Scalable Models Using Model Transformation

Thomas Huining Feng
Edward A. Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-85
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-85.html

July 13, 2008

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.

Scalable Models Using Model Transformation *

Thomas Huining Feng and Edward A. Lee

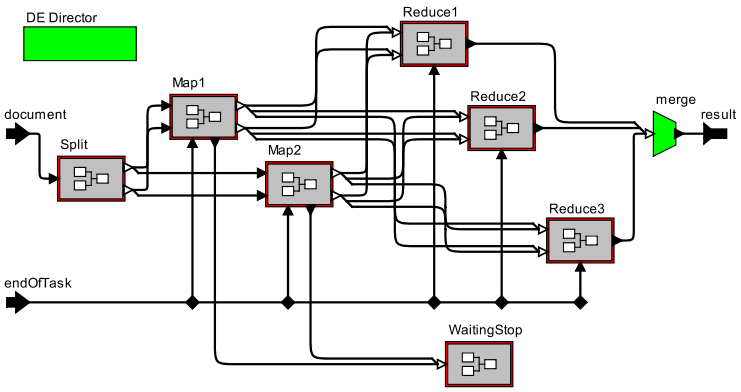
Center for Hybrid and Embedded Software Systems
EECS, University of California, Berkeley
Berkeley, CA 94720, USA
{tfeng,eal}@eecs.berkeley.edu

Abstract. Higher-order model composition can be employed as a mechanism for scalable model construction. By creating a description that manipulates model fragments as first-class objects, designers' work of model creation and maintenance can be greatly simplified. In this paper, we present our approach to higher-order model composition based on model transformation. We define basic transformation rules to operate on the graph structures of actor models. The composition of basic transformation rules with heterogeneous models of computation form complex transformation systems, which we use to construct large models. We argue that our approach is more visual than the traditional approaches using textual model descriptions. It also has the advantage of allowing to dynamically modify models and to execute them on the fly. Our arguments are supported by a concrete example of constructing a distributed model of arbitrary size.

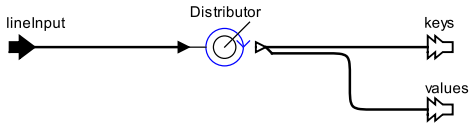
1 Introduction

We take an actor-oriented approach to the design of embedded systems. A model consists of actors as the basic building blocks, which 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. In a hierarchical design, models may contain models, in which case the contained models themselves act as actors. The execution semantics is defined by the director of the model, if it has one, or otherwise the director of the containing model. Each director implements a model of computation (MoC). Examples of MoCs include DE (Discrete Event), SDF (Synchronous Dataflow), FSM (Finite State Machine), and PN (Karn Process Network). A heterogeneous model uses various MoCs at different levels of its hierarchy. This proves extremely flexible for system design, because model designers can freely choose a convenient MoC for any part of the model. [1]

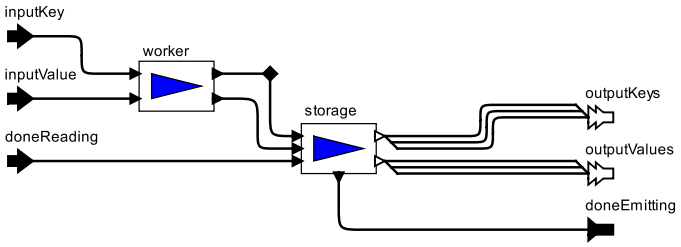
* This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab



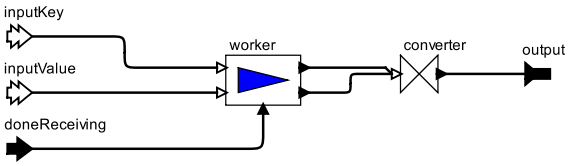
(a) Top level of the MapReduce model



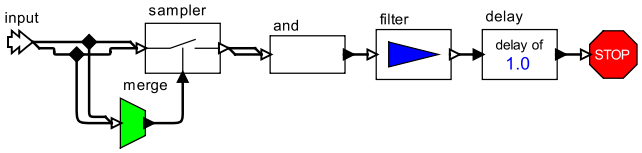
(b) Internal design of the Split actor



(c) Internal design of the Map actor



(d) Internal design of the Reduce actor



(e) Internal design of the WaitStop actor

Fig. 1. MapReduce model with 2 Map machines and 3 Reduce machines

In recent practice we have seen large-scale models containing thousands or even millions of actors. One concrete example is the model for distributed word-counting created using the MapReduce programming paradigm as described by Dean and Ghemawat in Google. [2] The model is designed to utilize hundreds of computers available in a large cluster to count the occurrences of each word in a huge number of web documents. We have created a simplified demo using 5 worker machines in the Ptolemy II modeling and simulation environment [3], as shown in Fig. 1. Two of the machines (modeled by actors) provide the Map functionality and three provide Reduce. The Split actor, located either on a central computer or on any of the worker machines, distributes the key-document pairs received at its input port to the Map machines. The Map machines map words in the input documents onto word-number pairs. In this case, the numbers in those pairs are always 1, each denoting a single occurrence of a word. Those pairs are then sent to the Reduce machines designated by the hash code of the words. Therefore, pairs containing the same word are always delivered to the same Reduce machine. The Reduce machines then count the pairs for each word that they receive, and send the result to the Merge actor for output. When all input documents are processed, the WaitStop actor receives a true-valued input, and terminates the execution.

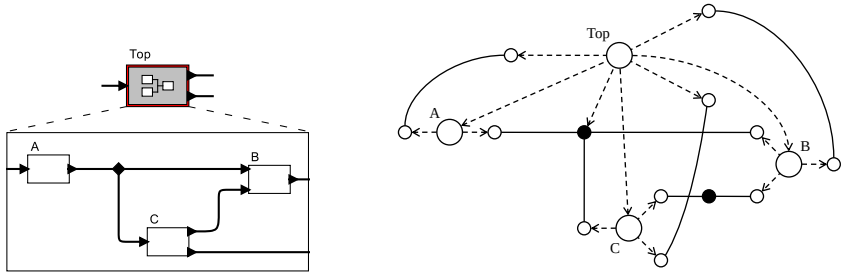
It is hard to imagine manually constructing one such model for a large number of worker machines. The complexity of the work grows sharply as the number of actors increases. Even if the model can be constructed manually, it is still extremely difficult to modify or maintain the design. We are thus motivated to explore an approach to automated model construction and modification.

In our prior work, we have developed Ptalon as a declarative language for higher-order model composition. [4] Textual model descriptions can be written by designers to manipulate model fragments as first-class objects, and to compose them to generate large models. As an example, the same MapReduce model is constructed with Ptalon. [5] It is shown that the size of the Ptalon description does not grow with the increase of worker machines used by the model. This is because the number of worker machines is defined as an integer parameter that can be set by the user.

In our recent effort on higher-order model composition, we aim to provide a more straightforward and user-friendly mechanism for constructing models. We view models as attributed graphs, and employ graph transformation technique. We have implemented a tool that supports a visual language for specifying transformations. The visual language is very close to the language that model designers use to manually create models. This removes the need for requiring model designers to learn new languages.

We envision a variety of applications for our technique. In this paper, we present an example for automated model construction and execution. Other applications include model optimization, refactoring, structural parametrization, and workflow automation.

(AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, HSBC, Lockheed-Martin, National Instruments, and Toyota.



(a) An example of a hierarchical model (b) Attributed graph representation

Fig. 2. Hierarchical model and its representation in attributed graph

This technique differs from Ptalon in the following ways.

1. Models are constructed with graph transformation rules that have a visual representation, whereas in Ptalon, textual descriptions are used.
2. Transformations can be applied to existing models as incremental modifications statically or dynamically.
3. A hierarchical heterogeneous model can be used to control the application of multiple transformations.

The following sections are organized as follows. In Section 2, we present models as graphs and define basic transformations. In Section 3, we discuss using a model to control basic transformations to form a complex transformation. We construct a MapReduce model with transformation as an example in Section 4. We study the related work in Section 5, and conclude our discussion in Section 6.

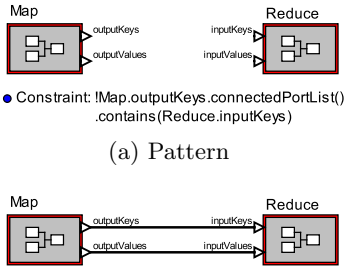
2 Model Transformation Based on Graph Transformation

Fig. 2 shows how a hierarchical model can be represented with an attributed graph. In Fig. 2(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 by big circles, ports represented by small hollow circles, and relations by filled dots. There are 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.

This alternative representation of models allows us to directly apply graph transformation techniques [6] to modify model structures.

2.1 Visual Representation of Transformation Rules

We use a visual syntax to specify transformation. This syntax is inspired by triple graph grammar [7]. A transformation is defined by a *transformation rule*,



(a) Pattern

(b) Replacement

	Pattern	Replacement
1	Map	Map
2	Reduce	Reduce

(c) Correspondence

Fig. 3. A transformation rule for connecting a Map and a Reduce

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. 3 shows a transformation rule designed for our MapReduce example, which we will further discussed 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 this way. In the pattern, two *matchers* aim to match two distinct actors in the given model. The names of the matchers are insignificant and need not be the same as those of the matched actors. Without considering the constraint, the two matchers in the pattern match any two such actors: one with output ports named "outputKeys" and "outputValues," and the other with input ports "inputKeys" and "inputValues." The constraint requires that the ports of the two actors not be connected before the transformation is applied. This avoids creating excessive connections between the same pairs of ports.

In the replacement, the two matchers are preserved, meaning that the matched actors should be kept after transformation. Two connections are to be created between their ports, because those connections (along with the hidden relations on them) do not 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 in [8] and reviewed in [9]. For completeness, we first review that approach here.

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 $m : V_G \to V_H$, such that for any $v_1, v_2 \in V_G$, if $(v_1, v_2) \in E_G$, then $\big(m(v_1), m(v_2)\big) \in E_H$. 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 $n \circ m : V_G \to V_I$ is the composition of function $m : V_G \to V_H$ and $n : V_H \to V_I$.

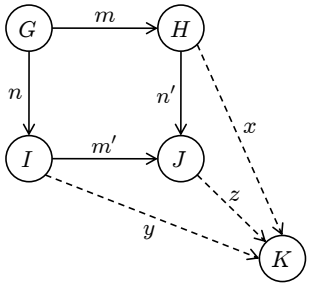


Fig. 4. Pushout of graph morphisms

Given graphs G, H, I, and morphisms $G \xrightarrow{m} H$ and $G \xrightarrow{n} I$ as depicted in Fig. 4, a *pushout* is a tuple $\langle J, I \xrightarrow{m'} J, H \xrightarrow{n'} J \rangle$, in which J is a graph and morphisms $I \xrightarrow{m'} J$ and $H \xrightarrow{n'} J$ satisfy the following conditions:

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.

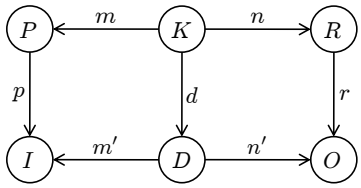


Fig. 5. Graph transformation based on double-pushout

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. 5, 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 of different types. (Recall that three types of vertices and two types of edges are used in Fig. 2.) 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 attribute set, and extend the definition of graph G to be $\langle V_G, E_G, A_G \rangle$, where $A_G : (V_G \cup E_G) \to 2^A$ 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*. 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 $x \in \big(V_G \cup E_G\big)$ and $y \in \big(V_H \cup E_H\big)$, we say that criterion c is *satisfied by* x *matching* y if $\big(A_G(x), A_H(y)\big) \in c$.

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 $m : V_G \to V_H$ satisfying the following additional conditions:

1. for any $v \in V_G$,
   (a) $\forall a \in \big(A_G(v) \setminus U\big).\ a \in A_H(m(v))$

(b) $\forall c \in \big(A_G(v) \cap C\big).\ \big(A_G(v), A_H(m(v))\big) \in c$

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

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 unchecked 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.

Practically, for a transformation depicted in Fig. 5, 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.)

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. Thus, we define total function $F_T : \mathbb{G} \to \mathbb{G}$ to return the result of applying T, where $\mathbb{G}$ is the set of all graphs. If T is applicable to the given input graph and an output graph can be obtained, then F_T returns that output graph; otherwise, F_T simply returns the input graph.

2.5 Model Transformation

The example in Fig. 2 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 lines) 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.

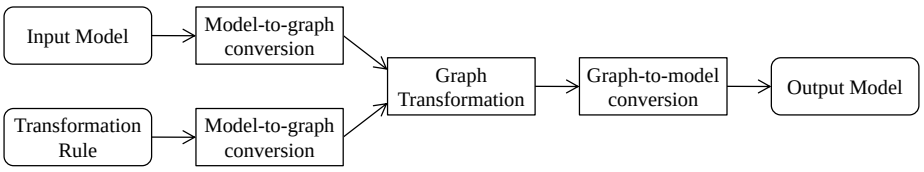


Fig. 6. The model transformation process

Using the graph representation, we establish a model transformation process as shown in Fig. 6. The inputs to the process consist of an input model and a

transformation rule, both specified in the modeling language. (Fig. 1 and Fig. 3 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. 2.)

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 Figure 5. 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

In Section 2.4 we discussed considering transformation rule T to be a function $F_T : \mathbb{G} \to \mathbb{G}$ that takes a graph (or model) as input and produces an output. This allows us to encapsulate the transformation rule in a TransformationRule actor whose functionality is defined by F_T. This actor can be embedded in models using arbitrary models of computation (MoCs). We call our approach of employing hierarchical heterogeneous models to control transformation rules *model-based transformation*. We call the transformation in a single TransformationRule actor a *basic transformation*, as opposed to the transformation obtained by composing TransformationRule actors in a model.

3.1 TransformationRule Actor

A TransformationRule actor has a single input port and two output ports. The input port accepts actor tokens that contain models to be transformed. An output port (the one appearing to the right in its visual representation) outputs the results by applying F_T to the inputs. Another output port (visually shown at the bottom of the actor's icon) produces true or false to signify whether the pattern of the transformation rule matches any subgraph of the input model.

When the TransformationRule actor is opened in the Ptolemy II GUI (Graphical User Interface), an interface appears to allow the designer to edit the transformation rule that this actor represents. This interface has a tab for each of the three components, as is shown in Figure 3. Among these, the correspondence

table is automatically maintained most of the time. When an actor is copied from the "Pattern" tab and pasted to the "Replacement" tab, a row is automatically created to establish the correspondence relation. Deleting an actor may also cause removal of a row.

3.2 A Library of Actors for Model-Based Transformation

In addition to TransformationRule, we have created other actors in an actor library for model-based transformation.

ModelGenerator is used to generate initial actor tokens. It has different usages. If an input string containing the description of a model in the Modeling Markup Language (MoML) is provided, it parses the string and sends the model via its output port. If only a model name is provided, it creates an empty model with the given name.

ModelCombine accepts multiple input models at each time. It merges those models and outputs a combined model. Suppose the n input models are represented with graphs $G_1, G_2, \cdots, G_n$, then the output model can be represented with $G = \langle V_G, E_G, A_G \rangle$, where V_G, E_G and A_G are the disjoint unions of the vertex sets, edge sets and attribute functions (considered as sets of argument-value pairs) of the input graphs. We take an extra step after the merging to update the name attributes so that they are unique at each level of the resulting model.

ModelView displays the input models in a separate window. It updates the window when a new actor token is received. After displaying the model in the actor token, it sends the token to downstream actors via its output port.

ModelExecutor executes the input models to completion. Inputs can be provided to the models being executed via user-customized input ports of ModelExecutor. The tokens available at those input ports are automatically transmitted to the input ports with the same names of the executed models that have the same names. Outputs from the models are also transmitted to the user-customized output ports.

MoMLGenerator exports the input models in the Modeling Markup Language (MoML). The exported strings can be written into files by FileWriter.

3.3 Applications

There is a large variety of applications for model-based transformation. We sketch some of them here. A concrete example of the model construction application will be discussed in the next section.

- *Model construction.* ModelGenerator can be used to generate empty models, which are first-class objects manipulated by a higher-order model (the one that contains the ModelGenerator). Arbitrary models can thus be constructed by modifying empty models with transformations. The constructed models can be executed by ModelExecutor on the fly, or be stored in files by MoMLGenerator and FileWriter.

- *Model optimization.* When appropriate information about model behavior is provided, model transformation can be used to optimize existing models while preserving their behavior. One example is to partially evaluate a given model by eliminating the parts of computation logic that output signals that can be computed statically. The validity is based on the information about whether the actors' outputs are constant, or how actors' outputs depend on their inputs. This information may be provided in the actors' behavioral interface [10], or be obtained with a static analysis of the code that implements the actors.
- *Design refactoring.* Refactoring also preserves model behavior. It usually aims to improve model designs for better understandability or easier maintenance. Take hierarchy flattening as an example. For some models, hierarchy may be eliminated by moving actors to higher levels. An opposite operation is to introduce extra levels to the hierarchy by encapsulating actors, which helps clarify the design and protect the encapsulated parts.
- *Structural parametrization.* Models can be parametrized with placeholders defined in their structures. Model transformations can be used to configure those placeholders to form complete models. Compared to value-based parametrization, structural parametrization is a generalization that provides more design flexibility and reuse opportunity. Furthermore, one can construct a class hierarchy for models, where models at each level (except the root level) are variants obtained from structurally parametrizing some models at a higher level. Formal checking, using for example interface automata [11], can be incorporated to guarantee behavioral properties. This leads to an actor-oriented subclassing mechanism that generalizes the work in [12].
- *Execution parallelization.* Using a model of computation that provides concurrency, multiple models can be executed in parallel with ModelExecutors. Those models can communicate with each other via the user-customized input ports and output ports of the ModelExecutors. This makes it possible to simulate a distributed system, where the models are executed on separate computers and communicate with each other.
- *Workflow automation.* Model-based transformation can also be used to automate tasks in the model development workflow. Those tasks include component configuration and composition, version control, and regression testing.

4 MapReduce Example

In this section we discuss a parametrized higher-order model that generates a MapReduce model and executes it. Fig. 7 shows part of the hierarchy of this higher-order model. A parameter at the top level, `numberOfMachines`, defines the preferred number of worker machines, which can be set by the user. For simulation, we use a FileReader to read the documents from disk and to output them in tokens via the upper output port. The lower output port produces true when all documents are sent, or false otherwise. When it outputs false, the document tokens are queued in the buffer for the ModelExecutor's upper-left input

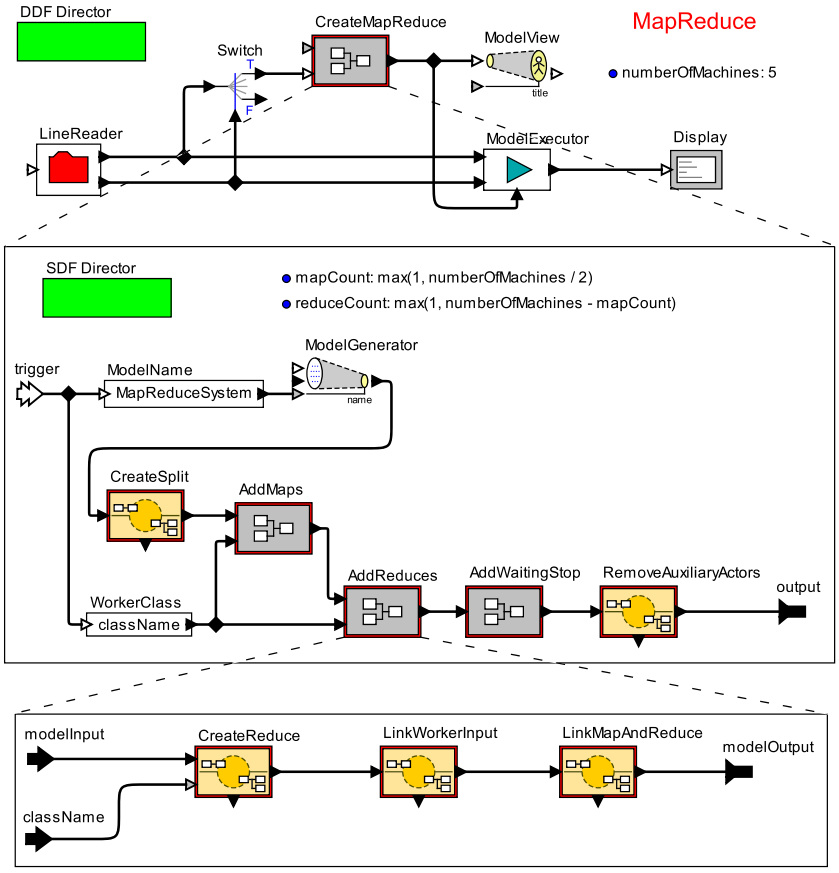
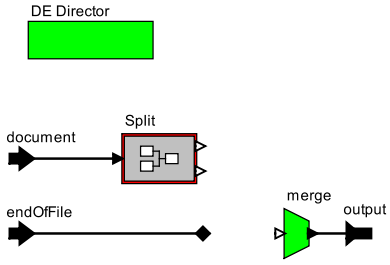


Fig. 7. A higher-order model that generates a MapReduce model and executes it

port. When it outputs true, the Switch sends a token to the CreateMapReduce actor, triggering it to generate a MapReduce model in an actor token. (Fig. 1 shows the MapReduce model generated when `numberOfMachines` equals 5.) The model is then sent to the ModelExecutor for immediate execution. The buffered documents are provided to the model as inputs at its "document" input port, and its outputs to the "result" output port are automatically transmitted to the Display actor connected to the ModelExecutor's output port.

The CreateMapReduce actor contains an SDF (Synchronous Dataflow) model that controls several TransformationRule actors (each represented with a yolk-like icon surrounded by a box). `mapCount` is defined to be the number of Map machines, and `reduceCount` is the number of Reduce machines. Fig. 8 shows the transformation rules in the CreateSplit actor. It creates a Split actor and a Merge actor, together with the input ports and output port of the MapReduce



(a) Pattern (empty) (b) Replacement

	Pattern	Replacement
1	(empty)	

(c) Correspondence

Fig. 8. The transformation rule in CreateSplit

model. Its pattern is deliberately made empty so that the transformation rule is applicable to the empty model generated by the ModelGenerator. In Fig. 3, we have shown the transformation rule in the LinkMapAndReduce actor. It connects the ports of an arbitrary Map actor and an arbitrary Reduce actor, with a constraint to make sure that no duplicated connections are made in multiple applications of the same transformation rule. We set a special parameter, which is not shown in the interface, so that the transformation is applied exactly (`mapCount` × `reduceCount`) times for each input model, so that all the Map actors and Reduce actors in it are interconnected.

This example shows how we use model transformation as a tool to construct a complex model. The size of the dynamically generated model is parametrizable with an integer parameter, and has no impact on the size of the higher-order model that the designer manually creates. Each TransformationRule actor can be separately designed, documented and maintained. It can also be parametrized and reused to construct other models.

5 Related Work

Model transformation has been under active research in recent years. In recognition of the public interest, the OMG (Object Management Group) has issued a request for proposal (RFP) on MOF (Meta-Object Facility) QVT (Query / Views / Transformations) to seek a standardized approach to model transformation [13].

Besides our model transformation tool developed in the Ptolemy II framework, existing tools include AGG [14], PROGRES [15], AToM^3 [16], FUJABA [17], VIATRA2 [18], and GReAT [19]. Among those, our tool is the only one that supports a large and extensible collection of MoCs for controlling basic trans-

formations. By carefully choosing MoCs, sequential transformation and parallel transformation can be achieved, as well as a mixture of both. This control mechanism is more flexible than the priority-based control provided by AGG and AToM^3, the imperative program control implemented in PROGRES, and the restricted selection of MoCs that the other tools offer. Furthermore, our model transformation tool provides a user-friendly language for transformation specification. It employs the same visual language as that used for manually creating models. This frees designers from learning another language for specifying transformations (such as a textual language and UML class diagrams), understanding the meta-models of their models, and describing their transformations in terms of the meta-models. Higher-order model composition for embedded system design is proposed in [20] and [21]. Compared to other related approaches in this field, such as Ptalon [4] and higher-order Petri nets [22], our model-based transformation approach allows designers to visually describe pieces of model structures and to transform them step by step. Besides, 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.

6 Conclusion

We present our approach to higher-order model composition based on model transformation. We provide a formal definition of graph transformation, which serves as the basis of our model transformation technique. We show that basic transformations can be used as actors in a hierarchical heterogeneous models. Our approach makes it easy to create complex transformations as composition of basic ones. We provide a word-counting model designed using the MapReduce programming pattern as a concrete example.

References

1. Goderis, A., Brooks, C., Altintas, I., Lee, E.A., Goble, C.: Heterogeneous composition of models of computation. Technical Report UCB/EECS-2007-139, EECS Department, University of California, Berkeley (2007)
2. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Communications of the ACM **51**(1) (2008) 107–113
3. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. Proceedings of the IEEE **91**(2) (2003) 127–144
4. Cataldo, A., Cheong, E., Feng, T.H., Lee, E.A., Mihal, A.C.: 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 (2006)
5. Cataldo, J.A.: The Power of Higher-Order Composition Languages in System Design. PhD thesis, EECS Department, University of California, Berkeley (2006)

6. Königs, A.: Model transformation with triple graph grammars. In: Model Transformations in Practice Workshop. (2005)
7. Schürr, A.: Specification of graph translators with triple graph grammars. In: WG '94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science, Springer-Verlag (1994) 151–163
8. Ehrig, H., Pfender, M., Schneider, H.J.: Graph-grammars: An algebraic approach. In: Annual Symposium on Foundations of Computer Science (FOCS). (1973) 167–180
9. Habel, A., Müller, J., Plump, D.: Double-pushout graph transformation revisited. Mathematical. Structures in Comp. Sci. **11**(5) (2001) 637–688
10. Lee, E.A., Xiong, Y.: A behavioral type system and its application in Ptolemy II. Formal Aspects of Computing **16**(3) (2004) 210–237
11. de Alfaro, L., Henzinger, T.A.: Interface automata. ACM SIGSOFT Software Engineering Notes **26**(5) (2001) 109–120
12. Lee, E.A., Liu, X., Neuendorffer, S.: Classes and inheritance in actor-oriented design. ACM Transactions on Embedded Computing Systems (TECS) **to appear** (2008)
13. Object Management Group (OMG): Request for proposal: MOF 2.0 Query / Views / Transformations RFP (2002)
14. Taentzer, G.: AGG: A tool enviroment for algebraic graph transformation. In: Proceedings of Applications of Graph Transformations with Industrial Relevance (AGTIVE), Kerkrade, The Netherlands (1999)
15. Schürr, A., Winter, A.J., Zündorf, A.: Graph grammar engineering with PROGRES. In: Proceedings of the 5th European Software Engineering Conference, Sitges, Spain (1995) 219–234
16. Lara, J.d., Vangheluwe, H.: AToM^3: A tool for multi-formalism and metamodelling. In: FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering, Grenoble, France (2002)
17. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: ICSE '00: Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland (2000)
18. Balogh, A., Varró, D.: Advanced model transformation language constructs in the VIATRA2 framework. In: SAC '06: Proceedings of the 2006 ACM symposium on Applied computing, Esslingen, Germany (2006) 1280–1287
19. Agrawal, A., Karsai, G., Shi, F.: A UML-based graph transformation approach for implementing domain-specific model transformations. International Journal on Software and Systems Modeling (2003)
20. Reekie, H.J.: Realtime Signal Processing - Dataflow, Visual, and Functional Programming. PhD thesis, University of Technology at Sydney (1995)
21. Colaço, J.L., Girault, A., Hamon, G., Pouzet, M.: Towards a higher-order synchronous data-flow language. In: EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software, New York, NY, USA, ACM Press (2004) 230–239
22. Janneck, J.W., Esser, R.: Higher-order Petri net modeling – techniques and applications. In: Workshop on Software Engineering and Formal Methods. (2002)