The Journal of Systems and Software 82 (2009) 292–306

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Specifying behavioral semantics of UML diagrams through graph transformations

Jun Kong [a,*], Kang Zhang [b], Jing Dong [b], Dianxiang Xu [a]

[a] North Dakota State University, Fargo, ND 58105, United States
[b] The University of Texas at Dallas, Dallas, TX, United States

ARTICLE INFO

Article history:
Received 8 October 2007
Received in revised form 16 June 2008
Accepted 16 June 2008
Available online 27 June 2008

Keywords:
Graph transformation
Graph grammars
Visual programming
Visual languages
UML
Behavioral semantics
Object-oriented systems

ABSTRACT

The Unified Modeling Language (UML) has been widely accepted as a standard for modeling software systems from various perspectives. The intuitive notations of UML diagrams greatly improve the communication among developers. However, the lack of a formal semantics makes it difficult to automate analysis and verification. This paper offers a graphical yet formal approach to specifying the behavioral semantics of statechart diagrams using graph transformation techniques. It supports many advanced features of statecharts, such as composite states, firing priority, history, junction, and choice. In our approach, a graph grammar is derived automatically from a state machine to summarize the hierarchy of states. Based on the graph grammar, the execution of a set of non-conflict state transitions is interpreted by a sequence of graph transformations. This facilitates verifying a design model against system requirements. To demonstrate our approach, we present a case study on a toll-gate system.
© 2008 Elsevier Inc. All rights reserved.

1. Introduction

Compared with texts, graphs are more intuitive in expressing structural information. Therefore, graphical notations have been extensively used in software development. As a visual modeling language, the Unified Modeling Language (UML) (Rumbaugh et al., 2005) includes various diagrams that specify software artifacts from various points of view. For example, the class diagram models the static structure of a system whereas the statechart diagram describes the behavior of the objects of a class. The intuitive nature of UML notations greatly facilitates distribution and communication of software artifacts among different developers. However, UML lacks a precise semantics, making it difficult to automate verification and analysis. Providing a precise semantics for UML diagrams has gained much attention (Bruel et al., 1998; Evans et al., 1999; Geiger and Zündorf, 2004; Kuske, 2001; Kuske et al., 2002).

Graph transformation offers a computational paradigm of mathematical precision and visual specification (Varró et al., 2002). It provides a means for specifying the semantics of UML diagrams. In general, graph transformation defines computation in a multi-dimensional fashion based on a set of rewriting rules, i.e., *productions*. Each production consists of two parts: a left graph and a right graph. The difference between the two visually indicates the changes caused by a computation. Using graphs to represent the states of a software application, the behavioral semantics can be captured naturally through a sequence of productions, i.e., transitions from one graph (representing the current state) to another (representing the next state). Explaining one set of visual notations (e.g., UML diagrams) by another with a precise meaning (e.g., graph transformation productions) reduces the gap between the specifying language and the specified language. This has been demonstrated by the successful applications of graph transformations to the behavioral semantics of state diagrams (Engels et al., 2000; Kuske, 2001).

Graph-transformation-based approaches (Baresi and Pezzè, 2001; Engels et al., 2000; Gogolla et al., 1998; Kuske, 2001; Varró et al., 2002) are suitable for specifying the semantics of UML statechart diagrams since there is no need to convert from graphical notations to textual/mathematical formalism (Crane et al., 2005). Although some of those approaches support composite states and firing priority, none of them covers the important features of history, junction and choice according to Crane et al. (2005). This paper presents a visual yet formal approach that supports those important features in the UML statechart diagrams. In particular, our approach can automatically translate a UML statechart diagram to a graph grammar with a precise semantics. The automation relieves the burden of learning graph grammar. Furthermore, our approach can lead to the development of a new verification framework. Existing verification frameworks often rely on transformation of system requirements in some formal language, which may cause a mismatch between system requirement and formal specification. In comparison, our approach can directly

* Corresponding author. Tel.: +1 701 231 8179; fax: +1 701 231 8255.
E-mail addresses: jun.kong@ndsu.edu (J. Kong), kzhang@utdallas.edu (K. Zhang), jdong@utdallas.edu (J. Dong), dianxiang.xu@ndsu.edu (D. Xu).

0164-1212/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2008.06.030

verify the defined behavioral semantics against system requirements specified in the form of sequence diagrams.

In our approach, the hierarchy of states in a statechart diagram is automatically formalized as a graph grammar, which extends a graph transformation system by defining an initial graph and classifying terminal and non-terminal objects. Accordingly, the state transition is implemented through a combination of a *parsing process* and a *generating process*. More specifically, starting from an initial graph, the generating process can generate well-formed graphs by iteratively applying productions in the forward direction (Blostein et al., 1994). The parsing process, on the other hand, can recognize the membership of a graph based on a sequence of production applications in the reverse direction (Blostein et al., 1994). It is used to recognize the source state of a state transition while the generating process is to generate the target state. In addition, a set of algebraic structures are abstracted to control the sequence of production applications due to the complex state entries in UML statechart diagrams. As such, our graph-grammar-based approach provides a foundation for directly executing UML models.

Executable UML models (Mellow and Balcer, 2002; Raistrick et al., 2004; Schattkowsky and Müller, 2004, 2005; Starr, 2001), which emphasize the behavioral aspect of a software artifact, can keep specification and implementation consistent. A UML model can be executed by translating it to some platform-dependent code through a code generator. An alternative approach is to directly execute UML models with a precise semantics on a *UML virtual machine* (*UVM*) (Schattkowsky and Müller, 2004, 2005). In our approach, the integrated behavioral semantics of class diagram, statechart diagram and object diagram is defined precisely by two sets of productions. One set of productions are organized in the form of graph grammar that interprets the state transition of objects; and the other set of productions specify the dynamic reconfiguration of object diagrams. The applications of two sets of productions are synchronized according to event dispatching. Our approach is well supported by the methodology of automatic visual language generation (Costagliola et al., 2004; Karsai et al., 2003; Zhang et al., 2001b). The generated language environment is considered to be a UML virtual machine that supports syntax-correct design of visual models and simulates the execution of integrated UML diagrams. Compared with other graph-transformation-based approaches (Ermel et al., 2005; Gogolla et al., 2002; Hölscher et al., 2006; Kuske et al., 2002; Ziemann et al., 2004a,b), which support the execution of integrated UML models, our approach can handle composite states that are typically needed in real-world modeling while it is challenging to effectively recognize and generate composite states due to the state explosion problem.

Various grammar formalisms (Costagliola et al., 1997; Costagliola and Polese, 2000; Rekers and Schürr, 1997; Schürr et al., 1995; Zhang et al., 2001a) have been proposed for different purposes. Most of them use nodes to represent objects and edges to model relations between objects in an abstract syntax. Different from these formalisms, the spatial graph grammar (SGG) (Kong et al., 2006) introduces spatial notions to the abstract syntax. In the SGG, nodes and edges together with spatial relations construct the pre-condition of a production application. The direct representation of spatial information in the abstract syntax can make productions easy to understand since grammar designers often design rules with similar appearances as the represented graphs. Using spatial information to directly model relationships in the abstract syntax is coherent with the concrete representation, which avoids converting spatial information to edges. Therefore, our approach uses the SGG as the underlying formalism to specify the behavioral semantics of statechart and object diagrams.

The contributions of this paper can be summarized as follows:

- Our approach integrates a sound formalism with visual notations to specify the behavioral semantics of statechart diagrams: using one set of visual notations with a precise meaning (e.g., graph transformation) to explain another set of visual notations (e.g., UML statechart diagrams) reduces the efforts of converting graphical notations to textual/mathematical formalisms.
- Among the graph-transformation-based approaches, our approach is the first to address all of the important features of statechart diagrams, including composite state, initial pseudo-state, final state, deepHistory, shallowHistory, join, fork, junction, and choice. Pseudo-states are useful in practical behavior modeling, but they require a sophisticated mechanism to control the state transition when entering a composite state. Our approach defines an efficient control mechanism by mapping each specific state entry onto a corresponding graph transformation rule.
- Our approach automatically converts the hierarchy of states to a graph grammar, and correspondingly applies a parsing process and a generating process to execute state transitions. This efficiently addresses the state explosion problem because a small number of productions can specify a large number of state combinations. Furthermore, it can enforce a consistent state transition.
- This paper also presents a case study, which illustrates the definition of an integrated behavioral semantics and the verification of consistency between a design model and system requirements. The integrated behavioral semantics is defined in the form of a spatial graph grammar, which naturally represents relations through spatial configuration and thus reduces the gap between the abstract and concrete representations of models. Different from other verification methods, our approach can directly apply a use case scenario (represented as a sequence diagram) to the defined behavioral semantics without the need of translating system requirements into some formal language.

In summary, this paper provides a visual yet formal approach to interpreting state transitions in UML statechart diagrams. According to the behavioral semantics of UML statechart diagrams, we define an integrated behavioral semantics, which provides a solid foundation for verifying a design model against system requirements.

The rest of this paper is organized as follows: Following a running example in Section 2, Section 3 introduces the spatial graph grammar – the theoretical foundation of our approach. Section 4 gives an overview of our approach. Section 5 illustrates the semantic specification and verification based on a toll-gate system. Section 6 discusses related work and Section 7 concludes the paper.

2. A motivating example

This section illustrates a toll-gate system (Kong et al., 2005), which is used as a running example to illustrate semantic specification and verification in the following sections. In a road traffic pricing system, drivers of authorized vehicles are charged at toll gates. The tolls are placed at special lanes called *green lanes*. A driver has to install a device (called an *ezpay*) inside his/her vehicle's windshield in order to pass a green lane. The registration of an authorized vehicle having an *ezpay* includes owner's personal data (such as name, date of birth, driver license number, bank account number and vehicle registration number). Each toll gate has a sensor that reads *ezpay*. The information read is stored by the system and used to debit the respective account. When an authorized vehicle passes through a green lane, a green light is turned on. If an un-authorized vehicle passes through it, a camera takes a photo of the vehicle's license plate.

Fig. 1 presents part of the static structure of a toll-gate system. The class *TollGate* models the functionality of a toll gate. It includes seven operations, simulating various actions taken by a toll gate:

- the operation *Create* sets up a new toll gate;
- *allowEnter* indicates that the toll-gate signals the first vehicle in the waiting list to proceed to the toll gate;
- *carEnter* denotes that a vehicle has entered the toll gate;
- *sendReq* specifies that a toll gate sends a request to the server to verify the identity of a vehicle;
- *getAcq* acknowledges the receipt of a confirmation from the server;
- *carExit* simulates the normal exit of an authorized vehicle; and
- *carRunaway* models the run-away of an un-authorized vehicle.

The class *Server* represents a database system including three operations:

- *Create* establishes a server;
- *getReq* indicates the receipt of a request from a toll gate; and
- *Verified* verifies whether a vehicle is authorized or not.

The association *Retrieve* between *Server* and *TollGate* specifies the communications between the server and a toll gate. The class *Camera* simulates a camera with two operations:

- *Picture* takes pictures of an un-authorized vehicle; and
- *turnOff* turns off the camera.

The class *Light* represents traffic lights, which signal vehicles by operations *turnGreen* and *turnRed*. A camera and a traffic light are controlled by a toll gate demonstrated by associations *Manage* and *Control*, respectively. The class *Car* represents vehicles:

- *Create* indicates the arrival of a vehicle waiting to enter a toll gate;
- *Enter* denotes that a vehicle enters a toll gate; and
- *Exit/runAway* specifies the leaving of an authorized/un-authorized vehicle.

The class *Car* connects with *TollGate*, *Light* and *Camera* through different associations.

Fig. 2 shows five statechart diagrams corresponding to the five classes in Fig. 1. Each statechart diagram consists of a set of states interconnected through transitions. According to Fig. 2(c), for example, a *Server* object can be in state *Idle* or *Busy*, and its state transition is triggered by method invocations. When taking composite states into account, active states are organized as a tree. For example, if state *Scan* is active in Fig. 2(a), the state configuration is a two-level tree, which has state *Busy* as the root and state *Scan* as the leaf.

A UML statechart diagram visually describes the behavior of an object. However, its semantics is only informally specified through natural language, which is not suitable for automatic execution and verification. This paper provides a visual yet formal approach to defining the behavioral semantics of UML statechart diagrams. Furthermore, we combine the behavioral semantics with dynamic configurations in object diagrams as an integrated behavioral semantics, which provides a solid foundation for verifying a design model against the system requirements. In the verification framework, the integrated behavioral semantics is defined through graph transformation and the system requirements are presented as sequence diagrams. Taking the toll-gate system as a running example, Section 5.1 informally describes behavioral semantics of UML statechart diagrams. Especially, it focuses on composite states and discusses how to formalize the hierarchy of states as a graph grammar while the appendix gives a formal and complete specification, which covers all of important features in UML statechart diagrams. Section 5.2 defines the dynamic reconfiguration in object diagrams through graph transformation. Section 5.3 combines the behavioral semantics in Section 5.1 and dynamic reconfiguration in Section 5.2 together. Based on the integrated semantics, Section 5.4 discusses the verification between a design model and system requirements.

3. Spatial graph grammars and transformations

Visual languages (Burnett, 2008) provide a direct representation to model structures and concepts in a visual fashion (Cox and Smedley, 2000). Graph grammars (Rozenberg, 1997) with their well-established theoretical background can be used as natural and powerful syntax-definition formalism to specify visual languages.

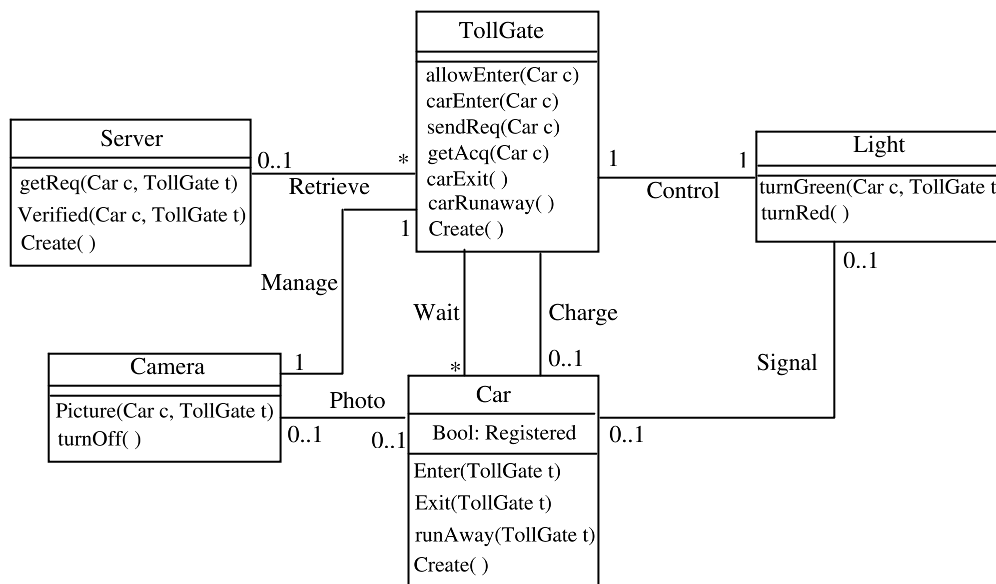

Fig. 1. The static structure of a toll-gate system.

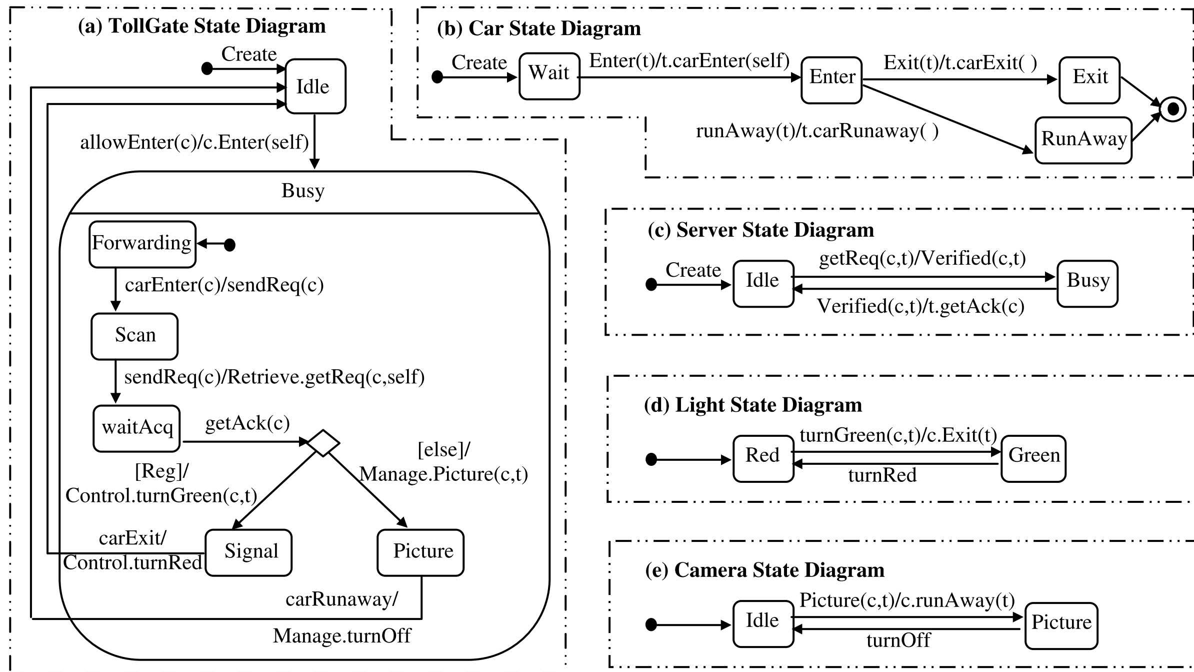

Fig. 2. Statechart diagrams for the toll-gate system.

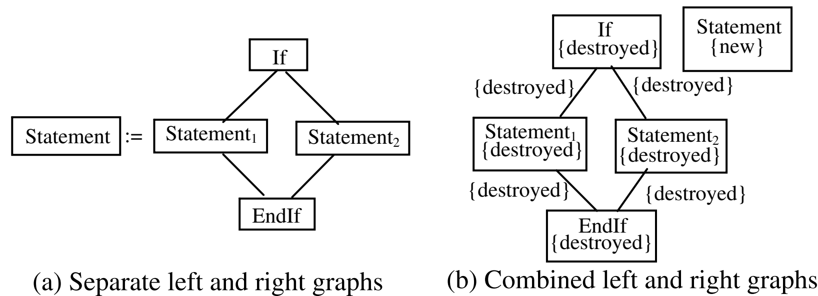

Fig. 3. Left and right graphs.

A graph grammar consists of a set of productions, each of which has two graphs called *left graph* and *right graph*. As presented in Fig. 3(a), we can use two separate graphs to represent the left and right graphs. This representation is analogous to the grammar definition in textual formal languages, and is used to present the automatically derived graph grammar from a statechart diagram. Alternatively, the left and right graphs can be combined into one graph as shown in Fig. 3(b) (Baresi et al., 2003; Fischer et al., 1998), in which nodes and edges tagged with {destroyed} are to be removed in a production application while those tagged with {new} are to be created. This presentation is more concise when there are many overlaps between the left and right graphs, and is used to specify the dynamic reconfiguration of object diagrams.

Applying a production to a graph (i.e., a *host graph*) is referred to as a graph transformation, which can be classified as an L-application (in the forward direction) or R-application (in the reverse direction). A *redex* is a sub-graph in the host graph which is isomorphic to the right graph in an R-application or to the left graph in an L-application. A production's L-application to a host graph is to find in the host graph a redex of the left graph of the production and replace the redex with the right graph. The language, defined as all possible graphs that have only terminal nodes, can be derived through L-applications (i.e., a generating process) from an initial graph. On the other hand, R-applications (i.e., a parsing process) are a reverse replacement (i.e., from the right graph to the left graph), and are used to check the syntactical correctness of a graph.

Being context-sensitive, the spatial graph grammar formalism (SGG) (Kong et al., 2006) allows both left and right graphs to have multiple nodes, and thus is expressive in specifying various types of graphs. Different from other graph grammar formalisms, the SGG introduces spatial notions to the abstract syntax. In the SGG, the spatial information not only contributes to the representation, but also explicitly conveys structural and semantic information. Instead of using attributes/edges to specify an order over a collection of objects, for example, the SGG can specify the order based on the spatial placement of the objects (e.g., the left object has a smaller index than the right one). The direct representation of spatial information in the abstract syntax reduces the gap between the concrete representation of models and the specification of productions. Due to spatial specifications in the abstract syntax, a graph transformation in the SGG may adjust the current spatial configuration. The SGG applies syntax-directed computations through *action codes* to derive new spatial properties from the current spatial attributes. An action code is associated with a production, and is executed when the production is applied. Writing an action code is like writing a standard event handler in Java. In

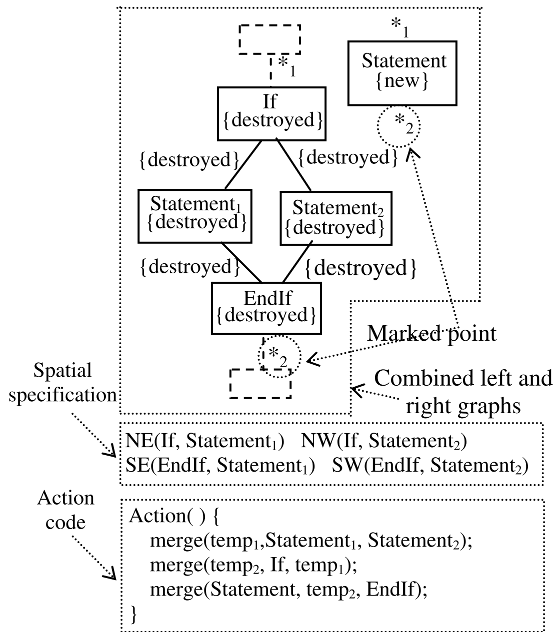

Fig. 4. An SGG production.

addition to left and right graphs, therefore, an SGG production also includes a spatial specification and action code as shown in Fig. 4,[1] which transforms an *if-structure* (i.e., nodes and edges labeled with {destroyed}) to a *statement* node (i.e., the node labeled with {new}). Especially, spatial information is used to distinguish the true and false branches: the left path indicates a true branch while the right one means a false branch.

Due to the multi-dimensional nature of graphs, we need to address the embedding issue in a graph transformation (Rekers and Schürr, 1997), i.e., establishing relationships between the surrounding of a redex and its replacing graph in the host graph. Inherited from the Reserved Graph Grammar (Zhang et al., 2001a), the SGG addresses the embedding issue through the *marking technique*, which identifies connecting points of edges as marked and un-marked ones. In an SGG production, a marked connecting point is highlighted with a "*" followed by a subscript,[2] which is called a marking identity and is used to distinguish marked points from each other. Informally, the semantics of marked points can be stated:

In a production, if the connecting point CP_1 of an edge E is marked and the node attached to the other connecting point CP_2 of E is deleted, CP_2 is redirected to a node having the same marking identity as CP_1.

For example, the production in Fig. 4 defines two marked points, which are used to embed the new *statement* node into the host graph. More specifically, Fig. 5(a) gives a host graph, which includes a redex matching the destroyed sub-graph of the production in Fig. 4. Removing the redex from the host graph causes two temporary dangling edges as shown in Fig. 5(b). Since the newly created *statement* node is designated with the marking identities $^*{}_1$ and $^*{}_2$, the dangling edges are redirected to the *statement* node as shown in Fig. 5(c).

[1] The dashed lines and rectangles in the production represent dummy edges and nodes, which are only used for specifying marked points.

[2] If a production only specifies one marked connecting point, it does not need a subscript.

4. Overview of the approach

Fig. 6 presents an overview of our graph-grammar-based approach to interpreting UML statechart diagrams. A statechart diagram, made up of a set of states and transitions, specifies the behavior of objects of a class during lifetime. States are connected through transitions, each of which consists of an event, a guard and an action. A state transition is triggered and the corresponding action is performed when the event occurs with the guard condition satisfied. States in a statechart diagram are organized in a hierarchical structure: a composite state at a high level of abstraction can be decomposed into a set of sub-states. Due to the hierarchy of states, more than one state can be active at the same time. These active states construct a tree (i.e., a state configuration) starting with the top-most composite state as the root down to single active states as leaves. The hierarchical structure of states empowers statechart diagrams in system modeling while it also challenges the recognition of source states due to state explosion: a small number of states can cause a large number of different combinations. The state explosion can be addressed by a graph grammar in which a set of productions can effectively specify and derive all well-formed state configurations. Our approach automatically formalizes the hierarchy of states in a statechart diagram as a graph grammar. More specifically, a non-terminal node in the derived graph grammar indicates a state configuration, and a terminal node represents a single or composite state. One production specifies the hierarchical relationships between a composite state and its direct sub-state(s) while a graph grammar integrates those local hierarchical relationships together and is able to derive all state configurations complying with the given statechart diagram. Based on the automatically derived graph grammar, the recognition of source states is implemented through a parsing process, and the generation of target states is realized by a generating process. Due to the complexity of state entries, a set of algebraic structures, which map state entries to productions, are used to control the generating process.

Our approach provides a formal foundation to directly verify the consistency between system requirements and a design model. At run-time, a system state is determined by both the structure and the state of each individual object. Accordingly, the evolution of system states is simulated by dynamic reconfiguration in object diagrams (e.g., creating/destroying objects and links) accompanied with state transition of objects, which are synchronized by *call events*. In a statechart diagram associated with class *C*, a call event (Kuske et al., 2002) is of the form $op(p_1,\ldots,p_n)$, where *op* is an operation triggered from object *o* of class *C* and p_1–p_n are parameters. A *call action* is of the form:

(1) $l.op(p_1,\ldots,p_n)$, i.e., the operation $op(p_1,\ldots,p_n)$, is triggered from the object which is retrieved by navigating the link *l* from object *o*; or
(2) $p_i.op(p_1,\ldots,p_n)$, i.e., the operation $op(p_1,\ldots,p_n)$ is triggered from object p_i which is passed from the parameter set.

In order to represent the system state at run-time, we extend object diagrams by associating a state configuration with each object. Furthermore, each object has an event queue storing call events. In an integrated diagram as presented in Fig. 7, we use rectangles to represent objects, rounded rectangles to denote states, and ellipses to indicate call events. The root of a state configuration connects to its designated object through a *current* link. Furthermore, the spatial relation *inside* is used to specify that a call event (e.g., *sendReg(c)*) belongs to the event queue of a *TollGate* object. The behavioral semantics of integrated diagrams is defined by a graph transformation system and a graph grammar synchronized

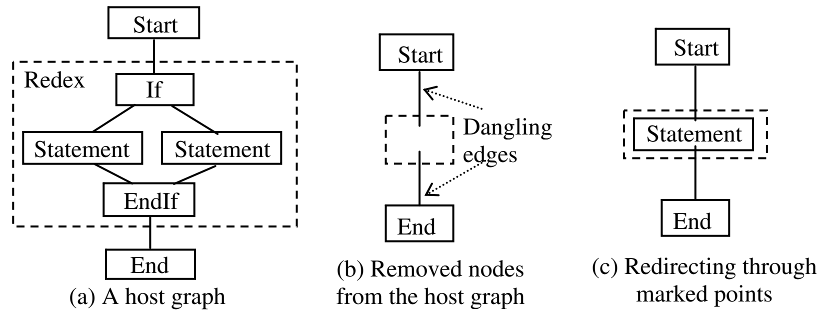

Fig. 5. A graph transformation.

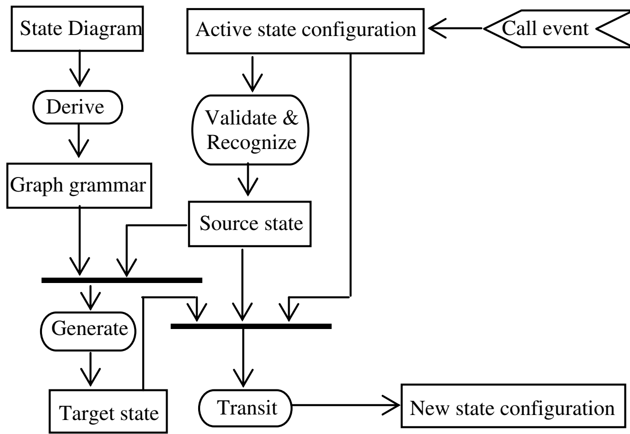

Fig. 6. Interpret state transition based on graph grammar.

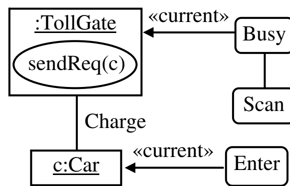

Fig. 7. An integrated diagram.

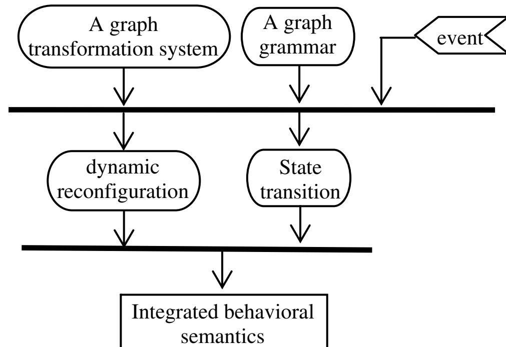

Fig. 8. Integrated behavioral semantics.

through call events as in Fig. 8: dynamic reconfiguration is performed through a graph transformation system while state transition of objects is implemented through an automatically derived graph grammar.

The designers must ensure that the defined behavioral semantics is consistent with system requirements. Fig. 9 presents our verification framework to validate the consistency between a defined model and system requirements. Use cases have been universally adopted for requirement specification. In our verification framework, each use case scenario can be represented as a sequence diagram. The ordered sequence of operation invocations recorded in a sequence diagram triggers the system evolution according to the integrated behavioral semantics. If all the method invocations in a sequence diagram are successfully applied to an initial system state, then the initial system state, the use case scenario and the integrated behavioral semantics are consistent with each other. Otherwise, the designers have to investigate an inconsistency among them.

In summary, we propose a visual approach which automatically assigns precise behavioral semantics to statechart diagrams. Then, we define an integrated behavioral semantics by combining the behavioral semantics of statechart diagrams with dynamic reconfiguration in object diagrams. The integrated behavioral semantics can be directly validated against system requirements.

5. An integrated behavioral semantics of UML models

This section uses the toll-gate system (Section 2) as a running example to illustrate the semantic specification and verification.

5.1. Behavioral semantics of statechart diagrams based on graph grammar

The hierarchical structure of states in a statechart diagram is automatically formalized as a graph grammar in our approach. Each state S in a statechart diagram generates a terminal node S_T and a non-terminal node S_N. From the semantic point of view, the terminal node S_T represents the state itself, and the non-terminal node S_N indicates a state configuration starting from S down to sub-states contained by S. For example, in the state configuration shown in Fig. 10, the non-terminal node $Busy_N$ denotes the state configuration starting from state *Busy* and the terminal node $Busy_T$ means state *Busy* itself.

Each production in a graph grammar defines the hierarchy between a composite state and its direct sub-states. Productions are automatically generated according to the following rules:

- As visualized in Fig. 11(a), a concurrent composite state s with i direct sub-states $(x_1,\ldots,x_i)$ generates a production such that the left graph includes only one non-terminal node s_N and the right graph is a two-level tree having node s_T as the root with i child nodes from x_{1N} to x_{iN}.
- Fig. 11(b) depicts that a sequential composite state s with i direct sub-states $(x_1,\ldots,x_i)$ generates i productions. In each production, the left graph has only one non-terminal node s_N, and the right graph is a two-level tree having node s_T as the root with one child node x_{mN} $(1 \leqslant m \leqslant i)$.

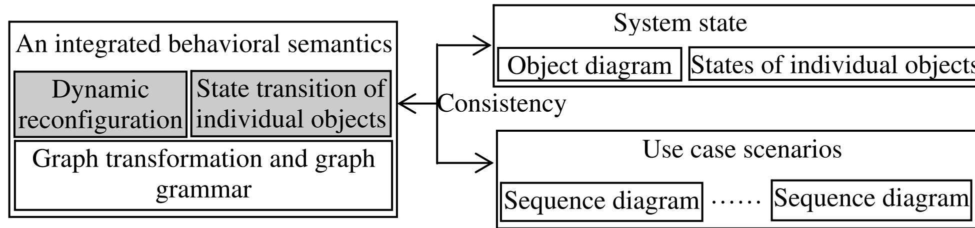

Fig. 9. Validate an integrated behavioral semantics against system requirements.

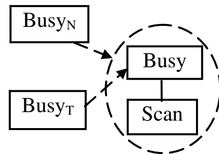

Fig. 10. A state configuration.

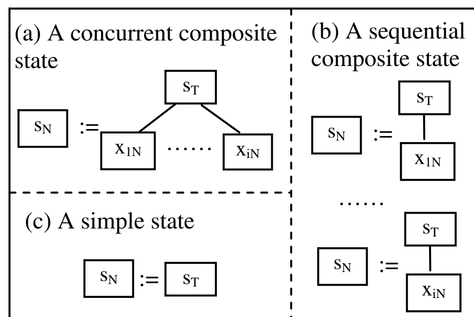

Fig. 11. Production generation.

- Fig. 11(c) shows a simple state s that generates a production such that the left/right graph is made up of one node s_N/s_T.

Based on the above rules, the hierarchy of states is formalized as a graph grammar. For example, corresponding to the statechart diagram in Fig. 2, the sequential composite state *Busy* generates five productions since it has five direct sub-states while each simple state produces one production.

Based on the generated graph grammar, the recognition of a source state can be performed by iterative R-applications in a bottom-up fashion until the corresponding non-terminal node of the source state is abstracted or λ is reached. On the other hand, generating a target state is realized through iterative L-applications in a top-down fashion. However, different state entries (such as *default entry* and *explicit entry*) and pseudo-states (such as *shallowHistory* and *deepHistory*) complicate the process of target-state generation, which requires a sophisticated control mechanism to control the sequence of production applications. For example, the sequential composite state *Busy* generates five productions, and the generation process needs to determine which production should be applied when a default entry to state *Busy* is triggered. Such a control mechanism is defined as a mapping between state entry and production. For example, state *Forwarding* in Fig. 2 is designated as the default state when the composite state *Busy* is initialized. Accordingly, this default entry is mapped to such a production that the left graph is a non-terminal node of state *Busy* and the right graph is a two-level tree, where the root is the terminal node of state *Busy* and the leaf is the non-terminal node of state *Forwarding*. A detailed description of the control mechanism is presented in the appendix.

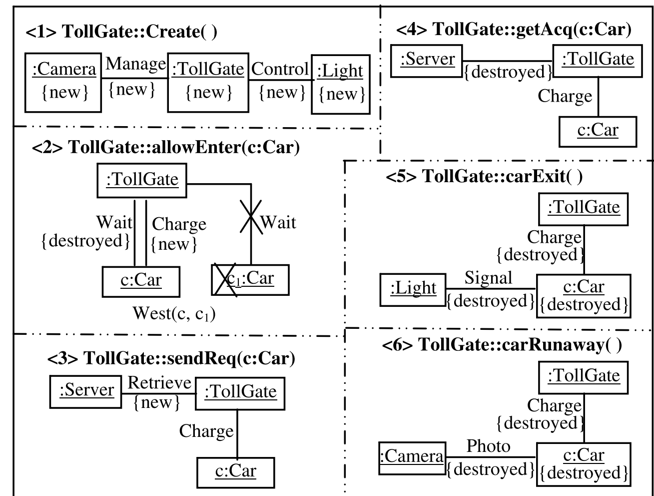

Fig. 12. The dynamic reconfiguration for operations in class *TollGate*.

5.2. Dynamic reconfigurations

As an instance of a class diagram, an object diagram presents a snapshot of a static structure at a point in time. Each object has a set of operations. The execution of an operation is simulated by a dynamic reconfiguration, such as creating or destroying an object in an object diagram. For example, the operation *Create* defined in class *TollGate* creates a *TollGate* object, a *Camera* object, and a *Light* object. A dynamic reconfiguration is specified through a graph transformation rule, in which the left graph defines the pre-configuration and the right graph specifies the post-configuration. A production application, i.e., a graph transformation, is triggered when its corresponding operation is invoked.

Fig. 12 presents six operations defined in class *TollGate*.[3] Production 1 specifies the operation *Create*, which simulates the scenario of creating a new toll gate. Due to the 1-to-1 cardinality between class *TollGate* and class *Camera*, the creation of a *Camera* object is associated with the creation of a *TollGate* object (same for the creation of a *Light* object). The operation *allowEnter*, as defined in Production 2, models that the first arriving vehicle is forwarded to an empty toll gate. Instead of using attributes, the arriving order of vehicles is visually represented through the spatial arrangement among *Car* objects, i.e., a *Car* object locating to the east indicates that it comes earlier. Based on the spatial specification and negative application condition (Ermel et al., 1999), Production 2 specifies that no other *Car* object locates to the east of the *Car* object *c*, which implies that *c* is the earliest arriving vehicle. Therefore, object *c* is forwarded to the toll gate, simulated by establishing the *Charge* link and destroying the *Wait* link between a *TollGate* object and object *c*. Production 3 illustrates the operation *sendReq*, which

[3] The execution of an operation may not necessarily cause a dynamic reconfiguration, and Fig. 12 only presents those operations which can change the structure.

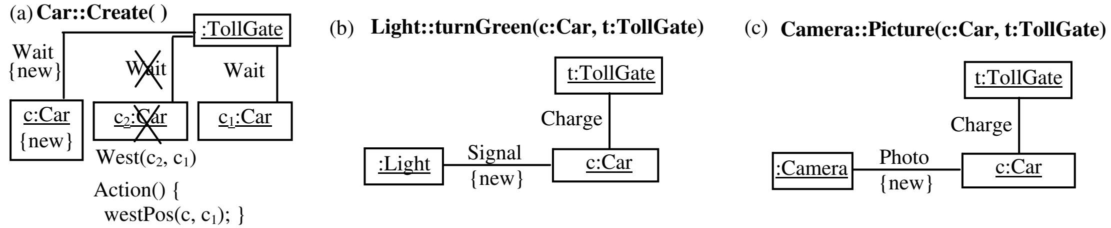

Fig. 13. The productions for other operations.

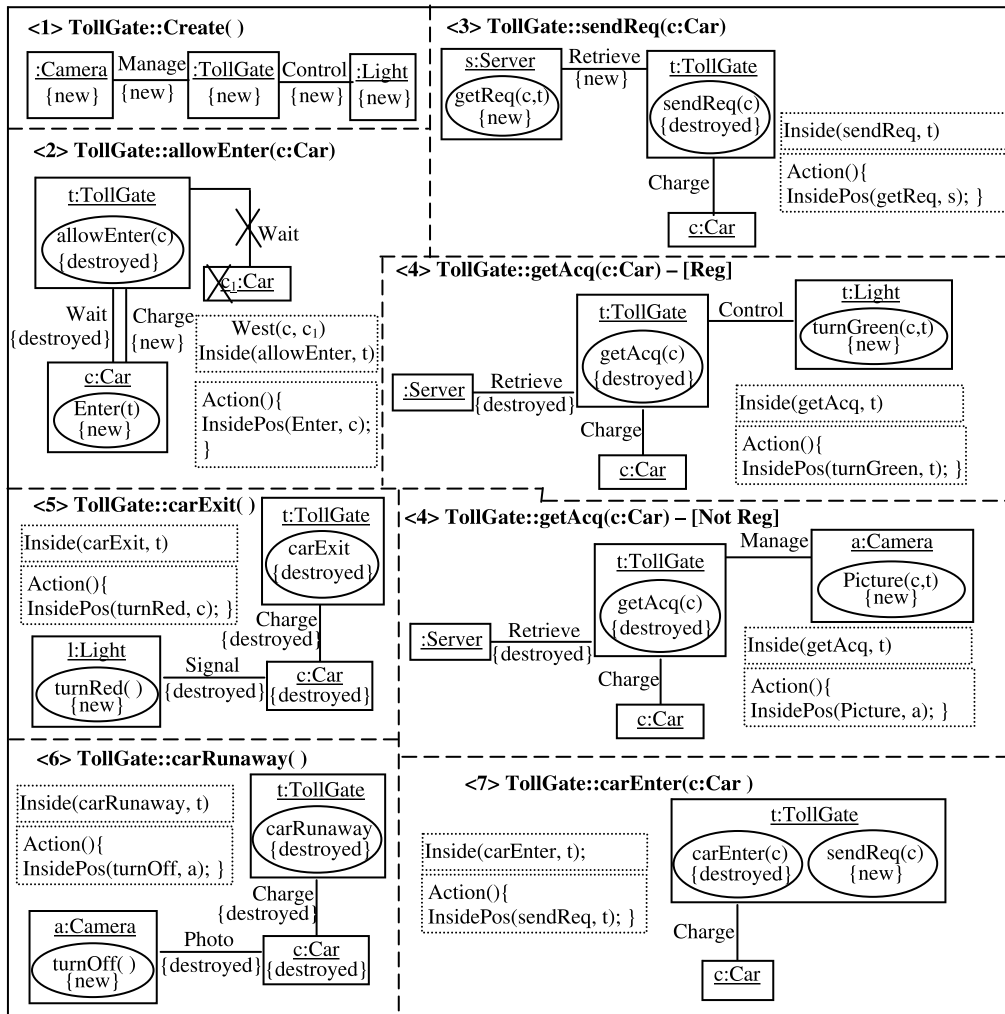

Fig. 14. Combining dynamic reconfiguration with call events in class *TollGate*.

sets up a communication between a toll gate and a server modeled by the establishment of a *Retrieve* link. Production 4 defines the operation *getAcq*, which disconnects the communication between a toll gate and a server after the request of an inquiry is completed. Production 5 illustrates the operation *carExit*: an authorized vehicle exits a toll gate by destroying the *Charge* link between a *TollGate* object and a *Car* object and by removing the *Signal* link between a *Light* object and the *Car* object. Similarly, Production 6 defines the operation *carRunaway* that simulates the run-away of an un-authorized vehicle.

Fig. 12 presents dynamic reconfigurations caused by operations in class *TollGate* while Fig. 13 defines dynamic reconfigurations corresponding to other operations. Fig. 13(a) specifies the operation *Create* in class *Car*, which simulates the arrival of a vehicle: the negative application condition and the spatial specification *West*(c_2,c_1) define that object c_1 is the last vehicle in the waiting list, and the newly arrived *Car* object c is placed west to c_1 according to the action code, so object c becomes the last vehicle in the waiting list. Fig. 13(b) defines the operation *turnGreen* in class *Light* that establishes a *Signal* link between a *Car* object and a *Light* object. Fig. 13(c) demonstrates the operation *Picture* in class *Camera* that sets up a *Photo* link between a *Camera* object and a *Car* object.

5.3. Integrated behavioral semantics

A state transition can stimulate a call action causing an operation to be invoked. For example, according to the statechart dia-

Fig. 15. A system state transition.

Fig. 16. A sequence of operations.

gram in Fig. 2(a), operation *allowEnter* triggers the state transition of a *TollGate* object from state *Idle* to state *Busy*, which stimulates a call action *Enter* in a *Car* object. In order to maintain call events, each object has an event queue, which records call events. The dispatch of a call event may trigger a state transition of an individual object as well as a dynamic reconfiguration. This sub-section specifies an event-driven graph transformation system, which combines the dynamic reconfigurations with the maintenance of call events.

Fig. 14 gives the event-driven graph transformation rules corresponding to the operations in class *TollGate*. The difference between Figs. 12 and 14 is that the productions in Fig. 14 include call events. In Fig. 14, rectangles represent objects and ellipses denote call events. Since the state transition triggered by operation *Create* does not stimulate a call action, Production 1 in Fig. 14 is identical to Production 1 in Fig. 12. According to the statechart diagram in Fig. 2(a), the state transition triggered by *allowEnter* (Production 2) invokes a call action *Enter*, which is stored in the event queue of a *Car* object *c*. Instead of using links, the belonging relationship between an event and its object is visually defined through a spatial arrangement: the event is placed *inside* the object. For example, in Production 2, the event *allowEnter*, which is dispatched and labeled with a destroyed tag, is placed inside the *TollGate* object *t* and the stimulated call event *Enter* is put correspondingly inside the *Car* object *c*. Similarly, Productions 3–7 define other operations in the class *TollGate*. Especially, the statechart diagram in Fig. 2(a) uses a *choice* pseudo-state to split transitions into multiple outgoing paths on which the decision is made based on the condition of a guard. Therefore, two productions corresponding two dynamic conditional branches are defined for the operation *getAcq*. Operation *carEnter* (as defined in Production 7) only includes the update of call events without a dynamic reconfiguration.

5.4. Verification

An integrated diagram is used to represent the run-time system state at a point in time: an object diagram indicates a snapshot of structures and each individual object in the object diagram has an active state configuration. For example, Fig. 15(a) presents an initial system state in which the root of each state configuration is connected to its associated object through a *current* link.

Given an initial system state, we can evolve the system states by applying ordered operations in a sequence diagram to the initial system state based on an integrated behavioral semantics. Each sequence diagram represents a use case scenario and each operation in the sequence diagram has a formal behavioral semantics defined in terms of graph transformation. For example, in Fig. 15(a), the execution of operation *allowEnter*, whose semantics is defined through Production 2 in Fig. 14, causes a dynamic reconfiguration which changes the structural configuration by destroying the *Wait* link and creating a *Charge* link between a *TollGate* object and a *Car* object. In addition to dynamic reconfiguration, this call event also fires a state transition: the state of the *TollGate* object *t* is transited from *Idle* to *Busy* with a default sub-state *Forwarding*. Fig. 15(b) gives the new system state after operation *allowEnter*.

If the ordered method invocations in a sequence diagram have been successfully applied to an initial system state, it indicates that the use case scenario, integrated behavioral semantics and the initial system state are consistent with each other. Otherwise, the design model is inconsistent with system requirements. For example, Fig. 16 presents a sequence diagram including two ordered method invocations, which are not applicable to the system state of Fig. 15(a) since a *Car* object in the state *Wait* (the initial state of a car object) cannot react to the operation *Exit*. The incompatibility between the integrated behavioral semantics and the sequence of operations requires the designer to investigate the designed model and the use case scenario. In the above example, the sequence diagram is not a valid use case scenario since a car cannot exit a tollgate even before it enters the tollgate.

6. Related work

In their seminal work, Harel and Namaad (1996) described the semantics of statecharts as implemented in the STATEMATE system, a commercial tool designed for the specification and design of real-world systems. Maggiolo-Schettini et al.(1994, 1996) interpreted statecharts through graph rewriting systems and described the evolution of a statechart by means of derivation sequences.

As an object-oriented variant of classical statecharts, UML statechart diagrams have received much attention. Different approaches have been published to interpret UML statechart diagrams. Crane et al. (2005) conducted a survey on the formal semantics of UML statechart diagrams, by classifying them into mathematical models, rewriting systems and translation approaches. Graph transformation, belonging to rewriting systems, has been recognized as a powerful tool to formalize the semantics of languages with dynamic features (Maggiolo-Schettini et al., 1996), which has been applied to specify behavioral semantics of

UML statechart diagrams. According to Crane et al. (2005), only half of the graph transformation based approaches support composite state and the priority of conflicting transitions, and none of them covers such advanced features as history, junction and choice. Our approach supports all of the advanced features in the UML statechart diagrams, including composite states, history, join, fork, junction, and choice. Furthermore, our approach provides a unique automation mechanism that automatically assigns a formal semantics to a statechart diagram without manual efforts.

Kuske (2001) explored the behavioral semantics of statechart diagrams through structured graph transformation. State hierarchies over a set of states are defined through a transformation unit, which includes a set of productions, initial and terminal graph class expressions, a control condition imposed on the production applications, and other transformation units. An active state is considered a sub-graph inside a hierarchy of states. A state transition is performed as the application of a production, where the left graph indicates the current active state and the right graph denotes the new state configuration. Engels et al. (2000) uses graph transformation rules to specify a goal-oriented interpreter for state machines. This approach extends the static meta-model with a specification describing a system's behavioral semantics. The semantics is given in the form of collaboration diagram, which makes the specification compatible with the UML standard. However, Engels's approach does not consider composite states.

Engels et al. (2001) later interpreted state diagrams by mapping them onto a semantic domain of the *Communicating Sequential Processes* (*CSP*) (Hoare, 1985). The mapping is defined by a hybrid rule-based language, which combines textual grammar rules (specifying CSP expressions) with graphical patterns (specifying the abstract syntax of statechart diagrams). Baresi and Pezzè (2001, 2005) provided a general framework to interpret diagram notations in terms of denotational semantics by mapping the abstract syntax of some visual language to a semantic domain through two sets of productions. One set defines the abstract syntax models and the other specifies the semantic models. The application of a production in the abstract syntax grammar can trigger the application of the corresponding production in the semantic grammar. Zhao et al. (2004) transformed the UML state diagrams to Petri-nets based on graph transformation. Hu and Shatz (2006) proposed a process for deriving a state-transition model from a UML state diagram, and then mapping to a colored Petri net, which is supported by a set of automatic analysis tools. Different from those approaches that depend on a set of mapping rules designed by experts, our approach automates the process of formalizing UML statechart diagrams based on graph grammars.

McUmber and Cheng (2001) introduced a general framework, which can transform UML models into specifications of a formal language based on homomorphic mapping between meta-models of the UML and the formal language. Cheng and Wang (2002) formalized OMT's (i.e., Object Modeling Technique – Rumbaugh et al., 1991) dynamic models in terms of the formal specification language LOTOS (Bolognesi and Brinksma, 1989). In contrast, we use graph grammars as the underlying formal basis, which reduces the gap between UML models and targeting specifications by using graphs to interpret graphs.

Varró et al. (2002) gave a rule-based, visual specification of statechart semantics through model transition system, which is a combination of meta-modeling and graph transformation with explicit control structures. This approach is featured by separating purely syntactic (such as states and transitions) and derived static semantic concepts (such as conflicts and priority) of statecharts from their dynamic operational semantics. This feature makes the approach flexible to be applied to different statechart variants. However, this approach does not discuss some important features, such as fork and join, in statechart diagrams.

Burmester et al. (2004) provides a round-trip engineering support for UML and Java. Especially, it tightly integrates the UML class and UML behavior diagrams with a visual programming language. Geiger and Zündorf (2004) used the Fujaba approach to exemplify the specification of the behavioral semantics of state diagrams. In order to process composite states, this approach needs to perform a model transformation to obtain a plain state machine by removing composite states from an original state machine while our approach can directly work on state machines including composite states.

In a statechart diagram, the intended meaning of the arrows touching the boundary of a composite state is asymmetric: the arrows pointing to the boundary denotes an initial pseudo-state while the arrows pointing away from the boundary denote the internal structure of the state (Gogolla et al., 1998). Gogolla et al. (1998) flattened a state diagram by means of graph transformation to make the intended semantics explicit. This approach, however, does not discuss the execution semantics of state transitions.

Our work is related to the research in formalizing UML state diagrams using various formal approaches. Jin et al. (2004) defined the syntax of UML statechart diagrams through the Graph Type Definition Language and specified the dynamic semantics by Abstract state Machines parameterized with syntactically-correct attributed graphs. Lilius et al. (1999) translated a state machine into a term rewriting system. Kwon used conditional term rewriting systems to translate state diagrams into the input language of the model checker SMV (McMillan et al., 1992) for model checking (Kwon, 2000). Latella et al. (1999) first mapped statechart diagrams to extended hierarchical automata and then defined an operational semantics for these automata. Labeled transition system is used as the underlying model to precisely specify UML active classes with an associated statechart (Reggio et al., 2000). von der Beeck (2002) developed a precise structured operational semantics (SOS) for UML-statecharts. *Abstract state machine* is used by Börger et al. (2000) to define the dynamic semantics of UML state machines. None of those approaches uses graph transformation.

Some researches are related to assign a formal unambiguous semantics to integrated UML models. Kuske et al. proposed an integrated formal semantics based on graph transformation for central aspects of UML class, object and state diagrams (Kuske et al., 2002): graph transformation rules are used to interpret operations in UML class diagrams and execute state transitions in UML statechart diagrams; and then an integrated semantics is given based on a combination of graph transformation rules associated with operations and those associated with state transitions. Later, this approach has been extended with UML sequence and collaboration diagrams, which capture the sequence of production applications (Gogolla et al., 2002). Hölscher et al. (2006) and Ziemann et al. (2004a,b) provided a framework for an automatic translation of an integrated UML model, including class, object, use-case, collaboration and state diagrams, into a graph transformation system, which simulates the evolution of the modeled system. This approach gives a formal semantics for a substantial part of the UML and has a minimal impedance mismatch between the original UML model and the semantic domain. Ermel et al. (2005) extended the above integrated transformation-based semantics with animation views, which define model-specific scenario animations in the application domain. Those approaches provide an integrated semantics for UML diagrams. However, they are limited to simple states while our approach is applicable to composite states, which complicate the semantic specification since active states could be represented as a tree instead of a single node. Our approach uses the spatial graph grammar (SGG) (Kong et al., 2006) as the specifying language to define the integrated semantics. When handling

complex models (such as an integrated diagram of object and state diagrams), we can offer a concise representation by using spatial relations to denote structural relations between objects. Such a spatial specification (e.g., the usage of spatial relation to model an order among objects in the above running example) can be directly defined in the SGG abstract syntax, which reduces the gap between the specifying language and the specified language.

Our work is also inspired by the need for software architecture specification and its dynamic evolution. Many *architectural description languages* (ADLs) (Medvidovic and Taylor, 2000) have been proposed to model and analyze software architectures. Based on formal models, those languages allow users to define software architectures without ambiguity, and thus are suitable for automatic reasoning. They are tailored for expert users. On the other hand, the UML (Rumbaugh et al., 2005) offers a visual conceptual tool to improve the distribution of software designs among designers by representing models with an intuitive appearance. Medvidovic et al. systematically presented two strategies to model software architectures in the UML (Medvidovic et al., 2002). Focusing on the usability of concepts, Garlan et al. proposed several alternatives to map concepts from ADLs to the UML (Garlan et al., 2002). Both works exhaustively discuss the strength and weakness of each method. Emphasizing the analysis and validation of designed models, Baresi et al. uses the UML notations to specify the static aspect of architectural styles paired with graph transformation for modeling dynamic reconfiguration (Baresi et al., 2003). Taentzer et al. (1998) used distributed graph transformations to specify dynamic changes in distributed systems. The changes are organized in a two-level hierarchy. One is related to the change in a local node and the other to the structure of the distributed system itself. Métayer (1998) presented a formalism for the definition of software architectures in terms of graphs. Dynamic evolution is defined independently by a coordinator. The above researches emphasize the specification of software architectures and dynamic evolution while they do not discuss the semantics of UML state diagrams.

7. Conclusion

This paper has presented a visual yet formal approach to interpreting state transitions in UML statechart diagrams based on graph grammar and graph transformation. Instead of manually designing a graph grammar, our approach automatically derives the graph grammar to formalize the hierarchy of states from a statechart diagram, which makes our approach compatible with UML since users do not have to understand graph grammar. Furthermore, our approach supports all of the important features of statechart diagrams, including composite state, initial pseudo-state, final state, deepHistory, shallowHistory, join, fork, junction, and choice. Based on our behavioral semantics of the UML statechart diagram, we present an integrated behavioral semantics, which includes dynamic reconfiguration in object diagrams and state transitions in statechart diagrams. The integrated behavioral semantics allows designers to validate a design model against system requirements.

The graph grammar and graph transformation approach provides a solid foundation for design verification without converting the design to some formal language. We will further investigate the verification between a design model and system requirement, such as consistency and reachability. In particular, we are interested in modeling secure software. Statechart diagrams are used to model the behavior of a system and sequence diagrams represent potential attacks. According to the behavioral semantics based on our approach, we can validate the vulnerability of a design model against potential attacks.

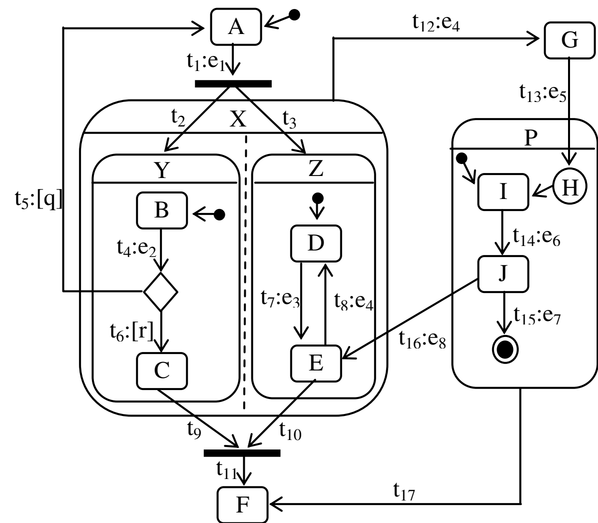

Fig. 17. A statechart diagram.

Acknowledgement

The authors would like to thank the Editor-in-Chief Hans van Vliet and the anonymous reviewers for their insightful and constructive comments that have helped us to significantly improve the presentation.

Appendix. A formal semantics of UML statechart diagrams

Section 5.1 gives an informal description on the behavioral semantics of UML statechart diagrams while this section gives a formal and complete description. UML statechart diagrams have been widely used to model the behavior of system objects. A statechart diagram as presented in Fig. 17 is made up of various states and transitions between those states. A state, which can be specified as a simple state, a sequential composite state or a concurrent composite state, indicates a stage in which an object can be. States are organized in hierarchy, i.e., a composite state at a higher level can be decomposed into several sub-states. A directed edge connects a source state with a target state, indicating a state transition in response to an event. Each transition has a label in the form of *trigger*[*guard*]/*effect*[4]: the *trigger* specifies an event that fires a state transition; the *guard* defines an enabling condition of a state transition; and the *effect* indicates actions to be executed.

Definition 1. A statechart diagram is abstracted as a tuple $\langle S, T, E, src, dst, hie, trigger, guard\rangle$, where

- S is a set of states;
- T is a set of transitions;
- E is a set of events;
- src, dst: $T \rightarrow 2^S$ are partial functions[5] those map a transition to its source state(s) and target state(s), respectively;
- hie: $S \rightarrow S$ is a partial function that defines the hierarchical relations between states such that $hie(s_1) = s_2$ if s_1 is the direct substate of s_2;
- $trigger$: $T \rightarrow E$ is a partial function that defines the trigger of a transition; and

[4] In Fig. 17, the symbol t_i, which prefixes *trigger[guard]/effect*, is used to differentiate transitions.

[5] A partial function on a set V is simply a function whose domain is a subset of V. If f is a partial function on V and $v \in V$, then we write $f(v)\downarrow$ and say that $f(v)$ is defined to indicate that v is in the domain of f; if v is not in the domain of f, we write $f(v)\uparrow$ and say that $f(v)$ is undefined.

- *guard*: T → *A* is a partial function where *A* is a set of attributes.

S_{sc}, S_{cc} and S_s are three disjoint sub-sets in *S*, which represent sequential composite states, concurrent composite states and simple states, respectively. In general, a single transition connects one source state and one target state. Pseudo-states *fork* and *join* are introduced to connect multiple transitions into a more complex state transition path. For example, the *fork* pseudo-state in Fig. 17 connects transitions t_1, t_2 and t_3 into a compound transition that leads to several orthogonal target states. In Definition 1, *src* and *dst* handle a *fork* compound transition as a whole: (1) the transition targeting the *fork* is used to represent the whole compound transition; (2) the source state of a *fork* compound state transition refers to the source of the edge targeting the *fork*; (3) the target states of a *fork* compound state transition are the targets of edges leaving the *fork*. A *join* compound transition is handled in a similar way. Therefore, in Fig. 17, transition t_1/t_{11} represents the *fork/join* compound transition: $src(t_1) = \{A\}$, $dst(t_1) = \{Y,Z\}$, $src(t_{11}) = \{C,E\}$ and $dst(t_{11}) = \{F\}$.

In a hierarchical state machine, an object can have several active states simultaneously. If a simple state is active, all the composite states that either directly or transitively contain the simple state are also active (Object Management Group, 2005). Active states are organized as a tree, i.e., a *state configuration*, starting with the top-most state down to single innermost states. The hierarchical organization of states can cause an interlevel transition, in which the source state and the target state are at different levels of a hierarchy of states. In order to handle interlevel transition, the *least common ancestor* (*LCA*) of a state transition *t* is defined as the lowest composite state that contains all source and target states of *t*.[6] Based on the least common ancestor, the *main source* of a state transition is either (1) the *LCA* of *t* if *LCA* is a concurrent composite state, or (2) a direct sub-state of *LCA*, which contains all source states, if *LCA* is a sequential composite state (Jin et al., 2004). The *main target* of a state transition can be defined similarly. For example, the state transition t_{16} from *J* to *E* in Fig. 17 is an interlevel transition, where the main source is state *P* and the main target is state *X*.

Definition 2. A tree is defined as a tuple $\langle N, par\rangle$, where *N* is a set of nodes and $par : N \rightarrow N$ is a partial function which maps a node to its parent.

In a statechart diagram, state *S* generates a terminal-node S_T and a non-terminal node S_N. Then, the state hierarchy is formalized as a graph grammar. Definitions 3–5 (refer to Section 5.1 for an informal description) define the production generation for a concurrent composite state, a sequential composite state and a single state, respectively.

Definition 3. Given a statechart diagram *$, T, E, src, dst, hie, trigger, guard*$\rangle$, $\forall s \in S_{cc}$, state *s* generates a production $p = (Lg, Rg)$ where

- $Lg = \langle\{s_N\}, par\rangle$ is a tree with only a root node;
- $Rg = \langle N, par\rangle$ is a tree, where $N = \{s_T\} \cup \{x_N | hie(x) = s\}$ and $par : N \rightarrow N$ such that $\forall x_N \in N$ and $x_N \neq s_T$, $par(x_{N)} = s_T$.

Definition 4. Given a statechart diagram *$, T, E, src, dst, hie, trigger, guard*$\rangle$, $\forall s \in S_{sc}$, state *s* generates a set of productions $P = \{(Lg_1, Rg_1), (Lg_2, Rg_2), \ldots\}$, where $|P|$ is equal to the number of direct sub-states contained by state *s* and each production $p_i = (Lg_i, Rg_i)$ respects the following pattern:

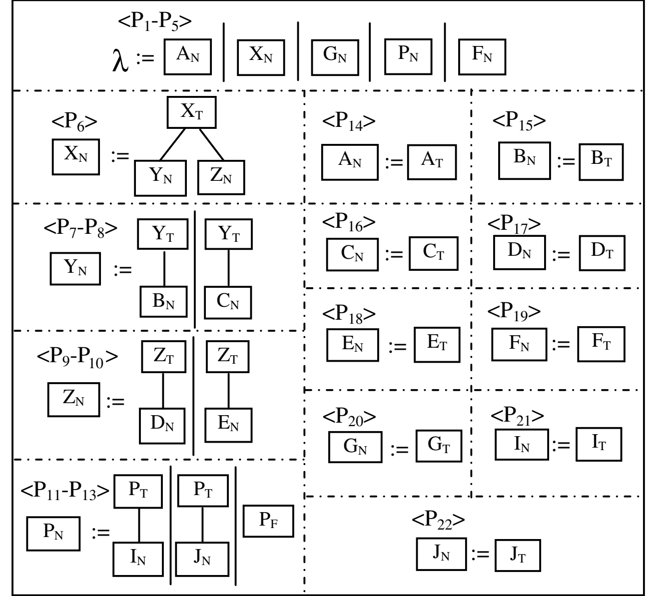

Fig. 18. A graph grammar corresponding to the statechart diagram in Fig. 17.

- $Lg_i = \langle\{s_N\}, par\rangle$ is a tree with only a root node.
- $Rg_i = \langle\{s_T, x_{iN}\}, par\rangle$ is a tree where $par(x_{iN)} = s_T$ and $hie(x_i) = s$.

Definition 5. Given a statechart diagram *$, T, E, src, dst, hie, trigger, guard*$\rangle$, $\forall s \in S_s$, state *s* generates a production $p = (Lg, Rg)$ where

- $Lg = \langle\{s_N\}, par\rangle$ is a tree with only a root node.
- $Rg = \langle\{s_T\}, par\rangle$ is a tree with only a root node.

According to Definitions 3–5, each statechart diagram automatically generates a graph grammar. Fig. 18 shows the graph grammar automatically generated from the statechart diagram in Fig. 17. Especially, each graph grammar has an initial graph (denoted by λ), which indicates the start point of a generating process and the end point of a parsing process. In order to make the generated graph grammar complete, each state *s*, which is not contained by any other state, will generate an additional grammatical rule such that the left graph is the initial graph λ and the right graph is the node s_N. For example, state *A* in Fig. 17 generates Production 1 in Fig. 18.

Definition 6. A graph grammar is a tuple $gg = \langle\lambda, P, S_T, S_N\rangle$, where λ is the initial graph, *P* is the set of productions, S_T is the set of terminal nodes and S_N is the set of non-terminal nodes.

Theorem 1. *Given a statechart diagram* $\langle S, T, E, src, dst, hie, trigger, guard\rangle$, *all possible state configurations can be generated based on the automatically derived graph grammar.*

Proof

(1) Consider a simple state $s \in S$, it is obvious that the graph grammar can produce the state configuration with only one state.
(2) Consider a sequential composite state $s \in S$. Based on the first induction, assume that every sub-state configuration, whose root is a direct sub-state of state *s*, can be generated from derived productions. Since (a) according to Definition 4, the set of generated productions $P = \{P_1, \ldots, P_K\}$, where *K* is equal to the number of direct sub-states contained by

[6] When calculating the *LCA*, a special state *top* is introduced to represent the top-most state, which contains all states in a statechart diagram.

state s, can produce all valid two-level state configurations $SC = \{SC_1, \ldots, SC_K\}$, where $SC_i = \langle\{s_T, x_{iN}\}, par_i\rangle$ ($1 \leqslant i \leqslant K$): $par_i(x_{iN}) = s_T$ and state x_i is a direct sub-state of state s; (b) according to the assumption, the leaf x_{iN} in the two-level state configuration can be further expanded to produce all valid sub-state configurations, we can generate all state configurations having state s as the root.

(3) It is similar to prove concurrent composite states. □

Based on the generated graph grammar, the state transition is implemented as a combination of a parsing process and a generating process. However, different state entries and pseudo-states require a precise mechanism to control the generating process. Definitions 7–10 determine the sequence of production applications for a generating process.

Definition 7. Given a statechart diagram ⟨$S, T, E, src, dst, hie, trigger, guard$⟩, *EntryType*: $T \times S \to$ { *DefaultEntry*, *ExplicitEntry*, *shallowHistory*, *deepHistory*} is a partial function, which defines the type of entry upon entering a composite state, e.g., $EntryType(t_{16}, Z) = ExplicitEntry$ and $EntryType(t_{13}, P) = shallowHistory$.

An initial pseudo-state, denoted by a small solid filled circle, is used to specify the default sub-state of a sequential composite state with a *default entry*. For example, state B in Fig. 17 is designated as the default state when the composite state Y is initialized. On the other hand, if a transition goes to a sub-state of a composite state, then an *explicit entry* is applied such that the sub-state becomes active. For example, state transition t_{16} in Fig. 17 enters state Z with an explicit entry, which goes to state E. The pseudo-state *shallowHistory* represents the most recent active sub-state of the composite state that directly contains *shallowHistory* (but not the sub-states of that sub-state) while *deepHistory* represents the most recent active configuration. One state transition may originate from the *shallowHistory*/*deepHistory* to a default history state in case that the composite state had never been active before. Based on the entry type defined in Definition 7, Definitions 8–10 determine the sequence of L-applications (i.e., a generating process) that generate a target state.

Definition 8. Based on a graph grammar $gg = \langle\lambda, P, S_T, S_N\rangle$ derived from a statechart diagram ⟨ $S, T, E, src, dst, hie, trigger, guard$⟩, a *default application* is defined as a partial function *default*: $S_N \to P$ such that $\forall s_N \in S_N$, $default(s_N) = p$ if state x is the default state when the composite state s is initialized and $x_N \in p.Rg.N$. For example, $default(Y_N) = p_7$, $default(Z_N) = p_9$, and $default(P_N) = p_{11}$.

Definition 9. Given a graph grammar $gg = \langle\lambda, P, S_T, S_N\rangle$ derived from a statechart diagram ⟨$S, T, E, src, dst, hie, trigger, guard$⟩, an *explicit application* is defined as a partial function *explicit*: $T \times S_N \to P$ such that $\forall t \in T$, $\forall s_N \in S_N$, $explicit(t, s_N) = p$ if transition t goes to a direct sub-state x of state s and $x_N \in p.Rg.N$. For example, $explicit(t_{16}, Z_N) = p_{10}$.

Definition 10. Based on a graph grammar $gg = \langle\lambda, P, S_T, S_N\rangle$ derived from a statechart diagram ⟨$S, T, E, src, dst, hie, trigger, guard$⟩, *DefaultHistory*: $S_N \times TYPE \to P$ is a partial function where $TYPE$ = {*shallowHistory*, *deepHistory*} and $\forall s_N \in S_N$, $DefaultHistory(s_N, type) = p$ if the default history state in s is referred to as state x and $x_N \in p.Rg.N$. For example, $DefaultHistory(P_N, shallowHistory) = p_{11}$.

Definitions 7–10 specify a collection of algebraic structures used to control the sequence of production applications. According to Definition 9, for example, Production 10 is applied when state transition t_{16} enters state Z since the right graph of Production 10 contains the non-terminal node of state E corresponding to the explicit entry. Furthermore, in order to support the pseudo-states of *junction* and *choice*, we assume two external functions "*eval(T)*" and "*exec(T)*" provided by the run-time environment (Jin et al., 2004). The external function *eval* is used to evaluate the guard and *exec* to carry out the effect of a state transition. Based on the automatically derived graph grammar and a set of algebraic structures, Figs. 19–22 give the details of executing a state transition.

A *completion transition*, triggered by a *completion event*, originates from a state without an explicit trigger. A composite state s can generate a completion event if s is a sequential composite and some final sub-state of s is active or if s is a concurrent composite state and all its regions have active final sub-states. In the derived graph grammar, we use the subscript "F" to indicate a final state. For example, the node P_F in Production 13 in Fig. 18 indicates a final state contained in the composite state P. Since a completion

```
 1  StateChart Diagram STC=<S, T, E, src, dst, hie, trigger, guard>
 2  StateConfiguration CurrentConfiguration=<N, par>
 3
 4  StateTransition ()
 5  {
 6    While (EventQueue is not empty)
 7    {
 8       if (t←CompletionCheck(CurrentConfiguration))
 9          Triggering(t);
10       Pop up event from EventQueue;
11       Fire(event);
12    }
13  }
```

Fig. 19. The StateTransition method.

```
 1  Fire(E event)
 2  {
 3    EnabledTrans={t∈STC.T | (STC.trigger(t)=event or STC.trigger↑)
 4                   and (STC.src(t)⊆CurrentStates.N) and eval(t)};
 5    RecognizedStates=Ø;
 6    While (EnabledTrans is not empty)
 7    {
 8       X_N ← apply a production p to CurrentConfiguration;
 9       Update CurrentConfiguration with X_N;
10       RecognizedStates=RecognizedStates ∪ {X_N};
11       if (∃t∈EnabledTrans, STC.src(t)⊆RecognizedStates)
12       {
13          Triggering(t);
14          EnabledTrans=EnabledTrans-{t}-{Conflict(t)};
15       }
16    }
17  }
```

Fig. 20. The *Fire* method.

```
 1  Triggering(Transition t)
 2  {
 3    exec(t);
 4    if (STC.trg(t) is a junction or a choice peusodostate)
 5        t ← t' such that STC.trg(t)=STC.src(t') and eval(t')
 6    MainSource ← ms(t);
 7    Exit MainSource and its substates from bottom to top;
 8    Update the history of MainSource and its substates;
 9    NewStateConf ← Enter(t) ;
10    Remove MainSource and its substates from CurrentConfiguration;
11    Add NewStateConf to CurrentConfiguration;
12  }
```

Fig. 21. The *Triggering* method.

```
1  StateConfiguration Enter(Transition t)
2  {
3     MainTarget ← mt(t);
4     NewStateConf ← MainTarget_N
5     While (NewStateConf includes non-terminal nodes)
6     {
7        X_N ← a non-terminal node in NewStateConf;
8        switch (EntryType(t, X_N))
9        {
10          case "ShallowHistory": expand X_N based on SH(X_N); break;
11          case "DeepHistroy": expand X_N and its substates based on DH(X_N); break;
12          case "ExplicitEntry": expand X_N by applying Production explicit(t, X_N);
13          default: expand X_N by applying Production default(X_N);
14       }
15    }
16    return NewStateConf;
17 }
```

Fig. 22. The *Enter* method.

event is dispatched before any other events, the method *StateTransition* in Fig. 19 first handles completion events: function *CompletionCheck*: *StateConfigurations* → *STC.T* is a partial function which verifies whether the current state configuration can trigger a completion transition. If a completion event is detected, *Triggering* (defined in Fig. 21) handles the triggered completion transition; otherwise, a regular event is dispatched and processed by the method *Fire* as defined in Fig. 20.

Fig. 20 presents the *Fire* method, which executes state transitions triggered by *event*. In the *Fire* method, *EnabledTrans* indicates the set of transitions that are eligible to be triggered by *event* under the current state configuration. Based on the automatically generated graph grammar, our approach parses the current state configuration to recognize the source state. The parsing process can inherently ensure consistent transitions. Two transitions are in *conflict* with each other if the intersection of their exit sets of states is not empty. The priority of a transition is defined based on its source state. In general, state transition t_1 originating from source state s_1 has a higher priority than state transition t_2 originating from source state s_2 if s_1 is a direct or transitive sub-state of s_2. The parsing process proceeds in a bottom-up fashion since the recognition of a state at a higher level is dependent on the recognition of its sub-states. Therefore, s_1 is recognized before s_2. Once a source state is recognized, the method *triggering* (shown in Fig. 21) is invoked to perform the corresponding state transition and conflicting transitions are eliminated from the set *EnabledTrans*. In summary, our approach is consistent with the firing priority.

Fig. 21 presents the *Triggering* method, which is designed to exit the source state(s) and enter the target state(s) in response to state transition *t*. In order to handle interlevel transition, i.e., the source state and the target state of a state transition are at different levels in a hierarchy of states (e.g., t_{16} in Fig. 17), function *ms*(t) is invoked to calculate the *main source* of state transition *t*. After exiting the main source, the main target is entered by calling *Enter*(t) as in Fig. 22. The function *mt*(t) calculates the main target and the generating process correspondingly starts from the non-terminal node $MainTarget_N$. $MainTarget_N$ is expanded to a state configuration, which only contains terminal nodes, through iterative L-applications while the sequence of production applications is specified through a *switch* statement based on Definitions 7–10. Especially, SH(X_N) represents the most recent active sub-state of composite state *X*. If state *X* had never been visited before, *DefaultHistory*(X_N, *shallowHistory*) (defined in Definition 10) is called to determine the default history state. A deep history is handled in a similar way.

References

Baresi, L., Pezzè, M., 2001. On formalizing UML with high-level Petri nets. In: Proceedings of the Concurrent Object-Oriented Programming and Petri Nets. LNCS 2001, pp. 271–300.

Baresi, L., Heckel, R., Thöne, S., Varró, D., 2003. Modeling and validation of service-oriented architectures: application vs. style. In: Proceedings of the ESEC/FSE'03, pp. 68–77.

Baresi, L., Pezzè, M., 2005. Formal interpreters for diagram notations. ACM Transactions on Software Engineering and Methodology 14 (1), 42–84.

von der Beeck, M., 2002. A structured operational semantics for UML-statecharts. Software and Systems Modeling 1 (2), 130–141.

Blostein, D., Fahmy, H., Grbavec, A., 1994. Issues in the practical use of graph rewriting. In: Proceedings of the 5th International Workshop on Graph Grammars and their Application to Computer Science. LNCS 1073, pp. 38–55.

Bolognesi, T., Brinksma, E., 1989. Introduction to the ISO specification language LOTOS. In: Eijk, P.H.J.V., Vissers, C.A., Diaz, M. (Eds.), The Formal Description Technique LOTOS, pp. 23–73.

Börger, E., Cavarra, A., Riccobene, E., 2000. Modeling the dynamics of UML state machines. In: Proceedings of the ASM 2000. LNCS 1912, pp. 223–241.

Bruel, J.M., France, R., 1998. Transforming UML models to formal specifications. In: Proceedings OOPSLA'98 Workshop on Formalizing UML: Why? How?

Burmester, S., Giese, H., Niere, J., Tichy, M., Wadsack, J.P., Wagner, R., Wendehals, L., Zündorf, A., 2004. Tool integration at the meta-model level: the Fujaba approach. International Journal on Software Tools for Technology Transfer 6 (3), 203–218.

Burnett, M.M., 2008. Visual language research bibliography. <http://www.cs.orst.edu/~burnett/vpl.html> (up to date).

Cheng, B.H.C., Wang, E.Y., 2002. Formalizing and integrating the dynamic model for object-oriented modeling. IEEE Transactions on Software Engineering 28 (8), 747–762.

Costagliola, G., De Lucia, A., Orefice, S., Tortora, G., 1997. A parsing methodology for the implementation of visual systems. IEEE Transactions on Software Engineering 23 (12), 777–799.

Costagliola, G., Polese, G., 2000. Extended positional grammars In: Proceedings of the 16th IEEE Symposium on Visual Languages, pp. 103–110.

Costagliola, G., Deufemia, V., Polese, G., 2004. A framework for modeling and implementing visual notations with applications to software engineering. ACM Transactions on Software Engineering and Methodology, 431–487.

Cox, P.T., Smedley, T., 2000. Building environments for visual programming of robots by demonstration. Journal of Visual Languages & Computing 11 (5), 549–571.

Crane, M.L., Dingel, J., 2005. On the semantics of UML state machines: categorization and comparison. Technical report, Queen's University.

Engels, G., Hausmann, J.H., Heckel, R., Sauer, S., 2000. Dynamic meta modeling: a graphical approach to the operational semantics of behavioral diagrams in UML. In: Proceedings of the UML 2000. LNCS 1939. pp. 323–337.

Engels, G., Heckel, R., Küster, J.M., 2001. Rule-based specification of behavioral consistency based on the UML meta model. In: Proceedings of the UML 2001, pp. 272–286.

Ermel, C., Rudolf, M., Taentzer, G., 1999. The AGG approach: language and environment. Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools vol. 2, 551–604.

Ermel, C., Hülscher, K., Kuske, S., Ziemann, P., 2005. Animated simulation of integrated UML behavioral models based on graph transformation. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 125–133.

Evans, A., France, R., Grant, E., 1999. Towards formal reasoning with UML models. In: Proceedings of the OOPSLA'99 Workshop on Behavioral Semantics.

Fischer, T., Niere, J., Torunski, L., Zündorf, A., 1998. Story diagrams: a new graph rewrite language based on the unified modeling language and java. In: Proceedings of the Theory and Application to Graph Transformations. LNCS 1764, pp. 296–309.

Garlan, D., Cheng, S.W., Kompanek, A.J., 2002. Reconciling the needs of architectural description with object-modeling notations. Science of Computer Programming 44, 23–49.

Geiger, L., Zündorf, A., 2004. Statechart modeling with Fujaba. In: Proceedings of the International Workshop on Graph-Based Tools.

Gogolla, M., Presicce, F.P., 1998. State diagrams in UML: a formal semantics using graph transformations. In: Proceedings of the ICSE'98 Workshop Precise Semantics of Modeling Techniques. Technical Report TUM-I9803, pp. 55–72.

Gogolla, M., Ziemann, P., Kuske, S., 2002. Towards an integrated graph based semantics for UML. In: Proceedings of the Graph Transformation and Visual Modeling Techniques.

Harel, D., Namaad, A., 1996. The STATEMATE semantics of statecharts. ACM Transactions on Software Engineering and Methodology 5 (4), 293–333.

Hoare, C.A.R., 1985. Communicating sequential processes. Prentice Hall International Series in Computer Science. Prentice-Hall International, New Jersey.

Hölscher, K., Ziemann, P., Gogolla, M., 2006. On translating UML models into graph transformation systems. Journal of Visual Languages and Computing 17 (1), 78–105.

Hu, Z., Shatz, S.M., 2006. Explicit modeling of semantics associated with composite states in UML statecharts. Journal of Automated Software Engineering 13 (4), 423–467.

Jin, Y., Esser, R., Janneck, J.W., 2004. A method for describing the syntax and semantics of UML statecharts. Journal of Software and Systems Modeling 3, 150–163.

Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T., 2003. Model-integrated development of embedded software. Proceedings of the IEEE 91 (1), 145–164.

Kong, J., Song, G.L., Dong, J., 2005. Specifying behavioral semantics through graph transformation. In: Proceedings of the IEEE VL/HCC'05 Workshop on Visual Modeling for Software Intensive Systems (VMSIS), pp. 51–58.
Kong, J., Zhang, K., Zeng, X.Q., 2006. Spatial graph grammar for graphic user interfaces. ACM Transactions on Human–Computer Interaction 13 (2), 268–307.
Kuske, S., 2001. A formal semantics of UML state machines based on structured graph transformation. In: Proceedings of the UML 2001, pp. 241–256.
Kuske, S., Gogolla, M., Kollmann, R., Kreowski, H.J., 2002. An integrated semantics for UML class, object and state diagrams based on graph transformation. In: Proceedings of the 3rd International Conference on Integrated Formal Methods. LNCS 2335, pp. 11–28.
Kwon, G., 2000. Rewrite rules and operational semantics for model checking UML statecharts. In: Proceedings of the UML 2000, pp. 528–540.
Latella, D., Majzik, I., Massink, M., 1999. Towards a formal operational semantics of UML statechart diagrams. In: Proceedings of the IFIP TC6/WG6.1 3rd International Conference on Formal Methods for Open Object-Oriented Distributed Systems, pp. 331–347.
Lilius, J., Paltor, I.P., 1999. Formalizing UML state machines for model checking. In: Proceedings UML'99. LNCS 1723, pp. 430–444.
Maggiolo-Schettini, A., Peron, A., 1994. Semantics of full statecharts based on graph rewriting. In: Proceedings of the Graph Transformation in Computer Science. LNCS 776. pp. 265–279.
Maggiolo-Schettini, A., Peron, A., 1996. A graph rewriting framework for statecharts semantics. In: Proceedings of the 5th International Workshop on Graph Grammars and their Application to Computer Science. LNCS 1073. pp. 107–121.
McMillan, K.L., 1992. Symbolic model checking: an approach to the state explosion problem. Ph.D. thesis. Department of Computer Science, Carnegie Mellon University.
McUmber, W.E., Cheng, B.H.C., 2001. A general framework for formalizing UML with formal languages. In: Proceedings of the International Conference on Software Engineering, pp. 433–442.
Medvidovic, N., Taylor, R.N., 2000. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26 (1), 70–93.
Medvidovic, N., Rosenblum, D.S., Redmiles, D.F., Robbins, J.E., 2002. Modeling software architectures in the unified modeling language. ACM Transactions on Software Engineering and Methodology 11 (1), 2–57.
Mellow, S.J., Balcer, M.J., 2002. Executable UML – A Foundation for Model-Driven Architecture. Addison-Wesley, New York.
Métayer, D.L., 1998. Describing software architecture styles using graph grammars. IEEE Transactions on Software Engineering 24 (7), 521–533.
Object Management Group, 2005. Unified Modeling Language: Superstructure. Version 2.0.
Raistrick, C., Francis, P., Wright, J., 2004. Model Driven Architecture with Executable UML. Cambridge University Press, Cambridge.
Rekers, J., Schürr, A., 1997. Defining and parsing visual languages with layered graph grammars. Journal of Visual Languages and Computing 8 (1), 27–55.
Reggio, G., Astesiano, E., Choppy, C., Hussmann, H., 2000. Analyzing UML active classes and associated state machines – a lightweight formal approach. Proceedings of the FASE 2000. LNCS 1783. pp. 127–146.
Rozenberg, G. (Ed.), 1997. Handbook of Graph Grammars and Computing by Graph Transformation: Foundations, vol. 1. World Scientific, Singapore.
Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., 1991. Object-oriented modeling and design. Prentice-Hall, Englewood Cliffs, NJ.
Rumbaugh, J., Jacobson, I., Booch, G., 2005. The Unified Modeling Language Reference Manual, second ed. Addison-Wesley, New York.
Schattkowsky, T., Müller, W., 2004. Model-based design of embedded systems. In: Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-time Distributed Computing. pp. 113–128.
Schattkowsky, T., Müller, W., 2005. Transformation of UML state machines for direct execution. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 117–124.
Schürr, A., Winter, A., Zündorf, A., 1995. Graph grammar engineering with PROGRES. In: Proceedings ESEC'95. LNCS 989. pp. 219–234.
Starr, L., 2001. Executable UML: How to Build Class Models. Prentice-Hall, New Jersey.
Taentzer, G., Geodicke, M., Meyer, T., 1998. Dynamic change management by distributed graph transformation: towards configurable distributed systems. In: Proceedings of the 6th International Workshop Theory and Application of Graph Transformations. LNCS 1764. pp. 179–193.
Varró, D., 2002. A formal semantics of UML statecharts by model transition systems. In: Proceedings ICGT 2002. LNCS 2505. pp. 378–392.
Zhang, D.Q., Zhang, K., Cao, J., 2001a. A context-sensitive graph grammar formalism for the specification of visual languages. Computer Journal 44 (3), 187–200.
Zhang, K., Zhang, D-Q., Cao, J., 2001b. Design, construction, and application of a generic visual language generation environment. IEEE Transactions on Software Engineering 27 (4), 289–307.
Zhao, Y., Fan, Y., Bai, X., Wang, Y., Cai, H., Ding, W., 2004. Towards formal verification of UML diagrams based on graph transformation. In: Proceedings of the IEEE International Conference on E-Commerce Technology for Dynamic E-Business, pp. 180–187.
Ziemann, P., Hölscher, K., Gogolla, M., 2004a. From UML models to graph transformation systems. In: Proceedings of the Workshop on Visual Languages and Formal Methods. Electronic Notes in Theoretical Computer Science (ENTCS). pp. 17–33.
Ziemann, P., Hölscher, K., Gogolla, M., 2004b. Coherently explaining UML statechart and collaboration diagrams by graph transformations. In: Proceedings of the Brazilian Symposium on Formal Methods. Electronic Notes in Theoretical Computer Science (ENTCS). pp. 263–280.

Jun Kong is an assistant professor of Computer Science at North Dakota State University. His research and teaching interests include software modeling and design, pervasive computing, human-computer interaction and visual languages.

Kang Zhang is a Professor of Computer Science at the University of Texas at Dallas. His research interests include visual languages, information visualization, and their applications in software engineering and Web engineering. Further details can be found in his home page: www.utdallas.edu/~kzhang.

Jing Dong is an assistant professor in the Department of Computer Science at the University of Texas at Dallas. His research and teaching interests include formal and automated methods for software engineering, software modeling and design, service-oriented architecture, and visualization. He is a member of the IEEE and the ACM.

Dianxiang Xu received the B.S., M.S., and Ph.D. degrees in Computer Science from Nanjing University, China in 1989, 1992, and 1995, respectively. He is assistant professor of computer science at North Dakota State University, USA. From August 2000 to July 2003, he was research assistant professor and engineer in the Computer Science Department at Texas A&M University. From May 1999 to August 2000, he was a research associate at Florida International University. Prior to that, he was associate professor in the Department of Computer Science and Technology, Nanjing University. His research interests are in the areas of software security, software testing, aspect-oriented software development, applied formal methods, and software agents.