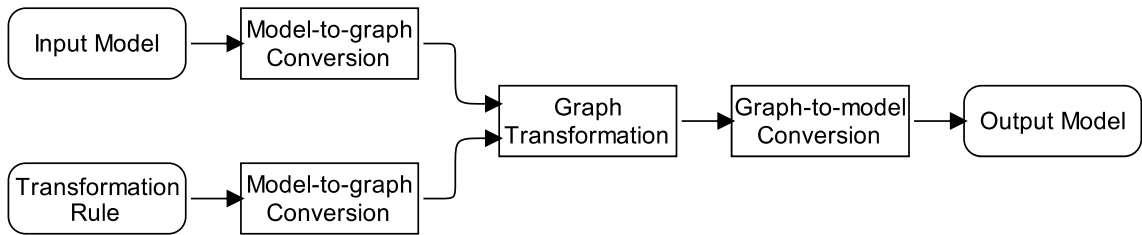
Figure 2.2: The model transformation process.

2.6 Model Transformation Based on Graph Transformation

Figure 2.1 demonstrates a way to represent any model or transformation rule with an attributed graph. In the attributed graph, three special attributes are assigned to the vertices to distinguish their types: `Actor` (visually represented with big hollow circles), `Port` (small hollow circles) and `Relation` (filled dots). Two additional attributes identify the types of the edges: `Containment` (dashed arrows) and `Connection` (solid lines).

Based on the graph representation, I establish a model transformation process as depicted in Figure 2.2. The inputs to the process consist of a model and a transformation rule, both specified in the modeling language. The two inputs are then converted into attributed graphs. A vertex is created for each actor, port or relation, and an edge is created for each connection or containment relation.

The transformation rule is converted into graphs with corresponding elements.

Its pattern and replacement are converted into the L graph and the R graph in transformation $L \xrightarrow{c} R$. The correspondence table defines c, which is a partial graph morphism. The "Pattern" column shows the names of some objects in the pattern. For a hierarchical transformation rule, the names may contain parts separated by dots, referring to the unique identifiers at different levels. The "Replacement" column shows the names of the corresponding objects in the replacement.

After the conversion, graph transformation is applied. The result is converted back into a model for output.

2.7 Extensions

To improve convenience and flexibility, I have introduced extensions to traditional model transformation techniques.

2.7.1 Hierarchy

Hierarchies are common in actor models. It is necessary to provide support for them in the visual syntax for transformation rules, even though any transformation rule or model with hierarchy can be represented with a flat graph as shown in Figure 2.1.

A special matcher called CompositeActorMatcher is implemented to match a container in a hierarchy. An arbitrary number of such matchers can be used in a pattern.

They may contain matchers and actors inside. In particular, the pattern itself is considered as a top-level CompositeActorMatcher.

To match a pattern to an input model, the transformation engine tries to find an extra level of the model hierarchy for each CompositeActorMatcher. The pattern itself as the starting point can match any composite actor in the model. From the viewpoint of transformation designers, the concept of subgraph isomorphism is enriched in that the pattern appears to be hierarchical, which matches a hierarchical subgraph of the model.

Levels of a model hierarchy can be created or removed with a transformation rule. If an extra level is found in the replacement, the transformation engine creates a composite actor in the model and copies the inside contents from the replacement into it. If a CompositeActorMatcher is found in the pattern but not in the replacement, the transformation engine removes the matched composite actor from the model, and moves its contents to its container. This in effect removes the "wrapper" of the matched level of the model hierarchy.

If a CompositeActorMatcher has a correspondence in the replacement, then the level of hierarchy is kept. Designers may specify moving objects inside or outside of the matched composite actor by performing the corresponding movements in the replacement.

This representation for hierarchy is convenient for such transformations as hierarchy flattening and movement of objects between different levels.

2.7.2 Alternative Visual Syntax

In an alternative visual syntax for some transformations, the pattern may be enhanced with extra annotations, and the replacement and correspondence table may be omitted. This can be done by merging the pattern and the replacement into one and tagging the objects with different color codes depending on whether they are only in the pattern, only in the replacement, or in both. If an object is tagged to be in both the pattern and the replacement, it may be associated with criteria and operations at the same time.

Though this syntax is usually more concise, it has practical limitations and is thus provided only as a shorthand. The color codes are intuitive to some but probably not all designers, and it is not possible to specify movements of objects between levels of the model hierarchy.

2.7.3 Selective Tags

Three types of *selective tags* can be associated with any object in the pattern: optional, negated and ignored.

If an object is *optional*, the transformation engine first tries to find a subgraph in the model that has a match for it. If the attempt fails, the pattern matching algorithm backtracks and ignores the object in a second attempt. Multiple objects may be tagged optional, and the points for backtracking are maintained in a stack in a deterministic total order for those objects.

If an object is *negated*, it must not be found in the subgraph for the pattern to match.

If an object is *ignored*, it is not considered by the transformation engine, as if it were not in the pattern. This tag is especially useful for testing purpose.

2.7.4 Higher-Order Specification in a Declarative Language

Ptalon is a higher-order composition language for actor models [7, 8]. It allows model designers to use higher-order descriptions to construct models from model fragments as first-class citizens. It is a declarative language that focuses on the structure of the constructed model rather than the precise procedure with which the model is constructed.

Ptalon is useful for designers who intend to create large-scale models or models with parametrizable structures. With an iterative language construct, actors can be created in a loop controlled by a loop variable. Parameters may be defined in a description that affect the way in which the model is constructed. A typical example is to define the number of repeatedly created parts of a model as a parameter. Value of the parameter is supplied by the model user. This allows the description to remain concise even if the model that it constructs may grow arbitrarily large.

To bring the benefits of declarative higher-order composition into model transformation, I enhance Ptalon and leverage it to specify transformation rules. Originally, a Ptalon description has a single top-level block following a model name. The block
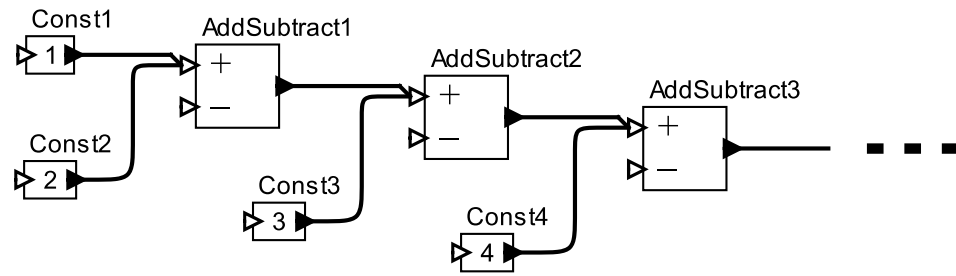
a) Before transformation
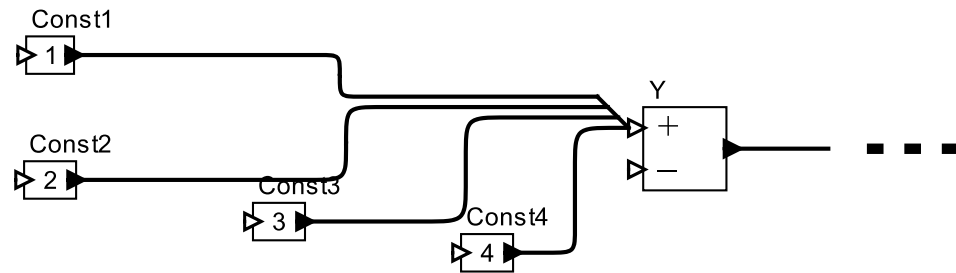
b) After transformation

Figure 2.3: Simplifying a model with a 4-input adder.

defines the structure of model. With my extension, two top-level blocks exist in a description following the name of a transformation rule, one defines the pattern and the other defines the replacement. The two blocks share a name space, so objects in the pattern can be easily referred to from the replacement.

To exemplify the usage, consider the scenario in Figure 2.3, where a model with multiple add operations needs to be reduced into one with less add operations. Figure 2.3a shows part of the model before transformation, which uses the AddSubtract actor with two input channels at its plus port. Suppose that the AddSubtract actor has now been improved to accept up to n inputs and to sum all the inputs in one oper-

ation. Model designer may want to use a transformation rule to automatically update the model so that it takes advantage of the new implementation to save execution time and memory space. The desired result for $n = 4$ is shown in Figure 2.3b.

The transformation rule is specified with the description in Table 2.1. The value of parameter `n` is unspecified, so the description can be reused with different `n`'s depending on the actual implementation of AddSubtract provided. The first top-level block following name "`Transformation`" and keyword "`is`" defines the pattern. A loop is used to create `n` constant actors and $n - 1$ addition operations. A pair of double square brackets encloses a Ptolemy expression to be evaluated. For example, "`a[[0]]`" is equivalent to "`a0`" and "`r[[k * 2 - 1]]`" is equivalent to "`r1`" when `k` is equal to 1. This helps to generate unique names for the objects in the model.

The second block following the "`=>`" sign defines the replacement. Some relations and all the constant actors are preserved with the "`preserve`" keyword. Unpreserved objects would be removed as an effect of the transformation. Fresh objects defined in the replacement would be created.

Instead of the "`=>`" sign, the pattern block and the replacement block may also be separated with the "`=>+`" sign. If that is the case, all objects in the pattern are considered to be preserved, except those that are explicitly declared to be deleted with the "`delete`" keyword.

```
Transformation is {
  parameter num = [[n]];
  actor X = ptolemy.actor.lib.Const;
  actor Y = ptolemy.actor.lib.AddSubtract;

  relation r[[0]];
  a[[0]] := X(output:=r[[0]]);
  for k initially [[1]] [[k < num]] {
    relation r[[k * 2 - 1]];
    a[[k * 2 - 1]] := X(output:=r[[k * 2 - 1]]);
    relation r[[k * 2]];
    port reference yInput[[k]];
    a[[k * 2]] := Y(plus:=yInput[[k]], output:=r[[k * 2]]);
    this(yInput[[k]]:=r[[k * 2 - 2]]);
    this(yInput[[k]]:=r[[k * 2 - 1]]);
  } next [[k + 1]]
} => {
  preserve r[[0]];
  preserve a[[0]];
  for k initially [[1]] [[k < num]] {
    preserve r[[k * 2 - 1]];
    preserve a[[k * 2 - 1]];
  } next [[k + 1]]
  preserve r[[num * 2 - 2]];

  port reference newYInput;
  Y(plus:=newYInput, output:=r[[num * 2 - 2]]);
  for k initially [[0]] [[k < num]] {
    if [[k == 0]] {
      this(newYInput:=r[[0]]);
    } else {
      this(newYInput:=r[[k * 2 - 1]]);
    }
  } next [[k + 1]]
}
```

Table 2.1: Ptalon description for model simplification with 4-input adders.

2.8 Requirement for Efficient Control

The complexity of basic transformations lies in pattern matching. The problem of finding a subgraph matched by a pattern is known to be *NP*-hard. Moreover, control cannot be specified in a transformation rule, which makes it impossible to progressively transform a model depending on the state and external inputs. Therefore, basic transformations are practically restricted to expressing simple transformations, and a control mechanism is required for more complex ones.

A first attempt to a control mechanism is to encapsulate a basic transformation in a purely functional actor that takes models as inputs and sends transformed models as outputs. Such an actor can be used in any actor-oriented model of computation, such as dataflow and DE (discrete-event). However, this approach is not efficient enough. Control structures such as loops and branches are hard to express in dataflow. The creation of tokens to be sent and received between actors incur run-time overhead that limits performance, especially when the tokens are large in size such as those encapsulating model structures. A better approach is needed, and that is the motivation for introducing the Ptera model of computation.

Chapter 3

Ptolemy Event Relationship Actors

Motivated by the need for a control language for model transformation, I created the Ptera (Ptolemy event relationship actor) model of computation based on event graphs [56]. By using this control language, designers can create scalable transformation workflows containing basic transformations as computation units. More complex transformations can be constructed in this way.

As a general-purpose programming language, Ptera is not limited to model transformation. For that reason, I discuss the syntax and semantics of Ptera models here without putting them in the context of model transformation. Discussion on the application to model transformation is postponed to the next chapter.

There are two types of Ptera models. A flat Ptera model has no hierarchy. It can be represented with an attributed graph containing vertices connected with directed edges. Vertices may be associated with such attributes as *ID, actions, final, initial*
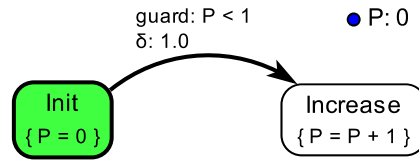
Figure 3.1: A simple model with two events that can be executed with a size-one event queue.

and *parameters*. Edges may be associated with such attributes as *ID*, *arguments*, *canceling*, *delay*, *guard*, *initializing*, *priority* and *triggers*. All these attributes are included in the following discussion.

A hierarchical Ptera model is a generalization of flat models where the vertices in it may be additionally associated with an attributed graph that represents a *submodel*. Submodels may themselves be hierarchical Ptera models, and in general, they may also be other types of models, such as FSMs (finite state machines), actor models, class diagrams and even imperative programs. The only requirement is that their behavior can be defined in an abstract semantics framework to be discussed later.

3.1   Flat Models

Flat models are a simpler kind of models in which vertices are not associated with submodels.

3.1.1 Introductory Examples

A vertex in a model is an *event*. A example model with two events is shown in Figure 3.1. Each event has a unique *identifier* (*ID*), which is displayed on its icon. In this case, the events' IDs are Init and Increase.

Variables in a model associate values with names. Each variable is visually represented as a name-value pair to the right of a dot. In the example, there is a single variable with name P and initial value 0. The initial value is provided by the model designer and is shown in the static model design. During execution, the variable may be updated with new values.

Init is an *initial event* (with the *initial* attribute set to true). This is indicated with a filled vertex having a thick border. At the beginning of an execution, all the initial events are scheduled to occur at model time 0. (As discussed later, even though they all conceptually occur at time 0, there is a well-defined order.) An *event queue* exists in the execution that can hold an unbounded number of scheduled events. An event is removed from the event queue and is processed when the model time reaches the time at which the event is scheduled to occur (called the *time stamp* of that event).

Events may be associated with *actions* that are specified with a list of assignments separated by semicolons. In Figure 3.1, Init has action "P = 0", which causes the side effect of setting variable P to 0 when Init is processed. The edge from Init to Increase is called a *scheduling relation*. It has Boolean expression "P < 1" as its *guard* and 1.0 as its *delay* represented by symbol δ. This means, after the Init event is processed, if
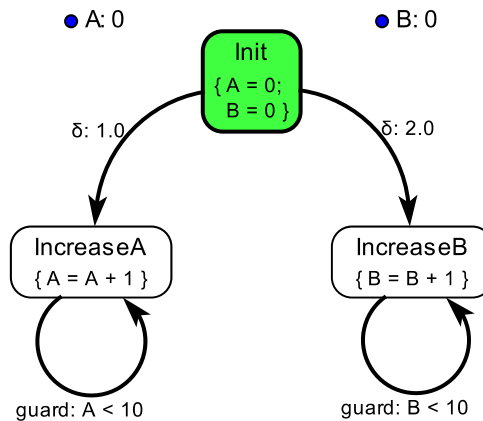
Figure 3.2: A model with multiple events in the event queue.

P's value is less than 1 (which is true in this case), then Increase would be scheduled 1.0 unit of model time later than the current model time (which is 0). When Increase is processed at model time 1.0, its action "P = P + 1" is executed and P's value is increased to 1.

After processing Increase, the event queue becomes empty, since no more event is scheduled, so the execution terminates. In general, execution terminates when there is no event left in the event queue.

In the model in Figure 3.1, it is clear that there is at most one event in the event queue (either Init or Increase) at any time. In general, an unbounded number of events can be scheduled in the event queue.

As another example, execution of the model in Figure 3.2 requires an event queue of size greater than 1. The Init event schedules IncreaseA and IncreaseB to occur after 1.0 unit of model time and 2.0 units of model time, respectively. The guards of the
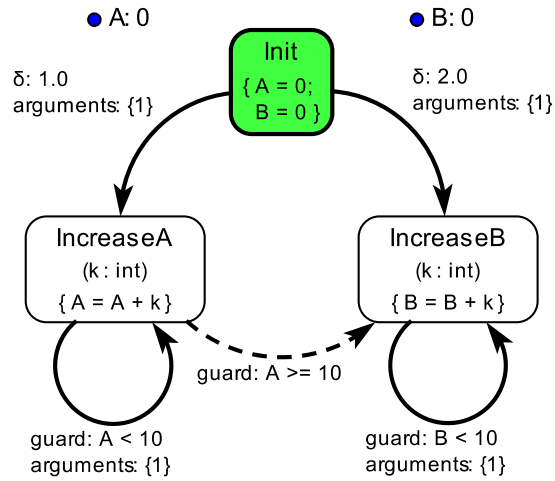
Figure 3.3: A model with parameters for the events and a canceling relation.

two scheduling relations from Init take the default value "true," and are thus hidden in the visual representation. When IncreaseA is processed, it increases variable A by 1 and reschedules itself, until A's value reaches 10. The model-time delay δ on the scheduling relation from A to itself is also hidden, because it takes the default value "0.0," which means the event is scheduled at the current model time. Similarly, IncreaseB repeatedly increases variable B at the current model time, until B's value reaches 10.

3.1.2 Parameters

Resembling a C function, an event may have a list of comma-separated formal *parameters* declared for it. Those parameters have distinct names. Acceptable value types are also specified in the list.

Figure 3.3 shows a model modified from Figure 3.2. (The dashed edge in the figure is a canceling relation, which will be discussed next.) Parameters with name "k" are defined for two events. Those parameters are used as increments for the variables in each step.

Each scheduling relation pointing to an event with parameters must specify a list of expressions in its *arguments* attribute. Those expressions are used to compute the actual values for the parameters when the event is processed. In the example, all scheduling relations pointing to IncreaseA and IncreaseB specify "{1}" in their arguments attributes, meaning that the parameters should take value 1 when those events are processed.

Values of the parameters declared by an event can be accessed in the event's actions and the guards and delays of the scheduling relations from that event.

3.1.3 Canceling Relations

A *canceling relation* is represented as a dashed edge between events. It can be guarded by a Boolean expression. Its delay must be 0 and its arguments must be an empty list "{}," which are both hidden.

When an event with an outgoing canceling relation is processed, if the guard is true and the event pointed to has been scheduled in the event queue, then that scheduled event would be removed from the event queue immediately without being processed. This yields the effect of canceling a previously scheduled event. If the event pointed

to is scheduled multiple times, with multiple *instances* of it in the event queue (each of which belongs to the same event but may be associated with a different list of arguments), then the canceling relation causes only the first one to be removed. If the event pointed to is not scheduled, the canceling relation take no effect.

Figure 3.3 provides an example of canceling relation. Processing the last IncreaseA event (at time 1.0) causes IncreaseB (scheduled to occur at time 2.0 by the Init event) to be cancelled. As a result, variable B is never increased.

Canceling relations do not increase expressiveness. In fact, a model with canceling relations can always be converted into one without canceling relations, as is shown in [35].

3.1.4 Simultaneous Events

Simultaneous events are events in a model that potentially have instances coexisting in the event queue and scheduled to occur at the same model time.

To motivate, consider setting both δ's in Figure 3.3 to 1.0, which yields the model in Figure 3.4. That makes IncreaseA and IncreaseB simultaneous events. Notice that instances of simultaneous events may not always occur at the same time. For example, if the δ's were set to $a + b$ and $a * b$ respectively, where a and b's values are not determined, then only some instances of IncreaseA and IncreaseB would occur at the same time. Moreover, even though multiple instances of IncreaseA occur at the same time, they do not coexist in the event queue, so IncreaseA is not simultaneous
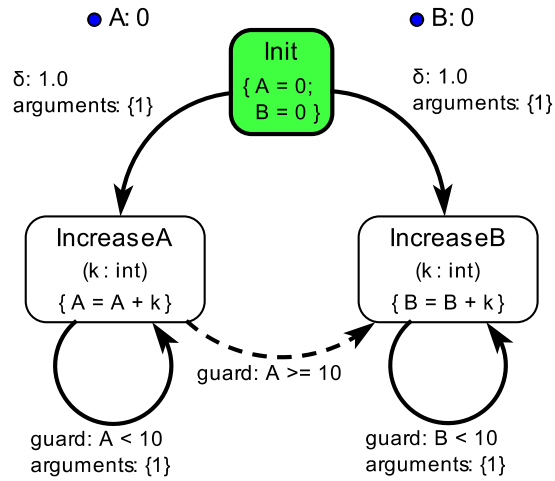
Figure 3.4: A model with simultaneous events.

with itself. In general, it is a model checking [9] problem to detect simultaneous events.

Table 3.1 contains four possible execution traces for the cases where IncreaseA always occurs before IncreaseB, where IncreaseB always occurs before IncreaseA, and where IncreaseA and IncreaseB are alternating in two different ways. (There are many other possible execution traces as well.) State of the event queue is not shown in this representation of execution traces. The "Time" row shows the model time at which events are processed. The "Event" row shows the names of the events that are processed. Below the double lines are the states of variables A and B after events in the same columns are processed. The columns are arranged from left to right in the order of event processing.

The traces end with different final values of A and B. The last instance of In-

| Time | 0.0 | 1.0 | 1.0 | ... | 1.0 |
Event	Init	IncreaseA	IncreaseA	...	IncreaseA
A	0	1	2	...	10
B	0	0	0	...	0

1) IncreaseA is always scheduled before IncreaseB

| Time | 0.0 | 1.0 | 1.0 | ... | 1.0 | 1.0 | 1.0 |
Event	Init	IncreaseB	IncreaseB	...	IncreaseB	IncreaseA	IncreaseA
A	0	0	0	...	0	1	2
B	0	1	2	...	10	10	10

| Time | ... | 1.0 |
Event	...	IncreaseA
A	...	10
B	...	10

2) IncreaseB is always scheduled before IncreaseA

| Time | 0.0 | 1.0 | 1.0 | 1.0 | 1.0 | ... | 1.0 |
Event	Init	IncreaseA	IncreaseB	IncreaseA	IncreaseB	...	IncreaseA
A	0	1	1	2	2	...	9
B	0	0	1	1	2	...	8

| Time | 1.0 | 1.0 |
Event	IncreaseB	IncreaseA
A	9	10
B	9	9

3) IncreaseA and IncreaseB are alternating, starting with IncreaseA

| Time | 0.0 | 1.0 | 1.0 | 1.0 | 1.0 | ... | 1.0 |
Event	Init	IncreaseB	IncreaseA	IncreaseB	IncreaseA	...	IncreaseB
A	0	0	1	1	2	...	8
B	0	1	1	2	2	...	9

| Time | 1.0 | 1.0 | 1.0 |
Event	IncreaseA	IncreaseB	IncreaseA
A	9	9	10
B	9	10	10

4) IncreaseA and IncreaseB are alternating, starting with IncreaseB

Table 3.1: Four possible execution traces for the model in Figure 3.4.

creaseA, which increases A to 10, always cancels the next IncreaseB in the event queue, if any. There are 10 instances of IncreaseB in total, and the one that is cancelled can be any one of them, if a well-defined order is missing.

I provide a solution below to avoid nondeterministic execution results as demonstrated above.

LIFO and FIFO Policies

A LIFO (last in, first out) property or a FIFO (first in, first out) property can be associated with a model. If neither of them is explicitly specified, the model has the LIFO property by default.

Either the LIFO policy or the FIFO policy is used when multiple events are scheduled to occur at the same time by different instances of events. With LIFO, the event scheduled by a later instance of event is processed earlier. The opposite occurs with FIFO.

As an example, consider using the LIFO policy to execute the model in Figure 3.4. Execution trace 3 and 4 in Table 3.1 would not be possible. Following the Init event, either IncreaseA or IncreaseB is processed first, depending on other mechanisms to untie simultaneous events to be discussed later.

- Suppose IncreaseA is processed first. According to the LIFO policy, the second instance of IncreaseA scheduled by the first one should be processed before IncreaseB, which is scheduled by Init. The second instance again schedules the
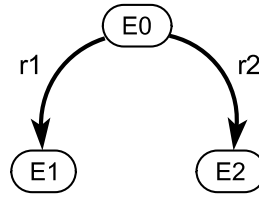
Figure 3.5: A scenario where event E0 schedules E1 and E2 after the same delay.

next one. In this way, processing of instances of IncreaseA goes on until A's value reaches 10, when IncreaseB is cancelled. That leads to execution trace 1.

- If IncreaseB is processed first, all 10 instances of IncreaseB are processed before IncreaseA. That yields execution trace 2.

With FIFO, however, instances of IncreaseA and IncreaseB interleave, resulting in execution traces 3 and 4 in the table.

In practice, LIFO is more commonly used because it atomically executes a chain of events, where one schedules the next with no delay. Atomicity is in the sense that no event that is not in the chain interferes with the processing of those events in the chain. This is convenient for specifying workflows where some tasks need to be finished sequentially without intervention.

Priorities

For events that are scheduled by the same event with the same delay δ, such as E1 and E2 in Figure 3.5, *priority numbers* can be assigned to the scheduling relations

r1 and r2 to determine the processing order. If r1 has a higher priority (i.e., a smaller priority number) than r2, then E1 is processed before E2, and vice versa. The default priority number for any scheduling relation is 0.

In Figure 3.4, if the priority of the scheduling relation from Init to IncreaseA is -1, and the priority of that from Init to IncreaseB is 0, then the first instance of IncreaseA is processed before IncreaseB. Execution traces 2 and 4 in Table 3.1 would not be possible. On the contrary, if the priority of the scheduling relation from Init to IncreaseA is 1, then the first instance of IncreaseB is processed earlier, making execution traces 1 and 3 impossible.

Identifiers

An event's identifier (ID) is unique within the flat model. A scheduling relation also has a unique ID that is usually hidden in the visual representation.

In Figure 3.5, if r1 and r2 have the same delay δ and the same priority, then the order of E1 and E2 is determined as follows:

> Assume total orders $\leq_e$ and $\leq_r$ exist for comparing events and relations based on their IDs, respectively. Let $e_1 =_e e_2$ be equivalent to $(e_1 \leq_e e_2) \wedge (e_2 \leq_e e_1)$ and $e_1 <_e e_2$ be equivalent to $(e_1 \leq_e e_2) \wedge \neg(e_2 \leq_e e_1)$. e_1 occurs before e_2 if and only if $(e_1 <_e e_2) \vee (e_1 =_e e_2 \wedge r_1 \leq_r r_2)$.

Moreover, the total order $\leq_e$ is also used to sort the initial events in the model (which are scheduled implicitly at the beginning of an execution but not by scheduling relations).

Designs with Atomicity

An interesting research topic is to ensure atomicity for a sequence of events with the presence of other simultaneous events. Without requiring designers to explicitly control critical sections, as is the case for imperative programming languages, here I present two design patterns in Figure 3.6 that designers can reuse to obtain atomicity.

Final events shown as filled vertices with double-line borders are used in the design patterns. They are special events that have the side effect of removing all events in Q. They are used to force termination of the execution even though there may be events remaining in the event queue.

The design pattern in Figure 3.6a is used to sequentially and atomically perform a number of tasks, assuming the LIFO policy is chosen. Even if other events exist in the model (which are not shown in the figure), those events cannot interleave with the tasks. As a result, intermediate state between tasks is not infected by other events.

The design pattern in Figure 3.6b is used to perform tasks until the guard G is satisfied. This again assumes LIFO. After the Start event is processed, all tasks are scheduled. In this case the first one to be processed is Task1, because Task1 $\leq_e$ Task2 $\leq_e$ Task3. After Task1, if G is true, End is processed next, which terminates the execution. If G is not true, then Task2 would be processed. The processing of tasks continues until either G becomes true at some point, or all tasks are processed but G remains false. A concrete use of this design pattern has been presented in the lower part of Figure 1.4, where the tasks are basic transformations.
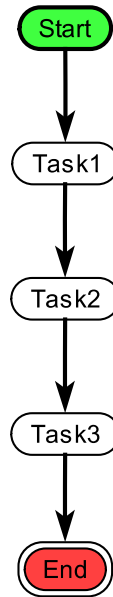
a) Sequentially perform all tasks
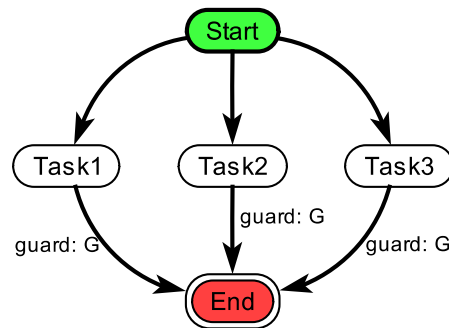
b) Sequentially perform tasks until G is satisfied

Figure 3.6: Two design patterns for controlling tasks.

3.1.5   Model Execution Algorithm

I operationally define the semantics of a flat model with an execution algorithm. In the algorithm, symbol Q refers to the event queue. The algorithm terminates when Q becomes empty.

1. Initialize Q to be empty

2. For each initial event e in the $\leq_e$ order

    (a) Create an instance i_e

    (b) Set the time stamp of i_e to be 0

    (c) Append i_e to Q

3. While Q is not empty

    (a) Remove the first i_e from Q, which is an instance of some event e

    (b) Execute the actions of e

    (c) Terminate if e is a final event

    (d) For each canceling relation c from e

    From Q, remove the first instance of the event that c points to, if any

    (e) Let R be the list of scheduling relations from e

    (f) Sort R by delays, priorities, target event IDs, and IDs of the scheduling relations in the order of significance

(g) Create an empty queue Q'

(h) For each scheduling relation r in R whose guard is true

    i. Evaluate parameters for the event e' that r points to

    ii. Create an instance $i_{e'}$ of e' and associate it with the parameters

    iii. Set the time stamp of $i_{e'}$ to be greater than the current model time by r's delay

    iv. Append $i_{e'}$ to Q'

(i) Create Q'' by merging Q' with Q and preserving the order of events originally in Q' and Q. For any $i' \in Q'$ and $i \in Q$, i' appears before i in Q'' if and only if the LIFO policy is used and the time stamp of i' is less than or equal to that of i, or the FIFO policy is used and the time stamp of i' is strictly less than that of i.

(j) Let Q be Q''

3.1.6 Example: Car Wash Simulation

In a car wash system, a number of car wash machines share a single queue. When a car arrives, it is placed at the end of the queue to wait for service by any of the machines. The machines serve cars in the queue one at a time in a first-come-first-serve manner. The car arrival intervals and service times are produced with two stochastic processes.
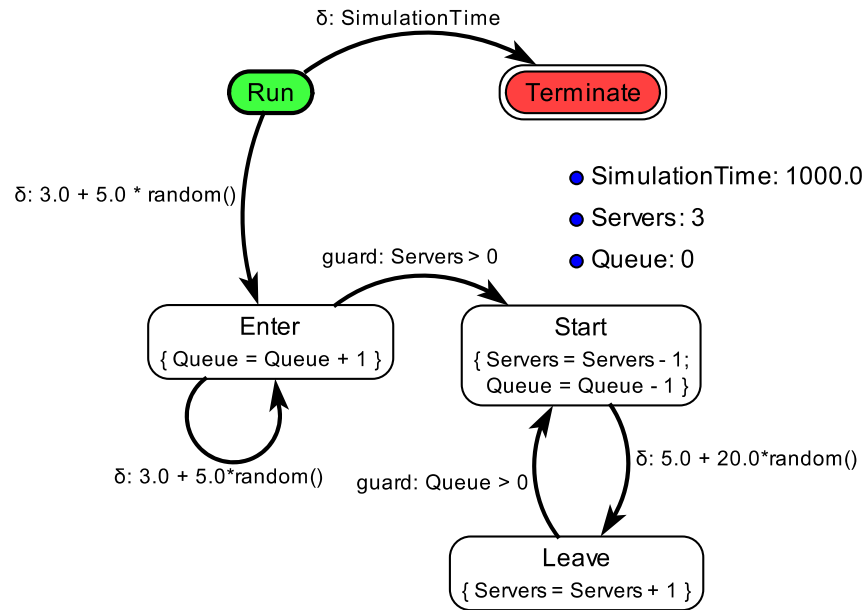
Figure 3.7: A model that simulates a car wash system.

The model to analyze the number of available servers and the number of waiting cars over time is provided in Figure 3.7. The Servers variable is initialized to 3, which is the total number of servers. The Queue variable starts with 0, since no car is waiting in the queue at the beginning. Run is an initial event. It schedules the Terminate final event to occur after the amount of time defined by a third variable SimulationTime.

The Run event also schedules the first instance of the Enter event, causing the first car arrival to occur after delay "3.0 + 5.0 * random()," where random() is a function that returns a random number in $[0, 1)$ in a uniform distribution. When Enter occurs, its action increases the queue size in the Queue variable by 1. The

Enter event schedules itself to occur again. It also schedules the Start event if there is any available server. The LIFO policy guarantees both Enter and Start to be processed atomically, so it is not possible for the value of the Servers variable to be changed by any other event in the queue after that value is tested by the guard of the scheduling relation from Enter to Start.

The Start event simulates car washing by decreasing the number of available servers and the number of cars in the queue. The service time is "5.0 + 20.0 * random()." After that amount of time, the Leave event occurs, which represents the finish of service for that car. Whenever a car leaves, the number of available servers must be greater than 0, so the Leave event immediately schedules Start if there is at least one car in the queue. Again, due to atomicity provided by the LIFO policy, testing for the queue size and changing it in the following Start event would not be interfered by any other event in the event queue.

Without the Terminate event pre-scheduled at the beginning, an execution of the model would not terminate because the event queue would never be empty.

3.2 Hierarchical Models

Hierarchical and component-based design can effectively defeat complexity and improve reusability. Examples of existing modeling languages that support model hierarchies include Statecharts [33], DEVS (Discrete Event System Specification) [11] and hierarchical Petri nets [22].

Hierarchical heterogeneous model composition is studied in research projects such as Ptolemy II [21], ForSyDe [36], SPEX [49] and ModHel'X [32]. In Ptolemy II, models of computation that can be composed in the model hierarchy include DE (discrete events) [42], CT (continuous-time models) [50], FSM (finite state machine), SDF (synchronous dataflow) [44], DDF (dynamic dataflow) [61], HDF (heterochronous dataflow) [28], SR (synchronous reactive) [3], PN (process networks) [37, 38, 47] and CSP (communicating sequential processes) [34]. They form a rich set of formal languages that can be mixed to achieve high flexibility and reusability [29].

An abstract framework has been defined in Ptolemy II for specifying the execution behavior of hierarchical heterogeneous models. The framework consists of 1) an execution algorithm with designated extension points, and 2) a protocol containing high-level specifications of the expected behavior at those extension points.

After an overview of the abstract framework, I provide algorithms for use at the extension points, which, in combination with the execution algorithm, define the operational semantics of hierarchical Ptera models.

3.2.1   An Abstract Framework for Model Execution

Given a possibly hierarchical model M, the following execution algorithm specifies its execution at a high level. [1] The algorithm contains extension points appearing as methods invoked by names with parameters. Some of the methods have Boolean

---

[1]There are models of computation studied in Ptolemy II whose execution does not follow this algorithm, such as PN and CSP, where concurrent processes communicate with messages.

return values.

○ *Execute M:*

> Preinitialize M
>
> Initialize M
>
> while true
>
> > if Prefire M then
> >
> > > Fire M one or more times
> > >
> > > if not Postfire M then
> > >
> > > > break
>
> Finalize M

For model M to be executed, Preinitialize, Initialize, Prefire, Fire, Postfire and Finalize in the algorithm must be defined. Given M as the parameter, those methods delegate to the corresponding methods defined for M's director. If M is hierarchical, which means M is a composite actor containing actors inside, then its director also invokes those methods for the contained actors. The way the director invokes the methods for the contained actors depends on the model of computation that the director implements. In general, for each contained actor, invocation of the methods should follow the sequence in the execution algorithm, though the sequences for different contained actors may be interleaved.

Expected behavior of the methods is specified as follows.

- *Preinitialize.* Perform operations required to start a fresh execution for the given model.

- *Initialize.* Initialize the given model for execution. If M is contained in another model and its model of computation is timed, it may issue an initial firing request to explicitly request its container to prefire, fire and postfire it at the current model time or at a model time in the future.

- *Prefire.* Return a Boolean value that determines whether a firing can be performed for the given model.

- *Fire.* Fire the given model, which means to execute the model for one step. Tokens received at the model's input ports may be read and the model may produce output tokens via its output ports.

- *Postfire.* Update the state of the given model as a side effect of the last firing, and return a Boolean value that tells whether future firings are permitted. A timed model may a issue firing request to its container, if any, or it may advance model time if itself is at the top level of the model hierarchy.

- *Finalize.* Terminate the execution and release the resources allocated for the given model. Once finalized, the model should not be prefired, fired or postfired, unless it is initialized again.

An additional *fireAt* method is defined as a callback that can be invoked in the Initialize and Postfire methods to request firing from the container. It takes as the

first parameter the model that issues the request and as the second parameter a model time when firing of that model should occur. That model time should be equal to or greater than the current model time. For example, when Postfire is invoked with model M contained in M', fireAt may be issued with parameters M and t. The director of M' receives that request and schedules to fire M when the model time reaches t.

3.2.2   Ptera Semantics in the Abstract Framework

Compared to the execution algorithm for flat Ptera models in 3.1.5, an alternative way to define the semantics is by defining all the methods in the abstract framework. This alternative way yields an equivalent semantics for flat models, but it additionally supports hierarchical models.

In the following discussion, M denotes a Ptera model. M' is the model that contains M in the model hierarchy, if it exists. m represents a submodel contained in M, if any.

The event queue of M is denoted by Q. It is a priority queue that stores scheduled events and fireAt requests from the submodels. Events are sorted in the order discussed previously. fireAt requests are sorted by their time stamps. If a fireAt request and an event have the same time stamp, then the fireAt request is sorted before the event. If two fireAt requests have the same time stamp, they are sorted with the LIFO or FIFO policy, depending on when those fireAt requests were received.

An additional *initializing* attribute is defined for each scheduling relation. It takes a Boolean value that determines whether the submodel of the scheduled event, if there is any, should be initialized *every time* that event is processed. If the attribute is false, the submodel would be initialized only if it has not been initialized or its postfire has returned false last time (meaning that its previous execution has finished).

An additional variable S denotes a set of references to the submodels of M that have been initialized.

- *Preinitialize M:*

    Initialize Q and S to be empty

    For each event e in M in the $\leq_e$ order

        If e is associated with a submodel m

            Preinitialize m

- *Initialize M:*

    For each initial event e in M in the $\leq_e$ order

        Create an instance i_e and append it to Q

        Set the time stamp of i_e to be the current model time

    If container M' exists and Q is not empty

        Issue fireAt to M' with the current model time

- *Prefire M returns Boolean:*

    If Q is not empty

Peek the first item i in Q

Let t be i's time stamp

If $t <$ current time

Report error and return false

else if $t >$ current time

Return false

Return true

○ *Fire M:*

If Q is not empty

Peek the first item i in Q

Let t be i's time stamp

If $t <$ current time

Report error and return

else if $t >$ current time

Return

Remove i from Q (step 3a in 3.1.5)

If i is an instance of event e

Execute the actions of e (step 3b in 3.1.5)

If e is a final event

Clear Q (step 3c in 3.1.5)

else if e has submodel m

If the scheduling relation that scheduled i has initializing

attribute set to true, or m is not in S

Initialize m

Add m to S

If m need not be initialized or no fireAt request was received

when it was initialized

RunOnce m // defined next

else // e does not have submodel

Evaluate scheduling relations and canceling relations from e

(steps 3d through 3j in 3.1.5)

else // i is a fireAt request

Let m be the submodel that issued i

RunOnce m

○ *RunOnce m:*

If Prefire m

Fire m

If not Postfire m

Finalize m

Remove m from S

Let e be the event that m is associated with

Evaluate scheduling relations and canceling relations from e

(steps 3d through 3j in 3.1.5)

○ *Postfire M returns Boolean:*

    If Q is not empty

        Peek the first item i in Q

        Let t be i's time stamp

        If M has container M'

            Issue fireAt to M' with time stamp t

        else

            Advance model time to t

        Return true

    Return false

○ *Finalize M:*

    For each event e in M in the $\leq_e$ order

        If e is associated with a submodel m and m is in S

            Finalize m

    Clear Q and S

As part of the protocol established in the abstract framework, in a hierarchical model, each composite actor with a discrete-event director maintains its event queue locally. Model M reports only the next firing time t to its container M' with a fireAt request in postfire. If t is greater than or equal to the current model time, M' ensures

to fire M no later than t. M' does not need to know the events that are scheduled in the event queue of M. This effectively encapsulates the internal behavior of M. Another benefit is that by adjusting the *time advance rate* of M (a factor to be multiplied with the time stamps of events occurring in M, which could be different from 1), M may conceptually run at a different speed. This is particularly interesting for simulation and performance analysis.

3.2.3 Semantic Equivalence for Flat Models

Operationally, it is straightforward to prove that the above implementation in the abstract framework defines an equivalent semantics for flat models compared to the execution algorithm in 3.1.5, which does not use the abstract framework.

When M is flat, its container M' does not exist and no submodel is associated with any event. As specified by the abstract framework, an execution starts by preinitializing and initializing M. The set S is always empty. The event queue Q begins with only the initial events sorted in their previously defined total order.

After initialization, M is executed in a loop. In each iteration, it is first prefired. Prefire returns false only when the nearest event in Q has a time stamp greater than the current model time, which should not happen for a flat model, because either initialization or the last postfire should have set the model time to that time stamp already. In each firing, if there are imminent events in Q, the first one is processed following the steps 3a through 3j in 3.1.5. When Q is empty, M is still fired but

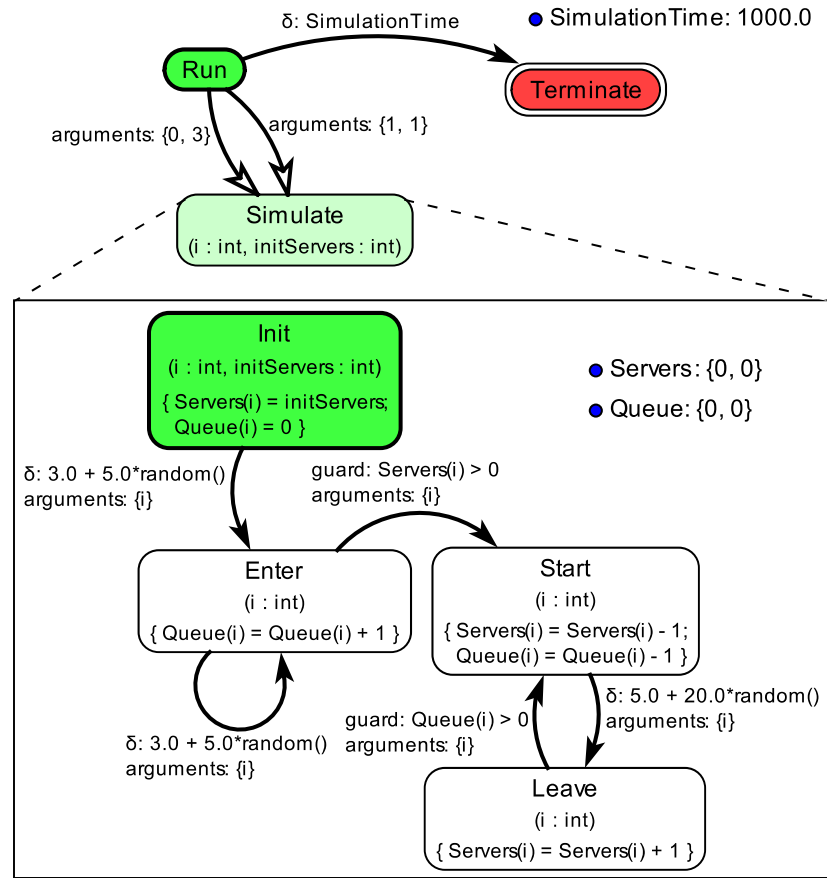Figure 3.8: A hierarchical model that simulates a car wash system with two settings.

nothing is done in the firing. Its postfire returns false to terminate the execution.

Because of the semantic equivalence for flat models, the execution algorithm specified in the abstract framework can be considered as an extension to that in 3.1.5, with the difference that the former also supports hierarchical models.

3.2.4 Hierarchical Car Wash Model

As a demonstration of hierarchical models, Figure 3.8 is a modification from Figure 3.7. Its top level simulates an execution environment, which has a Run event as the only initial event, a Terminate event as a final event, and a Simulate event associated with a submodel. The submodel simulates the car wash system with the given number of servers.

The two scheduling relations pointing to the Simulate event have hollow arrow heads, which reveal the fact that their initializing attributes are set to true. For that reason, they are called *initializing scheduling relations*. They make the submodel initialized each time the Simulate event is processed. In this example, the Simulate event is processed twice, causing two instances of the Init event to be scheduled in the submodel's local event queue. They are the start of two concurrent simulations, one with 3 servers and the other with 1 server. The priorities of the initializing scheduling relations are not explicitly specified. Because the two simulations are independent, the order in which they start has no observable effect. In fact, the two simulations may even occur concurrently.

Parameter i for the Simulate event distinguishes the two simulations. Compared to the model in Figure 3.7, the Servers variable in the submodel has been extended into an array with two elements. Servers(0) refers to the number of servers in simulation $i = 0$, while Servers(1) is used in simulation $i = 1$. The Queue variable is enhanced in the same way. Each event in the submodel also takes a parameter i and supplies

Time	0.0	0.0	0.0	0.0	0.0	3.654	3.654	4.028
Event	Run	Simulate^0	Init^0	Simulate^1	Init^1	Enter^0	Start^0	Enter^1
Servers(0)	0	0	3	3	3	3	2	2
Queue(0)	0	0	0	0	0	1	0	0
Servers(1)	0	0	0	0	1	1	1	1
Queue(1)	0	0	0	0	0	0	0	1

Time	4.028	8.158	8.158	8.406	11.007	12.384	12.384	12.997
Event	Start^1	Enter^0	Start^0	Enter^1	Leave^0	Leave^1	Start^1	Enter^1
Servers(0)	2	2	1	1	2	2	2	2
Queue(0)	0	1	0	0	0	0	0	0
Servers(1)	0	0	0	0	0	1	0	0
Queue(1)	0	0	0	1	1	1	0	1

Time	14.613	14.613	...	1000
Event	Enter^0	Start^0	...	Terminate
Servers(0)	2	1	...	2
Queue(0)	1	0	...	0
Servers(1)	0	0	...	0
Queue(1)	1	1	...	105

Table 3.2: An example execution trace for the model in Figure 3.8.

the value of i that it receives to the next events that it schedules. This ensures that the events and variables in one simulation are not affected by those in the other simulation, even though they share the same model structure.

Table 3.2 shows a possible execution trace. Numbers in superscript represent the i values for the events.

Let M and m be the top-level model and the submodel, respectively. The execution starts by running the execution algorithm with M. At the beginning, M is preinitialized, causing m to be preinitialized as well. Next, M is initialized, so its initial event Run is placed in its event queue denoted by Q_M.

In the first iteration of the loop in the execution algorithm, M is prefired and the prefire returns true because the Run event is scheduled at the current model time, which is 0. M is then fired. Its Run event is processed and two instances of the Simulate event and a Terminate event are scheduled in Q_M. Postfire of M returns true because there are events remaining in Q_M.

In the second iteration, M is prefired, fired and postfired again. In the firing, assume the instance of Simulate event to be processed is Simulate^0. (As mentioned above, an opposite assumption leads to the same result.) m is initialized for the first time and is added to set S. In its initialization, m schedules the Init event in its event queue Q_m. It also issues a fireAt request to M, requesting M to fire it at the current model time. After initializing m, the firing of M finishes. M is then postfired with true return result.

In the third iteration, the submodel m is prefired, fired and postfired. (Simulate^1 remains in Q_M, because fireAt requests are always ordered before event instances with the same time stamp.) In m's firing, the Init event is processed, which schedules an Enter event in Q_m after a random delay. In m's postfire, another fireAt request is issued to M.

In the fourth iteration, Simulate^1 is processed. This causes m to be again initialized (due to the initializing scheduling relation). Another instance of the Init event is scheduled in Q_m. That instance is processed in the fifth iteration.

The execution continues until the model time is eventually advanced to 1000 and

the Terminate event originally scheduled in Q_M is processed.

As a remark, one can conceptually execute multiple instances of a submodel by initializing it multiple times. However, the event queue and variables are not copied. Therefore, the variables need to be enhanced into arrays and an extra index parameter (i in this case) needs to be provided to every event.

Actor-oriented subclassing [45, 43] provides an alternative approach to enhancing a submodel into multiple executable instances. A class can be defined for the design of the submodel, with which instances can be created for execution.

Yet another approach is higher-order model composition [7], which allows to specify the model with a parametrizable higher-order model description. For example, using the Ptalon language discussed in 2.7.4, specification of a complex and large-scale model can be greatly simplified with a compact description [8]. Growth in model size does not require larger descriptions. Another example of higher-order model composition language is the one offered by the VIATRA2 model transformation tool [2].

3.3 Composition with Heterogeneous Models of Computation

Ptera models can be composed with actor-oriented discrete-event (DE) models and finite state machines (FSMs), as well as models using other models of computation implementable in the abstract framework.

I demonstrate the concept with two examples. One is to embed Ptera in DE and the other is to embed Ptera in DE and FSM in Ptera. A discussion that follows lists other types of composition that are currently known to be compatible. The general idea behind is extremely powerful because it allows designers to choose a convenient and expressive model of computation to model each part of the their systems, and to obtain a well-defined semantics for the overall composition.

3.3.1   Composition with DE

Compared to Ptera models, DE models are a different kind of discrete-event models. Their visual representation uses the actor-oriented modeling language discussed in 1.1. That is, actors are represented with boxes, ports of actors are triangles, and the lines between ports are communication channels.

Actors perform computation on the data received at their input ports, and produce data via their output ports. Those data are wrapped in events that also carry time stamps representing the model time at which they are produced. The events in DE, which I call *DE events* in the following discussion, are not visible in the model design, and are different from events in Ptera shown as vertices.

There are two kinds of actors. An atomic actor is implemented in an imperative programming language, such as Java and C. A composite actor encapsulates inter-connected actors and acts as a single actor. For a composite actor to be executable, a director, which is a special attribute, must be associated with it. The director

implements a model of computation. In particular, a DE model is a composite actor with a DE director.

Overview of the DE Director

The DE director implements DE semantics [42] for the composite actor that it is associated with. In an execution, the DE director maintains an event queue, which temporarily stores DE events received at the input ports of actors in that composite actor, as well as the fireAt requests issued by those actors. When the model time becomes equal to the time stamp of a DE event (or a fireAt request), the DE director fires the corresponding actor and provides it with the DE event.

The following is a brief description of the methods defined for the DE director to be used in the abstract framework.

- *Preinitialize.* Clear the event queue used for executing the composite actor. Preinitialize all the actors in the composite actor.

- *Initialize.* Analyze data dependency between actors. Initialize all the actors in the composite actor. Some actors may issue fireAt requests, which are stored in this director's event queue.

- *Prefire.* Check whether there is any DE event or fireAt request in the event queue to be processed at the current model time, or there is any DE event available at an input port of the composite actor, or the composite actor is at

the top level. If any of those conditions is true, return true; otherwise, return false.

- *Fire.* Move the events at the input ports of the composite actor into the event queue for the actors in the composite actor to process. Remove the DE events and fireAt requests scheduled at the current model time from the event queue, one at a time, in an order preserving data dependency. For each DE event or fireAt request, prefire the responsible actor and fire it if prefire returns true. The fired actor may generate DE events for the actors connected to its output ports. Those DE events are stored in this director's event queue until the receiving actors are fired at the model time equal to the time stamps of those DE events. After that, postfire the actor to determine whether it needs to be fired again. The actor may issue a fireAt request to be added to this director's event queue.

- *Postfire.* Check whether there are DE events or fireAt requests remaining in the event queue. If so, either advance model time to the smallest time stamp in case the associated composite actor is at the top level of model hierarchy, or otherwise issue a fireAt request to the containing model.

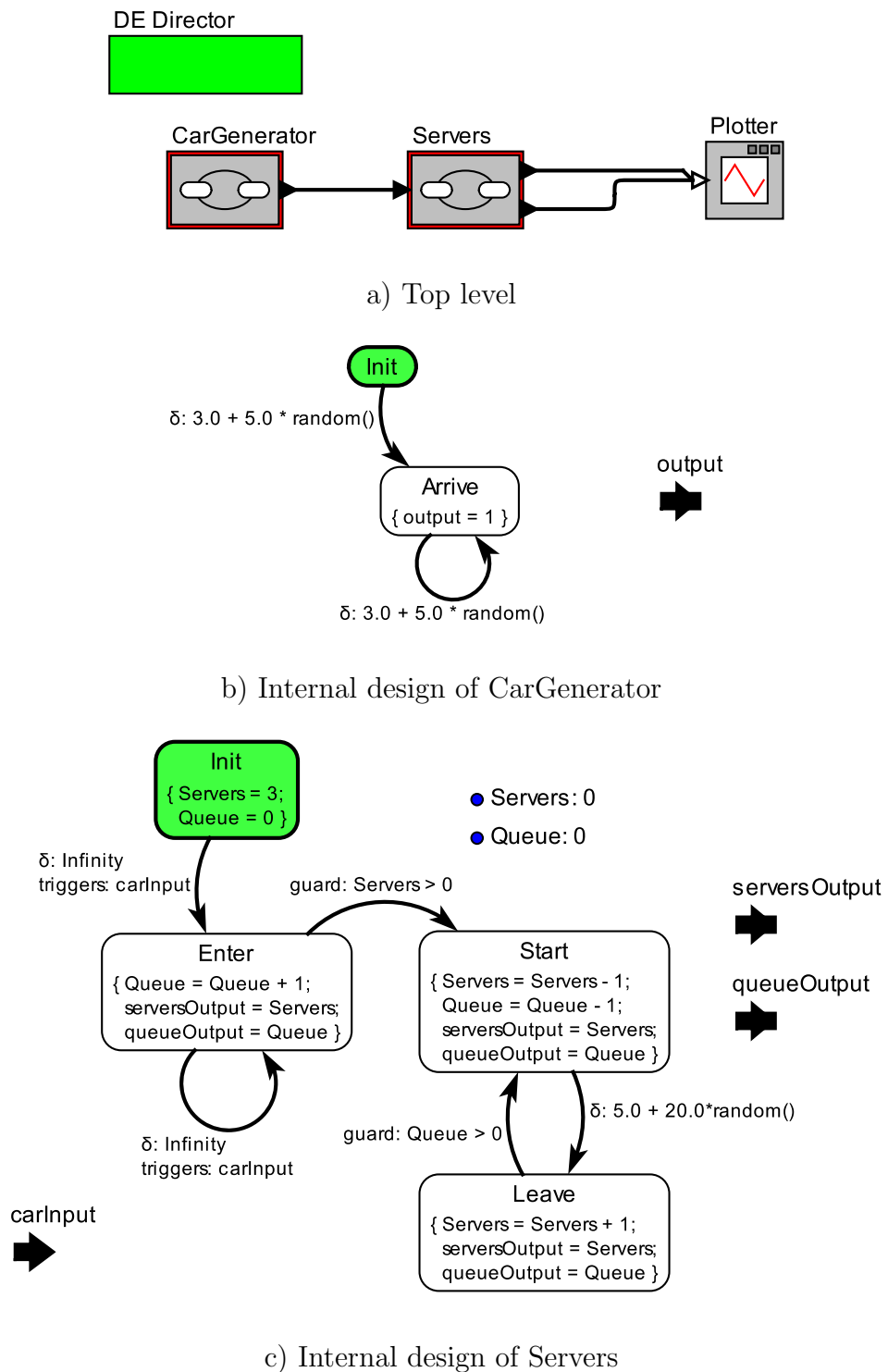- *Finalize.* Clear the event queue and finalize all actors in the composite actor.

a) Top level

b) Internal design of CarGenerator

c) Internal design of Servers

Figure 3.9: A car wash model using DE and Ptera in a hierarchical composition.

Example

Figure 3.9 shows a model that uses DE at its top level and contains Ptera submodels. In Figure 3.9a, boxes are actors except that the filled box with caption "DE Director" represents a DE director, which is essentially an attribute that defines the DE semantics for the diagram. CarGenerator and Servers are two composite actors associated with Ptera submodels. Plotter is an atomic actor defined in Java.

Figure 3.9b shows the internal design of CarGenerator. The Init event schedules the first Arrive event after a random delay. Each Arrive event schedules the next one. Whenever it is processed, the Arrive event generates a car arrival signal and sends it via the output port using assignment "output = 1" with the left hand side being the port name and the right hand side being an expression that computes the output value. In this case, the value 1 is unimportant and only the presence of a value at the output port is interesting.

Figure 3.9c shows the internal design of Servers. It is similar to the previous car wash models, except that there is an extra carInput port to receive DE events representing car arrival signals from the external and the Enter event is scheduled to handle inputs via that port. No assumption is made in the Servers component about the source of the car arrival signals. At the top level, the connection from CarGenerator's output port to Servers' input port makes explicit the producer-consumer relationship. This separation of concerns leads to a more modular and reusable design.

Plotter at the top level plots the outputs from Servers in a separate window.
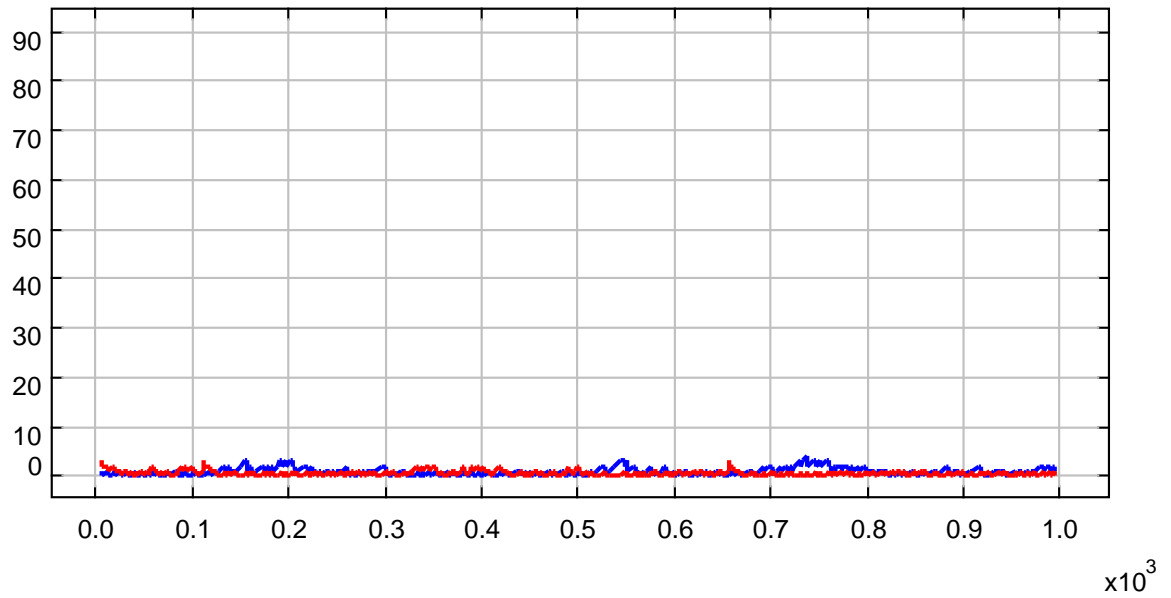
a) InterarrivalTime = 3.0 + 5.0 * random()
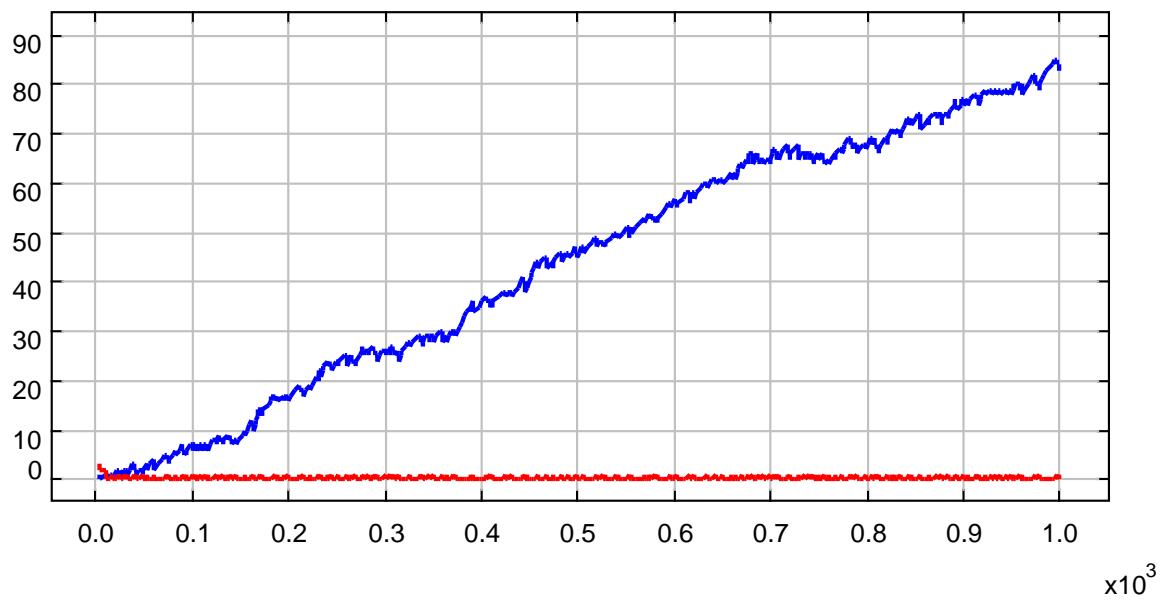
b) InterarrivalTime = 1.0 + 5.0 * random()

Figure 3.10: Plotter output for two configurations of Figure 3.9.

An example plot obtained by executing the model to time 1000 is provided in Figure 3.10a, where the upper (blue) curve represents the number of waiting cars over time, and the lower (red) curve represents the number of available servers. If the car interarrival time in Figure 3.9b is changed from "3.0 + 5.0 * random()" to "1.0 + 5.0 * random()," one may observe a different plot as in Figure 3.10b.

Processing of DE Events

In the initialization phase, the DE director computes the causality dependency between the interconnected actors. This information can be used in the Fire method to determine an unambiguous actor firing order [64]. Each actor is also initialized in this phase. For a Ptera submodel (such as CarGenerator and Servers in Figure 3.10), initializing it causes initial events to be scheduled in its event queue at time 0 and a fireAt request to be issued to the DE director. The request is stored in the DE director's event queue.

DE events and fireAt requests in the event queue of a DE director need not be totally ordered. The director only needs to ensures that if DE event e_2 is causally dependent on e_1, then e_1 must be processed before e_2. For DE events that do not have causality dependency between them, the order in which they are processed does not affect the observable output, and can be arbitrary [15, 16].

Each time the DE director is fired, it retrieves imminent DE events and fireAt requests from its event queue. For a DE event, the director fires the receiving actor.

The actor can read the DE event when it is fired. For a fireAt request, the actor is fired with no available input.

In the example, when fired the first time at model time 0, the two submodels process their Init events. In CarGenerator, an Arrive event is scheduled in its event queue. A new fireAt request is issued to the DE director in postfire that requests to fire again in the future when the model time reaches the time of the Arrive event. In Servers, processing the Init event causes the Servers and Queue variables to be reset to their initial values. The Enter event is scheduled with the delay "Infinity" and is registered to receive inputs at carInput port (discussed next). Postfire of this submodel does not issue fireAt request but simply returns true. This is because, even though the Servers component has events remaining in its event queue, it cannot decide the time when it should be fired again. The DE director at the top level should fire it when a DE event is received at its input port.

In postfire, the DE director advances model time to the time of the fireAt request from CarGenerator. In the next firing, the DE director fires CarGenerator, which processes the Arrive event and generates a DE event to the output port. That DE event is tagged with a time stamp equal to the current model time, and is stored temporarily in the DE director's event queue. When CarGenerator postfires, it schedules the Arrive again and issues another fireAt request. The DE director also fires the Servers submodel since it has a DE event at its input port. The latter processes the Enter event, which is triggered by the input as specified by the triggers attribute of

the scheduling relation that scheduled it.

Plotter passively waits for DE events from Servers. Every time it is fired, it is provided with two DE events, one at each channel of the input port, because Servers always generates DE events at both output ports at the same time. In reaction, Plotter extracts the values from those DE events and plots them in a separate window.

Reacting to Inputs and Sending Outputs

To schedule an event in a Ptera submodel to react to external inputs, a scheduling relation may be tagged with a *triggers* attribute that specifies port names separated by commas. That attribute is used in conjunction with the delay δ to determine when the event is processed.

Let the triggers be "$p_1, p_2, \cdots, p_n$." The event is processed when the model time is δ-greater than the time at which the scheduling relation is evaluated *or* one or more DE events are received at *any* of $p_1, p_2, \cdots, p_n$. To schedule an event that indefinitely waits for input, "Infinity" may be used as the value of δ.

To test whether a port actually has an input, a special Boolean variable whose name is the port name followed by string "_isPresent" can be accessed. To refer to the input value available at a port, the port name may be used in an expression.

For example, the Enter event in Figure 3.9c is scheduled to indefinitely wait for DE events at the carInput port. When one is received, the Enter event is processed ahead of its scheduled time and its action increases the queue size by 1. In that

particular case, the value of the input is ignored.

To send DE events via output ports, assignments can be written in the actions with port names on the left hand side and expressions that compute the values on the right hand side. The time stamps of the outputs are always equal to the model time at which the actions are executed.

3.3.2   Composition with FSMs

Ptera models can also be composed with untimed models such as FSMs (finite state machines). When a Ptera model contains an FSM submodel associated with an event, it can fire the FSM when that event is processed and when inputs are received at its input ports.

The opposite composition, having Ptera submodels associated with states in an FSM, is also interesting because by changing states, the submodels may be disabled and enabled, and the execution can switch between modes. That, of course, requires to extend FSM in a well thought through way, so that it can unambiguously handle the fireAt requests from the submodels. It could lead to a problem if a submodel is disabled and the time that it requested to be fired has passed. When it is re-enabled, it would be in a dilemma where some events cannot be processed because they should have occurred in the past, but without processing them, the state of the execution is undetermined. An extension to FSM that addresses this problem is modal models, which can interact well with discrete-event models [51].

Overview of the FSM Director

Here I only concern FSMs without the modal model extension embedded in Ptera
as submodels.

In the abstract framework, the FSM director can be defined as follows.

- *Preinitialize.* Nothing needs to be done.

- *Initialize.* Set the current state to be the initial state.

- *Prefire.* Return true, because an FSM is always willing to be fired unless its
  final state has been reached, in which case its Postfire should have returned
  false and Prefire would not be invoked again.

- *Fire.* Evaluate the guards of all transitions from the current state. Among those
  that are enabled (if any), pick one according to a predefined or user-specified
  scheme. Execute the actions of the chosen transition that produce output data
  at the FSM's output ports. Record the chosen transition to be used in Postfire.

- *Postfire.* If a transition is chosen in Fire, execute the actions of that transition
  that assign new values to variables. After that, set the current state to be the
  destination state. Return true if the new state is not a final state. Return false
 otherwise.

- *Finalize.* Nothing needs to be done.

Because an FSM is untimed, if it is contained in a discrete-event model, the data that it outputs in Fire are automatically associated with the current model time as their time stamps.

Example

To demonstrate the composition between Ptera and FSM, consider the case where drivers can perceive the number of cars waiting in the queue and may avoid entering the queue if there have already been too many waiting cars. That leads to a lower arrival rate (or equivalently, longer interarrival time in average). Conversely, if there are only few or no waiting cars, the drivers would always enter the queue, resulting in a higher arrival rate.

The model in Figure 3.9 is modified for this scenario and the revised model is shown in Figure 3.11. Figure 3.12 shows the result of executing the new model. At the top level, the queueOutput port of Servers (whose internal design is the same as Figure 3.9c) is fed back to the queueInput port of CarGenerator. The FSM submodel in Figure 3.11c is associated with the Update event in CarGenerator. It inherits the ports from its container, allowing the guards of its transitions to test the inputs received at the queueInput port. In general, actions in an FSM submodel can also produce data via the output ports.
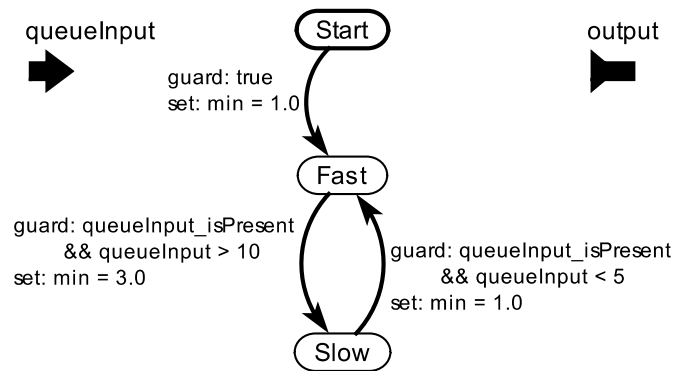
When the Update event of CarGenerator is processed, the FSM submodel is initialized to be in its initial state. When fired the first time, the FSM moves into the
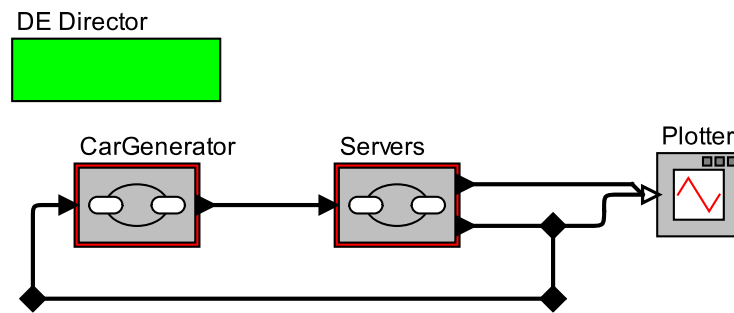
a) Top level
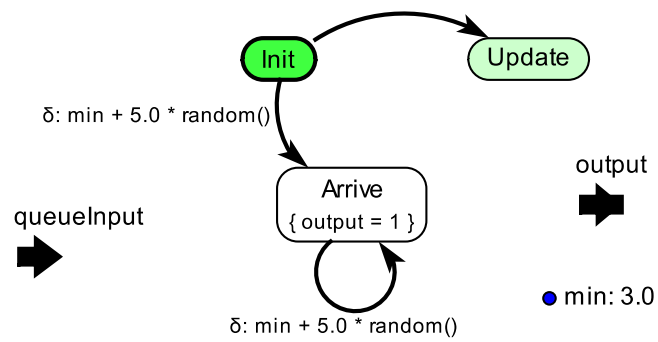
b) Internal design of CarGenerator

c) Internal design of Update

Figure 3.11: A car wash model using DE, Ptera and FSM in a hierarchical composition.
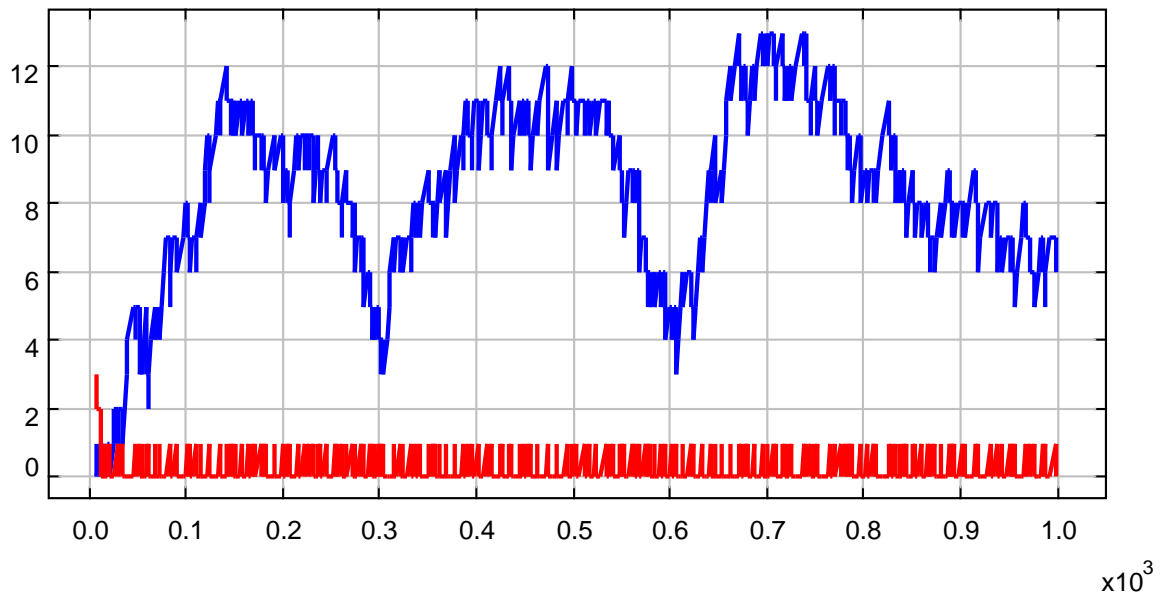
Figure 3.12: Plotter output for the model in Figure 3.9.

Fast state and sets the minimum interarrival time to be 1.0. Since then, the interarrival time is generated with expression "1.0 + 5.0 * random()." Notice that the min variable is defined in CarGenerator, and a scoping rule enables the contained FSM to read from and write to that variable.

Postfire of the FSM always returns true, because there is no final state. The FSM would be fired again when either the Update event is processed again (which does not happen in this example) or when input is received at any input port. When the Servers composite actor sends out the number of waiting cars via its queueOutput port, the number is transferred to the queueInput port of CarGenerator by the top-level DE model and is made available to the FSM submodel. The FSM submodel is fired at that time. It may or may not change state depending on whether that

received number exceeds the bounds.

In general, when a Ptera model receives input at a port, all the initialized sub-models are fired, regardless of the models of computation that those submodels use.

3.3.3 Hierarchical Heterogeneous Model Design

Models of computation implementable in the abstract framework can be composed with Ptera to create executable models, but some types of composition may not be as common as others.

I identify types of meaningful compositions involving Ptera models, and perform experiments in the Ptolemy II framework to back the theoretical results.

These are the currently studied models of computation that can be used to create models containing Ptera submodels.

- DE. As is demonstrated in the examples in Figure 3.9 and Figure 3.11, DE models can contain Ptera submodels and provide well-defined discrete-event semantics. The Ptera submodels can communicate with actors in the DE model using DE events via ports.

- Ptera. Composition between Ptera and itself has been demonstrated in Figure 3.8. The containing and the contained Ptera models have separate event queues, and the event queue of the outer model only needs to store the nearest fireAt request from the submodel. It fires the submodel when the model time

reaches the requested time, or an input is received at any input port, or the event that the submodel is associated with is processed.

- TM (timed multitasking). This model of computation emulates the behavior of a real-time operating system running on a single-core CPU. An actor is a task that can be triggered by timed events received at its input ports. The actor may also issue fireAt requests which the director considers as interrupts occurring at the time equal to their time stamps.

These are the models of computation that can be contained in Ptera.

- DE. DE can be embedded in Ptera as well, though the examples do not show this type of composition. A DE submodel reports to the Ptera model that contains it with fireAt requests. It would also be fired when the event it is associated with is processed and when inputs are received at the input ports.

- FSM. As an untimed model of computation, FSM cannot issue fireAt requests, so its firing can only be triggered by processing the event that it is associated with in the Ptera model or by receiving an input.

- Ptera, as is discussed above.

- SDF (synchronous dataflow) [44], DDF (dynamic dataflow) [61] and HDF (heterochronous dataflow). These are different flavors of dataflow and are all untimed. Their semantics can be defined in the abstract framework. Similar to

FSMs, submodels in these models of computation are fired when the containing Ptera model processes the events they are associated with or when inputs are received.

- SR (synchronous reactive) [3]. An SR model is timed and the time advances in fixed step sizes called periods (which may be 0). In each period, the data values on the communication channels are computed as a fixpoint. For an SR submodel contained in a Ptera model, at the end of each period, it requests firing for the next period, and the Ptera model stores the fireAt request in its event queue.

Chapter 4

Model-Based Transformation

With the motivation to develop a control language for model transformation, I created the Ptera model of computation based on event graphs, which I have shown to be suitable for specifying hierarchical discrete-event processes.

To distinguish from basic transformations specified with transformation rules, a more sophisticated transformation involving multiple basic transformations controlled by a model is called *model-based transformation*. (The name also implies that the transformation is itself a model and can be transformed.) Specifically, if the control language is Ptera, the model-based transformation is also called *transformation workflow* (or simply workflow in the following discussion).

The key idea of a transformation workflow is to consider an event as a task that contributes to the overall operation. Finishing of the current task causes the next task(s) to be activated, if any. The guard on a scheduling relation identifies the

condition under which the activation happens, and the delay δ represents the time (or cost) for the activation. If no scheduling relation exists between two tasks, the condition under which one can activate the other is considered false, or alternatively, the delay is infinite.

Without loss of generality, tasks are considered instantaneous and do not incur any model time delay. To model a task that takes time, the designer may create two events corresponding to the start and end of that task, and assign the requested amount of delay to the scheduling relation from the start to the end. The designer may also associate a submodel with a task, so the duration of that task becomes equal to the duration of a single execution of that submodel.

Intuitively, some tasks have basic transformations as their actions. Those actions modify the model stored in a variable with global accessibility in the workflow.

4.1   The Constant Optimization Example

Figure 4.1 presents a ConstantOptimization workflow that aims to optimize a Constant model as the one in Figure 4.2. The original model has 8 constant actors feeding sequences of constant numbers through arithmetic operators to the Display actor. Obviously, in this extreme case, all the computation can be performed statically, yielding a much simplified model in Figure 4.3.

The basic transformation in the Transform event of the workflow is shown in Figure 4.4. That single transformation rule handles all three types of arithmetic
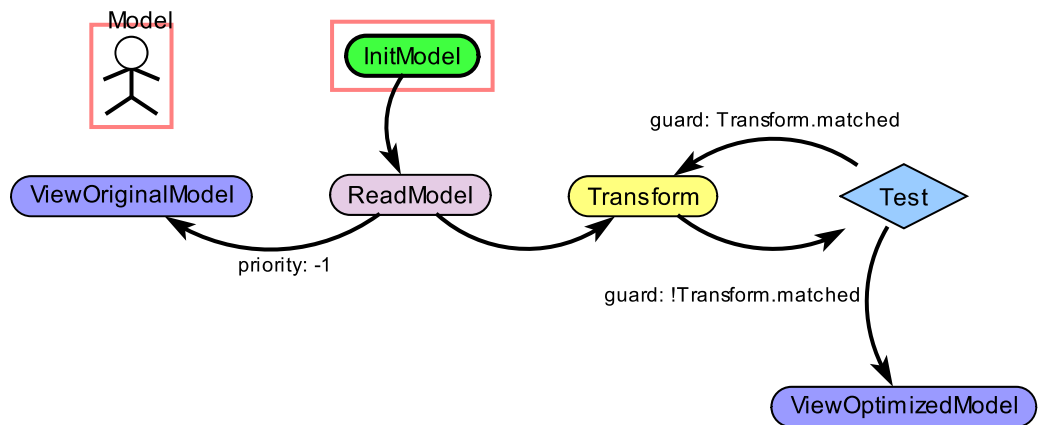
Figure 4.1: A ConstantOptimization workflow to optimize a Constant model.
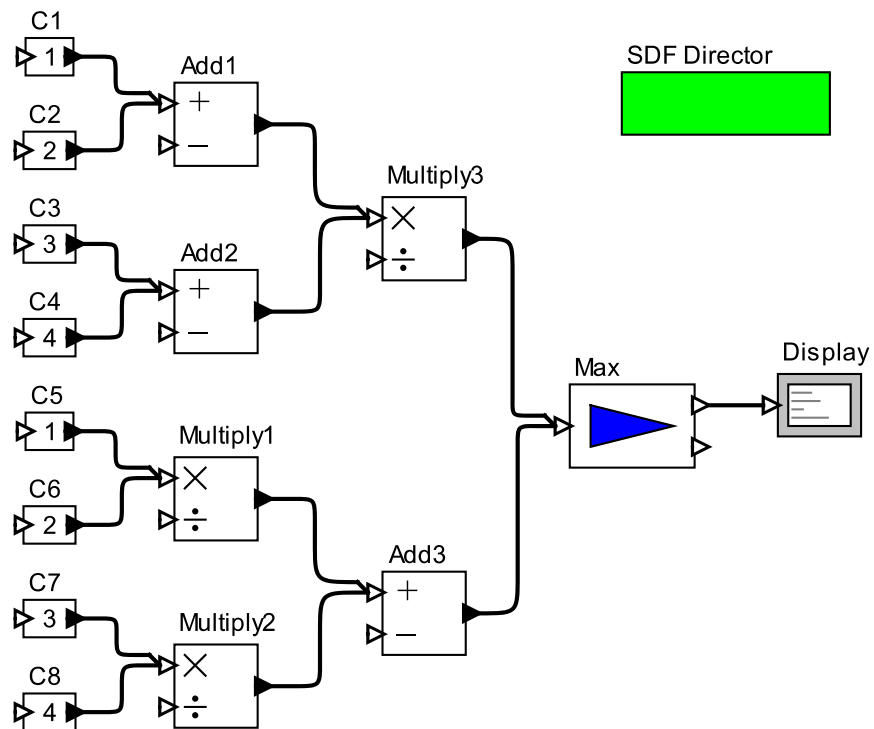

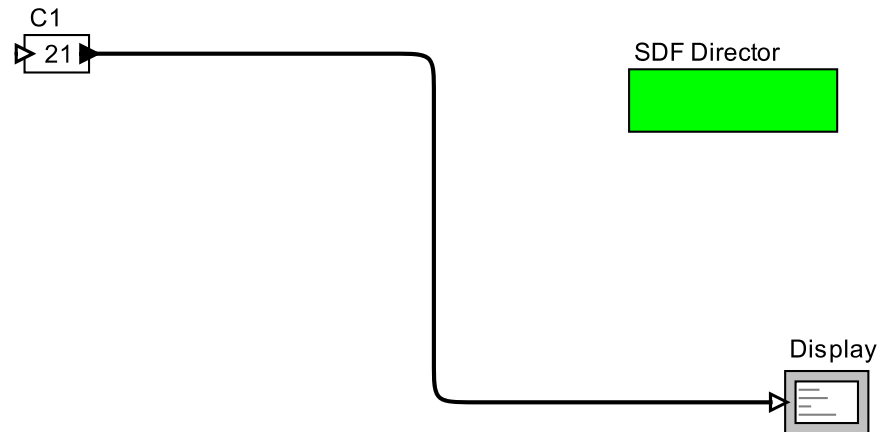Figure 4.2: The Constant model with 8 constant actors.

Figure 4.3: The model obtained by fully evaluating the Constant model.

operations in the original model (addition, multiplication and maximum). Matcher A itself in the pattern can match any actor with one input port and one output port, but Constraint3 further restricts the matching, requiring any actor that A matches to be in one of the three classes. Once a match is found, both C2 and A should be removed, and C1 should be kept because it has a correspondence C in the replacement. A hidden operation in C computes the value depending on what arithmetic operation is discovered. The expression to compute the value is as follows:

```
A.getClassName().equals("ptolemy.actor.lib.AddSubtract") ?
  C1.value + C2.value :
  A.getClassName().equals("ptolemy.actor.lib.MultiplyDivide") ?
    C1.value * C2.value :
      A.getClassName().equals("ptolemy.actor.lib.Maximum") ?
        (C1.value >= C2.value ? C1.value : C2.value) : 0
```

The purpose of presenting this ConstantOptimization example is to compare the transformation workflow with the equivalent DDF (dynamic dataflow) model in Fig-

a) Pattern
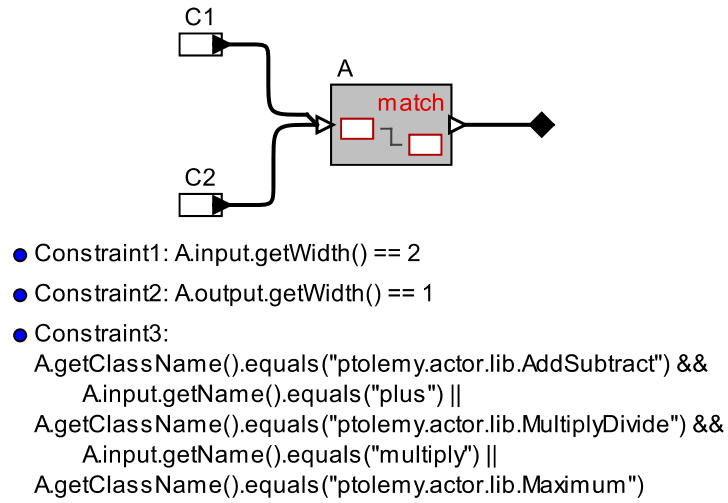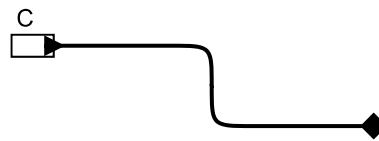
b) Replacement

	Pattern Object	Replacement Object
1	C1	C
2	Relation	Relation

c) Correspondence

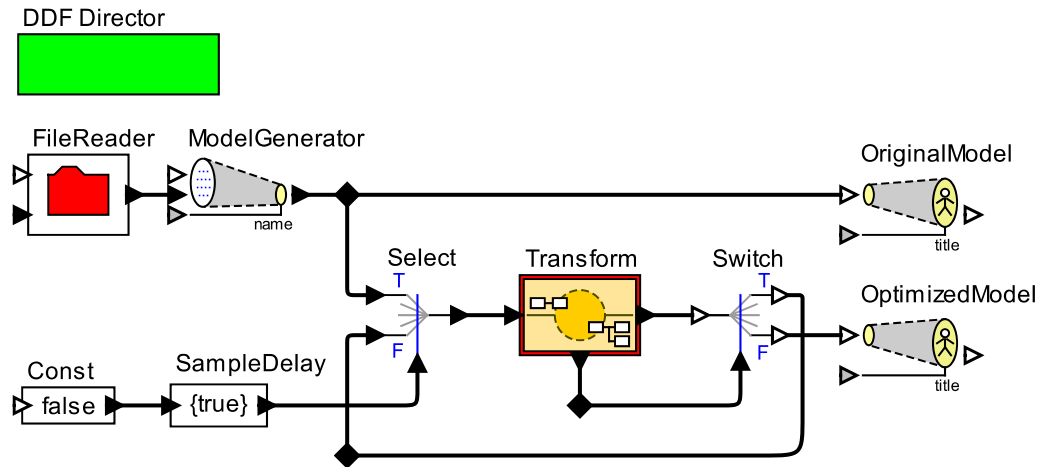Figure 4.4: The transformation rule to eliminate an arithmetic operation in the Constant model.

Figure 4.5: A DDF model equivalent to the transformation workflow to optimize a Constant model.

ure 4.5. That DDF model generates exactly the same result. The same transformation rule in Figure 4.4 is encapsulated in the Transform actor. Each time a model is received at its input port, the transformation rule is applied, and the transformed model is produced via the Transform actor's right output port. A true or false value is also produced via the Transform actor's bottom output port. That value signifies whether the transformation was successful.

As shown later, the transformation workflow has much better run-time performance than the DDF model, though they use the same transformation rule and the number of applications of that rule is also the same. The reason is that the workflow does not suffer from the overhead caused by message passing between actors, whereas the DDF model does.

4.2   Programming Interface for Events

Besides the tasks that perform basic transformations, such as Transform in Figure 4.1, other tasks with special actions are usually needed, such as those reading models in at the beginning and writing models out at the end. To allow users to customize tasks, I define a programming interface for events in Ptera models.

The programming interface consists of a few extension points and an invocation protocol. To customize a type of events and use it in a model, the following methods invoked at the extension points need to be defined.

- *Preinitialize.* Preinitialize the event.

- *Initialize.* Initialize the event.

- *Fire.* Execute the event's customized actions.

- *Finalize.* Finalize the event.

When a Ptera model is preinitialized, all the events in it are also preinitialized with the Preinitialize methods defined for them. Submodels associated with the events are preinitialized afterwards, if any. In initialization, events are initialized, but the submodels are not. The submodels are initialized only when the events that they are associated with are processed, as discussed previously.

When an event is being processed, the actions explicitly specified in its "actions" attribute are executed first. Then its Fire method is invoked, where extra customized

actions can be executed. Those extra actions are usually implicit and are not visible in the event's visual representation. After the Fire method, the submodel associated with that event is fired, if there is one. After that, all outgoing scheduling relations and canceling relations are evaluated, and new events may be scheduled in the event queue.

At the end of execution, all the events are finalized before the Ptera model is finalized. The submodels that have been initialized but have not yet been finalized are finalized as well.

4.3 An Event Library for Model Transformation

I created a library of events that are commonly used in transformation workflows by implementing the programming interface. Those events are listed in Figure 4.6 with special icons to help designers recognize them (because the customized actions are not visually represented). The model that some of those events operate on need not be explicitly specified, because it is stored in the Model variable.

The workflows in Figures 1.4 and 4.1 demonstrate the use of those events. Those workflows are instances of a predefined actor-oriented class [45, 43]. The class contains the basic structure of any workflow, including an initial event and the Model variable. By inheriting from the class, the instances automatically acquire those objects. In the visual representation, the icons of the inherited objects are surrounded with pink boxes.
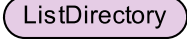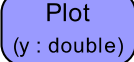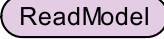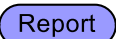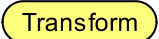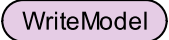
Variable	Meaning
Model 🧍	Temporarily store the model to be transformed

Event	Action
Configure	Open a dialog for the user to input values of some variables
Execute	Execute the model in the Model variable to completion
InitModel	Set the Model variable to contain an empty model
InitModelWithContainer	Set the Model variable to contain the container of this workflow
ListDirectory	List the names of all files in a directory
Match	Match a pattern with the model in the Model variable
Plot (y : double)	Plot the value of y (or other variables) at the current time
ReadModel	Read the model stored in a file into the Model variable
ReceiveInput	Receive input and make the input available after t in model time
Report	Report a message or an error to the user
Sleep	Wait for a certain amount of real time (in seconds)
Stop	Clear the event queue
Test	Perform testing (on scheduling relations) with no side effect
Transform	Transform the model in the Model variable
View	Show the model in the Model variable in a window
WriteModel	Output the model in the Model variable into a file

Figure 4.6: The event library for transformation workflows.

4.3.1 The Model Variable

The model to be transformed is stored in the Model variable of the workflow. In Figure 4.1, it is represented with an actor icon because the model stored in it is essentially a composite actor. For a hierarchical workflow such as Figure 1.4, the Model variable should be at the top level, so that it is visible to all submodels with bottom-up name resolution for the hierarchy.

At the beginning of execution, the Model variable is initialized to contain an empty model. Such a model is a composite actor with no object inside. Events processed during the execution may update that variable, either by storing a new model in it, or by modifying the existing model.

Shared variables as a means of data exchange between events do not lead to uncertainty of model behavior or performance loss. If dataflow is used, as is in Figure 4.5, to share data, tokens (data packages to be exchanged between actors) need to be created and destroyed. That causes a few problems.

1. Frequently creating and destroying tokens cause performance overhead and memory fragmentation.

2. To preserve semantics, it is necessary to ensure that tokens from one sender to multiple receivers carry the same data. If the tokens can be altered, an extra mechanism is needed to make sure that the changes done by one receiver are not visible by the others. One such mechanism is to duplicate the tokens sent

to multiple receivers. On the other hand, if tokens are immutable, then a new token is needed every time the data encapsulated in it are changed, even if the change is small.

3. If the actors exchange data with many other actors, a lot of connections between their ports exist in the diagram, making the design unreadable. Design flaws are easily overseen.

There have been extensions to dataflow that allow shared variables to be used. An example is found in the GReAT [1] model transformation tool, where implicitly defined shared variables store models to be transformed. The dataflow models of computation in Ptolemy II are another example, where variables can be defined and SetVariable actors can be used to update their values. In both approaches, to obtain deterministic behavior, actors operating on the same shared variable must be triggered in a specific order, which may not be obvious in the dataflow diagram.

4.3.2 Match and Transform Events

A key component of the event library in Figure 4.6 is the Transform event. Its action performs a basic transformation specified by the designer with a transformation rule. For example, the Transform event in Figure 4.1 encapsulates the basic transformation in Figure 4.4.

Depending on a mode parameter of the Transform event, when multiple occurrences of the pattern are found, one or more of them are transformed. The predefined

modes are listed below.

- *Transform first* and *transform last*. Transform only the first or the last occurrence. Before pattern matching, the model structure is linearized, so that the objects in the model are totally ordered in a sequence. Each occurrence of the pattern is a subsequence of that sequence. (*Subsequence A* of B is a sequence that consists of only elements in B and for any two elements x and y in A, x appears before y in A if and only if x appears before y in B.) Therefore, all distinct occurrences of a pattern can also be totally ordered using lexicographic order. In the transform first mode, the first occurrence among all is used for the transformation; in the transform last mode, the last occurrence is used.

- *Transform randomly*. Transform a randomly selected occurrence if there are multiple. A pseudo random number generator is used to make the selection. This mode is especially useful for testing.

- *Transform all*. Collect all occurrences of the pattern, and try to transform all of them in the previously described total order. Transforming an earlier occurrence may invalidate latter ones, for example, by making some objects in the latter ones no longer exist. In that case, the invalidated occurrences are ignored.

A Transform event has a "matched" variable. It stores a Boolean value that identifies whether the pattern matching was successful last time when the Transform event was processed. Initially its value is true. A simple way to repeatedly perform a

basic transformation until fixpoint is to create a scheduling relation from a Transform event back to itself with guard "matched." This causes the transformation to be performed until no more occurrence of the pattern is found.

The Match event in the event library is a light-weight version of the Transform event, since it only performs pattern matching but not transformation. It also has a "matched" variable to identify whether the last pattern matching was successful. It can be considered as a special case of the Transform event, whose transformation rule preserves all matched objects.

4.3.3 Auxiliary Events

Some of the auxiliary events in Figure 4.6 are described as follows.

- The InitModel event sets the Model variable to contain an empty model. It is usually an initial event of a workflow.

- The ReadModel event has a modelFile parameter that specifies the name of the file to be read. When that event is processed, it reads that file, parses the file's contents into a model, and stores that model in the Model variable. The original model is discarded.

- The Test event is a trivial event with no action. Conditions are tested by the guards of its outgoing scheduling relations.

- The View event shows the model in a separate window. Each time it is pro-

cessed, it updates the same window with the current model in the Model variable. In the example in Figure 4.1, ViewOriginalModel and ViewOptimized-Model are both View events. They show the original model and the transformed model in two different windows.

4.4 Performance

I assess the performance of model-based transformations using Ptera as the control language by comparing them against equivalent transformations controlled by DDF. Two test cases are selected for discussion here.

4.4.1 Transformation Test between Ptera and DDF

In the first test case, I compare the models in Figure 4.1 and Figure 4.5. They both optimize an input model stored on the disk by statically evaluating arithmetic operations with constant inputs. The basic transformation to be applied in each step is the one in Figure 4.4. That basic transformation is encapsulated in the Transform event in Figure 4.1 and the Transform actor in Figure 4.5.

Suppose the input model has n constant actors, where n is called the *input size*. The total number of actors is $2 \times n$ including actors for arithmetic operations and display. To obtain the final answer, the basic transformation needs to be applied $n-1$ times (leaving only the last constant actor).

Figure 4.7: Run time for the transformation test between Ptera and DDF.
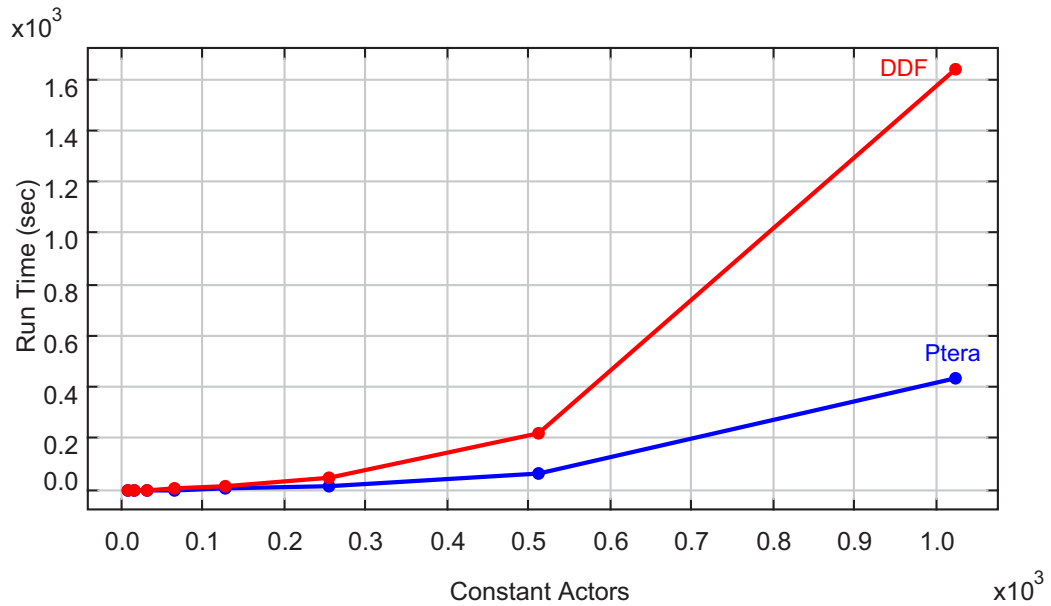
Input Size n	Ptera (sec)	DDF (sec)
8	0.081	0.108
16	0.092	0.192
32	0.204	0.614
64	0.654	2.420
128	2.544	10.546
256	11.094	43.695
512	64.429	222.628
1024	436.989	1635.023

An example with input size $n = 8$ is shown in Figure 4.2, and a possible optimization result is in Figure 4.3. There are other possible results, because the mode of the basic transformation is set to be "transform randomly," so that the transformation operates on a random occurrence of the pattern in the input model when multiple are found. Different constant actors may remain in the results, but their output values are always 21. In that sense, the behavior of the input model is preserved.

Figure 4.7 shows the run time for both models. Not only is the run time of the Ptera model much smaller than that of the DDF model, it grows slower as the input size increases.

4.4.2 Identity Test between Ptera, DDF and C generated from DDF

In the second test case, I modify the basic transformation in Figure 4.1 and Figure 4.5 to be an identity operation, which matches the same pattern as before but preserves all the objects intact. I obtain C code from the DDF model (with small modification) using the code generation framework in Ptolemy II [63, 62]. An identity function in the C code corresponds to the identity operation in both models. That function simply copies the data representing model structures received as a parameter and returns the copies.

The purpose of including the C code generated from the DDF model in this test case is to highlight the fact that the performance gain with Ptera compared to DDF is due to the difference in their data sharing mechanisms. Since the C code uses the same data sharing mechanism as DDF, it also suffers from the inefficiency caused by the use of tokens. The tokens in the DDF implementation are immutable. The C function corresponding to the identity operation creates new tokens to store the return values. (Even though the operation is identity, there is no way to assert that in general, so tokens still need to be created.) Tokens are destroyed to free memory

<div align="center">
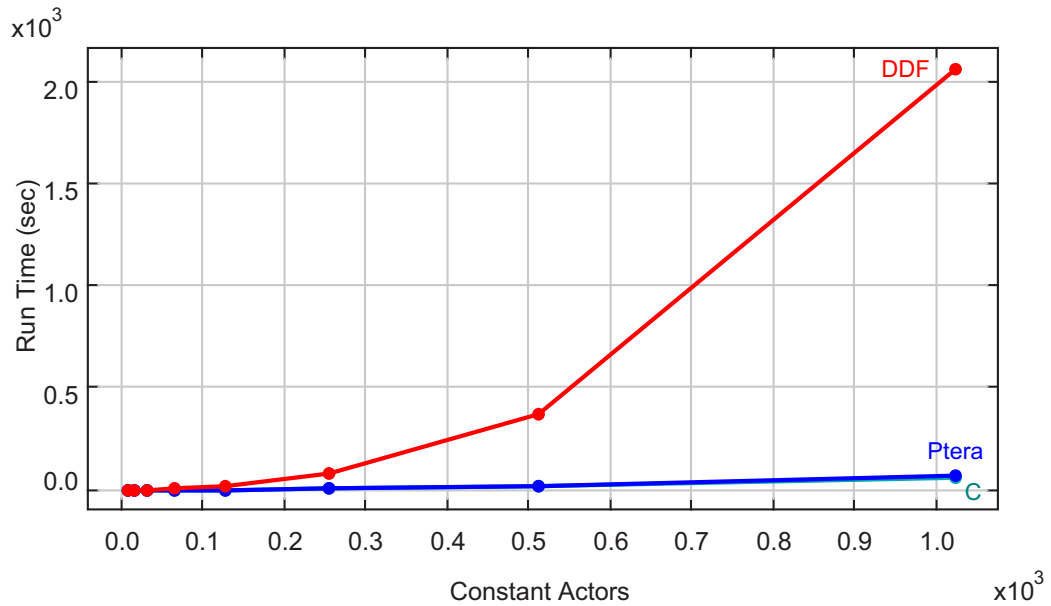
Input Size n	Ptera (sec)	DDF (sec)	C (sec)
8	0.016	0.113	0.015
16	0.032	0.250	0.031
32	0.063	0.875	0.078
64	0.188	4.088	0.218
128	0.750	17.254	0.717
256	3.261	74.850	3.276
512	14.431	366.679	12.386
1024	72.736	2054.172	59.764

</div>

Figure 4.8: Run time for the identity test between Ptera, DDF and C (generated from DDF).

when they are no longer needed.

The reason for changing the basic transformation to an identity operation is that model structures cannot be altered in the generated C code. Both the Ptera model and the DDF model are modified so that they still terminate within $n-1$ steps though the identity operations in them do not change the input model and can always be applied.

Figure 4.8 shows the run time for all three competitors. The two lower lines in the figure overlap. They show the run time of the Ptera model and the C code. The exact numbers are listed in the table below the figure.

The Ptera workflow still runs much faster than DDF. Interestingly, its performance is even comparable to the C code, despite the facts that the Ptera execution engine is implemented in Java with no static optimization on the workflow, that the C function simply copies input to output without performing pattern matching, and that various optimizations have been done in the C code such as static dataflow scheduling.

Chapter 5

Applications

Model transformation plays an important role in the model-driven architecture (MDA) [54]. Example applications provided in this chapter highlight the following key ideas.

- The Ptera model of computation facilitates the design of hierarchical transformation workflows. Similar to activity diagrams in the UML (Unified Modeling Language), Ptera models explicitly visualizing dependency relationship between tasks are easy to understand. Compared to activity diagrams, Ptera models additionally support hierarchical design and scheduling of future tasks. This significantly improves expressiveness.

- Model transformation is specified in the same modeling language as the one that designers use to create models. An environment for model design and simulation (for instance, Ptolemy II) can be reused for model transformation.

The need for learning a new language is eliminated, and hence the cost and the risk of errors are reduced.

- Model-based transformations are themselves models. Tools for ordinary models can be easily applied to them. Such tools include model analysis, model transformation, model checking, code generation, and so on.

5.1 Model Optimization

I have shown model optimization workflows in Figures 1.4 and 4.1. They both automatically search for occurrences of patterns in the model that can be reduced statically. The results are more compact models with several benefits.

1. Model size is reduced, making it easier to fit the models in a limited amount of memory.

2. Fewer operations need to be performed at run time, so the models potentially execute faster.

3. If an operation is eliminated completely, there is no need to provide an implementation of it. For example, in Figures 1.2, the division operation found in Figure 1.1 is removed. If code is generated from the model to run on an FPGA, there is no need to implement division at all.

4. If operations that compromise certain properties are eliminated, those proper-

ties may be regained after transformation. For example, in Figure 4.2, assume the execution time for the arithmetic operation actors is unknown. There is no way to assert real-time properties for the model. However, after the model is transformed into Figure 4.3, it may be possible to prove real-time properties.

5.2   Simulation

Model transformation can be applied to simulate system evolution. An example to be given here is Conway's game of life [27]. It starts with an $m \times n$ matrix where cells can be either occupied or unoccupied. The matrix evolves with iterations following a set of rules. Occupancy of its cells in the next iteration is determined by the current state of the cells and their neighbors (the cells directly adjacent to them horizontally, vertically or diagonally).

1. If a cell is occupied and has less than 2 or more than 3 occupied neighbors, it becomes unoccupied in the next iteration.

2. If a cell is unoccupied and has exactly 3 occupied neighbors, it becomes occupied in the next iteration.

3. All other cells remain unchanged in the next iteration.

The model-based transformation in Figure 5.1 simulates the evolution of the game-of-life system. The simulation terminates when the system stabilizes (that is, no cell changes in any future iteration).
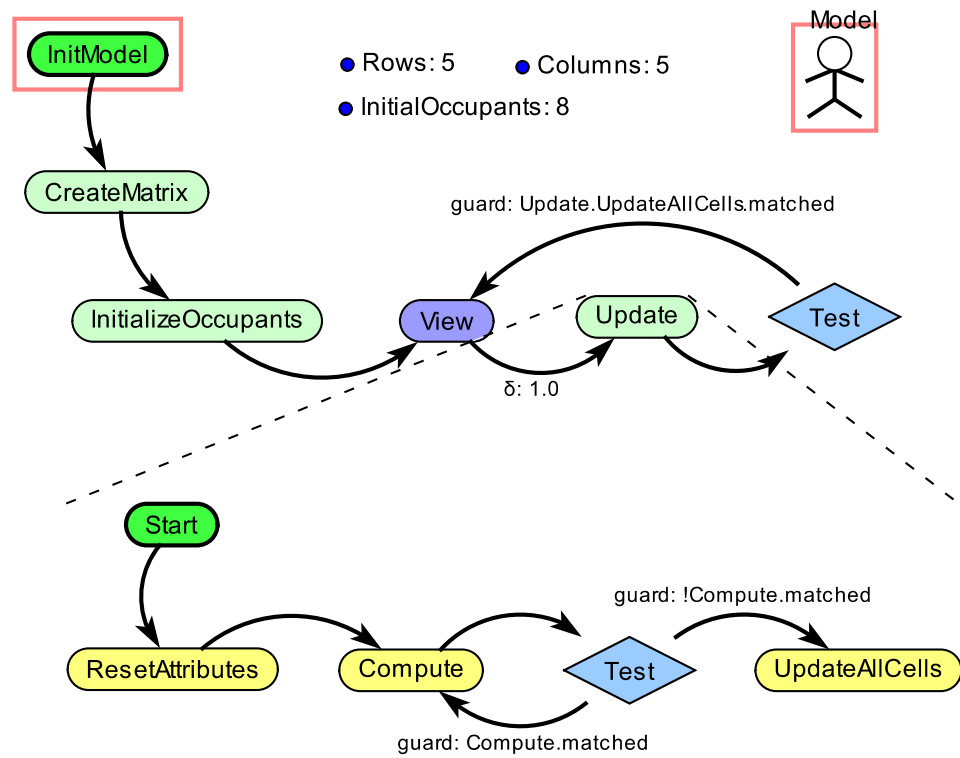
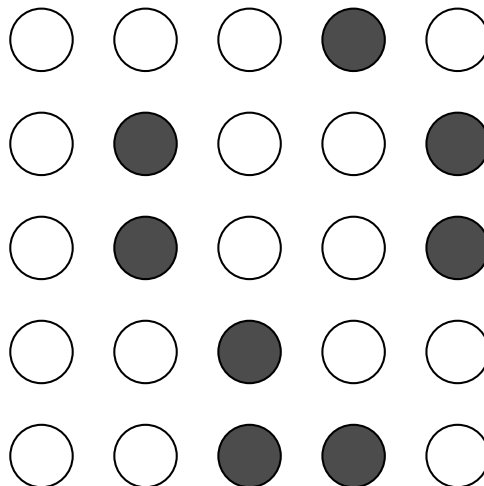Figure 5.1: A model-based transformation to simulate the game-of-life system.

Figure 5.2: The initial configuration of a randomly generated game-of-life system.

The matrix (as is shown in Figure 5.2, where filled nodes represent occupied cells) is generated by the CreateMatrix event, whose submodel is not shown in the figure. The size of the matrix is determined by the top-level Rows and Columns variables. There are hidden connections from each cell to its neighbors. Those connections help with pattern matching. Without them, all cells would have been unrelated. The four borders of matrix are conceptually stitched together. The leftmost cells are neighbors of the rightmost cells and vice versa. So are the top and bottom cells.

The InitializeOccupants event randomly initializes some cells of the matrix to be occupied represented with filled nodes. The number of occupied cells is determined by the InitialOccupants variable. A special attribute is associated with each occupied cell, which can be tested by a transformation rule.

When processed the first time, the View event displays the initial matrix in a separate window. It then schedules update of the matrix to occur after 1 unit of model time, assuming that is the duration for each iteration. If the simulation engine is set to be synchronized with real time, that delay roughly corresponds to 1 second in real time. [1]

After the delay, the submodel associated with the Update event (as is shown in the lower part of Figure 5.1) is executed. It updates the matrix according to the above-described rules. Its Compute event encapsulates the key basic transformation that computes the updating. Each time that basic transformation is applied, the next

---

[1] Accuracy of the synchronization with real time is not a subject of this research.

state of a chosen cell is computed. That cell is then marked to have been handled so that it will not be reconsidered in the same iteration.

The Compute event is repeatedly processed until its "matched" variable becomes false, which means all cells have been handled and marked. At that point, the Test event schedules UpdateAllCells to update the appearance of the matrix. If UpdateAll-Cells changes at least one cell, its "matched" variable is set to true, and at the top level the View event is scheduled again since the guard of the scheduling relation from the top-level Test event tests the truth of that variable. However, if no cell is updated by UpdateAllCells, a stable state has been reached and the matrix will not change in the future. The simulation then terminates, because the Test event at the top level would not schedule the View event again.

When it is executed with model time synchronized with real time, the transformation workflow produces an animation. The separate window showing the current state of the matrix is updated at the rate of approximately one frame per second. In the future work, leveraging the PRET (precision timed machines) real-time execution platform [17, 48] and the PTIDES (programming temporally integrated distributed embedded systems) discrete-event execution strategy [60, 23, 24, 15], timing of the animation can be made much more accurate, and real-time properties may also be statically verified.

5.3 Configurable Product Families

The concept and practice of product families [5, 10] have received substantial interests in industry. A *configurable product family* [59] consists of products obtained by configuring common components or design patterns with a *configurer*. Those components and designs can be easily reused.

To provide a proof-of-concept example, I re-examine a higher-order actor-oriented model originally introduced in [8]. That model is designed for a word-counting MapReduce application. It uses Google's MapReduce programming paradigm for distributed systems [14] to count occurrences of words in a document in a divide-and-conquer manner. The model can have an arbitrary number of Map and Reduce actors, so it can also be considered as a family of products.

A Map actor in the model corresponds to a Map function that receives a pair of key and value, and outputs a list of key-value pairs. A Reduce actor corresponds to a reduce function that receives a key and a list of values, and outputs a list of new values. The number of Map and Reduce actors is determined by a parameter provided by the user. Other actors in the model serve to generate initial inputs to the Map actors and assemble final outputs from the the Reduce actors.

In the original work in [8], the model is created with a Ptalon description. Two observations are made.

- *Scalability.* Increase of Map and Reduce actors only requires change of a parameter. The Ptalon description need not be modified.

- *Correctness.* The result of the model is not affected by the number of Map and Reduce actors, though more such actors potentially improve concurrency and may lead to better performance in a distributed system.

Here I re-examine the application and use model transformation as the configuration mechanism rather than Ptalon. The purpose is to show that higher-order component composition and generation of a family of products can also be achieved with model transformation, which has the additional benefits of a visual language close to the modeling language that designers are familiar with.

A product family configured with model transformation consists of two components:

- a design template that can be easily transformed into any desired product, and

- a set of model transformations that configure the template to generate those products.

(This definition can certainly be generalized such that multiple templates are involved, and the set of model transformations themselves form a product family.)

Figure 5.3 specifies the same family of MapReduce products as that in [8]. The template is the simplest MapReduce application containing only one Map actor and one Reduce actor. It can be enriched with more Map and Reduce actors. The three transformation attributes associated with it, each having a "T:" at the beginning of the label, specifies how the template can be configured to generate a product, how
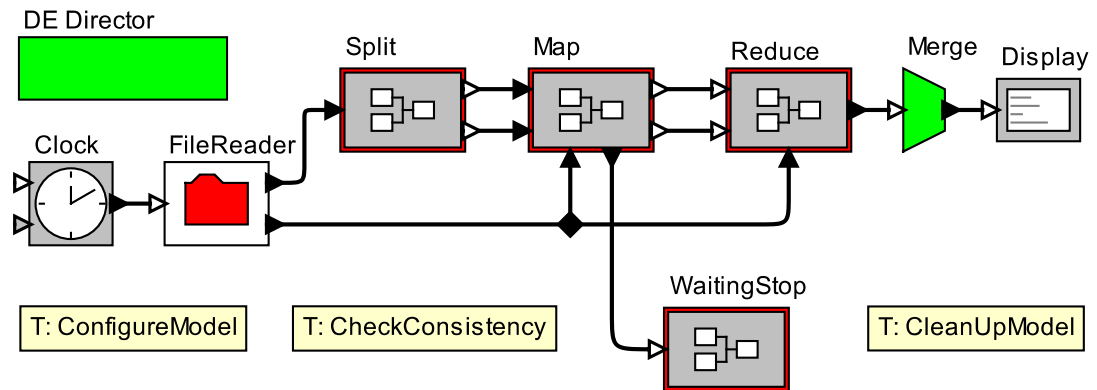
Figure 5.3: Design of the MapReduce product family.

certain structural properties can be checked, and how a generated product can be cleaned up to retrieve the template.

A possible product with two Map actors and two Reduce actors is displayed in Figure 5.4. In general, the right output ports of a Map actor should be connected to the left input ports of any Reduce actor. The left input ports of a Map actor should be connected to the unique Split actor that splits trunks of documents to all the Map actors. The right output port of a Reduce actor should be connected to the unique Merge actor that merges all results to the display. The vertical ports of Map actors and Reduce actors should be connected to FileReader and WaitingStop. Those ports receive end-of-file signals and output job finishing signals. Further increase of Map and Reduce actors in a larger MapReduce application would easily lead to a design that is not manually manageable.

Among the transformation attributes in Figure 5.3, ConfigureModel is used to
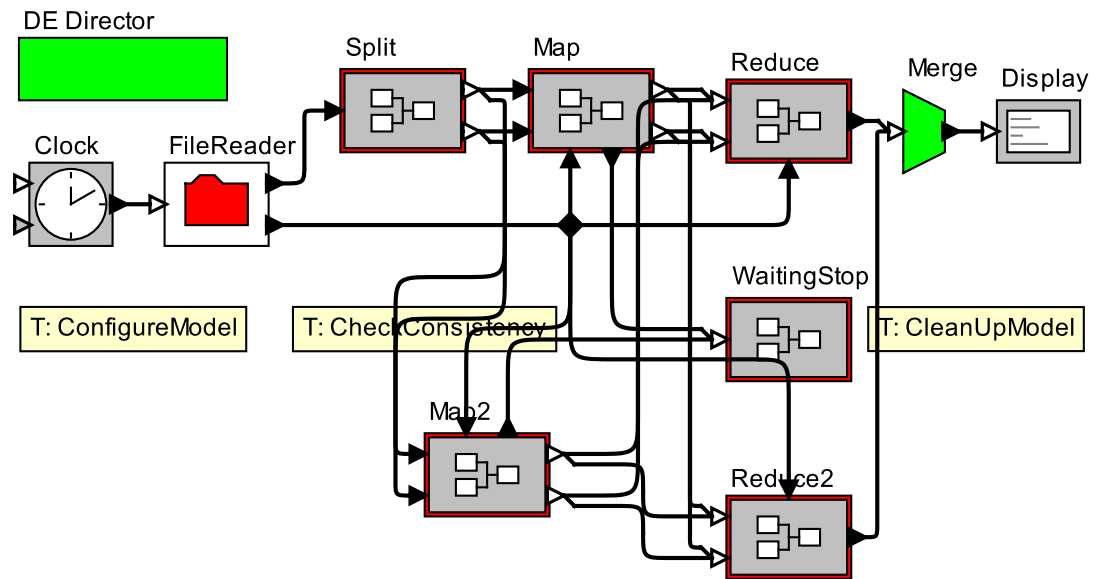
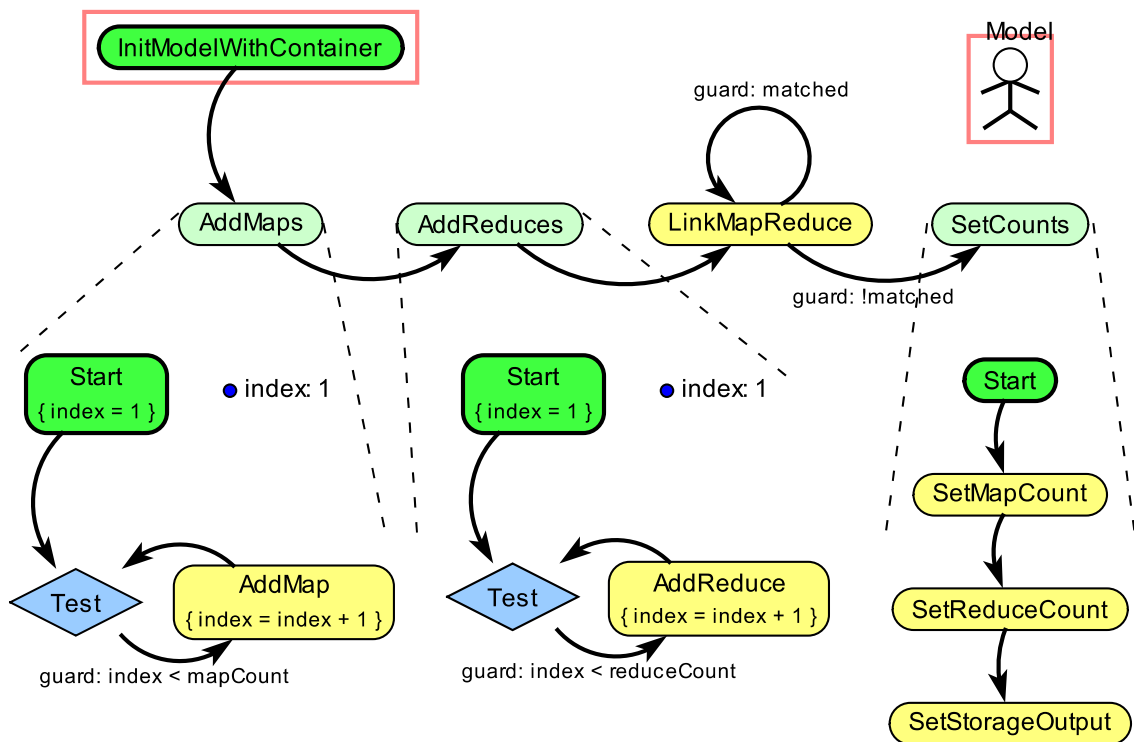Figure 5.4: A particular MapReduce product with 2 Map actors and 2 Reduce actors.

Figure 5.5: Model-based transformation encapsulated in the ConfigureModel transformation attribute.

configure the template and to generate a desired product. It has two parameters (which is not shown in the visual representation) to control the numbers of Map actors and Reduce actors. The internal design of the ConfigureModel attribute is shown in Figure 5.5. Notice that, since ConfigureModel is not an actor but an attribute, the transformation workflow in it is not executed as the model executes. Instead, the workflow is invoked explicitly by the user. Its initial event is InitModelWithContainer, which sets the Model variable to contain the template (because it is the container of the ConfigureModel attribute).

The first and second steps of generating a product are to create new Map actors and new Reduce actors in addition to the two in the template. After that, the new actors are connected with the basic transformation in LinkMapReduce. Lastly, certain local variables internal to the Map and Reduce actors are set with the SetCounts event.

The CheckConsistency transformation attribute is designed to check structural properties, such as all Map and Reduce actors being connected as required, and all Map actors being connected to the Split actor. The attribute encapsulates a transformation workflow to match predefined invalid patterns in the model. If any is found, a Report event is processed to pop up a dialog describing the problem. This helps the designer to detect design flaws.

The CleanUpModel transformation attribute is used to clean up all the generated actors and connections to restore the template from a product generated with it.

In this application, model transformations are associated with the design and can be provided to the model users. Since the transformations are specific instead of general-purpose, extra assumptions can be made. Checking of structural properties is easier with those assumptions.

Chapter 6

Conclusion

The research behind this dissertation focuses on developing a flexible and efficient model transformation mechanism for scalable model construction and maintenance. I study two classes of model transformations: basic transformations specified with transformation rules that can be applied to models as an atomic operations, and model-based transformations that are models themselves containing basic transformations as computation units.

For basic transformations, I show that any input model can be represented as an attributed graph, and graph transformation can be applied. I adapt the single pushout approach to formally define the semantics of transformation rules. I provide a syntax close to the modeling language that model designers are familiar with, so it is convenient to use and it saves time and cost. As an extension to traditional graph transformation, I leverage and enhance the Ptalon higher-order composition

language, with which complex transformation rules and those operating on variable model structures can be specified with compact and parametrizable descriptions.

Application of basic transformations requires pattern matching, which is essentially to solve subgraph isomorphism problems. Those problems are *NP-hard* in general. To make transformations scalable, I develop an approach to composing basic transformations and controlling them with a model. Based on event graphs, I created the Ptera (Ptolemy event relationship actor) model of computation as the control language. I operationally define the semantics in an abstract framework for model execution. By defining the semantics in this way, I enable Ptera to be hierarchically composed with existing models of computation to achieve high design flexibility.

Ptera has advantages over prevailing control languages for model transformation based on state machines and dataflow. The event queue and the notion of model time allow multiple transformation tasks to be scheduled for future processing. Better expressiveness is obtained compared to state machines, since run-time state need not be bounded. Performance overhead found in dataflow due to large data packages for communication is eliminated.

Among the wide range of potential applications for model-based transformation, I provide as examples model optimization, simulation of system behavior and configuration of product families.

Bibliography

[1] Aditya Agrawal, Gabor Karsai, and Feng Shi. A UML-based graph transformation approach for implementing domain-specific model transformations. *International Journal on Software and Systems Modeling*, 2003.

[2] András Balogh and Dániel Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1280–1287, Esslingen, Germany, October 2006.

[3] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[4] Gérard Berry. The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies, 2003.

[5] Jan Bosch. *Design and use of software architectures: Adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[6] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, and Haiyang Zheng. Heterogeneous concurrent modeling and design in Java (volume 1: Introduction to Ptolemy II). Technical Report UCB/EECS-2008-28, EECS Department, University of California, Berkeley, April 2008.

[7] Adam Cataldo, Elaine Cheong, Thomas Huining Feng, Edward A. Lee, and Andrew Christopher Mihal. A formalism for higher-order composition languages that satisfies the Church-Rosser property. Technical Report UCB/EECS-2006-48, EECS Department, University of California, Berkeley, May 2006.

[8] James Adam Cataldo. *The power of higher-order composition languages in system design*. PhD thesis, EECS Department, University of California, Berkeley, December 2006.

[9] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, January 2000.

[10] Paul Clements and Linda Northrop. *Software product lines: Practices and patterns*. Addison-Wesley Professional, 2001.

[11] Arturo I. Concepcion and Bernard P. Zeigler. DEVS formalism: A framework for hierarchical model development. *IEEE Transactions on Software Engineering (TSE)*, 14(2):228–241, February 1988.

[12] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Al-

gebraic approaches to graph transformation – part I: Basic concepts and double pushout approach. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*, pages 163–245, 1997.

[13] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation (OSDI)*, pages 137–150, San Francisco, California, USA, December 2004.

[15] Patricia Derler, Thomas Huining Feng, Edward A. Lee, Slobodan Matic, Hiren D. Patel, Yang Zhao, and Jia Zou. PTIDES: A programming model for distributed real-time embedded systems. Technical Report UCB/EECS-2008-72, EECS Department, University of California, Berkeley, May 2008.

[16] Patricia Derler, Edward A. Lee, and Slobodan Matic. Simulation and implementation of the PTIDES programming model. In *Proceedings of the 12th IEEE International Symposium on Distributed Simulation and Real Time Applications*, Vancouver, Canada, October 2008.

[17] Stephen A. Edwards and Edward A. Lee. The case for the precision timed (PRET) machine. In *Design Automation Conference (DAC)*, pages 264–265, San Diego, CA, USA, June 2007.

[18] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation – part II: Single pushout approach and comparison with double pushout approach. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*, pages 247–312, 1997.

[19] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In *Annual Symposium on Foundations of Computer Science (FOCS)*, pages 167–180, 1973.

[20] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graph transformation. In *International Conference on Graph Transformation (ICGT)*, pages 161–177, Rome, Italy, 2004.

[21] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, Stephen Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity – the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[22] Rainer Fehling. A concept of hierarchical Petri nets with building blocks. In *Proceedings of the 12th International Conference on Application and Theory of Petri Nets*, pages 148–168, London, UK, 1993. Springer-Verlag.

[23] Thomas Huining Feng and Edward A. Lee. Real-time distributed discrete-event execution with fault tolerance. In *14th IEEE Real-Time and Embedded Tech-*

nology and Applications Symposium (RTAS 2008), St. Louis, MO, USA, April 2008.

[24] Thomas Huining Feng, Edward A. Lee, Hiren D. Patel, and Jia Zou. Toward an effective execution policy for distributed real-time embedded systems. In *14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008), Work-in-Progress Session*, St. Louis, MO, USA, April 2008.

[25] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[26] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story diagrams: A new graph rewrite language based on the Unified Modeling Language and Java. In *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT)*, pages 296–309, Paderborn, Germany, 1998.

[27] Martin Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, October 1970.

[28] Alain Girault, Bilung Lee, and Edward A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–760, 1999.

[29] Antoon Goderis, Christopher Brooks, Ilkay Altintas, Edward A. Lee, and Carole A. Goble. Composing different models of computation in Kepler and Ptolemy II. In *International Conference on Computational Science (ICCS)*, pages 182–190, Beijing, China, May 2007.

[30] Yuri Gurevich. Evolving algebras. In *IFIP World Computer Congress*, pages 423–427, Hamburg, Germany, 1994.

[31] Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.

[32] Cécile Hardebolle and Frédéric Boulanger. ModHel'X: A component-oriented approach to multi-formalism modeling. In *Model Driven Engineering Languages and Systems (MoDELS)*, pages 247–258, Nashville, TN, USA, September 2007.

[33] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[34] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[35] Ricki G. Ingalls, Douglas J. Morrice, and Andrew B. Whinston. Eliminating canceling edges from the simulation graph model methodology. In *WSC '96:*

Proceedings of the 28th conference on Winter simulation, pages 825–832, Washington, DC, USA, 1996. IEEE Computer Society.

[36] Axel Jantsch and Ingo Sander. Models of computation and languages for embedded system design. *IEEE Proceedings on Computers and Digital Techniques*, 152(2):114–129, 2005.

[37] Gilles Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing*, pages 471–475, Stockholm, Sweden, August 1974. North Holland Publishing Company.

[38] Gilles Kahn and David B. Macqueen. Coroutines and networks of parallel processes. *Information Processing*, pages 993–998, 1977.

[39] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *ESEC/FSE: Proceedings of the 10th European Software Engineering Conference / 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 187–196, New York, NY, USA, 2005. ACM.

[40] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56, London, UK, 2001. Springer-Verlag.

[41] Juan de Lara and Hans Vangheluwe. AToM³: A tool for multi-formalism and

meta-modelling. In *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, Grenoble, France, April 2002.

[42] Edward A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7(1–4):25–45, 1999.

[43] Edward A. Lee, Xiaojun Liu, and Stephen Andrew Neuendorffer. Classes and inheritance in actor-oriented design. Technical Report UCB/EECS-2006-154, EECS Department, University of California, Berkeley, November 2006.

[44] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[45] Edward A. Lee and Stephen Neuendorffer. Classes and subclasses in actor-oriented design. In *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 161–168, San Diego, California, USA, June 2004.

[46] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.

[47] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[48] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 137–146, Atlanta, GA, USA, October 2008.

[49] Yuan Lin, Robert Mullenix, Mark Woh, Scott Mahlke, Trevor Mudge, Alastair Reid, and Krisztian Flautner. SPEX: A programming language for software defined radio. In *Software Defined Radio Technical Conference and Product Exposition*, Orlando, November 2006.

[50] Jie Liu and Edward A. Lee. Component-based hierarchical modeling of systems with continuous and discrete dynamics. In *Proceedings of the 2000 IEEE International Symposium on Computer-Aided Control System Design*, pages 95–100, Anchorage, Alaska, USA, September 2000.

[51] Jie Liu and Edward A. Lee. A component-based approach to modeling and simulating mixed-signal and hybrid systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 12(4):343–368, October 2002.

[52] Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1–2):181–224, 1993.

[53] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, June 2000.

[54] Object Management Group (OMG). *MDA Guide 1.0.1*, June 2003.

[55] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[56] Lee W. Schruben. Simulation modeling with event graphs. *Communications of the ACM*, 26(11):957–963, 1983.

[57] Andy Schürr, Andreas J. Winter, and Albert Zündorf. Graph grammar engineering with PROGRES. In *Proceedings of the 5th European Software Engineering Conference*, pages 219–234, Sitges, Spain, September 1995.

[58] Gabriele Taentzer. AGG: A tool enviroment for algebraic graph transformation. In *Proceedings of Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, Kerkrade, The Netherlands, September 1999.

[59] Juha Tiihonen, Timo Lehtonen, Timo Soininen, Antti Pulkkinen, Reijo Sulonen, and Asko Riitahuhta. Modeling configurable product families. In *International Conference on Engineering Design*, volume 2, pages 1139–1142, Munich, Germany, August 1999.

[60] Yang Zhao, Jie Liu, and Edward A. Lee. A programming model for time-synchronized distributed real-time systems. In *IEEE Real-Time and Embed-*

ded Technology and Applications Symposium (RTAS), pages 259–268, Bellevue, Washington, USA, April 2007.

[61] Gang Zhou. Dynamic dataflow modeling in Ptolemy II. Technical Report UCB/ERL M05/2, EECS Department, University of California, Berkeley, December 2004.

[62] Gang Zhou. *Partial evaluation for optimized compilation of actor-oriented models*. PhD thesis, EECS Department, University of California, Berkeley, May 2008.

[63] Gang Zhou, Man-Kit Leung, and Edward A. Lee. A code generation framework for actor-oriented models with partial evaluation. In *ICESS '07: Proceedings of the 3rd international conference on Embedded Software and Systems*, pages 193–206, Berlin, Heidelberg, 2007. Springer-Verlag.

[64] Ye Zhou and Edward A. Lee. Causality interfaces for actor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–35, 2008.