

Engineering Structurally Configurable Models with Model Transformation

Thomas Huining Feng



Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-159

<http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-159.html>

December 15, 2008

Copyright 2008, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards \#0720882 (CSR-EHS: PRET) and \#0720841 (CSR-CPS)), the U. S. Army Research Office (ARO \#W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI \#FA9550-06-0312), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, HSBC, Lockheed-Martin, National Instruments, and Toyota.

Engineering Structurally Configurable Models with Model Transformation

by Huining Feng

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II.**

Approval for the Report and Comprehensive Examination:

Committee:

Professor Edward A. Lee

Research Advisor

Date

* * * * *

Professor Sanjit A. Seshia

Second Reader

Date

Abstract

Complex configurable models of embedded software systems are hard to design and maintain, especially when model structures are variable and the number of allowable configurations is unlimited. We employ model transformation as an underlying technique to configure model structures. A transformation tool is created for actor models that automates tasks in the development workflow. Such tasks include structural configuration with user-specified parameters, resetting enhanced models to simple forms for modification and update, and validating consistency in model structures. As an example, we provide a structurally configurable actor-oriented model of a distributed system using the MapReduce pattern to justify our approach.

1 Introduction

Recent years have seen a rapid growth in the functionality and complexity of embedded systems. More memory and computation power on embedded devices have allowed more large-scale and feature-rich embedded software to be developed. Networking capabilities have enabled distributed collaboration. Availability of multicore processors has created opportunities for computation parallelization. Functional correctness no longer satisfies users' requirements, and timely delivery of computation results is playing a more important role. These trends give rise to new research challenges. To reduce design time and to make complex design possible, computer aided design (CAD) tools must be employed to automate tasks. A systematic approach must be taken to consistently manage large numbers of interacting components. Support must be provided for the verification of behavioral correctness (such as deadlock freedom and determinacy) and the precise analysis of performance properties (such as resource consumption and execution time).

Compared to manually writing code, it is generally more productive to model embedded systems in a modeling environment, using models of computation with rigorously defined execution semantics. A simulator helps designers understand the execution and discover design flaws before actual deployment. Provided with a code generator, the designers can immediately obtain executable code with equivalent behavior.

In industry, modeling and simulation tools such as Simulink from the MathWorks and LabVIEW from National Instruments have been widely applied. Those tools provide extensible component libraries for model designers, who construct models by hierarchically composing components. The connections between components designate data communication channels. The communication semantics are defined by the models of computation supported by the tools (a timed model in the case of Simulink, and a dataflow model in the case of LabVIEW). Research modeling tools such as Ptolemy II [14], ForSyDe [21], SPEX [30], and ModHel'X [19] support hierarchical heterogeneous models of computation. They allow designers to freely mix models of computation to facilitate their embedded systems design.

For a realistic embedded system, the model may include thousands or even millions of components. Growth in system complexity increases not only the number of components horizontally on each level of the hierarchy, but also the number of vertical levels. Furthermore, dynamic changes of model structures, topology and characteristics are common in distributed embedded applications designed for ad-hoc mobile wireless networks [37]. Rapid growth of complexity is also found in parallel applications targeting multicore processors, in which tasks and communication must be configured according to the available cores [9]. A research challenge is to invent a scalable modeling paradigm for the design and maintenance of such large-scale and highly configurable models. Specifically, a desired mechanism must satisfy the following requirements:

1. It must allow to easily expand model structures, so that increasing model size does not necessarily require redesign of the trusted models.
2. It must have a well-defined semantics.

3. It must provide an intuitive interface to model designers.
4. It must support verification of properties.

We present our recent research on scalable construction and configuration of embedded systems. Our approach is based on model transformation. We show that our approach is able to scale models to arbitrary size based on a template, and that structural properties can be checked with pattern matching. We focus on the functionality of our model transformation tool built in the Ptolemy II framework, and its application to large models of distributed and parallel embedded systems. As secondary theme of this project, we give a visual representation based on an actor-oriented modeling language, we discuss the expressiveness of the hierarchical ERG (event relationship graph) model of computation, and we describe applications to other modeling languages and environments.

1.1 Actor Models

We focus on actor models in our approach to model construction and configuration. Although our tool was created for actor models in Ptolemy II, the same idea can be applied to other modeling tools such as Simulink, LabVIEW, ForSyDe, SPEX and ModHel'X. Moreover, the recent OMG (Object Management Group) standard MARTE (Modeling and Analysis of Real-time and Embedded systems) [35] provides a foundation for modeling real-time embedded systems with distinctly actor-oriented and visual flavors. Our tool can be adapted to transforming real-time system specifications with MARTE. Therefore, it has wide applications to a number of industrial and academic modeling tools.

In an actor model, components are *actors* that implement functions that map signals at their input ports to signals at their output ports. The wiring between output ports and input ports represents transmission of unaltered signals. An *atomic actor* is implemented with a piece of code in an imperative programming language, such as Java and C. A *composite actor* is a hierarchical composition of interconnected actors encapsulated as a single actor. The semantics of the communication is governed by the *director* in the composite actor, which implements a particular model of computation. Examples of models of computation supported by Ptolemy II are DE (discrete event) [25], SDF (synchronous dataflow) [28], FSM (finite state machine) and ERG (event relationship graph) [40]. A model may use different directors in its composite actors. This leads to a design with hierarchical heterogeneous models of computation [15].

1.2 Higher-Order Composition

Actor-oriented subclassing and higher-order model composition are two techniques to facilitate construction of large actor models and to improve design reusability.

In actor-oriented subclassing [22, 27], classes can be defined as the templates to generate composite actors. Each composite actor that is an instance of a class has exactly the same internal structure. If parameters are defined in the class, then each instance can have distinct values for the parameters, by this means acquiring distinct behavior. A class may also declare itself as a subclass of another class, so that it inherits the design from the latter. New actors and connections may be added in a subclass, but the inherited ones may not be removed. Actor-oriented subclassing helps to reuse existing designs in a structural way, and to extend their behavior. To create a large model with identical parts, such as a distributed embedded system with nodes that provide the same functionality, the designer may obtain multiple instances of a class. There is no need to copy the same design manually, and hence the possibility of making a mistake in the copying is eliminated. However, we still need a mechanism to automatically generate the instances for a large model. Ideally, for the example of a distributed system, the designer’s work can be further reduced if he or she can configure the number of nodes, and have the system itself generate all necessary instances and connect them correctly.

The idea of higher-order composition is to treat actors as first-class objects like primitive data values. A higher-order description defines the composition of those actors. If parameters are accepted by the description, then the composition may be configured by specifying values for those parameters. In the Ptolemy II framework, we have implemented a number of higher-order composition mechanisms. One of these mechanisms is a declarative language called Ptalon [6]. It provides a language construct to iteratively create model structures.

In [5] a model for a distributed application is shown. The model counts words in the documents collected from the web. It uses Google’s MapReduce programming paradigm. MapReduce is an abstraction for programming parallel and distributed systems. It provides a framework for performing computation with two user-defined functions [12]. Those functions are shown below, with k_1 and k_2 being possibly distinct sets of keys, v_1 and v_2 being possibly distinct sets of values, $\text{list}(v_2)$ being the set of lists of values in v_2 , and $\text{list}(k_2 \times v_2)$ being the set of lists of key-value pairs in $k_2 \times v_2$:

$$\begin{array}{lll} \text{Map:} & (k_1 \times v_1) & \rightarrow \text{list}(k_2 \times v_2) \\ \text{Reduce:} & (k_2 \times \text{list}(v_2)) & \rightarrow \text{list}(v_2) \end{array}$$

Initially, the MapReduce framework generates pairs of keys and values as inputs to Map. Each pair results in an invocation of Map that returns a list of keys and values. The framework then gathers those lists and stores the values associated with each distinct key into a new list. The Reduce function is then invoked with those keys and lists of values associated with the keys. The lists returned by the invocations of Reduce are combined to produce the final answer.

The Map and Reduce functions can be computed in a distributed system. Conceptually a computation block handles the a particular key for one of the functions. An arbitrary number of computation blocks can run on the machines in a distributed system, which effectively divide a complex computation task into small pieces to be accomplished concurrently. The MapReduce pattern has also been applied to heterogeneous multicore systems with shared memory [31, 7].

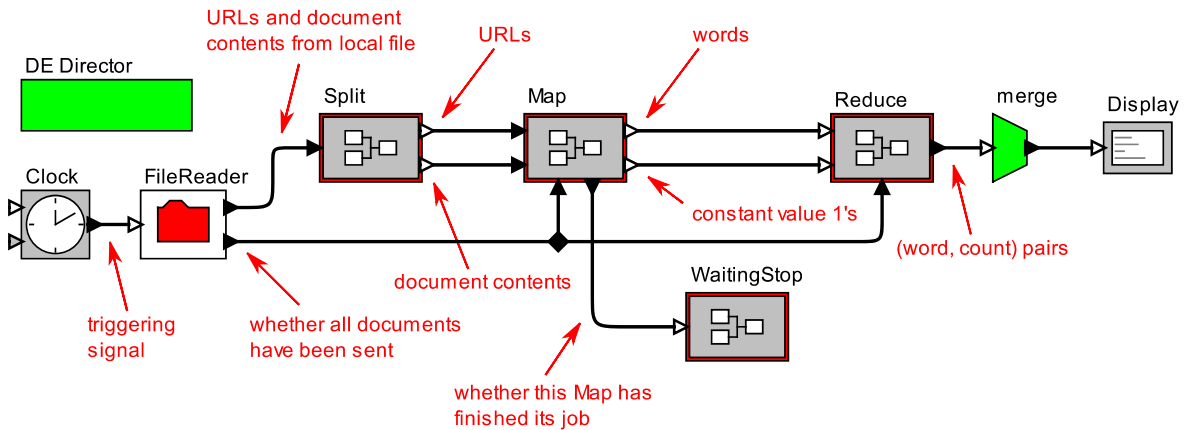


Figure 1: The word-counting model with one Map machine and one Reduce machine

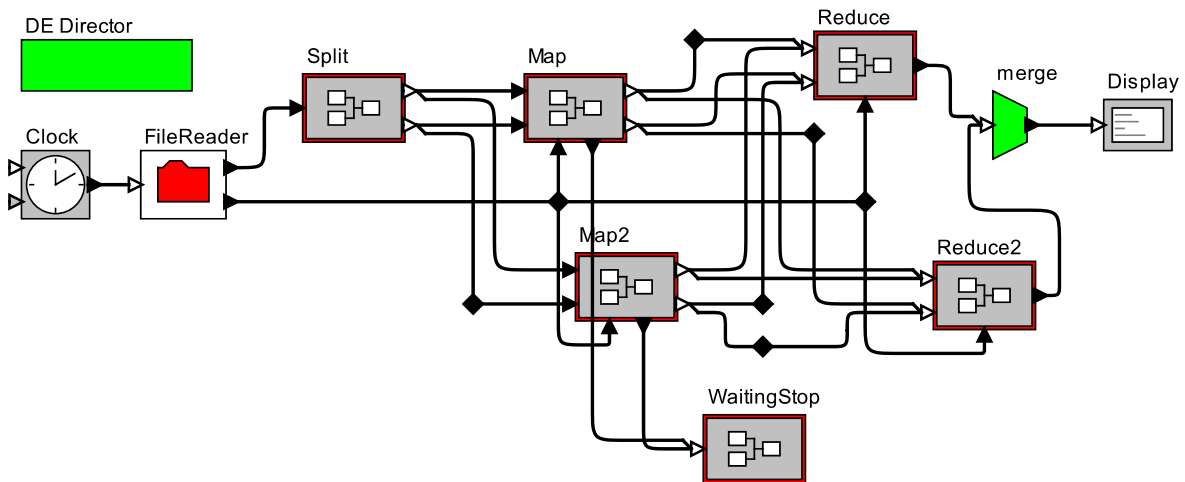


Figure 2: The word-counting model with two Map machines and two Reduce machines

Embedded systems applications that might have such structure include sensor fusion and target tracking. Many other applications have similar scaling requirements, albeit different structure.

In the word-counting example, a configurable number of composite actors are created with Ptalon. Each such actor simulates either a Map machine or a Reduce machine, on which one or more computation blocks for the Map or Reduce function are run. In one application of MapReduce, a Map machine accepts a document containing English words. For each appearance of a word, it computes the Hash code of the word and sends the word with value 1 to the Reduce machine designated by the Hash code. This guarantees that all appearances of a word are always sent to the same Reduce machine. On a Reduce machine, a computation block receives value 1's for the word that it handles, and it adds those values to a local counter. When all the documents are processed, the whole distributed computation is finished, and the values of all counters are sent to a display for output.

Fig. 1 shows the particular model consisting of one Map machine and one Reduce machine. A similar model with two Map machines and two Reduce machines is shown in Fig. 2. Those models produce the same display, with the only difference being the order of the word counts in the output, which is insignificant. We use DE as the top-level model of computation. The Clock is an atomic actor that serves as a source to trigger the FileReader. We assume that the URLs and document contents are stored in a file, which we consider as a database containing all data to be processed. The FileReader reads that file and produces alternating URLs and contents at its upper output port. It also produces true or false values at its lower output port indicating whether all documents have been sent. The Split composite actor evenly distributes the URLs and contents via its two output ports to the downstream Map actors. Each Map actor outputs words via its upper output port, and value 1's via its lower output port. Those ports are connected to the corresponding input ports of all Reduce actors. The Reduce actors send their word-counting outputs to the Merge at the end of the execution, which merges the data and sends them to the Display. When all documents have been processed and the WaitingStop receives stop signals from all Map actors, the execution terminates.

To generalize from these two examples, if we have m Map actors and n Reduce actors, there are exactly $2 \times m \times n$ connections between them. The number of other connections has order $O(m + n)$. Additionally, certain parameters within the Map actors and the Reduce actors need to be properly configured, so that those actors contain the correct number of conceptual computation blocks, and they send or receive data from the correct communication channels of their ports. This tedious work is automatically accomplished with higher-order composition.

A higher-order composition language similar to Ptalon is found in VIATRA2 (VIsual Auto-mated model TRAnsfOrmations) [2, 48], which is a text-based model transformation tool in Eclipse GMT (Generative Modeling Technologies). It also treats model structures as first-class objects, and provides loops and recursion in the abstract state machine language to construct models of arbitrary size.

Nevertheless, the current higher-order composition techniques have several limitations:

1. They require model designers to write descriptions in textual languages, despite the underlying notations being graphical. The loop and recursion constructs obscure model structures that are generated in the result.
2. Even though scalable models can be constructed with less effort, they cannot be easily understood, modified or maintained.
3. They do not support verification of structural properties, which is important because design flaws in the descriptions can cause problems that are very hard to detect.

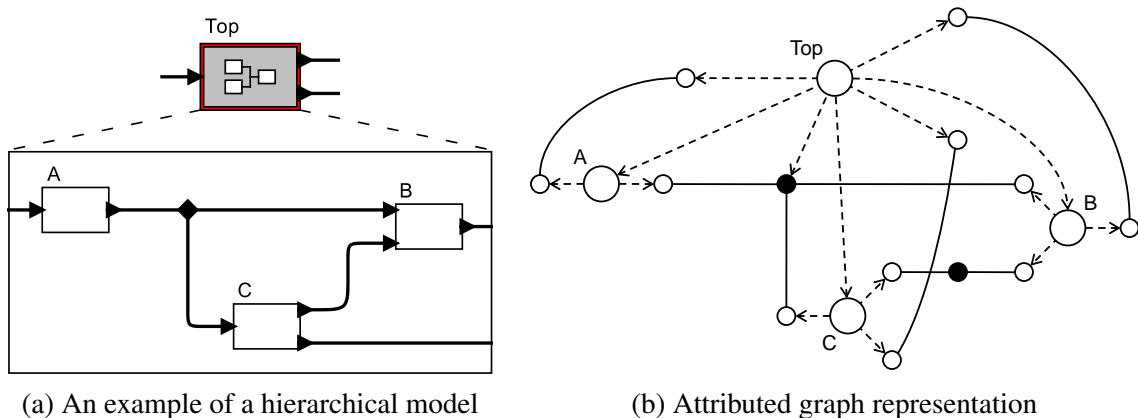


Figure 3: Hierarchical model and its representation in attributed graph

1.3 Model-Transformation-Based Model Construction and Configuration

Our scalable model construction and configuration mechanism based on model transformation aims to remedy the above problems. Using our tool, model designers can create transformation rules to match structures in their models and to transform those structures as first-class objects. A higher-order model can be created to systematically apply sequences of transformations.

In this report, we will continue to use the word-counting example to demonstrate our idea. The example can be modified to construct large-scale distributed embedded systems, such as distributed search and indexing applications on mobile phones. It can also be adapted to modeling parallel applications that utilize multicore processors [31]. In Section 2, we define basic transformations as the unit of model transformations. Compositions of basic transformations using event relationship graphs (ERGs), which we call model-based transformations, are discussed in Section 3. We demonstrate our model transformation technique with the distributed word-counting example in Section 4. In Section 5, we assess the merits of our work and compare it with other work in related fields. Section 6 offers a conclusion.

2 Model Transformation Based on Graph Transformation

Our model transformation tool is implemented with graph transformation algorithms. We consider each actor model as an attributed graph, in which nodes and edges may be associated with attributes.

Fig. 3 shows how a hierarchical actor model can be represented with an attributed graph. In Fig. 3(b), vertices represent actors, ports, and relations in the model. The styles of the vertices denote their types encoded with attributes, as will be discussed later. Actors are represented with big circles, ports represented with small hollow circles, and relations with filled dots. There are

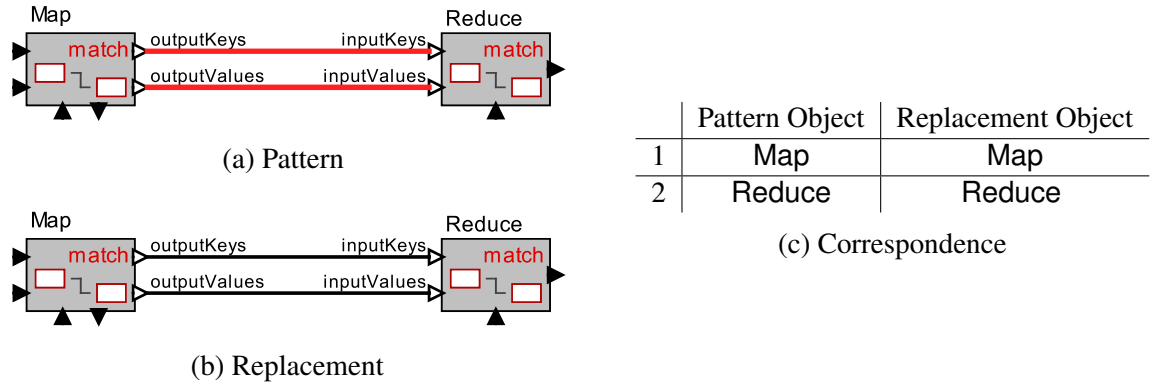


Figure 4: A transformation rule for connecting a Map and a Reduce

two types of edges. Dashed lines represent containment relationship, where the end vertices are semantically contained by the start vertices. Solid lines represent connections between ports. To represent a bidirectional connection between ports, we use two reversed directed edges. (Whether a port is an input port or an output port is an attribute of that port, which we do not show in the picture.)

This alternative representation of models allows us to directly apply graph transformation techniques [23] to modify model structures.

2.1 Visual Representation of Transformation Rules

We use a visual syntax to specify transformations. This syntax is inspired by triple graph grammar [42]. A transformation is defined by a *transformation rule*, which is similar to a rewrite rule in a context-free grammar. It can be used to match a subgraph in a given graph, and to replace that subgraph with a replacement graph. We specify a transformation rule with three components: a *pattern*, a *replacement*, and the *correspondence* between objects in those two.

Fig. 4 shows a transformation rule designed for our MapReduce example, which we will further discuss in Section 4. This rule creates connections between the output ports of a Map actor and the input ports of a Reduce actor. Repeatedly applying the rule results in having all the Map actors and Reduce actors connected in the same way. In the pattern, two *matchers* (Map and Reduce) are the placeholders used to match two distinct actors in a given model. The names of the matchers are insignificant and need not be the same as those of the matched actors. The two matchers are configured such that the input and output ports of Map and Reduce have the same names and types as those of a Map actor and a Reduce actor in the model, respectively. The two connections in the pattern are colored red in the graphical user interface, requiring that no such connections exist between those ports. The same effect can be achieved by removing those negative connections and adding the following constraint to the pattern, which is written in the Ptolemy expression language:

```
!Map.outputKeys.connectedPortList().contains(Reduce.inputKeys) &&
!Map.outputValues.connectedPortList().contains(Reduce.inputValues)
```

In this constraint, `outputKeys`, `outputValues`, `inputKeys` and `inputValues` refer to the ports belonging to the actors in the model that are matched by the Map and Reduce matchers. The `connectedPortList()` function returns a lists of all ports connected to the Map's output ports. We check whether the Reduce's input ports are in the returned lists.

In the replacement, the two matchers are preserved, meaning that the matched actors should be kept after transformation. Two connections (along with the hidden relations on them) are to be created, because they are required not to exist in the pattern. In general, designers of transformation rules can specify adding or deleting objects by editing the replacement as they wish.

Note that the names of the matchers in the replacement need not be the same as those in the pattern, because the third component, the correspondence table, establishes the relations between the two graphs. In this case, the correspondence table states that the Map object in the pattern corresponds to the Map object in the replacement, and the Reduce object in the pattern corresponds to the Reduce object in the replacement. For brevity, we do not show correspondence relations between other types of vertices such as ports and relations.

2.2 Formal Definition of Graph Transformation

Our graph transformation is defined as a modified version of the double-pushout approach introduced by Ehrig et al. [13] and reviewed by Habel et al. [18].

A graph G is a tuple $\langle V_G, E_G \rangle$, where V_G is the set of vertices in G and $E_G \subseteq V_G \times V_G$ is the set of edges. Graph G is a *subgraph* of graph H if $V_G \subseteq V_H$ and $E_G \subseteq E_H$.

A *graph morphism*, or simply *morphism*, from graph G to H is a total function $m : V_G \rightarrow V_H$, such that for any $v_1, v_2 \in V_G$, if $(v_1, v_2) \in E_G$, then $(m(v_1), m(v_2)) \in E_H$. We denote this morphism with $G \xrightarrow{m} H$. For any vertex $v \in V_G$, we say v *matches* v' in m if $m(v) = v'$. For any edge $(v_1, v_2) \in E_G$, we say (v_1, v_2) *matches* (v'_1, v'_2) in m if $m(v_1) = v'_1$ and $m(v_2) = v'_2$. If m is an injective function, then we say G is *isomorphic to a subgraph of H* , or G *matches a subgraph of H* .

The *composition* of $G \xrightarrow{m} H$ with $H \xrightarrow{n} I$ is $G \xrightarrow{n \circ m} I$, where $n \circ m : V_G \rightarrow V_I$ is the composition of function $m : V_G \rightarrow V_H$ and $n : V_H \rightarrow V_I$.

Given graphs G, H, I , and morphisms $G \xrightarrow{m} H$ and $G \xrightarrow{n} I$ as depicted in Fig. 5, a *pushout* is a triple $\langle J, I \xrightarrow{m'} J, H \xrightarrow{n'} J \rangle$, in which J is a graph and morphisms $I \xrightarrow{m'} J$ and $H \xrightarrow{n'} J$ satisfy the following conditions:

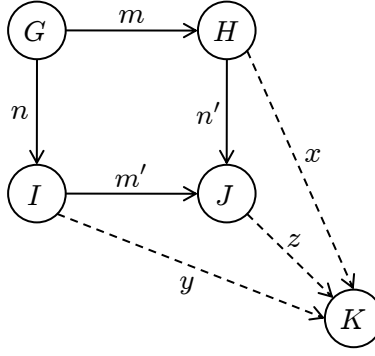


Figure 5: Pushout of graph morphisms

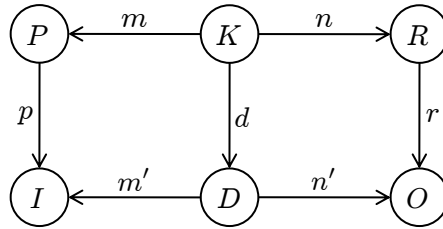


Figure 6: Graph transformation based on double-pushout

1. $n' \circ m = m' \circ n$, and
2. For any graph K with morphisms $H \xrightarrow{x} K$ and $I \xrightarrow{y} K$ satisfying $x \circ m = y \circ n$, there exists a unique $J \xrightarrow{z} K$ satisfying $z \circ n' = x$ and $z \circ m' = y$.

A *transformation rule* T , denoted by $\langle P \xleftarrow{m} K \xrightarrow{n} R \rangle$, consists of graphs P , K and R , and injective morphisms $K \xrightarrow{m} P$ and $K \xrightarrow{n} R$. P is called the *pattern graph* (or left-hand side). R is the *replacement graph* (or right-hand side). K is the *correspondence graph* (or glue graph) that relates the vertices and edges in the pattern and those in the replacement.

Given transformation rule $T = \langle P \xleftarrow{m} K \xrightarrow{n} R \rangle$ and input graph I , if an injective morphism $P \xrightarrow{p} I$ exists (i.e., P matches a subgraph of I), then T is *applicable to* I . If applicable, the result of applying T to I , as depicted in Fig. 6, is an output graph O , such that there exist graph D , injective morphisms $K \xrightarrow{d} D$ and $R \xrightarrow{r} O$, and morphisms $D \xrightarrow{m'} I$ and $D \xrightarrow{n'} O$, such that $\langle I, P \xrightarrow{p} I, D \xrightarrow{m'} I \rangle$ and $\langle O, R \xrightarrow{r} O, D \xrightarrow{n'} O \rangle$ are both pushouts.

2.3 Attributes

In order 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 Fig. 3.) It is also necessary to take into account other attributes that further differentiate vertices, such as the ones that decide whether a port is input port or output port. Therefore, we let A be a globally defined set of attributes, and extend the definition of graph G to be $\langle V_G, E_G, A_G \rangle$, where $A_G : (V_G \cup E_G) \rightarrow 2^A$ is a total function that returns a (possibly empty) set of attributes for each vertex and edge.

Our other definitions in the previous subsection remain unchanged, except that the definition of graph morphism is enhanced next to take into consideration the attributes.

2.4 Criteria Attributes and Operation Attributes

We define a subset of attributes $U \subseteq A$ to be *unchecked attributes*. It contains attributes that need not be directly checked in the extended graph morphisms to be defined below. A subset of unchecked attributes, $C \subseteq U$, is called *criteria*. (Two examples of criteria have been given in Section 2.1.)

Let B be an auxiliary set that equals $2^A \times 2^A$. We require any criterion $c \in C$ to be an element in 2^B . Given two vertices or edges $x \in (V_G \cup E_G)$ and $y \in (V_H \cup E_H)$, we say that criterion c is *satisfied by x matching y* if $(A_G(x), A_H(y)) \in c$.

We now extend the definition of graph morphism discussed in Sec. 2.2 to become the following. An *attributed graph morphism* from graph G to H is a graph morphism $m : V_G \rightarrow V_H$ satisfying the following additional conditions:

1. for any $v \in V_G$,
 - (a) $\forall a \in (A_G(v) \setminus U). a \in A_H(m(v))$
 - (b) $\forall c \in (A_G(v) \cap C). (A_G(v), A_H(m(v))) \in c$
2. for any $(v_1, v_2) \in E_G$,
 - (a) $\forall a \in (A_G((v_1, v_2)) \setminus U). a \in A_H((m(v_1), m(v_2)))$
 - (b) $\forall c \in (A_G((v_1, v_2)) \cap C). (A_G((v_1, v_2)), A_H((m(v_1), m(v_2)))) \in c$

Case (a) of the two conditions requires that all attributes belonging to a vertex or edge in G , except the unchecked ones, be associated with the matched vertex or edge in H . Therefore, the checked attributes of the latter form a superset of those of the former. Case (b) of the two conditions ensures that the criteria associated any vertex or edge in G be satisfied by the matching.

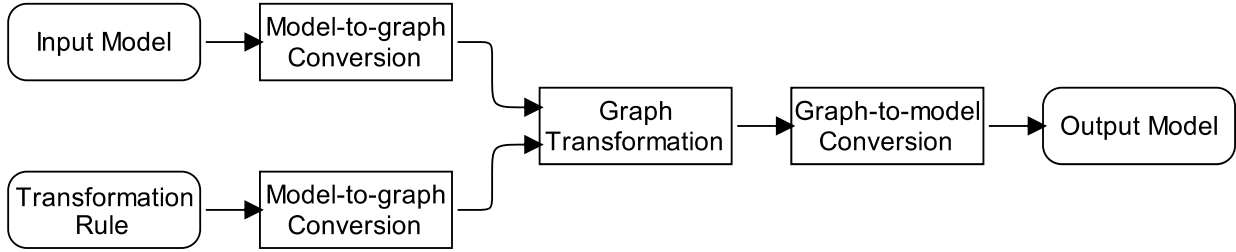


Figure 7: The model transformation process

Practically, for a transformation depicted in Fig. 6, only the graphs P and R contain vertices and edges with criteria attributes. In particular, we call the criteria in the replacement graph R *operations*, since they essentially enforce restrictions on the output graph that may be satisfied by performing additional attribute adding or removal operations. (In this discussion, we assume that those criteria in R can be satisfied by adding or removing attributes in the output graph O .)

Notice that because of the criteria in P and R , it may not be possible to apply transformation rule T to input graph I even if it is applicable in the sense that the pattern P matches a subgraph of I .

2.5 Model Transformation

The example in Fig. 3 shows a way to represent a model with an attributed graph. In the attributed graph, three special attributes are assigned to the vertices to distinguish their types: `Actor` (visually represented by big 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). The names of the vertices are unchecked attributes that are unique at each level of the hierarchy of a transformation rule. Names at different levels may be identical.

Using the graph representation, we establish a model transformation process as shown in Fig. 7. The inputs to the process consist of an input model and a transformation rule, both specified in the modeling language. (Fig. 1, Fig. 2 and Fig. 4 provide examples of both in this language.) The two inputs are then converted into attributed graphs. To convert a model into an attributed graph, a vertex is created for each actor, port or relation, and an edge is created for each connection or containment relation. (We use two reversed edges with the same attributes to simulate an undirected edge in Fig. 3.)

The transformation rule is converted into multiple graphs. Its pattern and replacement contain model fragments and are converted into the P graph and the R graph in Fig. 6. The correspondence table simplifies the specification of the K graph. Its “Pattern” column shows the names of the actors in the pattern. (For clearer presentation, we hide the vertices corresponding to ports and relations as well as all edges.) 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 actors in the replacement. There is a one-to-one relationship between entries in both columns. Conceptually the conversion process computes a subgraph of P as K , such that K contains only vertices listed in the “Pattern” column. The one-to-one nature ensures that a subgraph exists in R that is isomorphic to this selection of K .

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

3 Model-Based Transformation with Hierarchical Event Relationship Graphs

We call the transformation specified with a single transformation rule *basic transformation*. Even though applying a basic transformation on an input model is straightforward, as depicted in Fig. 7, its expressiveness and usability is limited. One reason is that, since graph transformation is context-free, all information required to perform a basic transformation must be captured in the pattern, and all changes must be specified in the replacement. This makes it hard to perform sparse modification on a large model structure. The complexity of pattern matching problem also limits the size of any basic transformation in practice.

Model-based transformation is our solution for affordable and expressive model transformation. By using a higher-order model to apply a sequence of basic transformations to an input model, the complexity of each basic transformation can be greatly reduced, and the transformation designers acquire more control over the transformation process.

3.1 Event Relationship Graphs

We use hierarchical event relationship graphs (ERGs) as a control flow model of computation for model-based transformation. This model of computation is based on Schruben’s event graphs [40]. An ERG model is represented with a multigraph, in which nodes represent events and directed edges between nodes represent scheduling relations between the corresponding events. (In a multigraph, multiple edges can exist from one node to another.)

An example ERG model is shown in Fig. 8, which contains two events with names E1 and E2. Implicitly, there is an event queue for each ERG model, which is a priority queue that maintains the events scheduled to be processed in the future. An *initial event*, which is colored green with a thick border as E1, is automatically scheduled at the start of execution (at model time 0). An edge from E1 to E2 is a *scheduling relation*, which can be associated with a boolean expression as its *guard*, and an expression that evaluates to a number as its *delay* (denoted by δ). When the

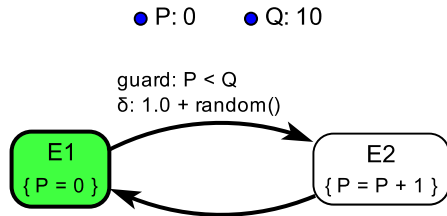


Figure 8: A simple event relationship graph

guard is set to be always true, it is usually omitted from the visual representation. Similarly, the delay is omitted if it is constant 0. The scheduling relation from E1 to E2 requires that when E1 is processed and the guard expression returns true, E2 is scheduled to occur δ units of model time later. The same event can be scheduled multiple times, with more than one instance of it in the event queue.

An ERG model can contain variables. In Fig. 8, variables P and Q are defined with initial values 0 and 10, respectively. As a side effect of processing an event, actions can be performed to update the values of those variables. In this example, E1 and E2 each has one action that sets P's value.

An ERG model can have one or more initial events. It can also have one or more *final events*, shown with a red background with double border. When any of the final events is processed, an implicit side effect (after its actions are executed) is to remove all the events remaining in the event queue, so that there is no more event to process, and the execution terminates. Note that, an ERG model is not required to have a final event for its execution to terminate. Its execution terminates whenever its event queue becomes empty after processing an event.

3.2 Hierarchy

Hierarchy and information hiding effectively help reduce complexity in the design and make it more understandable. Two approaches to add hierarchy to event graphs are discussed in [41]. One is to use submodels to compute the delays on the scheduling relations. In that case, the only meaningful output of a submodel is a number that is used as the δ on the scheduling relation that the submodel is associated with. Another approach is to associate a submodel with an event instead of a scheduling relation, so that processing that event causes the unique start event in the submodel to be scheduled. That start event may schedule other events in the submodel. When a unique and predefined end event is processed, the execution of the submodel is finished, and the event that the submodel is associated with is considered processed. The scheduling relations from that event are evaluated and for those whose guards return true, new events are scheduled.

We have invented a third type of hierarchical event graph models for our ERG model of com-

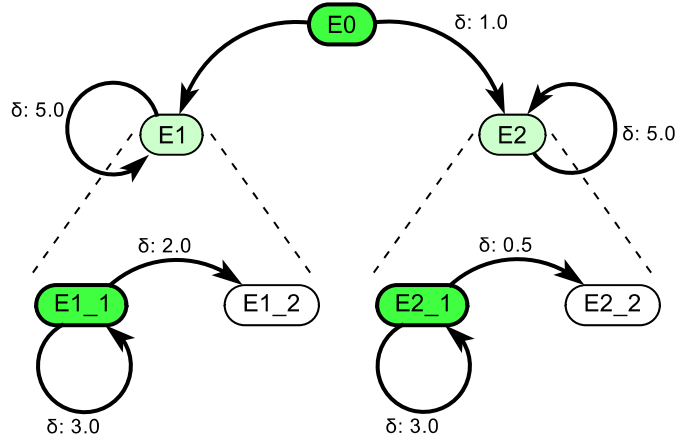


Figure 9: An example hierarchical ERG model

putation. As a design decision, we would like to define submodels as refinements of events, and to use events to specify tasks and subtasks in a hierarchical model-based transformation. We would also like to define the processing of an event as a complete execution of its refinement, which is itself an ERG model. This means the refinement has no distinguished start event and end event as are found in the second approach above. Its execution finishes as soon as its event queue becomes empty.

In our approach, refinements of an ERG model have their own event queues. All event queues share the same notion of time. A formal discussion about the operational semantics of a hierarchical ERG model is out of the scope of this project. It suffices to explain its operation informally with the example in Fig. 9. In that example, events E0, E1 and E2 are at the top level of the hierarchy, among which E0 is an initial event. E1 and E2 have refinements. Fig. 10 shows the events in the event queues for this model over time. At the beginning, E0 is scheduled. It then schedules E1 at time 0 and E2 at time 1. Since E1 is scheduled to occur earlier, the next step is to process E1, which schedules the initial event in its refinement, E1_1, at time 0 and itself again at time 5. Because the refinement of E1 has its own queue, E1_1 is actually scheduled in that queue, and an entry in the top-level queue is allocated to point to that queue. At the next step, because the refinement's event queue is to be processed next, its first event E1_1 is processed. That event further schedules E1_2 and itself in the same refinement's event queue, which is now moved to the second entry of the top-level event queue because the E2 event occurs earlier. The termination condition at each level of the hierarchy is defined as the event queue for that level being empty.

The above example shows how an ERG model interleaves its own execution with that of its refinements. It executes one iteration of the refinement when the first entry of its event queue points to that refinement's event queue. The result is one event in the refinement being processed. In this case, each refinement is itself an ERG, which could have refinements as well. In general, a refinement could also be any kind of model that is executed in iterations and reports time advance to its container between iterations. (The Ptolemy II framework provides the `fireAt()` method for a director to implement this mechanism in a model of computation.) This allows designers to

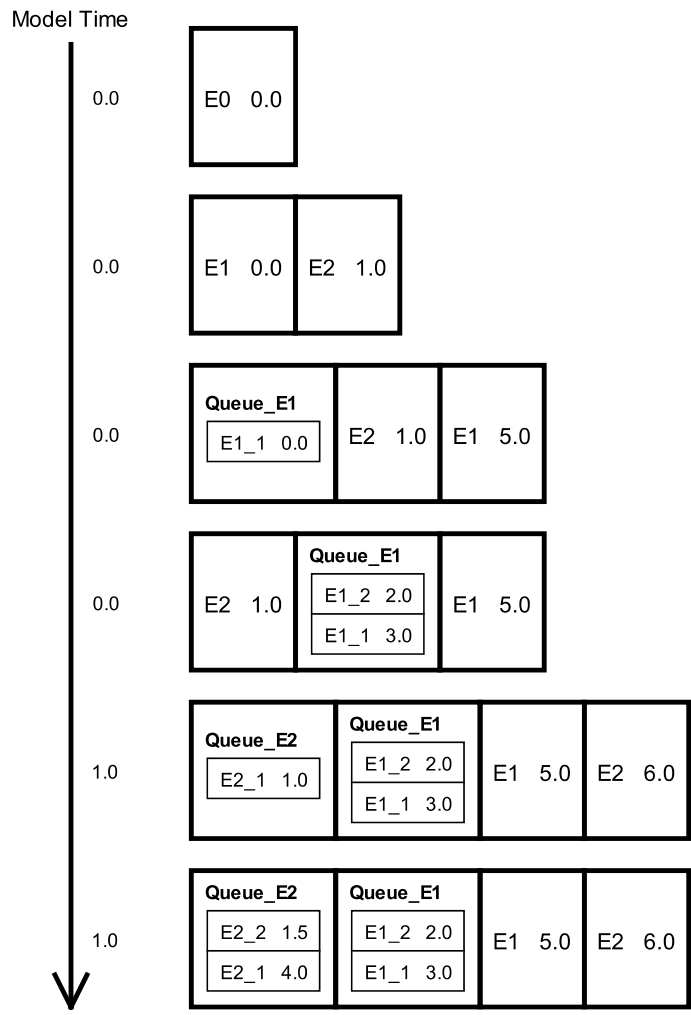


Figure 10: Run-time images of event queues



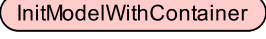





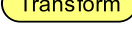

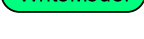
Event	Function
	Execute the model in the Model variable
	Initialize the Model variable with an empty model
	Initialize the Model variable with the container of this ERG model
	List the names of all files in a given directory
	Match the pattern defined in this event with the model in the Model variable
	Read the model stored in the file with a given name into the Model variable
	Report a message or an error to the user
	Perform pure testing (on the outgoing edges) with no side effect
	Transform the model in the Model variable with a basic transformation
	Show the model in the Model variable in a separate window
	Write the model in the Model variable into the file with a given name

Figure 11: Predefined events for model-based transformation

hierarchically compose heterogeneous models of computation, which leads to flexible systems design [14].

3.3 Event Library

In the previous section, we have discussed actions of events as a means to update variables' values. In fact, arbitrary actions can be defined by creating customized events.

We have created a library of events for model-based transformation. Some of those events perform transformation or pattern matching as their actions on the model stored in a special Model variable. That variable contains the model that is transformed by all the transformations within the ERG model and its refinements. We design special icons for those events to make them more recognizable. Fig. 11 shows a legend for those events and explains their functions. Among them, the Match event contains the specification of a model pattern, such as the one in Fig.4(a). The designer can edit this pattern in a separate window, when he or she chooses the "Look Inside" menu item in the event's popup menu. The Transform event contains a basic transformation that can be edited in a separate window. Besides the Model variable, a Match or Transform event also updates its local variable with name "matched." That variable stores a boolean value denoting whether the recent pattern matching or transformation was successful. Scheduling relations may read this variable's value in their guards to make a decision accordingly.

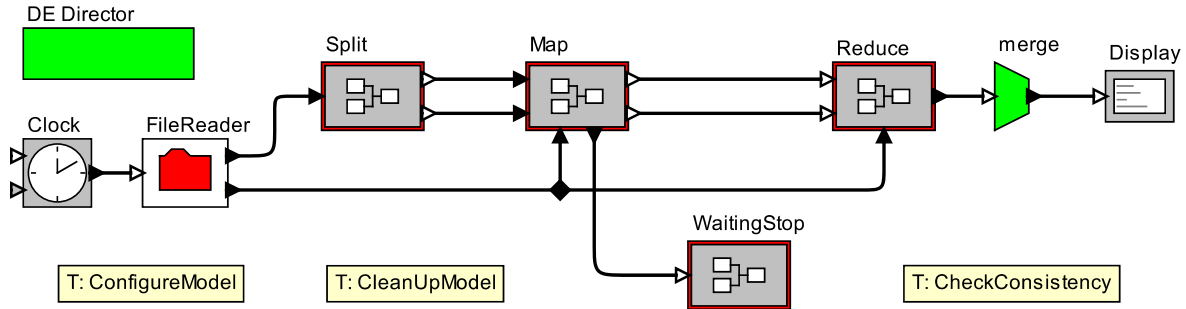


Figure 12: The configurable word-counting model

3.4 Initialization

There are several ways to initialize the Model variable. Once initialized with a model, the Match and Transform events in the ERG operate on that model. In the library, the InitModel event is specifically designed to initialize the Model variable with an empty model. The InitModelWithContainer event initializes it with the model that contains the current ERG model. The ReadModel event initializes it with a model that is read from a file.

When the Model variable is initialized to contain an empty model, subsequent transformation performed on it can add contents to it. This makes it possible to generate a model as the result of running the ERG model. The generated model can be executed dynamically with the Execute event, or be saved to a file with the WriteModel event, or simply be viewed in a separate window with the View event so that the user can manually edit it.

The InitModelWithContainer event is particularly interesting. It initializes the Model variable with the model that contains the ERG model acting as model-based transformation. If the ERG model is encapsulated in a *transformation attribute*, which is a special attribute that represents a transformation, and the transformation attribute is assigned to an actor model, then the InitModelWithContainer event initializes the Model variable with that actor model. When the transformation attribute is invoked from the user interface, the ERG model is executed, causing the actor model that contains it to be transformed. Viewed from the user's perspective, the transformation attribute in the actor model modifies the model's structure. This feature can be used to configure the actor model with parameters, as will be shown next.

4 The Distributed Word-Counting Model as an Example

We use the model in Fig. 12 to demonstrate our idea of large-scale structurally configurable models. The model shown in the figure is a template for constructing large models that contain multiple Map actors and Reduce actors. It is the same as the model in Fig. 1, except that it acquires three

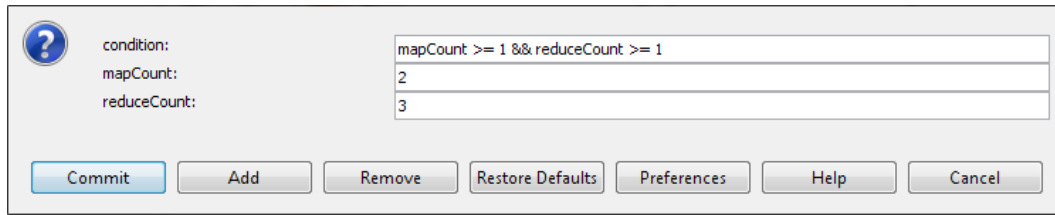


Figure 13: Parameters of the ConfigureModel attribute

transformation attributes (distinguished from other attributes with the starting “T:” in their names).

When the transformation attributes are opened in the Ptolemy graphical user interface, the model-based transformations that they contain can be edited and executed. In our example, the ConfigureModel attribute configures the template to generate a model in the same editor that has a given number of Map actors and Reduce actors connected correctly. The CleanUpModel attribute removes the generated actors and connections to restore the template in the editor. The CheckConsistency attribute validates certain structural properties, which any consistent configuration of the template must acquire.

Each transformation attribute can be associated with parameters, either predefined ones or customized ones. For example, ConfigureModel has two customized parameters mapCount and reduceCount to decide the numbers of Map and Reduce actors to be created, as shown in Fig. 13. The predefined parameter condition for every transformation attribute declares a boolean condition under which the transformation is applicable. In the case of ConfigureModel, the applicability condition is set to be “mapCount >= 1 && reduceCount >= 1”. This makes sure that the mapCount and reduceCount parameters have acceptable values. In general, applicability conditions help preclude certain misuse of transformation attributes.

4.1 Structural Configuration

The ConfigureModel transformation attribute contains a hierarchical model-based transformation, as shown in Fig. 14. It is an instance of an actor-oriented class, which obtains the InitModel-WithContainer event and the Model variable from the class definition. This is denoted by the pink boxes around the icons. The InitModelWithContainer event initializes Model with the model in Fig. 12 that contains the ConfigureModel attribute. We add the other events to the design to perform the transformation on that model. Note that this ERG model itself within the ConfigureModel attribute is not subject to transformation, because we do not intend to make our transformations self-modifying.

The events at the top level of the ERG model in Fig. 14 perform the following tasks:

1. AddMaps adds an appropriate number of Map actors to the template. It contains an ERG

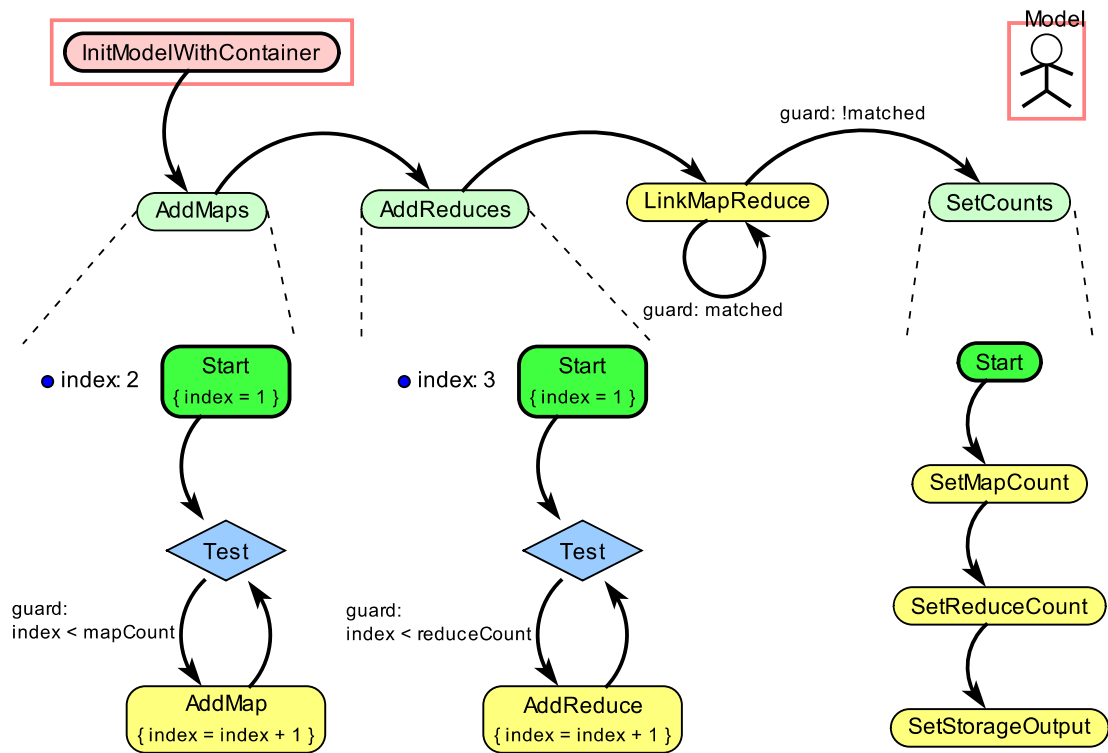
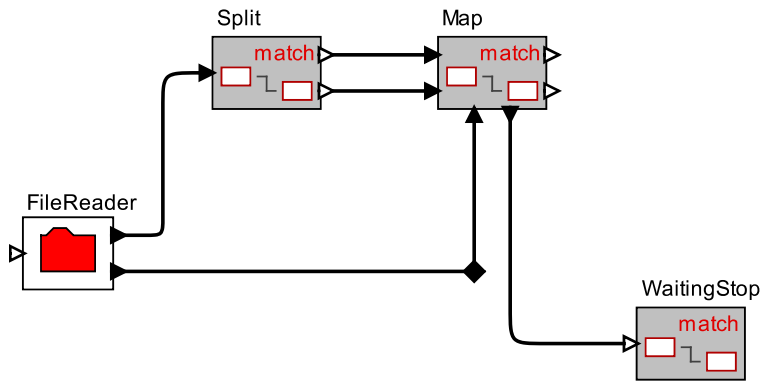


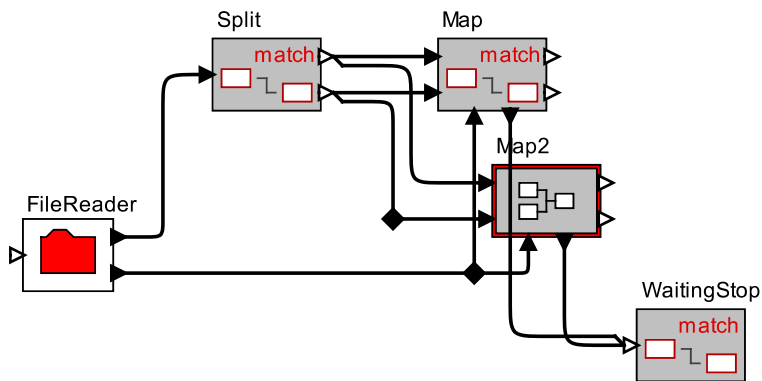
Figure 14: The model-based transformation in `ConfigureModel`



(a) Pattern

	Pattern	Replacement
1	FileReader	FileReader
2	WaitingStop	WaitingStop
3	Split	Split
4	Map	Map

(c) Correspondence



(b) Replacement

Figure 15: The basic transformation rule in AddMap

model as a refinement. In the refinement, a variable called “index” is defined, which counts the number of Map actors created so far. The Start event sets index to 1, because in the template, there is originally one Map actor. Then the Test event is scheduled to occur next (at the same model time). If index has not reached mapCount, Test schedules the basic transformation called AddMap to add one Map actor. The pattern, replacement, and the correspondence table of the AddMap transformation is shown in Fig. 15.

2. When the refinement of AddMaps finishes executing (i.e., no event is left in its local event queue), AddReduces is scheduled. It has a similar refinement to add Reduce actors.
3. When AddReduces is finished, the next task is to connect the created Map actors and Reduce actors. This is done by repeatedly performing the LinkMapReduce transformation, until no more connections can be created. We have already seen the basic transformation in LinkMapReduce in Fig. 4. As discussed before, the pattern of this basic transformation requires that no connection be established between the matched Map actor and Reduce actor. It creates the connections as a result. Repeated application of this transformation is guaranteed to stop when all Map actors are connected to all Reduce actors. When that happens, the “matched” variable of the LinkMapReduce event is set to false.
4. Finally, SetCount has a refinement to set the parameters within the composite actors, so that they obtain the correct values. The basic transformations in this refinement is less interesting because they only change parameter values but not the model structure.

By executing the model-based transformation in the ConfigureModel attribute with proper values for the mapCount and reduceCount parameters, user of the word-counting model can easily obtain a model of arbitrary size. This is convenient because the transformation automatically expands the model in the current editor, which is ready to execute. If the transformation is correct by construction, then the obtained model is correct regardless of how big it is.

One may ask, after configuring the template with a large number of Map and Reduce actors, how the designer restores the template in the editor, which allows future modification and update. For this purpose, we create a second transformation attribute called CleanUp. The result of applying the model-based transformation in it is to remove all the automatically created actors. Those actors are tagged with a special attribute that distinguishes them from the actors originally in the template. The transformation in CleanUp searches for all such actors and removes them. The connections are automatically removed as well when the actors at both ends no longer exist.

4.2 Checking Structural Consistency

It is difficult to ensure correct model behavior. Formal model checking methods are limited when faced with large or infinite state spaces. Abstraction is usually required to make model checking problems feasible [10]. Existing abstraction methods include abstract interpretation [8] and predicate abstraction [16].

Our aim in this project is not to ensure correct model behavior in all aspects. Instead, we try to find a systematic way to reliably construct and configure large models. We make an observation that erroneous behavior can arise from three main sources:

1. a design flaw in the template,
2. a design flaw in the transformation used to configure the template, and
3. misuse of a correctly designed transformation.

We do not focus on the first and third sources here. The reason is that it is much easier to ensure correctness in the template since its size is relatively small compared to the actual model generated from it. Also, the applicability conditions of transformation attributes help to eliminate many misuse cases.

For the second source of errors, we employ pattern matching as a mechanism to check the correctness of transformation results. We are forced to avoid performing too detailed checks on the model structures, because those checks potentially suffer from the same design flaws in the transformations themselves. For example, if we decided to check whether exactly the required number of Map actors and Reduce actors were created and connected, then we would be essentially reinventing the configuration transformation. This would not help detect or understand design flaws. Therefore, we decide to check structural properties in the constructed model from a different angle.

In our word-counting example, `CheckConsistency` is a transformation attribute that checks consistency of the current model without modifying it. If a structural error is detected, an error message is shown to notify the user. Internally, it contains an ERG modal as shown in Fig. 16. (Here `ReportError` takes a string parameter with name “text,” and each scheduling relation pointing to that event provides a value to the parameter as the only element in an array, which is the string to be shown in a message dialog.) We define three Match events to match the particular patterns that we do not expect to occur in the model. They are listed in Fig. 17. Among them, `FindUnconnectedPort` contains a pattern that matches any actor at the top level of the model with an unconnected port (input or output), except for the Clock actor which does not require any input to its input ports. `FindUnconnectedKeysPort` and `FindUnconnectedValuesPort` matches any output port of a Map actor that is not properly connected to the corresponding input port of a Reduce actor.

Notice that this check departs significantly from the transformation in Fig. 14. This helps detect unexpected errors. For example, if a designer creates a more sophisticated sink to replace the Display in the model, but forgets to connect it to the Merge’s output, the check can detect the problem and report it to the designer. If, however, the lack of connection is desired, then the report informs the designer of the previous assumption that all ports are connected, and urges the designer to review that assumption. This is an engineering practice that we believe to be beneficial. To add more checks in the future, the `CheckConsistency` attribute can be extended, or more transformation attributes can be added to the model.

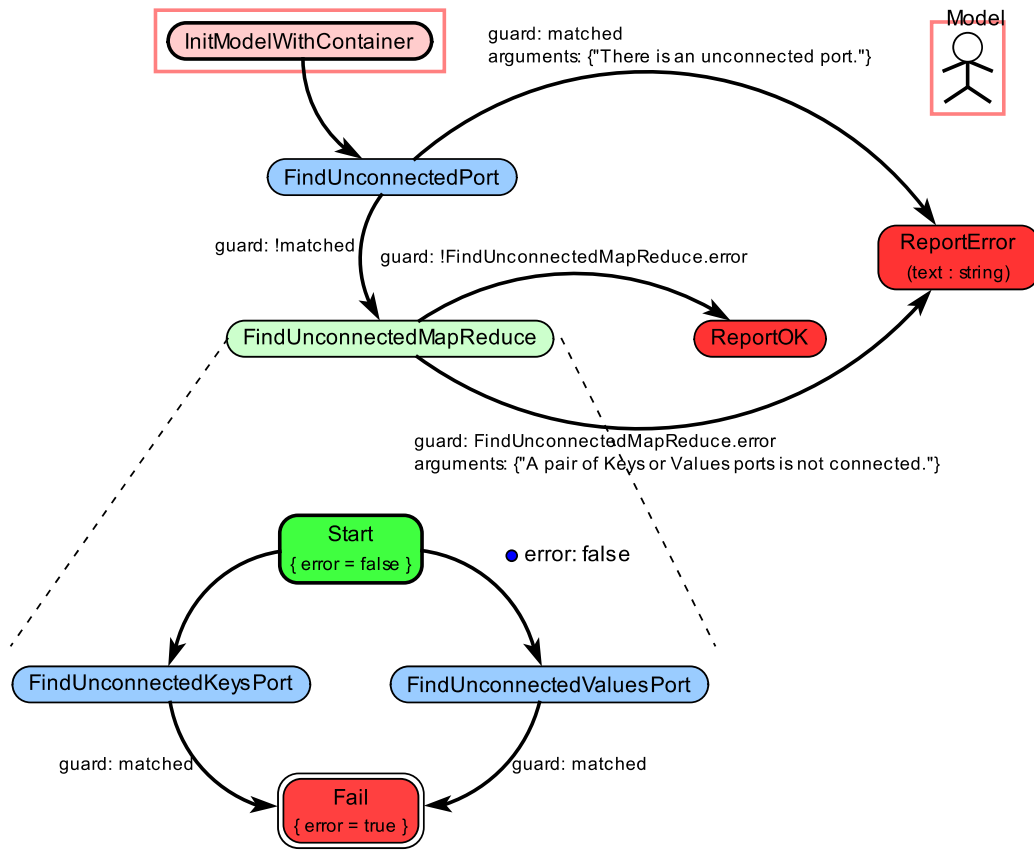


Figure 16: The model-based transformation in CheckConsistency

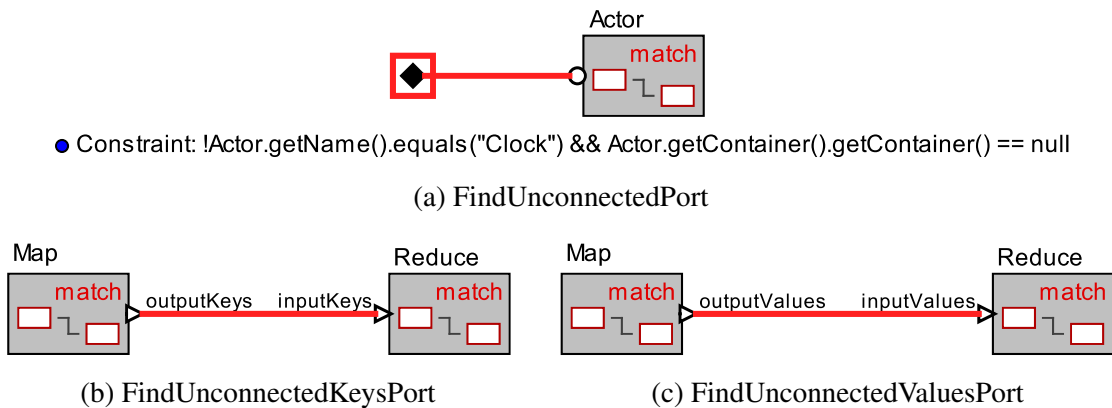


Figure 17: Patterns designed for the events in CheckConsistency

Attributes that contain ERG models can also be used to check or analyze other properties that are not purely structural. For a concrete example, consider the data polymorphic actors and behaviorally polymorphic actors in Ptolemy II. The former type of actors can operate on different data types, and their actual behavior depends on what types the given data have. The latter type of actors vary their behavior depending on the models of computation [26, 29]. Though they are designed to be polymorphic, their actual behavior can be statically analyzed once the data types or models of computation are known statically. This analysis can be done with an ERG model contained in an attribute.

5 Assessment and Related Work

We have implemented a model transformation tool for actor models in the Ptolemy II framework. Among the numerous potential applications that our model transformation tool has, in this project we focus on its application to systematic construction and configuration of large actor models. We compare this work with the related work in various fields.

5.1 Model Transformation

Model transformation has been under active research in recent years. In recognition of the public interest, the OMG has issued a request for proposal (RFP) on MOF (Meta-Object Facility) QVT (Query / Views / Transformations) to seek a standardized approach to model transformation [34].

Besides our tool, existing model transformation tools include AGG [46], AToM³ [24], FUJABA [33], GReAT [1], PROGRES [43] and VIATRA2 [3]. All those tools base their theories on graph grammars. A model, whether it is represented in a visual form or in a textual form, is considered as an attributed graph. A basic transformation operates on a context-free subgraph of the graph each time. The first step is to locate the subgraph with pattern matching, which is essentially to solve the subgraph isomorphic problem, whose complexity has been proved to be *NP*-hard. For our tool, the pattern matching algorithm is a variant of Ullmann's classic algorithm [47], with some performance improvement and extensions for better usability.

5.1.1 Basic Transformations

We provide extensive support for basic transformations. A useful feature is negative patterns (also called negative application conditions), which contain negative objects that must not be found in the model. Examples of such objects are the relation in Fig. 17(a) and the connections in Fig. 17(b) and Fig. 17(c), which are colored red. This feature is also supported by other tools, such as AGG, AToM³, FUJABA and VIATRA2. GReAT represents negative objects with cardinality 0.

To specify additional applicability and integrity constraints in the pattern, we leverage the Ptolemy expression language, which has a syntax similar to MATLAB expressions. We allow model designers to invoke arbitrary Java functions, including those provided by the Java standard library and the user-defined ones. We assume that all constraints are free of side effect. A similar expression language is found in AGG, which is based on Java expressions. Some model transformation tools provide their own constraint languages. For example, PROGRES uses queries for transaction preconditions [32], and VIATRA2 allows preconditions to be written in its textual pattern descriptions. Other tools, such as AToM³, GReAT and FUJABA, support object constraint language (OCL), or tool-specific dialects of OCL.

A unique capability that our basic transformations provide, which is not found in other tools, is that we incorporate the Ptalon higher-order composition language into transformation rules. A textual Ptalon description can be used to create a parametrized structure in the pattern to match variable model structures. VIATRA2 also provides with a higher-order composition language for creating parametrized patterns, but it does not offer a visual interface at the same time, whereas our tool does.

5.1.2 Model-Based Transformations

Much of the expressiveness of model transformations is in fact due to composition of basic transformations. This is because there is a high complexity for applying basic transformations with large patterns. Additionally, given certain control mechanisms, transformations in a composition can be conditional and iterative.

AGG organizes basic transformations on layers. Transformations on a higher layer are repeatedly applied until no more transformation is possible before the tool moves to the next layer. Similarly, AToM³ employs explicit priority numbers to separate groups of transformations. Its recent improvement adds model-based transformation support with the DEVS (discrete event system specification) model of computation [45]. PROGRES uses procedural programs to control transformations. FUJABA uses control flow diagrams and state machines. VIATRA2 uses abstract state machines. GReAT uses a model of computation that combines control flow and data flow, where patterns are sent in a data flow fashion but the models to be transformed are stored globally.

We formalize our idea of model-based transformation with hierarchical composition of transformations using one or more models of computation. In particular, we discuss ERG in this project, which is essentially a control flow model of computation that supports hierarchy and time. We demonstrate that hierarchical ERG models can be used to organize complex transformations. Time in the ERG models can be useful because they enable timed model transformation [17, 44]. An important application of timed model transformation is to model the dynamic evolution of a timed system. Discussion of timed model transformation is out of the scope of this project.

Other models of computation, such as DE, SDF and FSM, can also be used to control model-based transformation. We also support hierarchical combination of heterogeneous models of com-

putation, which has been proved extremely flexible and effective [15].

5.1.3 Visual Representations

Our model transformation tool has been created for the actor-oriented modeling language provided by Ptolemy II. Compared to the model transformation tools based on metamodeling with UML class diagrams, such as AToM³ and GReAT, our tool provides a native visual representation of transformation rules to the designers. It allows them to include actors from the actor library in any pattern, as well as matchers that can match arbitrary actors satisfying a set of criteria. Connections between actors' ports in a transformation rule are represented in the same way as they are in models. This visual representation that model designers are familiar with eliminates the need for learning a new language, such as class diagrams. It also reduces the risk of introducing errors due to misunderstanding of that language. A more important benefit of this representation is that it supports information hiding in hierarchical patterns. Those patterns are designed to match hierarchies in the models. In contrast, if class diagrams are used, hierarchies in the patterns are usually represented with special associations between classes. Such associations do not hide information, and they may confuse designers because associations are also used for other purposes, such as ownership and connection relationship.

A potential limitation of our visual representation is that it is oriented to actor models and does not immediately support other modeling languages. However, we believe the idea and algorithms behind the visual representation can be easily adapted to other modeling tools using block-diagram languages, such as Simulink, LabVIEW, ForSyDe, SPEX, and ModHel'X.

5.2 Automated Model Construction

Higher-order model composition for embedded system design is proposed in [38] and [11]. The idea is to construct models by considering model fragments as first-class objects and assembling them with a higher-order description. Parameters can be defined in the description to allow its users to configure the constructed model.

Compared to other related approaches in this field, such as Ptalon [6] and higher-order Petri nets [20], our model-based transformation approach allows designers to visually describe pieces of model structures and to transform them step by step. Our model descriptions are themselves hierarchical heterogeneous models, which can be divided into parametrized components for reuse. Therefore, not only the models constructed by the descriptions can easily scale to large sizes, the descriptions themselves are also scalable.

5.3 Event Graphs

Our ERG model of computation is based on event graphs, a discrete-event model of computation created by Schruben [40]. The original event graphs are significantly more expressive than finite state machines, because in each event graph there is an implicit event queue whose size is unbounded. In fact, event graphs are Turing-complete and so are ERGs (because Petri nets with inhibitor arcs are Turing-complete [36] and it has been shown that any such Petri net can be modeled with an equivalent event graph [39]). This model of computation is timed. Model-time delays can be associated with scheduling relations, making event graphs suitable for modeling discrete-event systems. In [41], two forms of hierarchical event graphs are discussed. In [4], another attempt is reported, which uses the listener pattern to compose event graphs hierarchically. In our approach, events in an ERG model can have another ERG model as its refinement. This allows us to model tasks and subtasks hierarchically, so that the completion of a task requires all its subtasks to finish. This is especially convenient for model transformation, as well as other applications that requires sophisticated control for operations on shared data structure. In addition, our approach also allows seamless composition and interaction between event graphs and other models of computation.

6 Conclusion

There is a high demand for a systematic approach to the design and maintenance of large structurally configurable models for embedded systems. We describe an approach based on model transformation techniques. We argue that in our approach, more useful features can be incorporated into the model design interface, such as structural verification, while the design is still kept clean through the use of a template that is relatively simple. Our argument is supported by an example that simulates a distributed system using the MapReduce pattern.

References

- [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, New York, NY, USA, 2006. ACM.
- [3] 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, Oct. 2006.

- [4] Arnold H. Buss and Paul J. Sanchez. Building complex models with LEGOs (listener event graph objects). *2002 Winter Simulation Conference (WSC'02)*, 1:732–737, 2002.
- [5] Adam Cataldo. *The Power of Higher-Order Composition Languages in System Design*. PhD thesis, University of California, Berkeley, December 2006.
- [6] 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.
- [7] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. A map reduce framework for programming graphics processors. In *Third Workshop on Software Tools for MultiCore Systems (STMCS 2008)*, Boston, MA, 2008.
- [8] Patrick Causot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. *SIGPLAN Not.*, 12(3):77–94, 1977.
- [9] W. Cesário, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava. Component-based design approach for multicore SoCs. In *DAC '02: Proceedings of the 39th Design Automation Conference*, pages 789–794, New York, NY, USA, 2002. ACM.
- [10] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.
- [11] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a higher-order synchronous data-flow language. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 230–239, New York, NY, USA, 2004. ACM Press.
- [12] 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, Dec. 2004.
- [13] 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.
- [14] 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.
- [15] 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.

- [16] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 72–83, London, UK, 1997. Springer-Verlag.
- [17] Szilvia Gyapay, Reiko Heckel, and Dániel Varró. Graph transformation with time: Causality and logical clocks. In *Proc. 1st Int. Conference on Graph Transformation (ICGT 02)*, pages 120–134. Springer-Verlag, 2002.
- [18] Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Comp. Sci.*, 11(5):637–688, 2001.
- [19] Cécile Hardebolle and Frédéric Boulanger. ModHel’X: A component-oriented approach to multi-formalism modeling, Oct. 2007.
- [20] Jörn W. Janneck and Robert Esser. Higher-order Petri net modeling – techniques and applications. In *Workshop on Software Engineering and Formal Methods*, January 2002.
- [21] 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.
- [22] Paul Kinnucan and Pieter J. Mosterman. A graphical variant approach to object-oriented modeling of dynamic systems. In *Summer Computer Simulation Conference (SCSC)*, pages 513–521, San Diego, CA, 2007.
- [23] Alexander Königs. Model transformation with triple graph grammars. In *Model Transformations in Practice Workshop*, Oct. 2005.
- [24] 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, Apr. 2002.
- [25] Edward A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7(1-4):25–45, 1999.
- [26] Edward A. Lee. Model-driven development – from object-oriented design to actor-oriented design. Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation, September 2003. Extended abstract of an invited presentation.
- [27] Edward A. Lee, Xiaojun Liu, and Stephen Neuendorffer. Classes and inheritance in actor-oriented design. *ACM Transactions on Embedded Computing Systems (TECS)*, to appear, 2008.
- [28] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sep. 1987.
- [29] Edward A. Lee and Yuhong Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing*, 16(3):210–237, 2004.

- [30] 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, Nov. 2006.
- [31] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: A programming model for heterogeneous multi-core systems. In *ASPLOS XIII: International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 287–296, New York, NY, USA, 2008. ACM.
- [32] M. Munch, A. Winter, and A. Schürr. Integrity constraints in the multi-paradigm language PROGRES. In *In Theory and Application of Graph Transformations, 6th International Workshop, TAGT98*, pages 84–85. Springer-Verlag, 1998.
- [33] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, Jun. 2000.
- [34] Object Management Group (OMG). Request for proposal: MOF 2.0 Query / Views / Transformations RFP, 2002.
- [35] Object Management Group (OMG). A UML profile for MARTE, beta 1. OMG Adopted Specification ptc/07-08-04, Aug. 2007.
- [36] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.
- [37] Ram Ramanathan and Jason Redi. A brief overview of ad hoc networks: challenges and directions. *Communications Magazine, IEEE*, 40(5):20–22, May 2002.
- [38] Hideki John Reekie. *Realtime Signal Processing – Dataflow, Visual, and Functional Programming*. PhD thesis, University of Technology at Sydney, 1995.
- [39] Lee Schruben and Enver Yücesan. Transforming Petri nets into event graph models. In *Winter Simulation Conference (WSC 94)*, pages 560–565, San Diego, CA, USA, 1994. Society for Computer Simulation International.
- [40] Lee W. Schruben. Simulation modeling with event graphs. *Communications of the ACM*, 26(11):957–963, 1983.
- [41] Lee W. Schruben. Building reusable simulators using hierarchical event graphs. In *Winter Simulation Conference (WSC 95)*, pages 472–475, Los Alamitos, CA, USA, 1995. IEEE Computer Society.
- [42] Andy Schürr. Specification of graph translators with triple graph grammars. In *WG '94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer-Verlag, 1994.
- [43] 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, Sep. 1995.

- [44] Eugene Syriani and Hans Vangheluwe. Programmed graph rewriting with devs. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007)*, Kassel, Germany, Oct. 2007. Springer-Verlag.
- [45] Eugene Syriani and Hans Vangheluwe. Programmed graph rewriting with time for simulation-based design. In *International Conference on Model Transformation (ICMT)*, pages 91–106, Zurich, Switzerland, July 2008.
- [46] Gabriele Taentzer. AGG: A tool environment for algebraic graph transformation. In *Proceedings of Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, Kerkrade, The Netherlands, Sep. 1999.
- [47] Julian R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.
- [48] Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):187–207, 2007.